Semantically Annotated Multi-Protocol Adapter Nodes: A New Approach to Implementing Network-Based Information Systems Using Ontologies

Nickolas John Gowland Falkner

The School of Computer Science
The University of Adelaide

October 30, 2007
# Contents

1 **Introduction** 15
   1.1 Computing Resources ................................................. 16
   1.2 Distributed Resource Usage ....................................... 19
   1.3 Network-based Resources ........................................... 19
      1.3.1 The Domain Name System ..................................... 20
      1.3.2 The Network Time Protocol ................................... 20
   1.4 Large-scale network-based information services .................. 21
   1.5 Standards-based distributed systems ............................... 22
   1.6 User Requirements .................................................. 23
   1.7 Global and Local Requirements ..................................... 24
   1.8 The requirement for semantic alignment ............................ 25

2 **Literature Review** 28
   2.1 Introduction .................................................................. 28
      2.1.1 Data, Information and Knowledge .............................. 29
      2.1.2 Chapter organisation ............................................. 29
   2.2 Data ............................................................................. 30
      2.2.1 Operating Systems .................................................. 31
      2.2.2 File systems and distributed file systems .................... 32
      2.2.3 Databases and Database Views ................................ 34
      2.2.4 Data storage and retrieval across networks .................. 35
         2.2.4.1 Network-based file systems ................................ 35
         2.2.4.2 Potential issues with network-based file systems ...... 36
   2.2.5 Distributed Systems ................................................ 38
      2.2.5.1 Definitions ....................................................... 38
      2.2.5.2 Models for distributed systems .............................. 39
      2.2.5.3 Developing Distributed Systems with proprietary formats 40
      2.2.5.4 Moving to open development frameworks ................. 41
3 Challenges to implementing a network-based information system

3.1 Addressing the challenges to implementing a network-based information system

3.1.1 Heterogeneity

3.1.2 Public Network

3.1.2.1 The network as black box

3.1.3 Non-ownership

3.1.4 Non-local storage

3.1.5 Protocols and policies

3.2 Summary

4 Methodology

4.1 Developing a New Approach to Implementing NBIS

4.1.1 Knowledge Representation

4.1.1.1 Mechanisms for Representing Knowledge

4.1.1.2 Syntactic and Semantic Elements of Knowledge Representation Mechanisms

4.1.1.3 High-Availability and Remote Access Mechanisms

4.1.2 Context

4.1.3 Data, Information and Knowledge

4.1.4 A three-tiered software architecture

4.1.4.1 Ontological Specification Tier

4.1.4.2 Data Model Tier

4.1.4.3 Programming Language Semantics Tier

4.1.4.4 Encapsulation in the software architecture

4.2 The Nature of the Knowledge Contained in a System

4.2.1 Contextual Inference in Traditional Programming Systems

4.2.2 What constitutes the knowledge of a target system?

4.2.3 Modelling a system ontologically

4.2.4 An example of ontological capture in the software architecture

4.2.5 Choosing the correct level of metadata

4.2.5.1 The nature of the underlying store

5 Application of the Methodology

5.1 Target systems

5.1.1 Evolution

5.1.2 Data Exploitation
5.1.3 Local Specialisation ........................................... 119
5.1.4 Well-defined Purpose ........................................ 121
5.1.5 Metadata ....................................................... 122
5.2 Comparison with Existing Approaches .............................. 122
5.2.1 Workflow-based approach .................................... 123
5.2.2 Orchestrative and Choreographic Approaches .................. 124
5.2.3 Service Oriented Architectures (SOA) .......................... 125
5.2.4 ACID-based transaction model ................................ 126
5.3 How the three-tier software architecture and KR-based approach meet the shortfalls ........................................ 129
5.3.1 Optimising data handling .................................... 130
5.3.2 Simple versus Complex System Nodes ......................... 131
5.4 Modelling Program Operation and Knowledge Domains ............ 132
5.4.1 Aspects Common to All Distributed Systems .................. 133
  5.4.1.1 Process Management ...................................... 134
  5.4.1.2 Input/Output ............................................. 134
  5.4.1.3 Concealment of implementation .......................... 135
  5.4.1.4 The use of knowledge in the system ..................... 135
5.4.2 Distinct Aspects of Different Distributed Systems ................ 137
5.4.3 Modelling operating systems and distributed information systems 138
5.4.4 Fundamental Functions ....................................... 141
5.4.5 Remotely Installable Native Functions ......................... 144
5.4.6 Fitting into the model ....................................... 146
6 Conceptual Model ..................................................... 147
6.1 The Problem Domain ............................................ 147
6.2 High-level Conceptual Model .................................... 149
  6.2.1 Query formats and the store ................................ 153
    6.2.1.1 Terminal and non-terminal graph nodes ................. 154
    6.2.1.2 Summary of query operations .......................... 155
  6.2.2 Operational Semantics ..................................... 155
  6.2.3 Object Conceptualisation ................................... 157
6.3 Formal and Theoretical Basis: RDF and OWL ...................... 157
  6.3.1 RDF .......................................................... 159
    6.3.1.1 Blank Nodes and Interpretations ....................... 161
    6.3.1.2 Entailment ............................................. 161
6.3.2 OWL ............................................. 162
  6.3.2.1 Entailment in OWL and Decidability ............. 162
  6.3.2.2 Significance ................................. 163
6.3.3 Ontological grounding in RDF and OWL semantics .... 164
  6.3.3.1 Entailment of operational semantics .............. 164
  6.3.3.2 Is entailment required? ......................... 166
6.4 Summary of Methodology .................................. 168

7 Implementation ............................................ 170
  7.1 Pragmatic Requirements of the Model ..................... 171
    7.1.1 Top level structure of the ontology ................. 172
    7.1.2 Lower level structure of the Functional Branch .... 174
    7.1.3 Data Model Issues .............................. 175
      7.1.3.1 Type Information ............................ 175
      7.1.3.2 Derived Datatypes ............................ 176
      7.1.3.3 Introducing new types ......................... 177
  7.2 Choosing a target system ................................ 178
  7.3 The Domain Name System (DNS) ........................ 179
    7.3.1 Operational Issues in the DNS .................... 180
    7.3.2 The effect of strict control on system modification .. 181
    7.3.3 Enforced configuration versus evolutionary configuration ... 181
    7.3.4 The DNS and semantic enhancement technologies .... 183
  7.4 The two implementations of the DNS in a partial and more developed KR framework ............... 184
  7.5 Problems in self-definition ........................... 184
  7.6 Developing a semantic model of the DNS ................ 185
  7.7 Mapping the DNS into the model ........................ 186
    7.7.1 Mapping into the Ontology and Data Model .......... 187
  7.8 Describing the operational semantics .................... 188
    7.8.1 A language for capturing operational semantics ...... 191
    7.8.2 Language Choice ................................ 192
  7.9 Ontological development of the Operational Semantics sub-branch ......................... 192
  7.10 Native Functions ................................... 195
  7.11 Application: Extending the functionality of Internet services .................... 196
    7.11.1 Extending an existing service ................. 196
  7.12 Implementation of an ontology based DNS server .... 199
7.13 Ontological Reasoning and Entailment ........................................... 203
7.14 Policy specifications ................................................................. 206
7.15 Aggregating metadata and data ................................................... 207
7.16 Ontology support in the client ..................................................... 209
7.17 Summary .................................................................

8 Results 211

8.1 Experimental Outline ............................................................... 211
  8.1.1 DNS structure ................................................................. 212
8.2 Testbed Environment ................................................................. 213
8.3 $\alpha$-system experiments .......................................................... 214
  8.3.1 Time Complexity of Queries .................................................... 217
8.4 $\beta$-system experiments ............................................................ 218
  8.4.1 Multi-protocol server ........................................................... 218
8.5 Breakdown of performance components in the
  $\beta$-implementation ................................................................. 221
  8.5.1 Experiment 2-1 ................................................................. 223
    8.5.1.1 Analysis of Experiment 2-1 results ..................................... 224
  8.5.2 Experiment 2-2 ................................................................. 226
    8.5.2.1 Complexity rating for ontologies ....................................... 226
    8.5.2.2 Test Ontologies for Experiment 2-2 .................................... 227
    8.5.2.3 Analysis of Experiment 2-2 results ..................................... 227
  8.5.3 Subsequent Experiments with Faster Equipment ......................... 230
    8.5.3.1 Experiment 2-1-N results ............................................... 231
    8.5.3.2 Experiment 2-2-N results ............................................... 231
    8.5.3.3 Analysis of new performance results ................................... 233
  8.5.4 Further optimisation ............................................................ 233
    8.5.4.1 Experiment 2-1-N-C results ............................................. 234
    8.5.4.2 Experiment 2-2-N-C results ............................................. 234
    8.5.4.3 Analysis of results of the optimised code experiments ............. 234
  8.5.5 Complexity of queries ........................................................ 235
  8.5.6 Overhead due to multi-protocol issues ..................................... 236
  8.5.7 Performance bottlenecks ....................................................... 238
  8.5.8 Optimisation points ........................................................... 238
8.6 Summary ................................................................. 238
9 Future Work and Applications

9.1 Extensions .............................................. 240
  9.1.1 Operational Semantics ................................. 240
  9.1.2 Representation of native functions .................... 241
  9.1.3 Self-healing mechanisms ............................... 242
  9.1.4 Confidence-based protocol determination ............... 243
  9.1.5 Ontology repair for DL features ....................... 244
  9.1.6 Name abstraction .................................... 245
  9.1.7 Denotational Semantics ............................... 245

9.2 Applications ............................................ 246
  9.2.1 Interoperation with WSDL ............................. 246
  9.2.2 Interoperation with SAWSDL ......................... 247
  9.2.3 Interoperation with OWL-S ............................ 247
  9.2.4 Semantic Web ....................................... 248
  9.2.5 General distributed systems ......................... 249
  9.2.6 Grid computing ...................................... 251

9.3 Summary ................................................ 252

10 Conclusions .............................................. 253

10.1 Specifying Semantics ...................................... 254

10.2 Contribution ............................................. 255

10.3 Meeting User Expectations of a Distributed Information System .... 256
  10.3.1 Meeting Global and Local Requirements ............... 257
  10.3.2 Performance Implications ............................. 258

10.4 The Advantages of a Knowledge-Based Approach ...................... 259
  10.4.1 Explicit Ontological Representation Minimises Confusion .... 259
    10.4.1.1 Shared Concepts Allows Much Wider Reuse of Data ... 260
  10.4.2 Reduced Double-Handling of Data ..................... 260
  10.4.3 Increased Opportunities for Collaboration .............. 261
  10.4.4 Decoupling of Stored Data from External Representations .... 262

A The core ontology for DNS operations ............................ 264
# List of Figures

1.1 A simple Java program ................................. 17

2.1 Dublin Core Metadata .................................. 57
2.2 Dublin Core metadata in HTML ...................... 57
2.3 Dublin Core metadata in RDF [61] ................... 58
2.4 An example of OWL DL ............................... 70
2.5 An example of KIF preconditions [98] ............... 74
2.6 myGrid services and middleware stack from [64]. .... 76

4.1 Block structure of an ontologically-defined NBIS node .... 104
4.2 Encapsulation in the three-tier software architecture .......... 108
4.3 The three branch ontology ............................. 112

5.1 The three-tier software architecture, with an operating system mapping ... 136
5.2 Fundamental function sets based on system requirements .... 142

6.1 The problem domain for widely-distributed network-based information systems ......................... 148
6.2 A network-based information system .................... 148
6.3 The high-level functional conceptual representation of a network-based information system server ..................... 150
6.4 The high-level functional conceptual representation of a network-based information system client ................. 150
6.5 The high-level functional conceptual representation with context establishment ......................... 152
6.6 The high-level functional conceptual representation showing detail of the ‘Act’ processes .................... 153
6.7 A UML Sequence Diagram for the Object Conceptualisation 158
6.8 An RDF Graph showing urirefs and literals, connected by arcs. 160
6.9 The RDF Graph from Figure 6.8 shown in RDF/XML ........ 160
6.10 Example graph form of a Lambda expression ................. 165
7.1 The minimal base ontology that maintains strong context .............. 173
7.2 The location of traditional DNS semantics in the UML sequence model . 188
7.3 The pre-implementation model of the DNS with operational semantics . 189
7.4 The minimal base ontology with DNS behaviour overlaid ............... 190
7.5 The location of the fixed point. .......................................... 193
7.6 Using specifications to encapsulate data for transport to the client ....... 194
7.7 An example of the applied λ-calculus. .................................... 195
7.8 The post-implementation model of the DNS with operational semantics . 201
7.9 The operational semantics component, showing cryptographic functions . 202
7.10 A simple DNS query illustrated. ........................................... 207
7.11 A multi-protocol response to a simple DNS query ....................... 208
7.12 A metadata-rich response to a DNS query from an enhanced system .... 208

8.1 An example DNS structure showing the location of ontologically enhanced
nodes ................................................. 213
8.2 The delegation of name domains to servers in the DNS .................. 214
8.3 Multi-format service on separate ports. .................................... 220
8.4 Multi-format service on a shared port. .................................... 220
8.5 The graph of complexity metric versus relative store size ................ 229
8.6 The graph of relative store size versus TREE and LOAD times .......... 230
8.7 The graph of complexity metric versus TREE and LOAD times .......... 231

9.1 Adapter nodes on the boundary displayed against adapter nodes without
occlusion ............................................. 250
9.2 Adapter nodes on the boundary displayed against adapter nodes as occluders 250

10.1 The expected deployment of adapter nodes versus standard nodes ....... 259
List of Tables

2.1 A Classification of Distributed Systems ........................................ 44
8.1 Performance measurements ..................................................... 215
8.2 Client-side Performance measurements with ontology load ............. 219
8.3 Server-side Performance measurements without ontology load ........... 219
8.4 Client and server operations .................................................... 221
8.5 Results of experiment 2-1, showing time spent in each activity .......... 224
8.6 Test ontologies for experiment 2-2 showing complexity metric .......... 228
8.7 LOAD and TREE times for the ontologies in experiment 2-2. ............. 228
8.8 Results of experiment 2-1-N, showing time spent in each activity ....... 232
8.9 LOAD and TREE times for the ontologies in experiment 2-2-N. .......... 232
8.10 Results of experiment 2-1-N-C, showing time spent in each activity ... 234
8.11 LOAD and TREE times for the ontologies in experiment 2-2-N-C. ...... 234
Abstract

Network-based information systems are an important class of distributed systems that serve large and diverse user communities with information and essential network services. Centrally defined standards for interoperation and information exchange ensure that any required functionality is provided but do so at the expense of flexibility and ease of system evolution. This thesis presents a novel approach to implementing network-based information systems in a knowledge-representation-based format using an ontological description of the service. Our approach allows us to provide flexible distributed systems that can conform to global standards while still allowing local developments and protocol extensions.

We can share data between systems if we provide an explicit specification of the relationship between the knowledge in the system and the structure and nature of the values shared between systems. Existing distributed systems may share data based on the values and structures of that data but we go beyond syntax-based value exchange to introduce a semantically-based exchange of knowledge. The explicit statement of the semantics and syntax of the system in a machine-interpretable form provides the automated integration of different systems through the use of adapter nodes. Adapter nodes are members of more than one system and seamlessly transport data between the systems.

We develop a multi-tier software architecture that characterises the values held inside the system depending on an ontological classification of their structure and context to allow the definition of values in terms of the knowledge that they represent. Initially, received values are viewed as data, with no structural information. Structural and type information, and the context of the value can now be associated with it through the use of ontologies, leading to a value-form referred to as knowledge: a value that is structurally and contextually rich. This is demonstrated through an implementation process employing RDF, OWL and SPARQL to develop an ontological description of a network-based information system. The implementation provides evidence for the benefits and costs of representing a system in such a manner, including a complexity-based analysis of system performance.

The implementation demonstrates the ability of such a representation to separate global standards-based requirements from local user requirements. This allows the addition of
behaviour, specific to local needs, to otherwise global systems in a way that does not compromise the global standards.

Our contribution is in providing a means for network-based information systems to retain the benefits of their global interaction while still allowing local customisation to meet the user expectations. This thesis presents a novel use of ontologically-based representation and tools to demonstrate the benefits of the multi-tier software architecture with a separation of the contents of the system into data, information and knowledge. Our approach increases the ease of interoperation for large-scale distributed systems and facilitates the development of systems that can adapt to local requirements while retaining their wider interoperability. Further, our approach provides a strong contextual framework to ground concepts in the system and also supports the amalgamation of data from many sources to provide rich and extensible network-based information system.
Declaration

This work contains no material that has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Signed:

Nickolas John Gowland Falkner
Acknowledgements

My thanks go to my supervisors, Dr Paul Coddington and Dr Andrew Wendelborn, as they have both helped me greatly with their advice, effort and ongoing support. Their insight and suggestions have been invaluable and I am a better researcher for it.

My peers, among the postgraduate students, and, later, also among the staff, academic and professional, have been of great help with their ongoing comments, assistance and willingness to enter into discussions. Associate Professor David Munro shared a number of his experiences from his journey into Academia and provided me with some much-needed confidence-boosting at points where I was doubting myself and the journey as a whole.

I would also like to thank my mother for the love, support and educational background with which she provided me.

My thanks also to my friends, who have been very tolerant while I tried hard not to dominate simple ‘how are you’ queries with a detailed discussion of the problem of loose contextual frameworks. I didn’t always succeed but they keep inviting me to dinner.

Finally, I would like to thank my wife who has supported me with a great deal of love and good humour while I undertook this project. She has also helped with a great deal of good advice based on her own experience as a PhD student and as a hard-working academic. She has also shown quite amazing discretion and judgement in the way in which she shared her experience with me. This work would not have been completed without her.

Thank you all, very much.
Chapter 1

Introduction

Network-based information systems are designed to provide information to their users over standardised access mechanisms and network protocols. These large and complex systems have a global span due to their reliance on well-defined and strictly-controlled standards. However, these standards can prevent a subset of users, or an individual site, from customising their implementation of the information system to meet their own needs. The standards, which are traditionally neither machine interpretable nor easily altered, can restrict the possible uses of a system to a subset that meets the overall need. As the standards are not machine-interpretable, the data standards and protocols of one system must be explicitly coded into another system if inter-system transfer is to take place. Thus, our problem is that these useful information systems could be made far more useful by adapting to user needs both through customisation and through increased interoperability. We do not want to sacrifice the global standards that allow for remote, asynchronous development of system components but we no longer wish to be unnecessarily constrained in the ways in which we can use the information in a system.

Our aim is to increase the interoperability of network-based information systems while providing a well-defined mechanism to allow site-based specification for nodes within the network-based information system. Our approach is to provide a mechanism for implementing network-based information systems with a knowledge-based representation utilising ontologies. We use ontologies to explicitly specify the relationships, structures and operational semantics of all values contained in the system. This allows us to reuse data, share data with other systems and provide machine-interpretable mechanisms that will facilitate high-speed exchange between heterogeneous systems. Network-based information systems (NBIS) are large and frequently-used distributed systems, but their scale and characteristics have made it difficult, up until now, to address automated evolution, translation and inter-system data exchange. Developments in the use of metadata, ontologies and rea-
soning systems have now made alternative mechanisms for implementing NBIS feasible. In this thesis, we introduce a framework for supporting interoperability and local customisation in a standards-based environment that builds on these existing developments. The nodes that we will develop to support the implementation of our approach will be referred to as *adapter nodes* as they can adapt data from one system to another, *boundary nodes* as they logically sit on the boundary between the two knowledge domains in two systems and, once they have been fully developed and implemented, as *Semantically Annotated Multi-Protocol Adapter Nodes* (SAMPAN) as they employ a knowledge-based representation to support as many systems as are required, employ semantic alignment, and occupy the adapter role.

In this chapter, we will introduce the basis of modern computing resources and the nature of distributed resource usage. This leads to a discussion of standards and user requirements, including the occasional conflict between the requirements of all users in a system versus the requirements of a smaller group of users. There is also an occasional conflict between the requirement for standardised behaviour and the requirement for flexible and evolving systems. Finally, we use these discussions to argue for a requirement for semantic alignment, where we can establish the similarity in meaning of two apparently dissimilar data, in order to make the best use of the resources in the network - for all users.

### 1.1 Computing Resources

Computing resources are used to provide solutions to problems. An unsolved problem constitutes an absence of knowledge in a given problem domain. To address the absence of knowledge, a solution must be synthesised by assembling the required resources to deliver the missing knowledge. Very few computer programs are developed with a complete integration of the knowledge framework of the problem domain, as this requires the articulation of the problem domain into a machine-interpretable knowledge framework that is then used throughout the program. Traditionally, the raw values that are used to produce answers are treated as typed values and the programmer’s implicit knowledge representation framework is used to assemble these typed values in ways that produce an answer that is sensible in the contextual framework of the problem domain.

The pivotal concept behind this view of programming is that there are three different classes of resources used inside a program [1]: the data (raw values), information, which has associated typing and can hence be used unambiguously by the semantics of a programming language, and, finally, knowledge. A value that has a defined type and has a well-defined context for its value, type and location inside the problem domain’s knowl-
public class SimpProg {

    private static int ftc(int x) {
        int y = x / (212/100);
        return y;
    }

    public static void main(String[] args) {
        System.out.println(ftc(212));
    }
}

Figure 1.1: A simple Java program

edge representation has finally become knowledge. This is the level at which most real-world problems are expressed although the knowledge level is often abstracted away and is not explicitly codified. Within a given programming language, a computer program consists of a set of syntactically-correct statements that represent a semantically-correct subset of potential operations in that programming language. It is relatively uncommon that the semantics and syntax of the programming language provide a strong semantic foundation for the semantics of a given problem domain, unless the program in question is a compiler. Even where some semantic alignment exists, it is uncommon to find this extending to the type definitions used to assemble the syntactic elements of the program. Consider the program in Figure 1.1.

A brief glance from an experienced programmer will determine two pieces of information. The first is that this is probably a program to perform the conversion between Fahrenheit and Celsius temperatures. The second is that, in Java at least, this will produce the incorrect result. The semantic grounding for both of these pieces of information come from the universe of information beyond the program. The temperature conversion is likely because of both the division constant that links the boiling point of water in F and C respectively and is confirmed by the name of the method ‘ftc’. The algorithm is incorrect because it is missing the baseline adjustment that is required as the two scales do not share a zero. The implementation of the algorithm is also incorrect as, in Java, 212/100 will return the value 2 due to integer division.

While it is too much to expect that a programming system be incapable of constructing incorrect programs, it is not too much to expect that, where context can be provided for data, that context would be used to ensure that only correct operations are used on this data. In
other words, we cannot guarantee that the algorithm will be correct but, with the correct
contextual framework for the data, we may be able to ensure that the implementation is the
most correct that it can be from a given algorithm.

In Figure 1.1, if there existed a type framework that defined real-world temperatures
as a subclass of continuous real-world measurements and, hence, required that any im-
plementation of these types be floating point, rather than integer, we would eliminate one
major fault. Additionally, if we defined further subtypes to separate different temperature
scales and prevented direct assignment except through intermediate, or conversion, types
this would also clearly illustrate an intention to a human reader as well as prevent acciden-
tal mixing of temperature values. While it is entirely feasible for an object-oriented frame-
work to provide such a mechanism, the object-oriented approach would only apply where
the programmer inherited or implemented the appropriate classes. Even if implemented in
a programming language, the mechanism is still not truly open for interoperation and anal-
ysis. It is also not portable across different languages with a guarantee of interoperation or
shared understanding, even if pre-compiled class objects are available as a shared library.

There is a great deal of implicit knowledge associated with our understanding of Fig-
ure 1.1 and a surprisingly small amount of that knowledge is available to the compiler or
naive human reader. This limits the compile-time and the run-time checking that can be
carried out and can limit the human reader in writing correct code or detecting the presence
of incorrect code.

The absence of an explicitly stated knowledge layer, or semantic context, for informa-
tion restricts the amount of information that can be automatically and reliably extracted
from a previously written program. It also restricts the exchange of knowledge concepts
between systems, especially legacy systems, as any change to provided or required infor-
mation requires a rewriting of the underlying program to change the way that the values are
used and possibly even the way in which the types are handled. However, a more signif-
ient problem exists when information without context is to be used in combination with
information that does have context - whether that is the combination of legacy system out-
put with metadata-annotated web-based systems, such as the Semantic Web, or an attempt
to use undocumented output by a human operator who believes that the output is for one
purpose, when it is actually for another.

Apart from the resources of data, information and knowledge contained in a program
there are many other essential computational resources, including networking and compu-
tational storage, that can be used to form larger sets of computational resources or can be
used in other ways to assist in problem solving. These means for assembling computational
resources have allowed the development of distributed computation and other information
handling models. These models assist in problem solving but, for the majority, operate at an informational level rather than a knowledge-based level.

1.2 Distributed Resource Usage

The effective and efficient use of computing resources is an important research topic in the fields of distributed and high-performance computing. Computational resources include, but are not limited to, processing devices, short and long-term storage devices and input/output devices. Distributed computation relies on a wide-area network (WAN), in conjunction with a set of local-area networks (LANs), to provide access to computational resources whereas high-performance computing traditionally concentrates on ‘in-box’ data or bus networks, as well as using high-throughout LAN protocols supported by dedicated or special-purpose hardware for clustering between nodes.

The internet provides a mechanism that allows the joining together of computational resources throughout the world to provide large, widely-distributed computational systems. However, such joinings are often restricted to well-defined, standards-based communications to provide information systems that usually meet user requirements but, in some cases, fail to provide data or services in the form precisely required by a user or site. Standards, while very useful, can be inflexible, hard to modify and slow to change to reflect changing requirements. This thesis describes a more flexible approach to implementing network-based information systems that relies upon the specification of the semantics and syntactic structure of data and data-exchange formats of distributed information systems. This allows the user-level customisation of standards-based systems, without compromising the standard, and also allows the joining together of a number of distributed information systems for efficient data exchange. Thus, a single system can dynamically provide data composed of responses from many different information systems.

The widespread availability of the Internet has made it possible to construct very large sets of resources that are both geographically separated and, importantly, administratively separated from each other. Administration separation occurs when two resources are not managed or owned by the same entity but the two entities have come to an, possibly implicit, agreement over co-operative use of their resources.

1.3 Network-based Resources

Other technologies have also flourished as the Internet has been developed, such as the world wide web (WWW) and peer-to-peer (P2P) networking. These facilitate a large virtual
storage system that a user can interrogate using a variety of interfaces to retrieve and store information beyond the confines of their desktop computer. Such large scale information systems can be referred to as network-based information systems (NBIS) as their useful functioning relies upon the network being available so that the information servers, caches and managers can be contacted to provide information as required. This is a decentralised model but, as will be shown later, facilities are often provided in such systems to allow some level of continued function in the event of network failure. Regardless, the authority of any node in the system to provide correct answers is dependant upon its continued connection to the network. A node that is deprived of a network connection for sufficiently long may still contain the correct data but, as this cannot be verified, the data must be labelled as stale until the checks can be made.

The sheer scale of network-based resources had led directly to the development of a number of network-based information systems that not only use the network but facilitate the ongoing use of the network. Examples of these wide-spread and important systems include the Domain Name Service (DNS) and the Network Time Protocol (NTP) hierarchy. Both provide typed values with an implicit knowledge framework to a vast number of consumer computers on a daily basis. These are examples of crucial network services that must be capable of being implemented within any new framework proposed to address and enhance large network-based distributed information services.

1.3.1 The Domain Name System

The DNS provides the mappings from Internet Protocol (IP) address to hostname and vice versa that are used by every network-based user hundreds to thousands of times a day. As will be discussed in more detail in Chapter 2, the DNS is a widely distributed network-based information system with a very large number of servers and clients arranged across the globe. Failure of the DNS, at any level, immediately leads to problems for the affected users although some limited function may remain in the absence of active network connections.

1.3.2 The Network Time Protocol

NTP provides the means for keeping the clocks of the many network-connected computers across the globe in reasonable synchrony with each other. The assumption of synchrony is important for establishing when events take place, despite being geographically distant. Tracking security problems, keeping track of file datestamps, ordering e-mail in the correct sequence and many forms of network traffic control rely upon the communicating machines having a Co-ordinated Universal Time (UTC) time that is within a small error margin [2].
There are far fewer servers providing NTP than are part of the DNS but the information network that exists between members of both or either group has a very important similarity: the ongoing functioning of the system requires information exchange between all of the members or any data stored will rapidly become stale and unusable.

Both the DNS and NTP will be discussed in more detail in Chapter 2 and in the implementation of our system, discussed in Chapter 7.

1.4 Large-scale network-based information services

DNS and NTP require the ongoing collaboration of system users, internet service providers and organisations to provide their service since most of the servers are provided by people and organisations that share no administrative framework with each other. Therefore, there is no administrative element that can prepare each node and deploy it, knowing the state of the system as a whole. In a cluster-based computing environment, homogeneity is often used to minimise administrative overheads in assimilating new technologies or customised environments. In a distributed computing environment, no assumption of homogeneity can be made and, if any assumptions are made, then it must be that the resources available to a widely distributed system will be as heterogeneous as is possible due to socio-political and economic factors combined with geographical diversity. Global knowledge is not necessarily attainable at any given time point, due to issues of synchrony, but a local administration group should have total knowledge of their area of responsibility. This is essential as distributed information systems collate existing information to provide answers to queries - if the owner of the knowledge does not share it or does not know that they have it then it cannot be provided.

Two things must be known before an individual resource is added to an information system. The first is what types of messages will be sent to and from a participating node. Thus, what needs to be known is the nature of the values, their type and the order in which they will be presented to a given node. The second is what transformations the node is responsible for within the system, especially as it applies to the data that traverses the node.

The communication across the network between nodes can be assumed to occur providing that each node is correctly connected to a network and is using the network communication protocols, Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP provides a reliable, fault-tolerant stream-based communication mechanism over unreliable datagrams and are the backbone of the modern Internet. It is, however, what is contained within the transmitted packets that is of importance to the system since this should be data
destined for the system. This payload is not useful unless it can be correctly decoded and interpreted by the receiving node - which assumes that the transmitting node has correctly interpreted and then encoded the information in the first place. This states the clear importance of agreement between sender and receiver over data exchange formats. Data that cannot be correctly sent to, or received from, another node is effectively useless in the network-based information system environment.

In a small system under centralised administrative authority, all nodes can be configured in the same way and any small changes to transmission protocols can be made to all nodes. This scales very poorly, especially for complex changes, and can lead to incorrect operation and inefficiency when computational resources are geographically and administratively separated.

1.5 Standards-based distributed systems

The most important aspect of any large-scale distributed computational effort is the specification of the standards that are required to make the system work. Without these data interchange formats, and specification of operational behaviour, the systems would not work.

Consider the DNS. The namespace of the DNS is composed of domains, descending from a central root ‘.’. Organisations are given responsibility for domains and subdomains, based on the partitioning of the name space and also based on the segmentation of the IP addressing range. Without standards to define interoperation, the administrator of cs.adelaide.edu.au could provide a service that works inside their domain but is completely incompatible with that of adelaide.edu.au or flinders.edu.au. The focus of network-based information services is to make the information available to at least all subscribers and, more likely, all other users who wish to subscribe and who may or may not be members. The DNS uses a large number of standards that define the basic set-up [3], operation [4] and many other aspects [5, 6, 7, 8, 9] of the system to ensure that global operation is provided uniformly, reliably and with the minimum of re-engineering required to achieve the interoperation.

While a standards-based approach is an excellent way to guarantee interoperation between disparate elements of a widely-distributed network-based information system, the benefits come with a price. Such systems are strongly standards-bound and this can lead to two major problems. The first problems occurs when a standard changes and a part of a protocol is added or dropped. In this instance, a mechanism must be employed to allow the continued operation of those nodes supporting a previous version of the system.
a protocol or standard changes, the implementation will also change. Occasionally, a standards change is caused purely by a change in the de facto system standard implementation. Regardless of how it occurs, the decentralised nature of machine location and administration means that there will be a hybrid mode in operation for some time while servers are upgraded or replaced. These legacy modes allow the continued operation of the system in a hybrid mode but also, by not forcing or providing an easy pathway to change, leave the system in a vulnerable state where it has not only the advantages and flaws of one version of the system but, at least, two [10]. The second problem occurs when trying to bring about a change in the first place. A large system with many subscribers will require a well-controlled change process to limit the impact, severity and frequency of changes to those that are strictly necessary, as defined by the majority of users and providers. Because of this, a standards-based change generally requires the production of working groups, working papers and a relatively long-timeframe collaborative consultation process to ensure that the interests of the largest possible subset of users is being met and that any solution that is implemented is the best possible solution at the time.

The major disadvantage that a centrally standardised system brings to an individual user or site is that the system that best meets the requirements of the majority of users may not completely meet the requirements of the individual. Consider the size of an Asynchronous Transfer Mode (ATM) cell: 53 bytes, 5 header and 48 payload. When this new protocol was being designed, two groups ended up in competition regarding the payload size: one wanted 32 (to remove the need for echo cancellation devices in ATM telephony) and the other wanted 64 (to enhance data transmission rates). As neither side could agree, the CCITT (now ITU-T) came up with a compromise and used the value 48. This, rather arbitrary, value is now in the standard and properly meets neither sides original requirements - but equally. A large and complex network-based information system may have hundreds of these decisions made during its lifespan and each and every decision may affect the ease of use at the user end or change behaviour in a way that does not meet with local requirements.

1.6 User Requirements

All users have expectations and requirements of any system that they use. In the case of a large service-oriented system, these expectations become more complex because of the implicit assumptions regarding reliability, ongoing access and the continued operation of the system in the way that it operates today. These requirements are also made more complex when users cannot cause change to occur because they depend upon a system component or computational resource to perform a task correctly but have no explicit and
direct control over a component if it fails to work correctly. The user requirements are a combination of global requirements and local requirements and are discussed briefly here, although they will be discussed in more detail later.

1.7 Global and Local Requirements

A global requirement of a system is a requirement that is shared by every user of the system. These include the continued operation of the system, the provision of a certain set of services in a defined way and a level of reliability that is maintained over time. All users share their global requirements, especially when it comes to the expected behaviour of the system when accessing information outside of the user’s site or domain. A local requirement is a system facet that is particularly well-suited to or has been customised to meet the specific requirements of a user’s site without regard to the global situation. Local requirements are met through two mechanisms: the customisation of a standards-based implementation and the accidental coincidence of a local requirement with an area of the global standard.

In the second case, user needs will continue to be met with the standards-based system, because the standard gives them what they want, even if not everyone wishes to use it. The only threat to this is if a standards-based change occurs that removes this feature as ‘no-one is using it’. In the first case the requirements are far more complex because the user or site must commit to the production of a system that implements all of the standard, while also providing the user-specific alterations that will be available at the local site. This is a large software engineering load to place upon an organisation, especially when the network-based information system has a large number of defining standards or has a large code base. Even allowing for a readily accessible code base, for an open-source system, the time involved in learning how to program the system and developing an alternative operational core for a site is non-trivial. Combined with the overhead of potentially re-engineering the system whenever a standards change is made or in the face of unexpected impact of local changes on standards-based activity elsewhere, there is a strong deterrent to going down this path and devoting the resources to a local version of a global system that behaves in a different manner.

The origin of this problem can be traced directly to the complexity of, and services provided by, a widely-distributed information system. Where a system provides one item in a straight-forward fashion, such as a network time service in the form of NTP, then there is little room for alteration in the provision of such an item. When a system provides a more wide-ranging set of services, such as the many mappings associated with the DNS, then
there is a greater potential for a user or site to find a need for a mapping that is not provided by the standard. The other significant contributor to the need for custom services is the requirement to combine several existing services through some form of service combination mechanism that sensibly composes information from more than one source and provides it to users. Since not all users have the same compositional requirement, any architecture is more likely to provide those combinations that are most commonly requested or, alternatively, more easy to provide. As will be discussed in Chapter 2, the rise of e-Science, the semantic web, software re-use methodologies and the additional requirements for keeping accurate provenance data have greatly contributed to the ability to compose existing services in new and different ways. These new compositions may take forms that the original designers did not necessarily foresee and, as a result, for which they had not provided robust support mechanisms.

While the composition of service output can be achieved with a simple mechanistic process, this is often achieved as an ad hoc join of two elements, rather than a systematic join of all of the corresponding elements in two systems. Without the corresponding association of the data, and underlying context, in interacting systems, there is only a limited vocabulary available for expressing the meaning of the joined data. The joined data may be useful but it is not guaranteed to be as semantically robust as either of its parents and, in fact, may only be interpretable inside a very limited synthetic window that is available only to the user who performed the join. As part of the requirement that the combined systems have meaning, the logical joining of two information systems and their services requires that the data stored by the systems can be transferred from one system to another in a meaningful manner and through a well-defined automated process.

1.8 The requirement for semantic alignment

Systems that require manual intervention to guarantee a successful join for every transaction will not be in use for long or, if they are, risk limiting their use because of the manual overhead. Thus, the data stores of systems that join data must support a semantic alignment, where the meaning and context of the data can be determined, and there must be a syntactic translation between the two forms that the data takes in both systems. If the exchange between systems is specified as data, raw values, then no such semantic alignment can be made in an automatic way because a type equivalence cannot be established and, without that, there is not even the building blocks of alignment. If the data is presented as programmatically useful information, with an associated type, then semantic alignment is still not deterministically established as there is no context for the information. It is only
when the information is represented as knowledge and placed into a strict, and well-defined, context that such alignment can be carried out automatically with a guarantee of success. If two entities agree upon a contextual framework that contains a valid mapping for every piece of information in the discussion then the knowledge associated with the context can be assumed to be the same and semantic alignment can proceed.

The thesis seeks to show how such a semantic and syntactic relationship is specified, and whether it can be specified, in order to allow for site-level customisation of widely-distributed network-based information systems. Such specification also allows the joining of several systems to provide composite data streams that were not necessarily envisioned by the designers of the component systems. By using a specification of the semantics of system data, the structure that the data, and data-exchange formats, it is possible to specify all inter- and intra-systemic operations in terms of alignment between elements of the specification. This approach is designed so that the definition of real-world applications, such as DNS or NTP, can take place in a way that allows their increased flexibility and enhancement without compromising those aspects that must remain immutable to support standards.

The specification of the system, in terms of the system as a knowledge domain, allows the use of external verification tools to validate that the system has been described in a logical and decidable manner and also allows the production of adapter nodes which sit on the edges of two, or more, information systems. The adapter nodes are fully-functional members of all of the systems for which they contain definitions but are able to pass data from one system to the other using the semantic and syntactic alignment information contained in their system specifications. The adapter nodes are also capable of aggregating data from any or all of their sources in order to provide a composite data stream that appears, to the user, as the output from a composite system.

Our approach to implementing network-based information systems, using the specification discussed in the previous paragraphs, solves the problem, outlined earlier in abstract terms, of providing a well-defined and explicit contextual framework for information that only has an implicit context and it also allows the use of a programming model where those elements of a system that are grounded in the knowledge framework can be used unambiguously throughout the system and will maintain the alignment between the program and the semantic framework of the problem domain.

In order to provide a basis for the decisions made in the conceptual model and implementation, this thesis provides a literature review in Chapter 2 to encompass all relevant work in the area and to explicitly define terms used throughout the remainder of the document. There follows a discussion of previous work leading to this point, then a discussion of
relevant technologies of particular interest. These technologies include system modelling, the semantic web, ontologies and the use of component architectures. Chapter 3 presents a discussion on the challenges that occur when implementing a network-based information system.

Chapters 4, 5 and 6 contain the methodology used to solve the outlined problem and concentrates on those aspects that are novel and extensions of existing work. These chapters comprise a discussion of the approach taken to provide the new solution, with the conceptual model and discussions of the implications of the model. Chapter 7 details the implementation of Semantically Annotated Multi-Protocol Adapter Nodes (SAMPAN) as well as an introductory formal basis that is used to justify the application of a contextual framework as a semantic grounding for the knowledge contained in a system. These chapters also outline the capture of knowledge in terms of the underlying values, the structural information, the logical relationships between entities and the operational semantics that are used to capture the operational nature of the program. The operational semantics are captured using an applied λ-calculus, in conjunction with the notion of fundamental functions, where fundamental functions are discrete functional elements that can be composed to provide other, and more complex, operations. These operational semantics are embedded in an OWL ontology which, in conjunction with our developed software architecture and an RDF-based triple store, allows the execution of operations within the system and the provision of data. The ontology contains descriptions and definitions of the data handling mechanisms, data representations and logical relationships between elements.

The final section of the thesis provides the results of experimentation in Chapter 8. These experiments were carried out to verify that the approach works and to also provide performance figures to show the impact of the individual aspects of the solution. The results discussion includes complexity analysis of the underlying system to explain the results obtained. The experiments and complexity analysis are also used to investigate how the performance of the system scales in comparison to the size and the relative structural complexity of the ontology that is used. The results of experimentation, in conjunction with the previous chapters, form the basis of Chapter 9, Future Work and Applications, which discusses other possibilities for this approach. Finally, in Chapter 10, conclusions are drawn from the theoretical and practical work to explicitly state what was and what was not demonstrated by the final model and implementation and what impact this has upon the field.
Chapter 2

Literature Review

2.1 Introduction

This chapter provides both the information required for the development of the new ideas and experiments carried out in later chapters, and a critical assessment of aspects of current research and implementation in order to show the advantages of the new ideas. Network-based information services, under a variety of names, have been in existence from the point at which distributed computing began to show the benefits of allowing several entities access to the same resource. Although, initially, such resources were physical and took the form of disk drives, tape drives or network access points, it was a small step to regard information as a shareable commodity in its own right. From a form that we would today recognise as a distributed data base, the requirements and capabilities of such systems become more complex and more powerful. This led to the development of distributed information systems, network-based information systems and, ultimately, internet-based information systems - terms that can be used to refer to similar systems but with subtle differences.

To begin this chapter, we will introduce a terminology for referring to system values at various stages of processing. This terminology identifies the data items in the system as data, information or knowledge and this will allow us to clearly identify the role of all of the system components that we discuss as it will ultimately fit into our new approach to implementing network-based information systems. We can also associate the different components of existing technologies and our approach with the operations and requirements for a given data value at all stages of its lifespan.

Because this separation of data, information and knowledge is a core component of our approach, this chapter is divided into three major sections: data, information and knowledge.
2.1.1 Data, Information and Knowledge

We adopt the terminology of Quigley [1] when referring to data, information and knowledge, namely that:

- Data is text that answers no questions within the problem space.
- Information is text that can be used to answer physical questions within the problem space.
- Knowledge is text that can be used to answer objective questions within the problem space.

As we will discuss in Chapter 4, in a computational framework, data items may be considered as values with no type information, information has type information but no semantic grounding but knowledge has both a well-defined type and a well-defined context. This context will establish exactly where the knowledge-qualified value can be used while still maintaining semantic sense.

The key concept is that, as the data becomes more context-rich, more can be gained from using it until quite complex concepts can be inferred, and problems answered, with a large amount of context. Modelling a problem requires the determination of the context of each piece of data so that there is sufficient context to obtain knowledge from values passed in as data and also to maintain the context when presenting the information back out. When a problem domain is decomposed into one or a set of intersecting knowledge domains, enough grounding must be in place to answer the questions arising in that domain. Simple mathematical problems are easily placed into the, somewhat abstract, domain of mathematics. It is when the data being handled is context-rich, and that context needs to be maintained, that the number of potential knowledge domains increase. We will explain the details of this in Chapter 4.

2.1.2 Chapter organisation

We build from the simple technologies that provide a basis for distributed computation to the complex and specialised aspects that differentiate technologies that are built from existing distributed systems. We have provided a boundary marker between those aspects that focus on sharing values, rather than a focus on the structure of the information. We then move to a discussion of information-level technologies that have a well-defined mechanism for establishing syntax but do not necessarily agree on semantic interaction. Finally, we introduce the required technologies to establish and agree upon the semantics of a set of
values. We then introduce some key technologies in the area of semantic interoperation, where knowledge is the most valued commodity.

The Data section of the chapter begins with a discussion of the building blocks for distributed computation. Firstly, the operating systems and file systems that form the basis of almost all computational machinery. Then, a description of database technologies to introduce the notion of database views in order to illustrate the importance of the role of localised data ownership and mutability. This leads to a discussion of data storage and retrieval across networks. Following this, distributed systems and network-based information systems (NBIS) are introduced in order to clarify the similarities and distinctions between such systems. As we leave distributed systems, we are moving from treating values as values with a potential for shared syntax, rather than a concrete requirement for shared syntax, to NBIS that must agree upon structure in order to guarantee information-level exchange. This is the start of the Information section of the chapter, where structure is important but semantic agreement is based on syntactic coincidence or implicit agreement rather than through explicit statement.

We then discuss a range of technologies that provide metadata, the description of data, that allows us to exchange enough information about data to be able to reach agreement on structure and type in a flexible and scaleable manner. We also introduce the Grid as a prelude to discussing other mechanisms for assembling computational resources and highlight the fact that these approaches work on a structural level, rather than a semantically-specified level. This ends our discussion of the Informational section and leads immediately into the Knowledge level.

The next sections provide discussion of the knowledge oriented aspects of distributed computing. This starts with a discussion of the underlying technologies required to describe a system, including description logics and the logical and ontological relationships between the data of a system. Finally, existing systems that use these technologies are discussed in order to illustrate precisely where these systems are strong and to illustrate where they can be strengthened. Examples are then given for modern and legacy projects. The chapter concludes with a summary of the perceived shortfalls and gaps in the current literature.

2.2 Data

At the base of every digital computational operation in a Von Neumann computer is a set of binary digits that represent either an operation or data. The interpretation of these values depends upon whether they are interpreted as an operation, to order the processor to do something, or a value, to be used by an operation. Values must be arranged and stored
in ways that allow us to access them in an efficient and application-appropriate manner. However, in this section of the chapter, it is the existence, storage or the sharing of values that are important. We are not yet concerned with the structures or semantics of these values. We begin our discussion at a more abstract level than that of bits in CPU registers - we start at the user interface provided by the operating system.

2.2.1 Operating Systems

The function of an operating system is to provide the user of a computer with an abstraction of an extended or virtual machine that is a simpler programming interface than if the user were dealing directly with the underlying hardware [11, 12]. The operating system also functions as a resource manager with a critical role, especially in multi-user systems, of allocating resources to the system users according to a previously determined strategy and priority based upon the identification of the user. In a single-user system, there must still be prioritisation of effort to allow critical system functions to continue operating despite the additional programmatic tasks that the user requires the computer to undertake. Resources are shared in two ways: time and space. When a resource is shared time-wise, a process or user has exclusive access to it for a given time. Indivisible resources, such as a single processor, are shared time-wise. Divisible resources, such as memory or disk, can be partitioned into separate spaces and multiple users can utilise their portion of a resource without having a direct effect on other users.

With the development of networking technologies, operating systems have also evolved and distributed operating systems have been developed that unify a set of machines into an entity that appears to be one large, virtual machine [13, 14, 15]. Users of a system like this are not required to know where their programs are being run, nor where their files are located. Such information is handled by the distributed operating system, which provides an interface whereby the user can use their files and execute programs without having to know this. Such an operating system, however, requires a detailed knowledge of the operating system executing on the physical computers that comprise the distributed operating system. This immediately makes it difficult for an ad hoc heterogeneous WAN to become the basis for operations within a distributed operating system unless, coincidentally, the computational resources are all of the correct type and have been configured in the correct way.
2.2.2 File systems and distributed file systems

The issue of how and where files are stored in systems is crucial to the effectiveness of the system as a means of processing and storing data. In this section, some existing alternatives are discussed in order to highlight key aspects that apply to distributed systems in general, some registry-based systems and the ultimate descendant of the traditional network-based information system - the persistent archive.

Recently, in the areas concerned with non-local storage, virtual network file systems have come into more widespread use, especially in the P2P user community where large virtual file systems offer what appears to be a coherent repository, despite the fact it is made of many different local file systems, glued together with P2P technology, such as Napster, Gnutella and PAST [16]. These systems use non-trivial routing and resource location protocols to establish where resources are and how to get to them. P2P enabling technologies such as JXTA [17] from Sun Microsystems also provide techniques and environments for developing more loosely arranged storage systems.

The PAST project [16] suggests that several of the key issues in the design of such a loosely distributed file system are:

- System integrity - storage load should be evenly distributed and:
  - Balance - Demand for storage should not exceed supply.
  - Semantics - the file system semantics must be sound for the distributed environment but cannot make many assumptions about the environment as it is in a state of flux.
  - Fair use - Individual nodes should not be capable of sustaining denial of service (DoS), overconsuming bandwidth or mediation resources.

- Data privacy and integrity - Data must be encrypted by a user or the user cannot depend upon ‘good’ behaviour form other nodes. It may be changed or viewed by anyone else using the system, or the owner of the physically hosting node.

- Pseudonymity - The identity of the user is only established through the authentication mechanisms employed systemwide, but implemented locally, and, hence, the strength of the identity check is only as good as the strength of the authentication system. Certificate authorities and strong cryptosystems can establish the identity of a user in association with a given certificate but this does not necessarily establish a identity in any legal or real sense.
The most commonly used distributed file system in use is Sun’s NFS [18, 19, 20], for two reasons. The first is that it has been around for a long time and the second is that it is a well-understood stable technology. NFS is a very tightly coupled system and this allows assumptions to be made about behaviour and performance with more reliability than in a loosely distributed system.

Analysing NFS with regard to the key issues raised by PAST [16]:

- **System Integrity** - there is no implicit or automated load balancing system and:
  - **Balance** - Demand for storage should not exceed supply but there is no mechanism for detecting this other than the disk filling up.
  - **Semantics** - File system semantics may be stricter than for a loosely distributed system as more assumptions can be made about the NFS environment. Since a server must provide the file systems that clients can mount, the server can make assertions about how they are presenting/receiving the file system. Normally this is a non-negotiable set of assertions as clients should not be able to change server behaviour, since this would be an unacceptable security risk.
  - **Fair use** - Individual nodes may be capable of sustaining DoS attacks but these can be managed by the server’s owner under normal system management protocols.

- **Data privacy and integrity** - Data can only be read by the user, those people the user authorises and the superuser. Since the superuser may not be above reproach, encryption is required to guarantee that the superuser has not read or altered the document. Where the superuser is trusted, encryption may not be necessary. However, it must at least be considered in any implementation.

- **Pseudonymity** - NFS needs to map Unix ID (uid) from machine to machine in a 1-to-1 mapping in order to provide the correct file system permissions [18]. Thus machines using NFS must agree upon a username space and hence assume the same identity check. A user cannot connect to a random NFS server unless the server has been incorrectly configured - client and server must work together to ensure that the service works. NFS provides configuration options that also control which accounts are mapped into the local username space. The superuser account, root, has a great deal of power and is either not mapped in or is only mapped in from trusted hosts that are explicitly stated in the configuration file.
2.2.3 Databases and Database Views

Information should be stored consistently and logically. An incoherent, chaotic or unrepe-peatable organisational scheme will ultimately prevent access to the data or, at least, greatly reduce the efficiency in which the data can be used. To this end, when discussing file systems that have the potential to store large amounts of related data, it is pertinent to discuss database technology.

While a database is an organised body of related information, a relational database (RDB) provides organisation and access mechanisms based upon the relationships between data, generally using a table-based format. Large-scale information storage is often conducted using RDB systems because of the increased ease of operation, once established, and reduced search times. The relationships between elements in an RDB can be exten-sional, meaning that the relation is defined and stored within the database, or intensional, where the relation is constructed by rules [21]. An extensional relation is true for all ob-servers, as it is fixed, but an intensional relation may change depending upon the when, how or who of data access. The RDB community’s definition of views provide a way of looking at the different ways that a database can be presented to its users through the use of stored relationships and the conjunctive association of a number of stored relationships [21], the extensional and intensional relations.

This terminology is adapted here to include the information presented to a given user within a distributed system. For example, consider the widely distributed network-based information service that is the Domain Name System (DNS) [3, 4]. The DNS clients and servers exchange DNS message format messages to ask and answer questions regarding domains. The result of a query to the DNS, presented in DNS message format, would be seen as a user’s view of the DNS system as the underlying information is presented in the context of the query. The client does not have access to the internal format, despite the tight translational coupling between internal and wire formats. The term wrapped resource or wrapper, denoting a resource that has a translational wrapping provided through some external mechanism, and mediator, for an aggregator of resources (wrapped or otherwise) are also from the database community [21] and are used here without modification.

A distributed system conforming to a globally defined standard implicitly provides a global view shared by all users. Also, a user cannot change the global standard without potential impact upon all other users. So the global view is usually protected from users and cannot be altered on an ad-hoc basis. Since only an end-user knows their own requirements, this lack of flexibility can have a significant impact upon a user’s ability to use the system in a way that is sensible, efficient and effective from their point of view. Without a good, or legal, reason for restricting access to the form of the information offered in a distributed
system there is no good reason to restrict all users to a single view [22].

2.2.4 Data storage and retrieval across networks

An alternative mechanism for storing files in a distributed manner is the WWW. Files of many different types may be stored as marked-up documents or as they are, where access is provided via an HTTP server. This is typically a read-only file system from the network perspective and space is not given to anonymous or unauthorised users. Instead of having a resource discovery mechanism provided as part of the protocol family, search engines and hyperlinks provide references for documents and files.

This immediately leads into the problem of how to access data stored in the WWW. NFS uses the standard file system approach to storing data where files are organised in directories and have distinct names within those directories. A simple tree-structure search starting at ‘/’, the root node, will eventually traverse all of the space. Similarly, in a P2P system, if the topology converges then all resources within a given perimeter can be discovered eventually as there is a resource discovery and advertising mechanism [23, 24]. Again, there are no guarantees for P2P because of the constant possibility of topology changes in the system.

The WWW is vast and the names and locations of files do not follow a strict format. While an exhaustive search of all possible alphanumeric combinations within all recognised domain names would eventually finish it is unlikely that it would retain currency for any length of time.

Most web-based file systems require explicit knowledge of where the data is, as a URL, and do not abstract beyond the naming conventions of the WWW.

2.2.4.1 Network-based file systems

WebFS [25] provides a distributed file system through using WebFS daemons on cooperating nodes to establish traditional NFS file semantics. If a WebFS daemon is not present on a node, but an HTTP daemon is, then the file system will have read access to the files offered via the HTTP server. A virtual node system is used to establish whether pages should be cached locally and this mechanism is used to conceal network activity from the user. Locating a remote node still requires knowledge of the URL of the site.

Similarly, WebDAV [26] aims to provide collaborative editing and management facilities to remote web servers by using extensions to the HTTP protocol. WebDAV is focussed as a distributed authoring and version system similar in nature to CVS [27]. DAV already provides features for locking, namespace management and property inspection across
HTTP and planned extensions include versioning and configuration management and access control. WebDAV uses a strict, hierarchical namespace that can be manipulated but does not have automated name abstraction systems built in.

Another widely used network file system is the Concurrent Versions System (CVS) [27]. CVS was originally designed for version control but because of its network distribution capability, in a client/server mode, it can offer a primitive network file system. This file system is based on the notion of a repository from which users explicitly check data in and out [28]. This functions as a remote file system that strictly controls access to files to prevent the accidental removal or alteration of data by unauthorised persons but primarily to track changes between file versions. This prevents the use of ‘hidden’ file changes where a user thinks that they have used the most recent file when, in fact, they have used an obsolete copy. CVS does not store entire copies of the different versions but, instead, stores the differences between versions so that intermediate versions can be retrieved if required but the overall storage load is not significant. There is no additional search information stored with the repository although some metadata is kept that deals with who performed a given change and which users are to be informed of certain activities. Finding a file in a CVS repository is a matter of setting the CVS root correctly and then accessing the file in the correct part of the CVS tree. CVS can use a variety of protocols but is not restricted to a fixed component such as the HTTP aspect of WebDAV. It should be noted that CVS could be implemented on top of WebDAV since CVS can function over a number of different protocol layers.

### 2.2.4.2 Potential issues with network-based file systems

One of the reasons that NBIS were developed was to provide better access mechanisms to data and to provide better, and more consistent, standardised schemes for managing large, distributed information sources. In this section, we look at some of the problems with network-based file systems, in order to more clearly show why NBIS emerged and, also, to foreshadow potential improvements in our approach to implementing NBIS.

None of these systems store metadata that could be used for associated searching and, hence, name independent data location. While they could all be used to store metadata files that could be used in this way there is no ‘in-system’ understanding of the function of such metadata and an external agent would be required to analyse and act upon the metadata. It is not possible to place a semantically aligned translation layer upon these systems because there is a lack of any of the mechanisms required to associate semantic metadata with the data.

Another problem that arises in all of these systems is the lack of a facility for the caching
of searches or a mechanism for the semantic description of such caching. In this context, a search is the action taken to locate data or resources where either the exact name or the exact location of the desired entity are not known. This is a common action as many storage systems use a machine-friendly hierarchical system that is neither intuitive to humans nor a good match to the way that humans store information.

If a Unix [29, 30] file is misplaced then a ‘find’ [31] command will locate it if the filename is known, or some of the contents of the file are known but this is computationally intensive and has an impact on I/O performance. While I/O caching will occur in the short term, demand on the buffers and cache memory will eventually return the system to a state where a similar search will be just as demanding as the first. The system has not learned anything from performing the search. The FreeBSD-based OS X operating system implements a metadata-based system called Spotlight that actively searches the file system and creates a rich user-specifiable metadata repository to facilitate faster searching. However, the production of this search-enhancing store requires substantial processor time and is also dependent on the number of the changes in the underlying file system. With a technology such as Spotlight in a distributed environment, the appearance or disappearance of datasets could disable a system with metadata production requests or render an entire volume of work useless when the data it describes disappears. There is also the important question of where such a metadata record would be stored - client, server or third party? Each of these has implications on the ability of a client to make requests for the storage of additional items, and also to the number of resources that may have to be committed to the distributed system.

As an example of this, a P2P system may not be able to make assumptions regarding the stability of its entity nodes and hence might not cache search results at all, depending on relocating resources each time [23, 24]. Again, there is no facility for the long-term learning the locations of desired objects.

Locating a file in the WWW requires that appropriate search terms be entered that adequately match the textual data stored in the file and be associated in such a way as to reveal the desired file within some few hundred hits. Not only is this associative searching based on textual associations rather than semantic associations but the search engines in use changes the position of returned records depending on user feedback. Thus, the same data could show up as the first hit, 100th hit or 1000000th hit if enough time elapses between searches.

What the search engine learns from user response is shaped by all of the user responses - not just those of the user currently performing the search. This implies that per-user caching strategies stand the best chance of meeting user requirements but this is also the
most storage and computationally intensive strategy. However, such strategies are greatly enhanced by the existence of metadata to allow more precise statement of search terms.

2.2.5 Distributed Systems

The range of different systems found within parallel and distributed computing environments can be classified based upon whether they have single or multiple instruction streams and single or multiple data streams [32]. A traditional uniprocessor architecture is an example of a SISD machine, with no parallelism in instruction or data streams. A distributed system is generally denoted as a MIMD architecture, since there are multiple autonomous processors (providing parallelism in instructions) and these different instructions can be simultaneously executed across parallel data streams. A final, additional refinement, to the classification is SPMD, where a single program is executed across multiple processors on different data elements. SPMD is far more commonly seen in cluster computing for tasks such as parameter searching and parametric analysis of a solution space.

2.2.5.1 Definitions

A distributed system is a “collection of independent computers that appear to the users of the system as a single computer” [33] and also “a collection of individual computing devices that communicate with each other” [34]. There are many different possible models of distributed system, ranging from shared memory and message passing machines in one physical cabinet to a loosely coupled group of workstations spread across the Internet [33]. This overall model has emerged as an effective way of providing more computational power by combining the processing power of several machines spread over a physical area as well as placing a large number of processors into one physical unit with a common bus or data network. As will be discussed later, the physical and administrative separation of computational resources has a very large impact upon how resources can be combined and how flexible and adaptive such solutions are in the face of change. Obviously, the change management process for a group of three workstations in one room is far simpler than for a thousand servers spread from Antarctica to Greenland.

In distributed computing, the execution of the user processes will take place over one or more nodes that are not necessarily located on the host from which the job was initiated. While conceivable that a distributed system’s scheduling algorithm may result in a job being executed on the initiating host, it is the possibility that it may not be that labels the system as a distributed system.
2.2.5.2 Models for distributed systems

The basic models for distributed systems are [33]:

- Tightly coupled distributed system, all processors in one machine, using shared memory.

- Tightly coupled distributed system using message passing - Each node has its own memory.

- Loosely coupled distributed system - This uses a client/server model.

- ‘Pure’ peer to peer (P2P) networks - No hierarchy, nodes connected via a network, resource discovery and routing are critical problems to solve for efficient use. These are robust as the elimination of any one node does not remove a ‘master’ node - self-healing should always be a possibility except where massive infrastructure failure has occurred. Nodes are both client and server [23].

- ‘Hybrid’ P2P networks - some nodes are client and server, some are just client or server.

Within these models, additional complexity is introduced based on whether the system is composed of homogeneous or heterogeneous nodes, and also the type of interconnect network. Tightly coupled systems normally share some form of machine data bus or dedicated data network. Loosely coupled systems have more possibilities:

- Homogeneous LAN systems - All the same machine type in a local area network. May be built for specific purposes or taking advantage of bulk corporate purchasing policies.

- Heterogeneous LAN systems - This system may be used within a single business or research entity and thus has more policy control and fewer ownership problems. This can include far more candidates since there is no ‘one architecture’ for conformance.

- Homogeneous WAN systems - This is a highly unlikely configuration unless there is one organisational entity that spans the WAN and can enforce homogeneity in physically distant locations.

- Heterogeneous WAN systems - There are no requirements for member type, location or ownership. This may or may not be a P2P or a more hierarchical system. The
A wide range of possible network and execution environments make effective and efficient use of this system difficult and challenging. As the most challenging case, we will consider this case within this thesis as solutions for this case can also provide solutions for the other cases.

Whenever distributed computation is to be carried out the executable code and the data may reside on identical sets of nodes, executable code and data may share common nodes or there may be a complete division between nodes containing executable code and nodes containing data. The decision that must be made is: where should the code execute and where should the data be while the code is executing? Moving a small piece of code to a large piece of data makes sense in many circumstances, but only if the final result is not a data stream much, much larger that then has to be passed back across the network. Similarly, moving data only makes sense if the overheads of moving the data are smaller than the costs incurred by not moving it.

2.2.5.3 Developing Distributed Systems with proprietary formats

Evolution of distributed systems followed from the development of two distinct but significant components. The first was the development of the mini-computer, and the later development of the personal computer, and the second was the development of networking technology that could link together the increasing numbers of small computers. Before this time, computers tended to be monolithic entities that had a significant capital value and were generally purchased for business-specific purposes, rather than as a general processing workhorse. There was also not a basis for ad-hoc connections between administratively unrelated machines. While some forms of local area networking existed, these tended to be short-haul and point-to-point networks. Although, prior to this time, researchers had identified the requirement for a new mode of computation and the use of computational resources [35, 36], these requirements were not realisable until new hardware and software technologies had been developed to support these new computational modes. Although the interconnection of computers was initially driven by a strict requirement for research collaboration, and required very specific interconnection devices, work proceeded on providing a protocol-based approach to allow far more nodes to join a network providing they conformed to a given network protocol.

Despite successful projects linking together a site’s computers over a local-area network, these were still strictly bound within a single administrative domain and any standards-based problems could be resolved without resort to external arbitration as the new system was, in effect, a virtual overlay onto the existing resources rather than a new system com-
posed of external and internal elements.

2.2.5.4 Moving to open development frameworks

This movement away from proprietary, pre-specified, hardware to an approach based on a
generic hardware solution that agrees to support a given set of protocols has been the most
significant contribution to the development of widely distributed systems as it allows a far
greater range of potential participants. The development and wide-spread adoption of the
Transmission Control Protocol/Internet Protocol (TCP/IP)[37, 38] networking protocols
in the 1980s led to the replacement of a number of competing vendor-specific network
protocols, most directed at the local-area networking level rather than wide-area, with a
single replacement protocol suite that allowed any vendor’s equipment to participate. This
levelling of the playing field also caused a drop in the prices of networking equipment,
since no company could now claim a monopoly as the protocols had to be open to allow
a manufacturer to provide equipment that met the standard. Also, as part of the RFCs,
each implementation was expected to be robust and also capable of interoperation with a
standard-conforming protocol from any other vendor.

At the same time, and completely separate from the development of networking, the
size of the computer was shrinking with the introduction of mini, micro and super mini-
computers. Following on directly from the development of the central processing unit as a
single chip, rather than requiring a large number of boards, the form factor and electrical
consumption of computers was significantly reduced. Rather than assemble a larger unipro-
cessor or multiprocessor computer with a time-sharing operating system, some tasks could
now be performed by a single, smaller computer. Initially, such computers did not have
time-sharing operating systems nor multiprocessor features and were often tasked with a
single purpose, such as word processing, while more complex tasks were left on larger
computers, or mainframes. The financial overhead of maintaining a dedicated mainframe
for a company, along with the attendant operators, air-conditioning, power requirements
and constant expenditure on consumables and maintenance, had already moved a larger
number of smaller-scale users to using the services of an external third party with a group
of mainframes, paying for the third party to perform computation on their behalf and only
being charged for the computation time that was used, rather than paying for a machine in
the basement that only achieved 40% utilisation. The third party could also use economies
of scale to purchase consumable resources, such as paper and magnetic storage media, and
devolve the savings to their customers.

Prior to the development of the networking technologies, assembling large amounts of
computing power required building a faster computer or ensuring that, rather than a time-
sharing system where users had to take their turn, a user had sole access to a resource and thus guaranteed that their job would complete as quickly as possible. Advances in computational design, and the development of multi-processor machines, led to rapid increases in computing power but at a significant cost. Rather than focussing on the detailed allocation of resources to tasks, some abstractions can be used to make programming and using such systems easier.

One of these is the notion of a workflow-based system, which reduces the complexity of coordinating resources [39]. This is done by breaking the work to be performed into fundamental building blocks and defining operations in terms of relationships between these blocks [39] and provides an executable representation of a business or production process [40]. Other abstractions are discussed throughout this thesis but the workflow abstraction is introduced here as subsequent design decisions can be made with the assumption that at least some of the implementation complexity can be abstracted away from the users through well-established mechanisms and technologies.

2.2.5.5 Choosing the right design

Where a single data-network parallel machine, or local-area dedicated cluster with custom networking equipment, has an advantage over a distributed computational environment is in the assumptions that can be made about network performance, available bandwidth and data transfer optimisation. A set of machines within five metres of each other, connected via Myrinet, have a well-defined and high bandwidth network interconnect. A set of machines spanning three countries are sharing national interlinks, state and city connections and, finally, any choke effects caused when entering the site containing the resource in question. While statistical estimates can be made about performance over this type of network environment it is, obviously, inferior to a dedicated high-performance local-area interconnect. This will strongly define data flow through the network and the type of problems that are suitable to be solved in each environment with reasonable efficiency.

A parallel or distributed-computing environment is of little use unless the problem that is being solved is capable of being solved in this manner. Early work on parallel computation [41] and later analysis on systems running large-scale parallel tasks [42], both clearly demonstrate that, to get any benefit from using more than one computer, the problem must have aspects that can be solved in parallel with each other. For some problems the parallelisation is obvious where, for others, it is more subtle and requires detailed analysis that is usually only justified for problems with a high value or assessed worth. Whether using a strictly parallel, cluster or distributed computing solution, a problem with no parallelisable elements will be solved at the average speed of the processors that work upon it, with no
benefit for the number of processors involved. Ultimately, there must be a benefit in using more than one processing resource, and the partitioning of the problem into separate components must be done in such a way that it does not make the solution less efficient.

2.3 Information

Having discussed those aspects of computation that centre on the existence or presentation of values, we can now move on to a discussion of the requirement for formatted values - a strict requirement for structure. Information systems must explicitly specify their structure definitions as it is the expectation of all participants that they are sharing information of the same nature. The translation of information from system-format to user-format also requires a well-defined mapping to ensure that, for example, a system that stores dictionary data will present the appropriate entry when queried with a given key-word. Information systems are a sub-class of distributed systems because of this structural focus, as it is possible for a distributed system to function without explicit statements of value structures but this is not guaranteed to provide the correct results for information systems.

2.3.1 Network-Based Information Systems

A network-based information system (NBIS) provides a network-based information service (NBI service) and is characterised by a decentralised data store designed to serve that data to a number of different clients [43, 44, 45, 46, 47]. Such systems have been referred to in the past as distributed information systems and internet-based information systems. The term network-based information system is used to indicate two things: that the distribution of the system is beyond the local boundaries of a single node or special-purpose nodal cluster; and that such a system does not need to traverse the internet although it can. This includes systems that exist in private, or virtually private, network environments but excludes systems that function more as a cluster computing environment than a truly distributed system. The acronym NBIS is used throughout this thesis to refer to a network-based information system, where there is a potential for confusion the terms ‘NBIS’ and ‘NBI service’ will be used to indicate which is meant.

Early work revolved around the sharing of file systems between elements of a distributed system but, as technology evolved, extended to include the sharing of information, regardless of storage mechanism, between clients and servers in a system. From the definition in the first paragraph, a network-based information system could be regarded in much the same way as a distributed database [46] and, in early stages of development, this was
true. However, an NBIS performs, at least part of the time, transformations from the data store to make it available across the network in a standardised format. The NBI service may also provide a range of different ways of accessing the same data, rather than a fixed format mode for addressing, reading and changing the data. A distributed database generally requires the same application or application suite to access the database storage and to form a coherent image of the stored data. An NBIS can connect to any application or client that supports the correct transmission protocols - the underlying data storage is abstracted away from all users and they connect to it as a single logical entity from their connection point. The NBIS provides a distributed service that accesses a distributed resource, where the distributed database provides a resource.

The first major difference from other distributed systems is that an NBI service does not have a traditional start and end point for execution - it is designed to run continuously and provide the information service for the foreseeable future. Thus, there is no downtime to make changes to such a system without removing active nodes from service. Testing of new approaches must be carried out within non-production environments or in such a way that it does not disrupt production. Because of this, requirements for the system may change while the system is active and deployed due to its lifespan and duration of execution [47].

2.3.1.1 Characterising a network-based information system

A simple classification of distributed systems is shown in Table 2.1, with emphasis on those aspects that characterise a network-based information system. An alternative column is shown to show the possible configuration of a given aspect in a non-NBIS system.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>NBIS</th>
<th>Alternative choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>Long</td>
<td>Short</td>
</tr>
<tr>
<td>Longevity</td>
<td>Long</td>
<td>Short</td>
</tr>
<tr>
<td>Bandwidth Utilisation</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Model</td>
<td>Decentralised</td>
<td>Centralise</td>
</tr>
<tr>
<td>Resource Composition</td>
<td>Homogeneous/Heterogeneous</td>
<td>Homogeneous/Heterogeneous</td>
</tr>
<tr>
<td>Geographical Span</td>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td>Complexity</td>
<td>Multiple potential or actual purposes</td>
<td>Single, simple purpose</td>
</tr>
</tbody>
</table>

Table 2.1: A Classification of Distributed Systems

The *duration* of a system is, simply, the time over which it is expected to function and meet user requirements. For example, a simple workstation script to perform file cleaning may be used repeatedly but if it only takes 30 seconds to execute it has a short duration. A
piece of software that stays up for days, weeks or months - and is expected to - has a long
duration.

The longevity of a system is the time over which the system is required to provide its
services. Some NBIS have been in existence for almost twenty years although the software
and hardware components that provide the service have gone through many changes over
that time. This is, in effect, the “Ship of Theseus” from historical debate on identity. A
famous ship, preserved by the Athenians, was considered by most to be the same ship,
despite the fact that over time all of the original planks had been replaced by new planks.
If the entire ship had been burned and a new ship purchased, then this thread of continuity
would be lost. Similarly, where an old system is completely replaced by a new system, there
is no continuity and the longevity of the old system has finished. The DNS is an example
of a very long-lived system as key mappings, core to the DNS, are available today in the
same format as when the system was designed in 1987 [3]. Despite the many upgrades to
the underlying hardware and software, it is the essentially the same service and has a high
longevity. The major difference between the “Ship of Theseus” and an NBIS is that the
decay of an NBIS is caused by changing user requirements and is a perceived decline rather
than a strict, physical descent into decay. Very few current computational systems are in
existence for long enough that anything other than obsolescence threatens their existence.

The average bandwidth utilisation of a system is defined here as the difference between
nodes that are sending datasets between each other almost constantly, at near network satu-
ratation point or having to depend upon bandwidth guarantees and dedicated virtual circuits,
versus a system where, although it may occasionally reach network saturation, the ma-
ajority of messages are well below saturation size and the frequency is bursty rather than
continuous. A distributed SPMD application will most likely have much higher bandwidth
utilisation than a MIMD application that is specifically designed to meet an information
service requirement. The movement frequency and size of data are the most usual con-
tributors to bandwidth utilisation but, occasionally, code updates can be quite substantial
consumers.

The characteristic model of an NBIS has already been identified in Table 2.1. NBIS
tend to function as loosely-coupled decentralised systems with a logical two-tier hierarchy
based on an authoritative group of servers that provide data, and can possibly receive it,
from clients of the system. Within the server layer, there are examples of systems em-
ploying further layers of servers to distribute load and protect authoritative servers form
overloading. In the domain name system (DNS), there are servers that peer with each
other, above another group of servers that can provide the authoritative data directly, cache
it or redirect enquiries back to a higher-level server. These subordinate servers also have
a peer relationship with each other. Similarly, the network time protocol (NTP) provides a layered server structure with distance metrics to identify how far a given server is from time sources of defined precision. This allows the NTP and DNS systems to serve a much larger group of clients than would otherwise be possible although such service comes with introduced delays or minor loss of precision in information propagation. This is caused by the caching mechanisms in DNS and the potential drift in time signals from NTP servers at different levels.

The resource composition has a large impact on the nature of the distributed system as it affects the possible parallel models that can be employed and has administrative and performance impacts. The movement of code and data becomes most challenging when dealing with heterogeneous WAN systems. As mentioned in Table 2.1, NBIS can function over homogeneous or heterogeneous resource frameworks - there is no strict requirement that either model be selected. However, such systems are far more commonly heterogeneous and, as a result, solutions to implement NBIS assume a heterogeneous resource framework.

The geographical span is the extent to which the system is geographically distributed. A large span is considered to span multiple countries, with deployment at multiple sites through each country. This leads to potential issues with international, national and local network bottlenecks, as well as requiring a multi-national effort to establish or change such a system with all of the ensuing complexity.

Finally, the complexity is an approximate measure of how easily the system can be used for different purposes. Looking again at the DNS, a brief glance may cause it to be classed as a simple system as all it does is provide mappings. However, a closer inspection reveals that it provides many different kinds of mapping in several different forms. It also provides reliability, caching and some simple extension and implicit overloading mechanisms. It is, compared to NTP, a relatively complex system and one that can be put to several different purposes. Where a service has only one primary use and little chance of extension, it is simple.

2.3.2 The Boundary of the Operating System

An important question for any distributed system is to define where the operating system of the underlying machine becomes intrinsically linked, if it does, to the virtual operating system of the distributed system. The difficulties of non-ownership of distributed resources have already been raised but there are situations where a greater commitment to a distributed system could allow extended access through mechanisms similar to those found in an operating system.
The boundary of the operating system can be defined in two separate, but not competing, ways. To discuss this it is necessary to first discuss the different ways that an operating system can be designed. Operating system structures can be classified into five broad groups [11]. These are:

- **Monolithic Systems** - the operating system has one layer, a set of procedures that can call each other. There is some distinction between operations carried out in the kernel and elsewhere but only in terms of how these are executed, rather than in terms of information hiding or access.

- **Layered Systems** - the operating system has several layers, building from the low level processor allocation and scheduling up to a user-layer where a user can refer to the layer below them to access their programs.

- **Virtual Machines** - using a virtual machine monitor the operating system provides a virtual representation of the underlying hardware. User system calls trap to the virtual machine and, from there, trap to the underlying hardware. Each user sees an identical copy of the underlying hardware.

- **Exokernels** - this is similar to a virtual machine but users only receive a subset of the available resources.

- **Client-Server Model** - the kernel is reduced in complexity, tending towards a micro-kernel and, while essential tasks are left at the kernel level, most of the operating system is now defined in user space. A user process sends requests to the kernel to allocate resources or manage the system with its unfettered access. An obvious advantage of a client-server model is that, if the kernel is expecting messages in order to perform actions, such messages can come from inside or outside the system. Hence, a client-server model could be well suited to a distributed operating system application.

2.3.2.1 Specifying the boundary

One approach [48] for developing a client-server style operating system is to provide the required, but minimum, kernel mechanisms but allow them to be controlled by specifying policy at the user level. The kernel merely performs actions - it is the policies in user space that determine if those actions should be taken and, crucially, by whom they should be taken. This defines the first boundary, the boundary between the kernel and the user. For
most modern operating systems, the hardware belongs to the kernel and the user makes
requests to the kernel to perform operations according to a set policy on the user’s behalf.

The second type of boundary is less well-defined and is the somewhat nebulous boundary
between the computational resource as a single entity and other resources. For the
purposes of further discussion, a computational resource is assumed to have an implicit
boundary around it within which the only means that a user can physically enter data is
via a directly attached non-networked input device. Any other form of attempted machine
control, which would be limited to network-based communication, is an attempt to control
the machine from beyond the second boundary. To remove any confusion regarding the use
of these terms, the first boundary will be referred to as the kernel boundary and the second
will be referred to as the network boundary.

Despite the existence, and widespread deployment, of microkernel operating systems,
there is still considerable overhead in establishing a distributed operating system over a
WAN due to the difficult requirements of integrating heterogeneous systems. While these
may be policy based at the operating system level, the policies on one system are not guar-
anteed to match the policies on another and constructing a meta-policy that can usefully
and safely drive policies on two machines of different composition is a very difficult pro-
cedure. While there are obvious benefits for allowing a deeper access into the machines
participating in a distributed system, there are significant challenges as well as potential
pitfalls with security and resource utilisation. The non-ownership of resources is again a
problem if remote storage is being employed as a distributed system policy of ‘save all files
for two weeks’ may be in direct conflict with user requirements at a given machine.

When the system is composed of cooperating entities, rather than directly-controlled
subordinate entities, policy specifications are only valid while the participants wish to co-
operate. Giving participants the ability to control the effect that their participation has upon
their local resources, giving them some control over the standardised system-wide policy,
is more likely to lead to longer-term cooperation. This has been shown in both distributed
systems and code sandboxes for languages such as Java, where the damage that can be done
to a cooperating machine is strictly limited in its range and nature by the use of sensible
system design.

2.3.2.2 The impact of policy-based control on system operation

Regardless of how they are controlled, all distributed systems have at least an implicit
operational policy that defines how they operate. This may be echoed in the operating
systems of the machines that comprise the system or it may merely give an appearance
of this by combining resources or performing transformations on input and output. The
standards that define a system produce the protocols and policies that govern the behaviour and future operation of the system. Heterogeneity generally prevents a distributed system from working down at the kernel boundary. Once it has been been allowed access through the network boundary, it usually works with user-level software to carry out its tasks.

Network-based information systems work at this level and, as will be discussed in more detail, the standards that define policies define certain behaviour and proscribe others. However, since the major operations of the local component on an NBIS take place at user level, policy specification does not have to descend to the kernel boundary so the set of proscriptions that prevent unauthorised hardware access are not included in the policy set. This allows more leeway in policy modification at higher levels, providing the policy changes do not invalidate the standards by failing to provide a service guaranteed in the standard or through the provision of a facility that is explicitly proscribed by the standard.

We now build a basis for the discussion of syntax and semantics in complex systems in order to explain concepts described in later chapters. We start from a discussion of syntax and semantics in programming languages to provide a firm foundation for later extrapolation.

### 2.3.3 The Syntax and Semantics of Programming Languages

The syntax of a programming language refers to those reserved words, symbols and valid sequences of tokens that make up valid statements and, hence, valid program [49, 50]. Thus, in Python [51], this is a valid sentence in the programming language:

```plaintext
x = 3
```

where:

```plaintext
x 3 =
```

is not. The syntax of programming languages are frequently expressed in Extended Backus-Naur Form (EBNF) [49], which allows the statement of structures while taking into account potential repetition and recursion within these structures. The EBNF for a Python assignment statement is:

```plaintext
assignment_stmt ::= {target_list "="}+ expression_list
target_list ::= target ("," target)* 
                | attributeref
                | subscription
                | slicing
target ::= identifier
          | "(" target_list ")"
          | "[" target_list "]"
```

Thus, in Python [51], this is a valid sentence in the programming language:
The valid sentence can be clearly recognised as an instance of \texttt{target_list = expression_list} where the \texttt{target_list} is \texttt{x} and the \texttt{expression_list} is \texttt{3}. As the sentence \texttt{x 3 =} has no valid representation in the set of possible sentences that comprise the language, it is invalid.

Syntax checking, during parsing, allows the detection of errors on the part of the programmer including, but not limited to, the use of incorrect or non-existent keywords, the use of undefined variables (in some languages) and incorrect block structures [49]. If a table is kept of the type of each variable encountered, a symbol table, then type-checking can be carried out to determine if the expressions being assigned to variables are of the same type as the variable itself. Note that some languages have very loose type systems, such as Python, and some have far stricter type systems, such as Ada and Java, so the degree to which type checking is carried out at compile-time is strictly language dependent.

A type is a set of values, which have a low-level representation and a set of operations that can be applied to these values. Frequently occurring simple types include those providing integer variables, floating point variables, characters, strings and boolean variables. More complex types can be constructed through the combination of simple, and other complex, types. In object-oriented terminology, these are referred to as classes and the variable itself is an instance of the class. In general, programming languages allow the assignment of values of a given type to variables of the same type. Unless there is a well-defined translation mechanism, implicitly or explicitly defined, a user may not assign a value of one type to a variable of a significantly different type. If this checking is conducted during compilation it is referred to as compile-time type checking. Some languages define the operations of a variable based upon what is currently within it. Thus, even if any value can be assigned to a variable, regardless of type, the operations that are valid for that variable then become limited to the type of the value. Python, for the most part, behaves in this way and this technique is referred to as run-time typing.

The analysis of type equivalence can be relatively straight-forward, based on name equivalence, more complex, using structural equivalence, or very complex, using type hierarchies that support polymorphism [49, 52]. These complex cases support interfaces as well as structural comparison. In most programming languages, there exist syntax and type checking mechanisms that, at various stages, can analyse the program to determine if the stated instructions are clear, unambiguous and are correct within the defined type system. This analysis provides feedback to the programmer when errors occur so that these errors may be corrected. This is, however, not sufficient to establish that the program is performing correctly.

The semantics of a program constitute the final meaning of the program where this
meaning can be interpreted as the effect of all programmatic operations on the passed inputs and how they lead to a given output [49, 52, 53, 50]. This meaning must match the programmer’s intent or the program will not produce the correct result. Having passed the syntactic checking, the first layer of meaning involves the checking of types to ensure that a statement that is syntactically sound is performing operations on items that have the same context and can both take part in the same operation. A program, once compiled, has a grounding in the local, or virtual, machine code to which it is compiled. Every statement can be shown to have a certain action within a local machine. Looking at the transformations of data, with no other change in representation, presents the operational semantics of the program, whereas a translation of the program into a set of logical relationships would provide the axiomatic semantics. The denotational semantics [50] of the program map a program directly to its meaning, or denotation, this provides a more abstract and high-level definition of the semantics than the operational or axiomatic semantics. The denotation can be a numerical value or a function, for example, and requires a valuation function, rather than an interpreter or compiler, to map the program to its corresponding meaning.

The denotational semantics provide the executable ‘meaning’ of the program where the operational semantics show the effect of the transformations implicit in the program. However, the axiomatic semantics restrict the meaning of a given sentence to only that which can be proven about it axiomatically, with reference to the logical formulas that describe it.

None of these approaches necessarily capture the context of each aspect of the program, as perceived by a non-computational resource: the human perspective. In order to be useful to a user, the program must accept input and provide output in a way that meets the user’s needs. That is, the operational semantics of the program must transform input to output in a predictable and repeatable manner. The level of complexity of the context of the data tends to increase with the number of orthogonal elements taken as input and returned as data.

Consider the assignment of two differing semantics to a variable, those that are based on the assumptions of the programmer and those that are based on the assumptions of a user. Where these semantic assignments are in agreement, we have, as mentioned earlier, semantic alignment. The meanings are the same. In strict terms, with particular reference to computer systems, it is more precise to say that their computational contexts are the same. Thus if a user enters data with a purpose in mind and the input is semantically aligned with the output, then the context of the output will be the same as the input and the user will be able to assign the same meaning, due to context, to the output.

Hence, data is defined in terms of its structure using both the syntax defined for its assembly and any valid type structures. It is then defined, at the least, in terms of denotational
semantics (which may be implicit in the compiler) as well as the operational semantics. Finally, the data must be semantically aligned with the program’s external context, in order to be useful to users.

Distributed systems must pass information, as data, between their nodes in order to distribute computational information. These data exchanges must be well-defined as the data appearing on another machine will do so as a byte stream, within a IP packet, as this is the most common form of exchange between co-operating nodes. Each node must agree upon the transmission format so that structural analysis of the packet will provide the correct data, in the correct structures and types. A node that transmits a 32-bit integer as a single datum to a node expecting two 16-bit short integers will not necessarily get the results that were expected. However, even if the syntax of communication is intact, there is no guarantee that the semantics of both nodes are in agreement. If Node A sends a 32-bit integer that is the temperature in Fahrenheit and Node B receives a 32-bit integer as the temperature in Celsius - a logic error is a likely outcome.

One of the major deficiencies in network-based information systems is that while the structure and type of data can be well-defined in standards, most systems do not have a mechanism for semantic alignment or, if they do, carry out this alignment using structural and contextual analysis that is not guaranteed to meet the user’s requirements.

2.3.4 Information Storage and the use of Associated Metadata

The storage resources in a local computer are volatile (RAM) and non-volatile (disk, NVRAM, any attached Flash-type storage devices, tape systems and all other forms of non-volatile recording technology). A disk-based file system is a non-volatile storage mechanism that writes information to and from the physical disk. These local resources may provide equivalent services to the distributed system environment and it is useful to understand the underlying disk mechanism as it can be used to provide analogues for storage paradigms at higher levels. A local file system stores information on the physical disk by storing information about the data as well as the data it is storing. Data may be stored in many non-contiguous blocks so reassembly information must be kept that allows the kernel to provide the data as if it were being stored in one piece. This metadata contains information on the name of the file and inode locations but is not user-configurable and is stored only to allow the operating system kernel to perform the reassembly.
2.3.4.1 The Semantic File System

The Semantic File System [54] provides a storage system that provides flexible associative access to data using the automatic extraction of attributes. These attributes are then selected through the use of user programmable transducers that interpret a request for a certain attribute and provide an in-system representation that allows this view of the system to be presented as a virtual directory. Although this is referred to as a semantic file system, the term semantic is used in this context because the user programmable transducers use information about the semantics of file system objects. The user programmable transducers perform the associated access based on textual analysis of the data, rather than by building on existing metadata or using ontologies to perform the analysis. This implies that a change to the data structure will require the transducer to be recoded by a programmer. Semantic file steps can provide assistance in the storage and context-specific retrieval of information but, as will be discussed, the semantic file system focuses on what can be extracted from existing data in the file rather than through the annotation of metadata.

If the WWW is used as a source of information, for example, then a search for information that has previously been used may result in millions of ‘hits’, where none of them are appropriate, even if the same search keys are used. The text-based associative search techniques used by search engines in the world wide web do provide some form of name independent searching but the results are not always what was desired because of massive overloading of textual terms. For example, a search for oil could return pages of petroleum drilling, petroleum jelly, cooking oils, massage oils and the Ontology Inference Layer.

2.3.5 XML, RDF and Metadata

Early web pages provided a mechanism for formatting the text on the page using a well-defined markup language. This has obvious shortcomings as there is no way to separate data from text, especially with regard to the typing and structure of the data. The eXtensible Markup Language (XML) [55] is designed to provide a means of describing data with user-defined markup tags. Initially this was used with HTML to provide a separation between describing how the data should look (HTML) and what the data was (XML). XML-Schema is a language for restricting the structure of XML documents and also extends XML with datatypes [56]. However, the combination of HTML, XML and XML-S does not allow for the description of interoperation between pages, or for the storage of the data contained in the page as an exploitable resource.

Metadata is important in this context because it describes the data stored in the storage facility, whether that is a traditional file system or a simple web page. For a traditional
Unix file system, the data are the bytes contained in the i-nodes, where the metadata are the file name, owner, creator, creation date and i-node tracking information. For a web page, this can be information external to the web page (such as creator) or it can be metadata that describes elements of the web page itself. The storage of metadata can greatly assist with data discovery, cataloguing, association and facilitating agent-based automation. In other words, the addition of metadata can provide sufficient information for a non-human mechanism to determine important information about the data.

The interpretation of metadata is a difficult problem and has led to the development of taxonomies, where information can be classified into a hierarchy, and inference systems, where a fact can be used to infer other information. The classification of the information within a domain, where that classification includes taxonomic and inferential logical statements, is referred to as an ontology. Ontologies have recently become important in metadata systems as it further removes the need for human intervention and allows rapid classification and interpretation of previously machine-impervious data sources.

The Resource Description Framework (RDF) [57] builds upon XML and provides a basis for metadata interoperability. RDF forms object-attribute-value triples where an object in one triple may be a value in another, allowing chaining, and an RDF statement can also be an object or value, which allows nested graphs. Types may also be associated with an object using type edges that refer back to a definition in a schema [58]. Any object with a Uniform Resource Indicator (URI) is a web resource and any web resource can be described in RDF and made available as a machine interpretable form. An RDF description is composed of properties, where each property has a property type and an associated value. Thus the RDF triple is composed of (property, property type, value) where the value may also be an object in another triple. RDF extends XML and uses the XML namespace facility. We will discuss the use of RDF further in Chapter 6, as part of the conceptual model that defines our approach.

RDFS (RDF Schema) provides a mechanism for defining the meaning, characteristics, and relationships of a set of properties. It defines a particular vocabulary for RDF data. So RDF and RDFS together allow for the definition of properties and the definition of their relationships. RDFS expressions are also valid RDF expressions but in RDFS agreement can be reached on the semantics of certain terms and hence on how to interpret certain statements [58]. Figures 2.2 and 2.3 show the different ways of storing data and metadata for publication information.

One of the reasons for storing metadata is that the provenance of the data can be stored in the metadata and be used for either validating the reporting of production steps or, potentially, repeating what was performed in the production of the data [59]. The two key
terms are:

- **Provenance** - This is technically a history of ownership, where the ownership (location) and the process of generation the data can be tracked at each point since its creation. In this context, whenever data changes a new file has been created with contains the modified data. The old version can be regarded as a separate file although the division between the two files may be temporal rather than logical or physical.

- **Repeatability** - The results can be reproduced by exactly replication the original execution environment, inputs and transformations. Borrowing from the concept of workflows, the workflow can be re-executed and will produce the same result. This is barring any deliberately non-deterministic elements that cannot necessarily be reproduced, such as random number generators or system noise capture.

Both of these aspects allow the precise determination of context and, with the same context, the same external meaning can be applied to the computational steps and, thus, the final result. For a network-based file system, context storage can lead to context sharing. With context sharing, it is possible to lay down a solid groundwork for semantic sharing and hence, semantically rich interoperation of network-based resources. This use of rich context is explored further in later chapters of the thesis.

The sheer volume of information contained in the WWW strongly motivates a requirement for better resource discovery and cataloguing mechanisms, which can be offered by adding semantic information to the WWW model. This has led to the development of the Semantic Web [60], which extends the WWW by associating meaning with data and improving the sharing and reuse of the data it contains. The semantic web is based on RDF and is a World Wide Web Consortium (W3C) project. Information stored on web pages may include elements from the Dublin Core Metadata Initiative (DCMI)[61] or provide information of use to distributed applications. The WWW has also been extended with the notion of Web services that provide a well described service based interface to resources. The notion of a large distributed net of services has had several names and incarnations, but the current focus of research in this field resides in the Grid.

2.3.5.1 **Metadata and Provenance**

When a scientist performs an experiment *in silico* it is essential to capture the metadata describing how the experiment was conducted, what was produced, what was used in that production and where all of the components came from. Only then can other scientists
repeat the experiment because the provenance of all aspects of the workflow can be determined. This issue is confounded by the sheer volume of data that can be produced experimentally or the number of disparate inputs that can be consulted to produce a final output. The volume of data leads to storage problems, annotation burden (if annotation is being practised) and searching problems, since any searching algorithms search time is dependent upon the number of elements to be searched.

Data volume and the amount of time required to search for information within a large corpus can cause problems for scientists and other users alike since computing resources are finite and often heavily tasked. Where large datasets are being produced it is logical that all information pertaining to the production is captured as soon as possible so that the data is rendered independently meaningful before the ‘additional’ information is lost.

However, locally produced data is not the only source of information available so the strict and efficient annotation of local datasets is not enough to ensure that all provenance data is kept. Provenance data must be kept on all sources of information that implies that not only could the source be found once but it can be found again and in a relatively efficient, or at least bounded, search. This becomes far more complex in widely-distributed systems as every component of the system must be recorded if it is in use and also provide information as to its sources of information. When a distributed environment is providing an ongoing role as an information system, this information becomes critical.

2.3.5.2 Existing work on provenance

Existing work on provenance includes initiatives such as the Dublin Core Metadata Initiative (DCMI)[61] which stores metadata regarding authorship and publication information in a well-defined form. The DCMI is dedicated to developing metadata standards for inter-domain discovery, defining frameworks for the interoperability of metadata and assisting in developing community based metadata sets. Dublin Core (DC) is widely used for managing provenance data in organisations ranging from libraries to the San Diego Supercomputer Center’s Storage Resource Broker (SRB) support technology for data grids [62]. There are 15 elements of the Dublin Core that can be used to describe publications of many forms, as seen in the following examples.

    Dublin core metadata can also be encoded with HTML, XML and RDF

We will now discuss the key issues involved in assembling computational resources and some of the approaches that have been taken to assemble these resources. This will also encompass a discussion of the Grid computing model.
2.3.6 Assembling Computational Resources

Within the virtual computational environment of a distributed system, the physical resources must exist somewhere in the system to provide the processing units, storage resources and I/O facilities that are used in the distributed system. Combining these resources can take several different forms, as outlined above, depending on the model of system and computation that is being employed. For ease of use, it is advantageous to the user of a system if they need not concern themselves with the underlying details of the system but can define their requirements on top of an abstract representation of the system. Modern operating systems already do this at the individual machine level with the use of command-line input (CLI) environments and graphical user interfaces (GUIs) which do not require a user to address the hardware devices directly. This is also in keeping with the removal of the user from direct access to the underlying hardware and the role of the kernel boundary.

Traditionally, most users employing a distributed computation environment have a distinct problem to solve rather than just using the distributed system for whatever problems crop up on a day-to-day basis. In other words, they have a specific purpose for reasons of research or business to use the resources to solve a defined problem. This problem may be an ongoing problem with different parameters, such as protein sequence identification in biochemistry, or it may be a long-lived problem with a single outcome, such as mathematical verification of a proof using computational techniques. Either way, the predominant use of such an environment is task-focused and any abstraction mechanism should be an efficient method for representing the task in a way that best suits the user’s need. We will now briefly discuss some current possibilities for assembling processing elements to meet user requirements: workflow systems and the Grid computing model.
2.3.6.1 Workflow systems

The majority of workflow systems represent the flow of data through various processing and storage entities and use these flows to arrive at a final output of some sort. Workflow systems have been used in the corporate sector for some time and are also being used in bioinformatics and systems such as the Business Process Execution Language for Web-Services (BPEL4WS) [63], myGrid [64] the ASCI computational grid [65] and PAGIS [66, 67]. Discussions at the Global Grid Forum (GGF 11) [68] have also illustrated that a grid service based workflow model is seen as being an area where semantic grid can assist the Open Grid Services Architecture model.

2.3.6.2 The Grid

The requirement to assemble a group of potentially heterogeneous and geographically diverse distributed resources has also led to the development of the Grid computing model [69, 70]. The Grid model recognises that while computing resources should, in an ideal world, be available in the same way that the power grid is, on demand and ubiquitously,
there are limitations based on the desire to extend virtual organisations across organisational boundaries in a secure manner. The Grid is now designated as a processing environment that enables the integrated, collaborative use of computers, networks, storage systems, and scientific instruments potentially owned and managed by multiple organizations [69, 70].

The term ‘Grid’ was first used in the mid-1990s for a proposed distributed computing infrastructure to be used for advanced science and engineering [69, 71]. Since the organisations requesting the use of such infrastructure may not overlap with the physical organisation of the infrastructure the consideration of virtual organisations became important. The key concepts behind the Grid are the “coordinated resource sharing and problem solving in dynamic multi-institutional virtual organisations” [69].

The Grid builds on distributed systems technology but includes a notion of federation and unification of services, authentication and programming models that was not previously used. Grid technology is not an instance of an NBIS but, instead, is designed to help the development of distributed applications, which include NBIS. A particular Grid service may be an NBIS and, hence, that service can communicate with another NBIS providing the interchange format is use correctly. The integration of Grid-like services with the implemented NBIS will be demonstrated in the Results chapter of this thesis.

Initially services could be provided and implemented as persistent services with usage protocols or as software development kits with APIs. This is extended under the Open Grid Services Architecture (OGSA) where services are made available as Grid services. Each Grid service is also a web service and is described using the Web Services Definition Language (WSDL) [72, 73]. To provide stateful resources, services can use the Web Services Resource Framework [74] to provide resource services that allow data to be stored and retrieved. This alignment of Grid technologies with Web services allows the redefinition of the Grid as a set of Grid services, which in turn provide services to Virtual Organisations (VOs) [70].

The term architecture denotes that there are a well-defined set of basic interfaces that can be used to construct systems, using WSDL with extensions to support the distributed and collaborative nature of the Grid. Grid services, contrasted with web services, are stateful service instances although they are potentially transient.

A widely distributed network-based information system has great need for reliable, distributed storage resources but the architecture of such a system is not traditionally captured within a workflow system or within the model outlined for the Grid. Few legacy, persistent information services require the registration of their service with a central registry. This is due to the incorporation of many key NBIS into the underlying fabric of the Inter-
net and their availability to all connected users. The disadvantage is that, to a point, users need to know the name of the system to which they wish to connect and the nodename or IP address of the appropriate server. As will be discussed later, the introduction of SRV records [7] has provided a central point of consultation for such services but the underlying bootstrapping problem remains: to use a legacy NBIS, you are not generally able to use a match-making or registry-based service, you must know which server is providing it and connect to it directly. The Grid has also been extended with the development of the Open Grid Services Architecture that seeks to define the grid as a service-oriented architecture where a Grid is seen as a composed set of Grid services.

2.3.6.3 Organisational Structures

The drive to assemble resources can come from many quarters. A single organisation may require a large number of distributed resources for a specific problem, multiple organisations may need to collaborate, individuals may need to pool their resources or organisations and individuals may need to combine their resources.

Traditional organisations have a semi-static structure with dedicated administration resources and an associated core of computational resources. Such organisations include business and universities. Such organisations may span states, countries or the world but they are a “bricks-and-mortar” entity in that they have a set of physical resources that can be used to direct and support their activities. Once they have assembled additional resources to solve a problem, they continue to exist. This provides financial and administrative support for their ongoing projects and, as will be discussed later, gives them advantages in negotiating changes to standards documents.

2.3.6.4 Virtual organisations

The increasingly widespread use of dynamic, collaborative computational systems, such as those described in the Grid literature [69, 70], has provided an organisational model based on the desire to solve a problem and to attract interested parties to contribute computational and programming resources to the project. This model of an organisation does not require a common building, administrative structure or funding body to define it. It is a virtual organisation (VO) and exists because a group that spans different physical organisations consider that the resource contribution to the VO is of sufficient benefit to balance the penalty incurred by resource donation.

The virtual organisation, being problem-focussed, requires a system that solves the target problem(s). When a widely-distributed system is used to as part of the solutions, any
limitations in the way that it represents the information or the interfaces that it provides will impair the efficiency of the problem-solving mechanism. Ultimately, the ability of the virtual organisation to manage and collect its knowledge is impaired when it does not have customisation of its resources - and this extends directly into the ability to control the operational semantics of the system.

Since a virtual organisation does not necessarily exist beyond the lifespan of the problem, the assembled computation resources will immediately be returned to their owners. There is no formal ongoing relationship with the previous members and the possible reformation of the assembly in the future be undertaken as if the previous organisation had not existed, beyond any feelings of “good will” on the part of the previous members. Obviously, it is very difficult to apply an economic model to such an organisation as there is no formal abstracted point of organisational responsibility.

Having discussed the organisation of traditional processing resources, we now return to an examination of the information that is stored in or shared by these resources. Where an NBIS is the goal, information storage and retrieval must be both fast and efficient. As we will discuss in the next section, the correct choice of metadata is crucial to assisting both human and machine in making good use of data. As an example, how useful would the Unix Operating System’s file system be if there were no name- or permission-based metadata on top of the i-node based file system? All file access would have to be carried out by memorising the number of the i-node and there would be no access control for the files.

2.4 Knowledge

Having now demonstrated the impact of providing structural information alongside value information, and thus providing information according to our terminology, we can examine possible uses of this metadata. In particular, we can move beyond the accidental or implicit semantic alignment that can be assumed from syntactic, or structural, alignment to an explicitly stated semantic relationship between values. Once values have both structural and semantic information associated with them, we can then associate operations or transformations with them based on their defined semantics. We discuss mechanisms to provide semantic specifications for the values in our system and lead towards a description of systems that employ explicit descriptions of semantics. However, as we will see, there is a significant shortfall between the stated semantics mechanisms of the systems described and our goal to use semantic alignment to provide strong relationships between well-described data and the associated system functions. We begin this section with more
discussion of metadata and its impact on meaning.

2.4.1 Impact of metadata on information storage

Studies of metadata [54, 75, 57, 76, 55] and its impact on improving information retrieval strongly indicate that the annotation of meaning, where such meaning can be interpreted via an ontology for that domain, greatly improves the speed of retrieval.

To demonstrate the impact of metadata on information storage, we start by looking at the local machine environment and revisiting the semantic file system described in Section 2.3.4. Recall that the semantic file system is a system where file specific metadata is automatically generated, although only on a textual level, and user defined transducers can be used to search through this metadata. This allows the formation of virtual directories on demand that only contain the files of interest. An advantage of this approach is that the search has already effectively been performed. A request to group the files by certain metadata changes the view of the file system but not the metadata or data. Thus, in the above lost Unix file example, a find would not be necessary as a virtual directory containing all files that contain the word of interest would be generated as part of the automated metadata annotation as an ongoing background process. The search is automatically cached since the metadata is generated in advance and stored. The efficient indexing of metadata would then lead to a fixed-time operation to present a given view of the system, rather than the variable time under a system that had to perform exhaustive searches and then cache those results.

The capturing of metadata requires a language model that is sufficiently powerful to capture all of the features and suitably extensible to allow ease of expansion in the future.

With XML, RDF and RDFS, data may describe itself within its metadata. However, it does not capture the domain of those items described within the data and it does not support the embedding of concepts that can lead to further inference on elements that have already been defined. A key concept, which is essential to the classification of knowledge, is that of the subclass, whether strict or not. A subclass of a set of objects defined as a class must be at least or more restrictive in its member admission criteria than the original class. If another set of objects is always more restrictive and has fewer members than the original set, this second set is a proper subclass, otherwise it is a subclass.

In set theory, any subset $B$ of $A$ may be defined as $B \subseteq A$. Any element in $B$ is also in $A$ but $A$ and $B$ may have the same set of elements. If $B$ is a proper subset of $A$ then $B \subset A$ and, while every element in $B$ must be in $A$, not every element in $A$ can be in $B$.

When information is classified, it is placed into a grouping that corresponds to the
concept of a set. In RDFS such a grouping is referred to as a class. It is, strictly speaking, not a set as it can contain duplicate items. Conceptual hierarchies are complex and, while the ability to classify information is useful, it is far more useful to be able to verify class membership and infer membership in parent classes.

RDFS does not provide the subclass relationship in a manner that allows it to be used to further classify information and conduct inference upon it. This is, however, a very important requirement for the classification of knowledge. This requirement led to the development of ontologies [77, 78, 79, 75] which are extensions of RDFS, much as RDFS extended RDF and RDF, in turn, extended XML.

In order to illustrate some of the relationships between data and the advantages that can be gained from employing an ontological approach, it is necessary to discuss some basic logic.

### 2.4.2 Required Logic

In order to be a logic, any representation of knowledge must have four essential features [80]:

- It must have a **vocabulary**: a collection of symbols containing logical symbols, constants, variables and punctuation.
- It must have a well-defined **syntax** which provide formation rules for making well-formed sentences from the vocabulary.
- If a logic is going to make meaningful statements, then it must have a mechanism for associating the constants and variables with elements in the universe of discourse. These **semantics** ground the logic within the universe of discourse. Without this, it is merely a collection of well-formed symbols. It should also have a mechanism for distinguishing true statements from false statements.
- It must have **rules of inference** to show how one logical pattern may be inferred from another pattern. To be **sound**, then any rules of inference must preserve the truth that is determined by the semantics.

Propositional logic allows the representation of statements as propositions, which can then be composed using the logical operators **and**, **or** and **not** [81]. A complex statement such as “The sun is shining” could be represented by the letter $p$. In the notation of Boolean Algebra, the statement:

$$
\neg p
$$
represents the statement \( \neg(\text{The sun is shining}) \). However, we cannot change the meaning of \( p \) to represent some other shining object. For example, the sentence “the light is shining” would require a completely separate representation.

A first-order logic (FOL) uses existential and universal quantifiers to extend propositional logic. It is also referred to as first-order predicate calculus as the atomic statements of FOL are of the form \( P(t_1, \ldots, t_n) \), denoting a predicate with one or more subjects. The use of quantification allows statements such as:

\[ \forall x : P(x) \]

which states that the predicate \( P \) is true for all values of \( x \) [82]. Similarly, a statement such as:

\[ \exists x : P(x) \]

states that there exists an \( x \) such that \( P(x) \) is true. Referring to the previous example, if \( \text{shining}(x) \) is defined as “is shining”, \( L(x) \) is defined as “is a light” with \( S(x) \) defined as “is a Sun”, then we can define both propositions in terms of predicates:

\[ (\forall x)(\text{shining}(x) \& L(x)) \]
\[ (\forall x)(\text{shining}(x) \& S(x)) \]

which mean, respectively, all lights shine and all suns shine. We can also relate predicates in terms of inference.

\[ (\forall x)S(x) \Rightarrow \text{shining}(x) \]

This statement means that, for all possible values of \( x \), if \( x \) is a Sun then it is shining.

The key difference between first- and second-order logic is that in FOL the quantifiers can only be applied to variables, not predicates. For example:

\[ \forall A : A(x) \]

is not a valid statement in FOL but is a valid statement in second-order logic.

The existential quantifier states that something exists. This is equivalent to saying that it is not true that all are non-existent. The universal quantifier states that something is true for all occurrences. This equivalent to saying that it is not true that there exists an untrue occurrence. The relationships between quantifiers are usually expressed in logical representation thus:

\[ (\exists x)A \text{ is equivalent to } \neg(\forall x)\neg A \]
\[ (\forall x)A \text{ is equivalent to } \neg(\exists x)\neg A \]

Finally, modal logic is also important as it provides concepts of necessity and possibility. Where a predicate calculus can store information in terms of quantifications and relationships, it cannot make statements such as “There must be a sun” or “It is possible that there is a sun”, it is limited to statements such as “There is a sun”, “There is NOT a sun” and “If there is a sun, then”. Modal logics are of interest in several different fields,
including database theory, as an entity does not have to be stated as existing or not existing before statements can be made about it in a temporal sense. Database schemas can be based upon modal logic descriptions to provide guidance as to the structures that are definitely required for data that may not yet have been defined and structures that may be required for future data. Rather than referring to contingent facts, it is possible to refer to constraints that bind the database regardless of what has been stored within it.

The extensional and intensional relations [21] are predicates within the universe of discourse of the database. Producing an intensional relation requires, in logical terms, a rule that states:

\[
p(X, Z) :\ a(X, Y) \& a(Y, Z)
\]

where \( p \) is the intensional predicate that is formed on demand but never explicitly stored and \( a \) is the extensional predicate that has associated data stored in the database. This is classified as a \textit{conjunctive query} because the result is the conjunction of two extensional predicates. What this illustrates is that databases, like the class relationships already explored, have a valid predicate representation. This lays the foundation for later work on the relationships between the stored form of data in a database and the logical relationship to a predicate-based representation and, ultimately, the knowledge domains that define the universe of discourse.

This separates discussion of predicates and entities into the discussion of what does exist, which is FOL or second-order logic depending on what happens with quantifiers, to what \textit{may} exist or what \textit{has to} exist. The predicate relationship:

\[
(\exists x)(\exists y) : \text{Mother}(x, y) \& \text{Person}(x) \& \text{Person}(y) \Rightarrow \text{ChildOf}(x, y)
\]

states that if \( x \) and \( y \) are linked by a motherhood relationship then \( y \) is the child of \( x \). But there is no requirement that person \( y \) has a mother. This relationship states what can be inferred if \( y \) does have a mother. However, the next statement in modal logic:

\[
(\forall x)\text{Person}(x) \Rightarrow \Box(\exists y)\text{Mother}(y, x)
\]

states that if \( x \) is a Person then there must exist a Mother \( y \). \( \Box \), in this context, represents the unary operator denoting necessity. This is a stronger statement than can be made in FOL and specifies a constraint that must always be true, rather than a contingent fact that exists, is true but proves no more than it exists.

A logic does not necessarily have to be first- or second-order to be of use. A partial, or incomplete, logic can still be used providing that it is only used in certain ways. Assumptions that would normally be made about FOL cannot be made about it. As it is defined, RDF is not a first-order logic but it was initially assumed that an extension of RDF could be shown to be FOL.
However, attempts to provide a same-syntax extension of RDF to first-order logic [83] have failed due to the discovery of diagonalisation paradoxes while attempting to construct the FOL. Although the syntax can be extended successfully, the model cannot be maintained because truth values cannot be assigned correctly due to the possible existence of paradoxical sentences. While this has led to other foundations for the semantic web [84, 85], there is no major implication for an ontology language built upon RDF, providing that the truth predicates are chosen correctly. The Web Ontology Language, OWL, has a weaker form of logic than RDF and, thus, may not fall prey to the same paradoxical problems [83]. At time of writing, no work had identified any impact upon the use of OWL or reasoning in OWL that was being affected by such paradoxes, nor had any been identified in theory. Thus, the logical underpinnings of OWL are sufficient for the concepts expressed in the language, it is internally consistent, but may not be extended beyond the supplied relationships and operations as it has not been proven that it would validly support a FOL at this stage.

2.4.3 Description Logics

Description logics (DLs) are a family of logic-based knowledge representation languages that have formal semantics and provide inference services [86]. They are more specifically defined as ‘a field of research that has studied a particular decidable fragment of first order logic’ [75]. It is possible to use DL-based systems to deduce implicit knowledge from explicit knowledge present in the system. As an example, the subsumption algorithm allows the determination of subconcept-superconcept relationships even where such relationships are not explicitly known but, to ensure reasonable behaviour, this subsumption problem must be ultimately decidable and, preferably, of low complexity. The algorithms for deciding subsumption of expressive DLs, which allow disjunction or negation, are known as tableau-based algorithms as they can be seen as specialisations of of the tableau calculus for first-order predicates [87]. Highly-optimised tableau systems exist for systems that have exponential time complexity and, due to these optimisations, these systems can be used without infeasible execution and decision times. This allows the use of the inference services over highly-expressive DLs that have such complexity.

A DL knowledge base is normally separated into two parts: the TBox and the ABox [86]. The TBox is concerned with terminological knowledge, hence the T in TBox, and contains concept definitions and axioms associated with the knowledge base. The ABox is concerned with assertional knowledge and contains both concept and role assertions. For example, a TBox could contain an axiom such as “A father is a man who has a child”,
where the ABox would contain the assertion that “Bill is a Father”.

For most DLs, the inference problems that may be encountered are decidable, as opposed to first order logic (FOL) which is semi-decidable, with time complexities ranging from polynomial to exponential time. Most DLs are decidable fragments of FOL and are closely related to modal logic with universal roles taking the place of TBox axioms and nominals in place of ABox entries.

The families of DLs are named in a way that identifies the features of a particular language. Thus, the names $SHIF(D)$ and $SHOIN(D)$ identify two distinct DLs with some similarities but some, important, differences. Both are based on the $ALC_{R+}$ DL, which is abbreviated to the $S$ character to reduce the number of characters in names [88]. $ALC$ is the smallest propositionally closed DL and includes Boolean constructors and restrictions on role successors. $ALC_{R+}$ adds transitively closed primitive roles [89]. Both languages support inverse roles (the $I$) and role hierarchies ($H$) but $SHIF$ does not support nominals and number restrictions but it does support conceptual agreement. The $F$ represents functional properties, where the $N$ denotes cardinality restrictions. $SHOIN$ extends $SHIF$ with the ability to specify individual names in the description language ($O$ denotes nominals), as well as in the ABox, and also allows the restriction of the number of associated entities in the language. Finally, the $D$ after the DL name indicates that the language has been extended with simple concrete datatypes - this is an extension to the DL variant, rather than a different DL variant.

Both $SHIF(D)$ and $SHOIN(D)$ are decidable DLs. If another language is either a same-syntax translation of the DL or it can be mapped into the DL in finite, polynomial time without loss then the mapped language is also decidable. These two DL languages can be used to understand the behaviour of two of the sub-languages of the OWL language family. This is discussed in the following section.

2.4.4 Ontologies

Although logic states that something can, does not, should or may exist, it has no vocabulary for describing the things that actually exist. An ontology provides this vocabulary and categorises information and knowledge [80]. The term ontology originates in Philosophy where it refers to the studies of the nature of being but the term has been appropriated by the Computer Science community, where it has been given a slightly different meaning.

In terms of knowledge representation, an ontology is defined as a description of the contents of a domain of interest using a specified language for describing the domain. In terms of computer science, an ontology is a machine-interpretable vocabulary specified in
such a way to allow the relationship of included terms, including the determination of the sub-class relationship [80, 90].

Regardless of whether a machine ontology has been produced for a set of data, if a person refers to and provides relationships between data using language: they are referring to the data ontologically, albeit informally, and using their description and language to assign meaning to the elements that comprise the dataset. A computer-based ontology uses a well-defined language and structures to discuss the relationships between the resources in a more formal manner but it is merely a concrete representation of the formal statement of the conceptualised semantic aspects of the data [80].

In order to be useful beyond the defining entity, an ontology must be capable of being used by other entities or people. To this end, the ontology’s structure and defining language should lend themselves to being understood and, also, being capable of being used by other agents. The focus of ontological research is now commonly directed towards sharing ontological descriptions across a network, whether that is across the Semantic Web, as explicit ontological specifications, or, less formally, through custom client/server systems and the Grid.

To be useful within the context of the WWW, an ontology language must fulfil three important requirements [60]:

- It must be highly intuitive
- It must have well-defined formal semantics
- It must be interoperable with existing Web languages, such as XML and RDF.

Ontology Inference Layer (OIL) [91] extends RDFS to an ontology language. The DARPA Agent Markup Language (DAML) was developed to support the semantic web and was initially released as DAML-ONT, an onotology language for the semantic web [78]. Other ontology languages that existed before these, and were consulted in their design, included KIF [92], SHOE [93] and OKBC [93].

OIL represents an ontology with an ontology container and an ontology definition. The ontology container is concerned with describing features of such an ontology, like author, name and subject where the ontology definition defines a particular ontological vocabulary. The ontology definitions are part of a set, consisting of import, class and slot definitions, type, subclass-of and slot-constraints [94]. DAML and OIL were combined into a new ontology language (DAML+OIL) which concentrated on clear semantics for the language and improved compatibility of DAML+OIL with formal foundations [78, 79].
Later research became focused on web-based applications for ontologies and the Web Ontology Language (OWL) was produced. OWL is a revision of DAML+OIL incorporating lessons learned from the production of DAML+OIL [79, 75]. OWL extends RDFS and adds vocabulary for describing classes and properties including, but not limited to, cardinality, equality and relations between classes. OWL uses a Subject Verb Object (SVO) format when written that makes it more easily human-readable [78]. We will discuss OWL further in Chapter 6, as part of the conceptual model that forms the basis for our approach.

OWL has three sub-languages or dialects, to provide differing levels of complexity as required by the ontology designer. OWL Lite permits restricted operations and supports classification hierarchies and simple constraints. For example, although OWL Lite permits a cardinality constraint, it restricts cardinality to be either 0 or 1. OWL DL supports more expressivity while guaranteeing computational completion and decidability. OWL Full grants full expressiveness but without any guarantees of decidability. Decidability is important because, without it, there is no way of guaranteeing whether a given formula in the ontology is valid or not. Without being able to guarantee the validity in finite time, we cannot apply reasoners to the ontology and expect a result in finite time.

In order to encode OWL Lite and OWL DL into their DL equivalents, each axiom and assertion in the ontology must be translated into axioms in the DL knowledge base. OWL Lite corresponds to the DL $\mathcal{SHI}(\mathcal{D})$ and OWL DL corresponds to the more expressive DL $\mathcal{SHOIN}(\mathcal{D})$ [75, 95]. OWL Full does not have a corresponding DL as it has no restrictions on its use of RDFS, whereas OWL-DL can only use a subset of RDFS. There is no decidable mapping for OWL Full. The DL support for OWL allows the use of external reasoners that can infer new statements from those defined in OWL-Lite or OWL-DL, as OWL-Lite is a strict subset of OWL-DL. OWL Full cannot be used by DL reasoners and there is no guarantee that a classification of the information in an OWL Full ontology can be carried out in finite time. However, while the classification of OWL-DL is guaranteed to complete, there is no guidance as to how long such completion will take and, as will be shown later, the lack of time guarantee versus loss of expressiveness can require a reassessment of the OWL language used. Neither OWL-Lite or OWL-DL can use all of the expressiveness available from the underlying layers of the model, RDFS for example, as this affects decidability.

From Figure 2.4, the RDF and XML heritage of OWL, in this case OWL DL, can clearly be seen. The XML-ised statements take the standard form of nested elements with well-defined attributes, and the RDF namespace can be seen functioning as a container for the whole, with RDFS and OWL namespace used to denote different attributes for individual elements. From the small fragment describing the WineGrape class, the Subject, Verb,
Figure 2.4: An example of OWL DL

Object (SVO) order means that this fragment is read as ‘WineGrape is a subClassOf the resource within food called Grape’ and is clearly comprehensible to the prepared human and machine alike.

While OWL has a rich set of class constructors, it lacks a powerful language for discussing the properties of an ontology [96]. Extending the OWL DL sub-language to provide this mechanism could compromise decidability unless the properties that could be captured were heavily limited. OWL Rules (now the Semantic Web Rules Language or SWRL) proposes a mechanism which, through the use of Horn clauses, adds rules that take the form of an implication between an antecedent and a consequent. Although there are still restrictions in the application of the rules, as the unfettered use of Horn clauses in this context will prevent any guarantee of decidability, this addition allows the use of additional annotations to the ontology, axioms about classes and properties and the ability to establish facts about OWL individuals. Finally, the rules allow for the manipulation of themselves. Although SWRL is not used in this thesis, the ideas and extensions provided by SWRL parallel some of the aspects of our final implementation.

2.4.4.1 Providing context for data

The amount of context that can be captured using RDF, RDFS and OWL is vast and should be fully utilised to extract the most utility from data. Any system that limits annotation or metadata organisation to simple mechanisms will ultimately have to be revisited and manually annotated with the required information. More importantly, as metadata annotation becomes more widespread, the requirement to integrate non-annotated data streams
into annotated data streams will become more important. However, rather than looking at the new technologies as a way of capturing existing structures, these can also be seen as a way of producing new structures. There is no reason why a programmatic structural representation of data cannot be produced from a knowledge-domain based description of the problem and it is far easier to produce data structures from knowledge domains than it is to interrogate the data to produce local knowledge domains. This is due to the semantic richness of the knowledge domain based model.

Many existing metadata systems, such as the semantic file system proposed by [54], fail to capture a vast array of potential metadata as they cannot be determined through analysis of in-place text or file system characteristics. Although these form an important part of a provenance system they do not constitute a complete solution for the production of a fully-annotated semantically rich storage system. The automated capture of valuable metadata requires an understanding of the complete system and cannot rely on only those aspects that are visible on individual hosts within a system. Also, the semantic file system does not provide a higher-level semantic model from which the file structures and relationships can be derived.

2.4.5 Semantic web and web services

The Semantic web extends the WWW by using metadata to associate meaning with data. This metadata can be used for enhanced searching, cataloguing and information retrieval. Another useful extension to the WWW using metadata is that of Web services. From [72], with the author's addenda in parentheses:

‘A Web service is a platform and implementation independent software component that can be:

- described using a service description language (which provides a metadata description of the service),
- published to a registry of services (which provides a repository for service-based metadata),
- discovered through a standard mechanism (which exploits this metadata),
- invoked through a declared API,
- composed with other services.’

Web services are stateless entities and have the potential to persist in their service role for a long time. Comparing this with the service of web pages over HTTP shows that the
default behaviour of the WWW is also to provide resources that are stateless but potentially long-lived. In this case, the underlying service is the provision of the text in the document rather than a computational service that transforms data. Since their state does not change they will continue to be provided in the same form until an agent external to the system, such as a webmaster or human author, interrupts the provision of the service. Web services are described using the Web Services Definition Language (WSDL)[73]. WSDL allows the specification of web services as end points and operations, with a common binding mechanism. A WSDL document is a set of definitions that may be in one file or made up from imported data from other files.

2.4.5.1 Semantic mark-up for web services

The two most significant mechanisms that can be used to add semantic information to web services are Semantic Annotation for WSDL (SAWSDL) [97] and OWL-S [98].

2.4.5.2 Semantic Annotation for WSDL

SAWSDL allows the description of semantics through annotations to existing WSDL documents. Model references can be embedded using existing WSDL extension element syntax. WSDL operations and interfaces can be annotated with model references to tie them semantically to a taxonomy or ontology that is defined externally.

XML-Schema elements in the definition are extended with the extension attributes liftingSchemaMapping and loweringSchemaMapping in order to provide a semantic grounding. These are used to associate schema types or elements with a mapping to an ontology. The liftingSchemaMapping is a, possibly empty, set of URIs that provide a reference to mapping definitions. This defines how an XML instance that conforms to the specified element or type is transformed to data that conforms to the semantic model. The loweringSchemaMapping provides the reverse transformation from data conforming to the semantic model to XML instance data. Where multiple URIs are specified for either attribute, these are seen as alternatives and the client processor may choose to apply any of them. An example usage of these alternatives is to provide multiple mapping URIs for different languages. While the semantic transformation may be functionally identical, a different language (French versus English) may cause a significant change to the XML instance data as well as the names of the concepts in the attached taxonomy. Multiple URIs as alternatives allow the maintenance of the semantic grounding while allowing flexibility.

SAWSDL provides an excellent mechanism for grounding WSDL definitions in an existing semantic reference document. However, it requires the existence of the grounding
taxonomy or ontology but does not provide a mechanism for extending or altering the reference document. A range of different ontological versions, potentially supporting alternative views of the service, can be supported using the alternative URI mechanism in the lifting and lowering Schema mappings. The semantic grounding provided by this approach only applies to those elements that have model references associated with them. The constructive elements of the definition are only grounded in the WSDL definition and, as a result, the definition cannot easily be made self-modifying. There is no explicit context that can be used to define, manipulate and alter the structures which, in turn, are used to construct the semantic mappings for SAWSDL in the first place.

2.4.5.3 OWL-S

OWL-S [98] provides an alternative approach using a service ontology to allow the statement of processes and their inputs and outputs in terms of resources. It provides a service profile for advertising and discovering services, a process model to specify operation and the grounding, which provides the information required to interact with a service. The Service ontology provides a ServiceProfile, ServiceGrounding and ServiceModel class to support these activities.

Within a certain context, service selection and match-making can be carried out very effectively using OWL-S. However, while OWL-S can also be used to describe some aspects of a knowledge-domain based solution, the lack of depth in the description of the processes limits its applicability. This is addressed in more detail in section 9.2.3. OWL-S can use Knowledge Interchange Format (KIF) [92] expressions to specify preconditions and effects in order to show the impact of processes on the execution of an agent/server interaction and to provide programmatic guards. These KIF-expressions are used to provide the representation for logical formulas within an XML-ised format, OWL-S. Figure 2.5 shows an example precondition for use in a banking system, where the number of the credit card must be known before it can be sent to a web agent. Preconditions and effects allow the programmer to have more control over the service and, through the use of these logical statements, make the knowledge base conform to system requirements.

What is lacking is the context for the KIF-expressions and, as a result, the ability change context for the variables and control structures used in a KIF-expression. Although they unify contextually bound variables, with OWL-S resource IDs, a large portion of OWL-S is used to describe the structural relationships of the described entities to their parameters and outputs. Without uniform global binding of variables across a system, it is possible to misuse or lose the context of the variable and, by extension, any of the rules that are used on that variable. Strong context ensures reliable knowledge and a shaky contextual foundation
can be repaired but it is much harder to maintain. This lack of context is in opposition to the knowledge-centric approach that focuses on describing the entity so that it becomes knowledge and then relating knowledge elements to each other.

### 2.4.6 The Semantic Grid

As has already been discussed, the Grid is a current focus of collaborative distributed resource usage. Including legacy systems within the Grid framework has become a separate area of research. The Australian Partnership for Advanced Computing (APAC) has recently produced discussion documents that deal with the provision of a GridDNS service [99], although this service focuses on a non-traditional use of traditional DNS technology and servers through database manipulation and the leverage of future developments in DNS. The disparity between the Grid and legacy system types leads to problems in their integration mainly due to the difference in transmission protocols and a lack of semantic depth in the definition of relationships between the two systems.

The Grid community have started to look at adding semantically rich metadata to the Grid computing environments [100, 71, 101]. This notion of a Semantic Grid provides a means for annotating Grid computational elements, resources and services in such a way that automation can be used more effectively, as well as providing a backbone for provenance and repeatability. The semantic grid is a service-oriented architecture with a highly developed knowledge layer that tracks the addition to, and maintenance of, the knowledge captured within the system. Data streaming from instruments must be annotated with metadata to provide their meaning. Once meaning and context have been associated with the data then the domain ontology can be used to provide the data as knowledge. This allows the widespread use of agents in the negotiation of knowledge and processing markets. Agents can also free human researchers to perform the work that they are paid to do, rather than having to manipulate computer resources to achieve their aims. The semantic grid
is regarded as a basis for the successful and effective adoption of e-Science, where non-computer scientists can use computing resources effectively, efficiently and collaboratively while still retaining control of their own research and the use of their own resources.

The volume of metadata that must be generated to annotate the data is large and this does not take into account the amount of metadata from previous computation and experiments that have not yet been annotated with semantically rich metadata. This motivates an efficient mechanism for annotation of previously generated data as well as the automated annotation of new data. This automated annotation provides the basis for reliable collection of provenance data at a user-defined level of granularity and also without requiring the user to pierce any system abstractions to manually annotate items implicitly contained within the process workflow. For example, a user may only wish to know which nodes have been used since they are using a well-defined workflow that has been used on many other systems. Another user may wish to keep information on node use, software version, input files, temporary files used and timing information collected as part of the execution. Users have different requirements for the amount and type of provenance data kept and this must be reflected in any system.

The semantic grid [100, 71, 101] provides a service-oriented model for advanced grid computing that relies upon a semantically rich metadata model and the extensive use of capable and autonomous agents. This works to extend the computational environment of a scientist by providing data, information and knowledge to human and machine alike, to increase system effectiveness and enhance the efficiency of operations. The focus on knowledge capture and maintenance establishes a clear requirement for automated annotation but does so at an agent level rather than as a service provided for the system overall.

DeRoure identifies that annotation services are required that will run over large, distributed data sets, possibly in real time [100]. Such a system may also be required to deal with multiple ontologies to annotate data so that all of the possible valid meanings are associated with the data.

However, the integration of legacy distributed services into this framework requires either the replication of the service, or its system, within the semantic web or the use of wrappers to provide a semantic overlay that is used by SW clients to retrieve and use data. This semantic annotation is shallow as it does not extend to the underlying service itself but is a masking layer through which other services can access it. The production of the wrapper is based on an analysis of the underlying service and a mapping between data structures in the unannotated system and concepts and structures in the annotated system. Any change to the underlying system, a standards-directed change, a local feature extension or a bug fix, may invalidate this mapping as it cannot be based on semantic
NOTE: This figure is included on page 76 of the print copy of the thesis held in the University of Adelaide Library.

Figure 2.6: myGrid services and middleware stack from [64].

alignment with the underlying service if the underlying service cannot provide semantic information on its data. There are a large number of legacy services that could be extremely useful in e-Science, Semantic Web and Grid applications if metadata annotation could be extended throughout the system. This usefulness increases dramatically when considering the volume of data that is generated in e-Science experiments and that has to have metadata generated and stored for the reasons already listed.

2.4.7 A modern workflow-based system: myGrid

The myGrid project [64, 102, 103] is an e-Science pilot research project with an aim to develop high-level middleware to support in silico biology experiments.

The myGrid information repository (mIR) stores all information considered to be relevant to the system user who is conducting an experiment. The Workflow enactment engine collaborates with the mIR to annotate all items described within the workflow and metadata storage is a key feature of the mIR.

The mIR could be used by many users in a single organisation to store their metadata, provenance information and data. myGrid is designed to assist the scientist with a collaborating group of services that provide a storage system that is used with workflow mechanisms and process abstractions in order to facilitate better data usage and reuse.

There is also no explicit statement of what underlying system resources are required to provide such a service, even when implementing the mIR. myGrid is focussed on the upper reaches of the middleware layer and does not address in detail how the lower layer must be
composed. Hence, an external entity may not be able to interact in any way with the myGrid
system, especially when it comes to dealing with provenance data and metadata annotation,
as the system is designed with strict services for internal use that may not accommodate
external entities.

2.4.8 A legacy distributed system: The Domain Name System

Legacy systems will almost always be a poor fit for integration with semantically-rich
metadata-based systems. The implicit capability requirements that drove the production
of new systems occurred after the legacy system development and many of these require-
ments are in response to perceived shortfalls in the design of legacy systems. The Domain
Name System (DNS) [3, 4] is characterised as follows:

- A logical name space is provided by the use of SRV records that record the Name-
  Server associated with a given domain. However, use of SRV records is neither
  compulsory nor guaranteed, as these are a relatively recent addition to the stable of
  DNS Resource Records.

- Storage abstractions are not provided directly to external clients. Trusted servers
  may exchange domain name zone information in encrypted and plain formats with
  a server but there is no abstraction for storage on the server node itself. To make a
  change to the information stored, the configuration files must be directly edited.

- Information repository abstraction is effectively non-existent. There is no way to
  change the data structures and metadata stored within the DNS without rewriting the
  source code.

- Distributed resilient architecture is implemented through a client/server architecture
  and the use of caching, automatic fail-over and replicas.

- There is no truly ephemeral data in the domain name system - it is all defined and
  stored in zone files.

2.4.8.1 A brief description of the DNS

The DNS is a service that stores and provides mappings as a hierarchical system, employ-
ing distributed management, extensive use of caching and a strong standards-based model
for making changes to the system.
The DNS can provide name to IP address mappings (A records), IP address to name mappings (PTR records), canonical names (CNAME records) for IP addresses so that multiple names map to one IP address, facilities to support global e-mail (MX records), location of name servers (NS records) and, more recently, even the location of particular network services within a domain (SRV records) [7]. These resource records make up the heart of DNS and are all found within the zone files that provide the DNS data that name servers, in turn, provide in response to client requests.

### 2.4.8.2 Basic DNS elements

DNS has three major components, described in RFC 1034[3]. These are the domain name space itself, name servers, and resolvers. Resolvers query name servers to resolve client requests and are usually found at system level and are directly accessible by user programs.

Within the domain name space, zone files are used to store the data associated with a particular domain. A zone file is composed of resource records. Resource records (RRs) store information about the owner, the type of resource, the time to live, the protocol family and the type-dependent format data. RR Type Numbers are used to identify the payload of a message sent to or from a DNS server. Rather than use the mnemonic ‘A’ or ‘PTR’ to identify the record, a pre-defined table of mappings from name to number is used and this number is used to identify the payload.

We will discuss the DNS elements throughout this thesis but will develop a discussion of the details of DNS’s role in our implementation in Chapter 7.

### 2.4.8.3 The operation of the DNS

The DNS is not registry based and, as it is currently designed, it cannot be used in a registry environment without a wrapping Web Service to take the request, convert it to DNS Message Format, and send it on, performing the reverse translation on the return. The DNS was designed before distributed file-systems, or even NFS, were in widespread use and has generally been seen as a source of authoritative mappings with relatively long lifespans. With the introduction of internet service providers (ISPs) using DHCP addressing to provide users with a constantly changing IP address, this assumption of longevity is now no longer accurate.

The structures of the DNS are strongly controlled by standards. Thus, there is currently no widespread perceived need to allow the mutation or introduction of new structures and metadata. In the absence of registries, semantic match-making or other mechanisms, there is no other way to guarantee that two DNS servers and clients will be able to intercommu-
nicate other than resorting to a strong standards-based approach.

The DNS does have a very strong and reliable distributed architecture, due to its importance in global computing. This model distributes load, provides secondary sources of information and, because of the widespread use of caching, allows for temporary disruptions of service without immediate loss of information.

The ephemeral nature of DNS data is questionable as the mappings are defined in a file stored on the server’s file systems. Hence, the data is technically not ephemeral as it does exist in concrete form. However, recent changes in the use of DNS servers means that that data may only be written to the file for a minute or two before there is an update, a server reload and the mapping that was there is now extinct. Because of DNS caching, the mapping may continue to exist for a finite time until all of the caches storing the data have reloaded from the primary. At this point, a mapping is available that is no longer defined in a stored format since cached copies exist for which there is no corresponding data file on the originating server. To clarify the difference between data that is produced without a physically stored representation and data that is merely short-lived or, as in the example, cache resident, the term synthetic data will be used. Synthetic data is data synthesised from the data or metadata already present in the system. Such data may be created using any of the metadata in a system, not just that which is formally defined ontologically and relates to individual data items.

For example, the number of IP addresses in use in a certain range is metadata. However, a “numberInUse” property associated with any one of the IP addresses will be “1” or “0” because that is the context in which it makes sense. There could be a “numberInUse” property associated with the instance of the IP Range that contains the IP addresses. One way of dealing with this is that, every time an address is added, the number is incremented. There are significant overheads when this process is carried out manually, as well as it being error-prone. Even with automated tools to assist in the process, there is a possibility of confusion unless a single data set is used to determine if IP addresses are free or not. Any tool to keep track of IP address usage should draw on the database that defines the zone and, hence, should access the DNS data directly. A scalable automated solution is to define a means of counting those elements in the IP range that are in use and then returning synthesised data that gives that number. Once the element counting routine is written, any IP range can be counted and the property “numberInUse” can return its object, a synthesised integer. This simple application cannot be added to existing DNS servers in a standard way without moving through the RFC process. It can be carried out locally within an administrative domain but, as a server-side operation returning a single annotated integer, this restricts its use to the local administrative domain.
2.4.9 Shortfalls

In summary, widely-distributed network-based systems are extremely valuable resources but their longevity and non-registry based approach to resource location make them a relatively poor fit for a number of modern applications, including e-Science and other semantic web applications. The standards-based update and revision procedure, combined with potential conflicts in purpose and application at local sites, makes them slow to update and prone to the formation of clusters of obsolescent nodes. While centralised standards guarantee interoperability, this comes at the cost of agility. Local administrative groups may see a need for an adaptation of an existing service or system that is at odds with the views of the global user community. In a consensus or majority-based decision process, this will prevent local users from being able to take advantage of the existing work that has produced the system and may force them to either write their own system from scratch or adapt the global system, recoding it each time any new standards break their changes. This is especially true of legacy-modern hybrid information systems or any information system where users can drive the views of such data.

If the data contained within such a system can be annotated in a way that allows deep semantic annotation and, hence, alignment with external technologies, the translation burden between legacy and current systems can be significantly reduced. Also, the specification of the data and operations of the system in a logical and structured manner can allow the separation of core functionality, as specified by the defining standards, and any local (extension) functionality to allow local sites and users to get the best use of a service. This applies even where such a use will never be part of a future standard, as it is a strictly local and small-scale change. When the correct specification mechanisms are used, the descriptions can be externally validated to ensure that they are logically correct and that they still entail the standards.

There are currently two parallel modes of operation for large-scale widely-distributed network-based information systems. The first is that of the legacy systems that have been developed as network services, using well-known port numbers and eschewing registry systems. These are metadata-poor and normally constructed using strong standards, which allow little room for variation. The second is that of the new generation systems that have been developed with an informational focus, using registries to advertise their services in conjunction with service description mechanisms such as WSDL, SAWSDL and OWL-S. These are potentially metadata rich and can support varying levels of annotation. Because of the ability to compose services based on semantic match-making, there is less requirement for interface based standards as parameter matching can be carried on demand using the service description.
However, despite these two dissimilar approaches, the core of each system is the storage and provision of data, in raw or transformed modes. Any user of such a system will want access to the stored data, and the ability to store data, in a way that aligns well with their use model. When these services reach subscription levels and ranges that span countries, there is an increasing possibility that users of the service may prefer the data in a slightly different organisational structure or provided in a slightly different way. Rather than the default global view that is placed upon systems through their default syntax and semantics, these users would like a locally defined view, while at the same time retaining their ability to use the global service.

The user’s view of the data, whether global or local, is a perception of the stored data and, ultimately, this data is a machine-based representation of a knowledge domain that comprises all or part of the universe of discourse for the problem. There is no good reason why implementation specific issues should prevent a user from changing their view of the data and there is also no good reason why the same data cannot support multiple views. The way that data is composed for one user does not require the in-store refactoring of all data, in the same way that a table join in a database query does not prevent other users from joining tables into different formats. However, at this level, the conceptual reordering of data is taking place at the individual structural level, before any concept of table membership need be considered.

What is missing from the legacy system approach and the metadata-rich more recent approach is the ability to specify the stored information and the operations required to interact with it in terms of knowledge domains. A knowledge-based approach completely decouples implementation issues from the information that is stored in the system and allows for the dynamic specification of format translation, external protocol transmission formats, the production of predicate operations and the inference of new information without requiring a formal standards-based system. Standards can still be used to allow interaction with existing standards-based systems but, by defining two knowledge domains where one focusses on the standard and one focusses of the local view, the same data can appear in both systems without compromising the requirements of either.

In this thesis we will show how a focus on the specification of data and operations in terms of the logic, representation and operational semantics of the system will free a system node from being bound to one mode or the other. Such nodes perform more persystem-operation processing than a dedicated node of a traditional system so impose a small performance overhead that may limit their deployment as the ubiquitous node type in a system. However, such nodes, at the boundary of two or more systems, can function as fully-functional members of all the systems for which they have a knowledge representation.
and, at the same time, make their stored data available to all the other systems. These boundary nodes, or adapter nodes, support semantic annotation, can respond to requests made in a variety of protocols from different systems and provide a virtual adaptation layer similar to the presentation layer in the ISO/OSI network model, except that it is provided between entire systems of nodes. For this reason, the overall system proposed and developed in this thesis is referred to as Semantically Annotated Multi-Protocol Adapter Nodes (SAMPAN).

The next chapter describes the challenges that must be overcome to implement a network-based information system and leads into the methodology chapter, which builds on the information detailed here and that of the next chapter to lay the foundations for the new approach developed in this thesis.
Chapter 3

Challenges to implementing a network-based information system

We wish to design a system that is a versatile and efficient mechanism for implementing a network-based information system. We must, therefore, describe the problems that face the implementers of network-based information systems and how these may be overcome in order to provide support for design decisions made in Chapters 4 and 7. This provides a reference point for later development that can be used to understand, as well as justify, the decisions made. This chapter contains important context for the development of the themes of this thesis. This chapter also illustrates that there is already sufficient complexity to make the design and correct implementation of an NBIS a non-trivial process. The issues raised in this chapter increase the complexity of implementation but do so while introducing significant benefits for the long term use and interoperability of the NBIS.

The challenges facing the developers of an NBIS can be significant and there are existing solutions to these challenges, but there are a number of trade-offs that must be considered before applying a given solution. The most usual trade-off is one of cost, as the introduction of special purpose hardware and software can resolve a great many of the problems encountered. When the trade-offs are resolved, the resulting NBIS are characterised by robustness in their deployment and their very large potential user base. The most important challenges are:

1. Heterogeneity - the processing units will not all be of the same architecture or configuration and hence cannot be assumed to execute a program compiled for a single architecture or configuration. They can also not be assumed to provide the same performance. This has an impact on the management of such a system. Such systems may require detailed attention to synchronisation if distributed results are to
be channelled through a single node for assembly or further processing. Also, such uncertainty regarding the executing platform makes the prediction and assessment of future performance crucial to providing any guarantee of performance. From an NBIS perspective, this creates uncertainty in query/response times, as well as the time taken to replicate or cache data in different parts of the system.

2. Public network - A local area network (LAN) or wide area network (WAN) will have different bandwidth, latency and routing properties at different points in time, compared to a dedicated data network in a single multi-processor machine. More importantly, a public network will service many other users and may not have a quality of service strategy or prioritisation mechanism to provide any overall guarantee of network performance. Realistically, any QoS guarantees are only as good as the underlying stability of the public network: a guaranteed bit rate cannot be maintained if the physical medium suffers catastrophic failure. Ultimately, the failure of the connectivity media causes the failure of any NBIS, even with extensive caching, because isolated NBIS elements will eventually contain stale data that is inaccessible.

3. Non-ownership - Not owning all of the components that are used for the computation has impact on the nature and resource load of the computation. While a dedicated multi-processor machine can be employed on one task, the nodes in a distributed system may be performing multiple tasks for the system or may even be carrying out non-system oriented activities while system jobs run. NBIS traditionally execute in environments where resources are dedicated to them by local management, rather than in a cycle-stealing model, but it is unusual for core servers to be dedicated to NBIS infrastructure at all levels of the tree. This is particularly true of those NBIS that provide less frequently used services or services that affect a smaller number of users. Large-scale NBIS, like the DNS, can have dedicated servers but even a service a vital as this is often run as part of a set of services at the sub-domain level.

4. Non-local storage - The storage used may not reside on the user’s initiating host and may not even reside as one physical resource within the system. There are many issues with non-local storage, some of which touch on resource usage implications of non-owned systems and some of which deal with security. In short, if you cannot control the storage location then you cannot guarantee the data’s integrity as the host writing to disk must be able to interpret the bytes being written as they are written to disk. Network-based information systems distribute information across a network - an NBIS without sound distributed storage is as useless as an NBIS without network connectivity.
5. Protocols and policies - In order to function as a virtual single entity, the distributed system needs to establish intercommunication protocols and policies that incorporate every node and meet the requirements of the user. While this is a guarantee of interoperability, it also comprises a set of constraints upon the system that restrict the different mechanisms of communication that may be employed. If the standard that forms the basis of protocols and policies is never varied, then no variation can be tolerated in the protocols and policies. Because NBIS traditionally span organisations, state and country boundaries, a standardised access and transport mechanism is the only way to guarantee interoperation and reliable operation without requiring a control and change mechanism for each node and instance of the system. However, NBIS are often so useful that many organisations can think of alternative uses for the NBIS - only some of which fit within existing standards. In the constricting environment of unchanging standards, there is no room for these new uses, especially where they only have a benefit to small, user-local groups.

3.1 Addressing the challenges to implementing a network-based information system

The larger and more diverse a distributed computing environment becomes, the more difficult it is to allow the ad-hoc introduction of nodes. In a registry-based system, the new nodes have to locate the registry and provide all the required protocols, whereas in a non-registry-based system the key contact nodes must be located first, then connected to, after which protocol matching takes place. These are, by no means, the only problems that beset the production and ongoing operation of a large distributed application.

3.1.1 Heterogeneity

Architectural heterogeneity may be masked through the use of virtual execution environments, such as Sun Microsystems’s Java [104] or Microsoft’s .NET [105]. Other mechanisms include the use of strict policies that control what can and cannot be exchanged between nodes, limiting the impact of heterogeneity by requiring conformance to a homogeneous standard. One example is HTTP, which allows WWW traffic to be visible on all platforms. The HTTP protocol is defined explicitly to ensure that programmers can produce compliant servers - if the protocol is not followed then information exchange cannot be guaranteed [106]. XML plays a similar role as a carrier syntax, often with HTTP, in other protocols, such as SOAP [107]. Because of this focus on protocol standardisation, the
challenge of heterogeneity is often strongly associated with the issues found in protocols and policies.

The widespread use of certain programming languages and standardised application programming interfaces (APIs) for UNIX, such as POSIX [108] and the Single User Specification [109], have allowed the widespread use of self-configuring code bundles for certain distributed systems and network services. Such code bundles contain source code and a configuration mechanism that undertakes most, if not all, of the customisation of the source code to conform to the local configuration. Such a mechanism is only useful if the software is going to be in use for some time as there is non-trivial overhead in downloading, configuring and compiling the software. It also requires that the software source is available to the user. In situations where the source code is not released, a user is dependent upon the owner of the source to release compiled versions that match their processor architecture and requirements.

The local reconfiguration and compilation of software is made more difficult if the target machine has not been configured for local compilation. This problem frequently occurs with non-UNIX-based operating systems due to an expectation that users are unfamiliar with programming and will be mostly concerned with executing existing applications. However, it can also occur with UNIX-based operating systems when an installation has been chosen for space efficiency and, as a result, the compiler or required compilation files have been omitted from the system. This merely reinforces the underlying problem with heterogeneous nodes: namely, that little can be assumed about their capabilities and function beyond their existence and network connection.

Profiling a remote node in order to provide some guidance as to its performance is a very complex and error-prone procedure as the environment of the remote note may not be fully known or understood by the profiling agent. In a distributed computing environment, the farming out of tasks to a node that is orders of magnitude slower than its peers dramatically slows the computation. The same is true of NBIS as query resolution may take a much longer time on certain platforms than others. Apart from performance estimation, it is possible to use watchdog timers and process termination to control nodes that perform outside of specified time constraints. The theme of active intervention in the life of dispatched processes is revisited in the discussion on non-ownership of nodes, Section 3.1.3.

3.1.2 Public Network

Users may increase inter-node network efficiency when they have control over the underlying network fabric or have the active cooperation of the providers of a network. For
example, they may customise the network protocols used providing that all other nodes can also comprehend that protocol. However, this requires that any intervening active devices are configured to handle that protocol. A more common solution is to optimise the use of standard protocols but with optimisation at the operating system level. If such protocol enhancements execute at the kernel level [110] then substantial performance gains can be achieved but any predictability as to the possible gains becomes difficult to impossible when the network packets leave the local environment, even if guarantees of service have been made in the public network. This is because service guarantees can only be enforced if a possible network pathway continues to exist - in the case of widespread or catastrophic failure, any external service guarantees are no longer valid.

The public network cannot be effectively controlled as it ‘belongs’ to none of its users but is a service provided as a business by any of a group of network providers. Existing routing algorithms are designed to route packets through the network with a high probability of eventual delivery. The route taken may change from packet to packet and this can prevent any guarantee of specific node traversal unless virtual networks are overlayed that provide more control and can approximate the degree of control in a truly private network. Specifying network paths can be useful when different cost metrics, in terms of real-world cost, apply to different links. Without a mechanism for physically controlling packet delivery to networks, it is not possible to balance cost metrics in an active manner across the different routes available. Underlying internet service provider (ISP) and telecommunications company (Telco) protocols may support quite specific quality guarantees and bandwidth provision. This is rarely available for the type of ad hoc computation carried out in distributed systems, as the resource requirement is probably not known in advance. Ad hoc processing-based distributed systems are also rarely sufficiently significant that they can reserve 60% of an ISP’s trunk without booking it in advance. Hence, no assumptions can be made about the performance of, or propagation of traffic through, the public network.

Underlying protocols such as Asynchronous Transfer Mode (ATM)[111] support dedicated network pathways, as permanent virtual circuits (PVCs), and also dedicated and throttled bit rates through the network. However, this level of control is traditionally used by an ISP or Telco to control the provision of its service across all of its customers or for major, large volume, customers, rather than to make adjustments for one small variable-use customer. The exception to this occurs when a large group of customers combine to form a group of sufficient size that they can justify the provision of dedicated network resources. The cost and overhead of dedicated resources is significant and this option is certainly not available to all users.

Catastrophic media failure is always a threat to the most well-established quality of
service regime and, to minimise the threat, users and service providers must consider the use of alternative network routes utilising different physical media paths. However, this is no guarantee of connectivity as certain network points are hard or very expensive to duplicate, such as the undersea cables used to connect countries. The failure of such media pushes large amounts of traffic back onto any remaining connection points, such as high-latency satellites or older, and lower capacity, undersea cable.

3.1.2.1 The network as black box

For the vast majority of users, the public network infrastructure must be treated as a black box where performance can be inferred from experimental testing but no performance or quality guarantee can be depended upon [112]. Distributed systems that span a WAN cannot have built in latency dependencies or lag sensitivity that would prevent their use in wide spread network communication. This also effectively prevents the deployment of a real-time system on a WAN as, without guarantees of completion, the system cannot meet its requirements. As well, while there may still be local node performance enhancements to be obtained from kernel-level enhancements of networking code [110], these may have no impact on inter-nodal communications efficiency once the packets leave the LAN.

Since the characteristics of the network have a direct bearing on code and data mobility, there must exist strategies for dealing with poor network performance as well as a concept of a ‘giving up’ point where performance has degraded to an unacceptable level. These strategies are built into all of the long-lived network-based information systems that provide crucial information for the internet. For example, the DNS has a time-out facility for queries to ensure that queries do not hang indefinitely, waiting for a response. Also, as will be described in more detail later, there is a complex and robust caching mechanism so that, if contact is lost with a higher level authority, information already obtained can be served to bridge the gap until the other servers return to service or become visible again.

The worst case, and the case that should be avoided, occurs when process resources are consumed waiting for network resources that are not available. Every component of the NBIS should have an inbuilt mechanism for shutting down or releasing resources in a well-behaved manner if the underlying network assumptions cannot be met. Such behaviour may be defined at a system-level or a user-level. What is key is that all participants are aware of the limits and nature of such behaviour prior to entering into computation arrangements. This prevents network problems outside of the participants’ control from unnecessarily consuming resources in a way that has not been agreed to by all participants.
3.1.3 Non-ownership

Non-ownership of a computing resource implies that there is no guarantee that the resource will be there to perform computation or storage at the time that the user wishes to use it. More importantly, there is no guarantee that any data stored on such a system will be, by default, available on demand since local resource prioritisation may require that it be removed or the node removed from the system. The local resource scheme is not necessarily visible to the user, although if a description were available as metadata then the user could have some idea of what would happen when using a given node.

There is a distinct difference between a node that is contributed and dedicated to a distributed system and a node that is expected to perform other operations simultaneously. Clearly, the second node is only partially available and the distributed system is competing with other executing processes for this node. This is a cycle-sharing model, rather than a cycle-stealing model, unless the distributed system is set at a lower priority than all the other tasks being undertaken. In the case of NBIS, it is unlikely that such a prioritisation would take place so these systems would usually run in a cycle-sharing environment.

A dedicated compute node may be used by the person who is contributing it, if they are also submitting tasks, or may have been made available by a user or site as part of an experimental model or commercial arrangement. Providing that the location of computing tasks is undertaken in a way that is perceived to be fair, equitable and transparent, the contributing user will have reduced incentive to remove the machine from the computing environment if another job is currently using their machine at the cost of one of their own jobs. This strongly implies that such a system should not have a locally-defined access or prioritisation policy as the temptation to favour local jobs over non-local jobs would quickly reduce the efficiency of the distributed system and reduce it to a balkanised set of intercommunicating clusters incapable of transferring tasks. The key difference between a network-based information system and a more processing focussed distributed system is that the majority of traffic in an NBIS takes a query/response form, where the response is stored in the NBIS rather than calculated on demand. A processing focussed distributed system tends to provide synthesised responses in response to queries. Thus, it is in the interests of all participants on the NBIS to provide uniform access policies for foreign and local access since there is no guaranteed localisation of the information contained in the NBIS knowledge base. In the example of the DNS, preventing external users from easily accessing a site’s publicly-distributed DNS data will be an impediment only to those users accessing the site’s deliberately shared resources. A user not wishing to share DNS data would not advertise the existence of their DNS server, nor have it cached or maintained by any other users. If they choose to share DNS data, then they must exchange data with
outside agencies in a symmetric policy fashion.

Importantly, non-ownership does not necessarily have to imply a complete lack of control over the external nodes. When nodes are combined into a virtual organisation, particularly where adherence to a set of protocols or policies is established, the nodes are under more control than completely disorganised nodes. There is still no guarantee that the ruleset will be adhered to but at least now the requirements of the virtual organisation have been stated. The motivation for participants conforming to the policy is that all participants benefit from such behaviour - a distributed, replicated and redundant information service will only function properly if enough participants take part. Because network-based information systems tend to be long-lived, such participation is essential to the ongoing usefulness of the system as, without participation, over time the system would degrade in both the span of information it contains and the resources with which it can be delivered.

3.1.4 Non-local storage

Non-local storage represents a similar problem to non-ownership but is confounded by the requirement to provide a storage mechanism that can be used relatively reliably over the top of an inherently unreliable organisational structure combined with a potentially unreliable network infrastructure. Early efforts dealing with non-local storage included the network file systems (NFS) from Sun Microsystems [18, 19] and the Andrew filesystem [113]. NFS is now at version 4 [20], strongly influenced by elements of the Andrew FS, and has a very wide-spread cross platform distribution. NFS allows a local file system to be exported to a set of remote nodes with a variety of different options, including whether to block or not on when data cannot be retrieved and what actions to take when a local root user tries to access superuser files on the exported file system.

Network-based information systems can take one of two approaches regarding the storage of their data. The first is to provide a distributed store, with each server providing some space for storage and potentially some space for replication and caching as well. Accessing a node is a connection to the store as a whole and there may be no guarantee as to where the information to be retrieved is actually stored. If a user coincidentally accesses information stored at their node, there may be reduced retrieval time due to the removal of network traversal time. However, in this case, as the distribution of components of the store would be for storage and retrieval efficiency of the system as a whole, such behaviour is not necessarily predictable or desirable.

In the second approach, the information space itself is split up into separate sub-spaces and each of these sub-spaces is explicitly located at a certain location. Information in a
given sub-space may be replicated elsewhere but, assuming that the location of a given sub-
space is known, a user can predict which server to access to deal directly, or authoritatively,
with a sub-set of the information stored by the data. The namespace of either mechanism
must be designed to prevent the accidental or deliberate destruction of information that is
not owned by the changing entity.

Efficiency also dictates the use of caching, where appropriate, to prevent large amounts
of byte-sized information traversing the network in an inefficient manner. This then re-
quires a sound cache consistency mechanism and the use of locking mechanisms, at vari-
ous levels, to prevent users from accidentally writing to stale data or overwriting the recent
changes of other users. Privacy and digital rights issues may be used to motivate the en-
cryption of data in the distributed storage systems, with decryption only occurring during
delivery or possibly at a separate decrypting node. This would prevent the privacy issues
mentioned in the associated challenge section but comes at a performance and scaleability
cost.

3.1.5 Protocols and policies

Finally, protocol and policies allow the statement of the ‘rules of conduct’ for a system.
These can take several forms including information exchange protocols, such as HTTP
and TCP/IP, or as service level agreements for establishing system performance charac-
teristics. Without the existence of formal protocols, it would be impossible for systems
to intercommunicate without a long and laborious establishment of a translation between
the, most likely incompatible, protocols on the differing systems. However, as will be dis-
cussed, such standards can also pose an impediment to the evolution and development of
long-lived systems.

The sheer ubiquity of the modern internet largely obscures the very large number of
protocols that are allowing the transfer of information on a global scale, from a variety of
vendors and with the underlying assumption that any computer that is connected to the net-
work will be capable of communicating on the network. These protocols are found at the
software level, for the encoding and encapsulation of data for transmission through nodes
and networks, and at the hardware level where signalling, routing and administration pro-
tocols provide the physical mechanism to transport the software-manipulated information.

Policies are less formally structured and are not usually encoded as part of hardware
specifications, or in the firmware of hardware devices. Protocols are clear statements of the
steps that must be taken and the ways in which the steps must be defined, where policies
are more concerned with actions that should take place to provide certain facilities within a
system, rather than dealing with the existence of the system itself. A protocol is a normative component of a standard, where a policy is a descriptive component of the standard. In explanation of this, a protocol must be true for the system to exist where a policy may describe a state of affairs that can be found to exist within the system.

For example, a home internet user has a computer with a network card that supports the 802.3 Ethernet standard. This ensures that the network connection between the computer and the user’s ADSL modem will operate correctly, as both card and modem support the 802.3 network protocol. There is also a TCP/IP protocol stack associated with the software driver for the network card so that, if packets arrive at the network card, any packets can be decoded correctly and passed to the correct application. All of this is protocol-focused. The arrangement that the home user has with the ADSL provider is based upon policies, after the initial provisioning of the service. The ADSL provider has agreed to provide the user with an ADSL service, over certain protocols, and, in exchange for an agreement to provide funds, provisions or arranges the provisioning of an ADSL service over the copper wire from the exchange to the user’s dwelling or place of business. The type of service that is then provided is policy-based. The ADSL provider will provide so many bytes/second with a maximum download of X bytes. This is a policy-based agreement and does not affect the existence of the ADSL link or the nature that communication takes. If a user exceeds their maximum download, it is a policy-based decision that then throttles their link to a slower speed or records additional charges for the additional data download.

In this case, a user has limited freedom to negotiate the terms of their agreement as most providers offer a set of options and the user picks the one which best suits their needs. Some distributed systems only offer this style of policy negotiation, where others offer a much wider range of options, potentially over a continuum rather than at discrete points. However, the policies make assumptions about the function of the system and this, in turn, assumes that the protocol issues have been resolved.

Returning to the ISP example, a user may decide to use a Frame Relay or ISDN service to their home instead of PPPoE over ADSL. This is a protocol decision and requires the user and provider to agree upon using a protocol other than PPPoE over ADSL. The range of packages offered for the new service, the policy, would then be conditional upon the new protocol being available to the user and provider, and that it can be successfully established between the two parties.

Returning to the context of network-based information systems, the underlying protocols start with the traditional network stack and are then built upon to provide an NBIS-specific stack that provide the basis for guaranteed interoperation. In order to make good use of the underlying system, policies are then developed to provide users and information
providers with a robust and useful system. In the case of the DNS, DNS message format specifies the protocol over which DNS messages can be exchanged but policies dictate how caching and secondary domain name server arrangements are to be made. Policies also dictate the structure of the domains and sub-domains while the behaviour of queries, and which servers are accessed, are specified through strict lookup protocols outlined in the original specification documents [3, 4].

Having established these protocols and policies, it is possible for nodes to join and leave the system, within system-defined security constraints, without having to establish any other arrangement with the existing participants except that which is defined in the protocols and policy documents. However, as mentioned in the early description of protocols and policies in Sections 2.3.2.2 and 3.1.5, this is also a limitation depending on how the non-standard behaviour is handled and defined. If any non-standard activity is ignored, then there is no capacity for a standards-based system to change its behaviour without causing a global change that reflects the new standard. There is no iterative or incremental pathway to allow a change in system-wide behaviour. The local effect of this policy is that, to guarantee standardised interoperation, an NBIS that is rigidly controlled by protocols and policies can not allow local variation that interferes with global operation. However, this does not prevent local variations that do not impact upon the system-wide standards. This is an important theme that will be revisited throughout this thesis: the benefits of standardisation can be enjoyed while still allowing local variations that do not directly compete with the standardised mechanisms. The means by that this can be performed, however, requires more rigour in the system definition than a straightforward adherence to standardised protocols and policies.

3.2 Summary

In this chapter, the major challenges facing any distributed system have been addressed within the context of the network-based information system. We have done so in order to introduce the key concepts that are used to provide a design for a standards-conforming network-based information system. Such a system also allows localised extensions outside of the standards framework and, as a result, can allow system evolution and co-development while still maintaining global system functionality. As will be shown later, the mechanisms used to provide this versatility exist and are sufficiently mature to allow the description and implementation of such a system.

The remainder of this thesis should be read in the context of the key points identified in this chapter. It is not sufficient to describe large-scale network-spanning applications in
terms of the algorithms that they implement to manipulate the data that they contain. It is also essential to consider the environment in which they execute, the network and storage resources that they consume and the significant impact of protocols and policies on these large-scale systems. The adaptability of an existing system to a new application can be greatly enhanced by allowing a more flexible policy scheme in the delivery of its services or, from a protocol layer, by allowing access over different protocols. If we accept that the core of a system is its algorithms and data transformation, and that its implementation is heavily influenced by its environment, it is a logical extension to identify aspects of the environment and separate them from core functionality while providing a mechanism to link the operating core into its environment.

The next step taken is the development of the methodology that is then used to derive the implementation of the SAMPAN system. In the next chapter, we extend and develop the design, as well as introducing a high-level conceptual model for the system.
Chapter 4

Methodology

Our goal is to provide a novel approach for implementing network-based information systems in a way that will allow their ongoing use in a standards-based environment while also allowing them to be agile and responsive to local requirements. The next three chapters will introduce, develop and then provide a conceptual model for our methodology. Our approach is based on the use of ontologies and annotation to provide a strong contextual framework for all data employed in the system, regardless of level. By wrapping data with context, we can utilise this new knowledge to establish exactly which operations are required or legal in the framework of the NBIS that we implement.

This chapter describes the underlying approach to the implementation of NBIS that we employed in the development of the final implementation of the software and motivates the work upon which we build the detailed aspects of the design. This chapter is divided into two major sections. These are:

1. Developing our new approach to implementing network-based information systems and discussing, as a pre-cursor, how a distributed information system may be modelled. This section also introduces a three-tier software architecture that is used as the basis for the development of network-based information system implementations that are standards-compliant and extensible, through the use of ontologies.

2. A discussion of the nature of the knowledge contained in a distributed system and the metadata and storage requirements for such a system. The implicit overheads of a knowledge-based approach are then considered.

This then leads to two subsequent chapters that discuss the application of the methodology. Chapter 5 contains the discussion of suitable target systems and a comparison with other approaches. Chapter 6 discusses the high-level conceptual model of the system and
definition of the nature of queries inside such a system, as well as the requirements for information storage.

4.1 Developing a New Approach to Implementing NBIS

A representation of a system in terms of the knowledge domains that it spans or contains is an idealised and abstract representation. There are a number of steps that must be taken to ensure that the system is correctly described and the system and relationships between system elements must be well defined. However, as will be discussed, it is not enough to describe the system in a high-level representation, such as an ontology language, as it must then be interpreted and implemented. This has led us to design a three-tier software architecture for supporting a knowledge-domain based approach, consisting of the ontological model, the data model and the programming language semantics underpinning both.

An obvious advantage of such an approach is that it allows an older, non-annotated system to be replicated inside a system that can support annotation provided the old knowledge domain can be nested within the new. We will discuss the benefits of representing systems as knowledge domains in more detail later in this chapter. To be effective, all entities within the semantic grid and semantic web must have meaningful metadata that can be used with ontologies to provide the greatest dissemination of captured knowledge both as it applies to the stored data and to using the system’s resources [100]. Thus, instead of producing a new DNS service for the Grid [99] or Semantic Grid, the existing DNS could be made available through the new approach, with an annotated data stream.

In this chapter, the three-tier software architecture will be introduced and discussed in detail, leading to the required basis for implementation and evaluation. An initial discussion of knowledge representation and context is presented, as background to a discussion of our three-tiered software architecture used to implement ontologically described network-based information systems. We then present a discussion of the knowledge storage requirements and mechanisms that can be used to achieve the implementation. We then compare our approach with other existing approaches and identify how our solution differs. First, however, our approach should be defined in terms of existing technologies in order to ground the approach in well-known approaches. The final sections of the chapter focus on the development of the conceptual model, which we used both for implementation and as the formal basis for our use of RDF and OWL, and the use of entailment, instead of DL-like subsumption, to determine decidability.
4.1.1 Knowledge Representation

As discussed previously, to be a logic, any representation of knowledge must provide a vocabulary, syntax, semantics and rules of inference. There are basic requirements that must be met when storing knowledge, which are additional to any vocabulary or syntax requirements, as the assumption is that the knowledge will be re-used at some stage, and by users other than the original generator of the knowledge. While syntax and vocabulary can be reinterpreted with a high probability of a successful interpretation, lost semantics are much harder to replace unless the interpreter has all of the context under which the original interpretation was established.

With this in mind, there are fundamental questions raised for a knowledge representation:

1. How will the knowledge be represented?
2. How will the representation be stored?
3. How can users be guaranteed that they are sharing context and, hence, meaning?
4. How will the representation be made available?
5. What mechanisms can be used to realise the knowledge stored in the representation?

The focus of this research is on large, widely-distributed network-based services that act as information repositories. A distinction is made here between large network-based systems that function predominantly as a distributed system for performing a set of computations over existing datasets to produce a solution and those network-based systems that act first as distributed data stores, utilising some computations to prepare or manipulate data in response to user requests. The latter form we identify as network-based information systems: these are our focus for this thesis. The potential remoteness of client and server encourages a model where the representation can either be safely distributed, remotely cached or is otherwise highly portable. This prevents bottlenecks leading to and from a single point of consultation via a representation browser sited on the same node as the representation store. The term store is used here to denote a persistent repository of information, ordered in such a way that storage, searching and retrieval can be carried out in bounded finite time.
4.1.1.1 Mechanisms for Representing Knowledge

The nature of the knowledge itself must also be analysed to provide answers to the questions posed in the previous section, especially with regard to the representation of knowledge. Although XML-based technologies are widely used to provide structure and context for data, XML does not scale well when used with binary data due to the potential need to encode individual bits. Binary XML implementations do exist, such as BinX [114] and XML-Binary Optimised Packaging [115], but this focuses on describing the data in ways that allow cross-network access and only annotating slices of the data as they are required. Several knowledge representation technologies have alternative, non-XML-based, implementations that could be used instead but the overhead of adding many bytes of information for each bit would still constitute a significant increase in required storage. Knowledge representation, where metadata is added to ensure portability or to provide explicit realisation of a taxonomy, can add a substantial number of bytes to existing data files, binary or not.

While the representation technology used and, to a lesser extent, the storage technology selected depend upon the nature of the knowledge and the size of the stored data, the remaining questions depend heavily upon the application domain of the knowledge and require a detailed understanding of user requirements. Different users have different requirements of a knowledge representation, ranging from simple presentation issues to a requirement for complex structures capturing higher-level concepts. There are similar issues with the provision mechanism and realisation.

Therefore, a clear user requirements statement is necessary to assist with answering the questions introduced at the start of section 4.1.1. The users of large-scale, widely-distributed network-based information systems have an implicit expectation that they can access the stores of the system through a mediating interface to allow reading, and possibly writing, of stored information. In legacy systems, these requests are made in well-defined highly structured formats through well-known ports. TCP/IP is used for transport and the content of the payload makes up the entire message. In more contemporary systems, these requests are often made by using system-defined ports and a well-known transport mechanism on top of TCP/IP or UDP/IP, encapsulating an XML-based message. In this case the TCP/IP payload contains a transport wrapper and then the final message.

4.1.1.2 Syntactic and Semantic Elements of Knowledge Representation Mechanisms

Users of distributed information systems need to be able to refer to those items that are stored within the store and to be able to differentiate the elements of the messages that are used to pass information to and from the store, where the messages are part of the system
semantics. To clarify this point, there are messages and operations that may be used by the system but they do not cause a change in the knowledge contained in the system or, more usually, are not explicitly provided for system use. These include a large number of operational messages used in a network to provide the transport framework, such as routing information, keep-alives, retransmissions and acknowledgements. Although these may be used, indirectly, by a distributed system they are not part of the system semantics. The system’s semantics are usually grounded on a black-box model of a network that meets certain behavioural characteristics. Exact routing metrics and routes and the mechanics of the underlying protocol are not taken into account when executing transactions on the store or sending and receiving transactional information.

A message sent from a user to a distributed information system can take one of three forms. It is a request for information, a request to store data or it is a request for storage metadata, such as the number of records, last item stored, number of connected users or similar. All of these messages, and their contents, are considered part of the system semantics since this information can cause changes in the store or be used by the store to construct and transmit an answer. It is in the nature of a distributed information system that requests for information, requests for storage and operations in the store themselves must all deal with the information that is considered important for the system. A user cannot request information that is not present, or not understood to be present, within the system and obtain a valid response. As will be discussed in Section 6.2.1, we can logically represent any form of query message to a system as a simple request to fill in the elements omitted from a request containing some specified elements - that is, there exists a syntactic rewriting that maintains semantic sense. Such a request is simply a request to map the message into the store and bind any missing variables to a set of values that match the existing bindings in the message. A query is satisfied by binding values to unbound variables in the message, within the context of the bound variables already present in the message.

The system semantics, supported by the syntactic elements over which such semantics are valid, must be encoded so that they accurately represent the knowledge of the system and, in doing so, must meet a requirement that they be perceived as the same knowledge throughout the system. Thus, the encoding scheme must be reliable, robust, predictable and contextually-rich. This motivates the use of ontologies to provide a strong description of the knowledge in the system as an ontology language can provide the strict hierarchical relationships necessary to provide ontological description [86], as well as logical relationships to show the relationship between entities. The domain of knowledge that spans the system must be encoded as the smallest span that covers the system semantics - eliminating those messages and external storage items that are not key to the system, but that may
merely be exploited by it. If the knowledge hierarchy is sufficiently well-defined then the context of any given piece of information is firmly fixed and, once a human applies meaning to the contextual framework, the meaning of a piece of information can be determined both within and without the system. It is also essential that the knowledge can be referred to in such a way that the contextual hierarchy can be established without having to recreate the entire hierarchy to provide the context.

4.1.1.3 High-Availability and Remote Access Mechanisms

Due to the highly-distributed nature of the system, and hence the user base, the availability mechanism that is employed for the system must allow remote access and potentially allow a component-based approach [12] to allow distinct components of the system to define their areas of knowledge. A component-based approach is one in which a system is strictly categorised into a set of separate, interacting components that all have a well-defined, and components-specific, function. This representation of system components can then be extended, by synthesis, to a ‘whole-system’ representation through the use of sharing and inclusion mechanisms. Thus, the availability mechanisms must support a notion of inheritance, publishing and remote referencing where qualified names can be used to refer to resources not contained in the local component of the representation.

Finally, however the knowledge is represented, there must exist some mechanism for machine interpretation of the representation. This is not just limited to expressing what is explicitly defined but also may require logical inference. It may require the use of reasoning engines, in conjunction with the logical constructs, to establish what constitutes a valid extension of the system within the reasoning framework, constrained by the expressions defined in the representation. Reasoners normally place constraints on the expressiveness of the knowledge representation, as certain aspects of knowledge representation prevent reasoning from terminating through the introduction of undecidable aspects. Since realising the knowledge requires an interpretation of what is explicitly stated, as well as the possible determination of what is implied, an architecture will be proposed in this chapter that provides a firm basis for both requirements.

4.1.2 Context

It is critical that the exact definition of the term context inside this thesis is established as this is a concept that is critical to the development of our approach. The context of a value, or data element, is used to determine how to interpret it. In Computer Science, context is generally defined as the minimal set of data that must be stored in order to allow
the interruption and resumption of a task. We extend this definition to a knowledge level by requiring that the minimal set of data saved include all of the required knowledge that allows the data to be interpreted as its creator intended. Thus, rather than providing enough for a simple context switch at a processor level, our view of context is that there is enough stored knowledge to allow the ongoing correct interpretation of the data value, at whichever stage it is being used.

There are many different levels at which any form of data can be interpreted and, rather than functioning as a set of possible and distinct choices, there are often overlaps between different interpretative layers. While this can sometimes contribute a very subtle range of nuances to an interpretation, in many situations it leads to confusion. Such confusion is not desirable in a distributed computing environment as possible shades of meaning lead to ambiguity and ambiguity must be resolved in order to establish the valid interpretation.

When determining the context of a value in a computing framework, there are several layers that are used to determine the final interpretation of the value. The first is the composition of the value in terms of its bit pattern. This is, however, useless without a defined interpretation applied in the programming language that receives the bit pattern as there are many different ways of interpreting a given bit stream. However, no interpretation would convey a meaning to the program. The value, as it was originally transmitted, has a meaning to the entity that initiated its transmission but only if the entity responsible for transmission is capable of comprehending the meaning. Consider the sound files encrypted and transmitted through the digital GSM telephone network. These individual packets are decrypted, decompressed and played through the speaker of the receiving telephone. Despite the representation of the value being well-known to both telephones engaged in the process, neither device can divine the meaning of the text carried by the packets. The message is from a human to a human.

This displays the key difficulty in discussions of context in a computing environment. There are interpretative contexts that are used to move values into a well-defined form to allow their manipulation in a programming environment. Above this is the interpretative context that provides a meaning to allow a reasoning entity to use this programmatically well-interpreted value in the correct context to maintain its meaning. The programming language semantics of a given programming language define the interpretative context of all legal statements in the language - these must be defined or the compilation of the statements is significantly harder, or sometimes impossible. However, the higher-level context that establishes meaning is usually implicitly defined and is interpreted by reading the source code to determine which roles were being taken by the values. If the domain of discourse is sufficiently well known, the roles can be reverse-engineered from the programming language.
semantics. In the following sections, the context of a knowledge representation refers to the complete conceptual framework to establish context, unless otherwise explicitly stated. Because knowledge requires a shared, high-level, context, it must be built upon a well-defined lower-level, programming language semantics, context. This will be illustrated further, with examples, in this chapter.

4.1.3 Data, Information and Knowledge

To establish a basic understanding of the importance of contextual information in modelling, and hence how modelling would be carried out in a system, it is necessary to establish how data is classified into the contextually-defined terms: data, information and knowledge. We have already defined the terms data, information and knowledge in Chapter 2, in Section 2.1.1.

As will be shown later, it is possible to place information in context without a domain explosion if the correct tools are used in the assembly process. This does not change the underlying fact that the pure representation of a knowledge domain is unfettered by implementation. It merely establishes that the implementation process for a knowledge-based solution starts well before any code is written and, in fact, starts once the human mind starts organising the problem in order to solve it.

Regardless of how much context has been associated with it, once data has been successfully transferred it must be stored so that further computation or the service of the information can take place. Traditionally, a distributed system would store the data for computation and discard it once no longer required. An NBIS may retain data for a long term as its primary role is the storage and re-presentation of data to clients. If the data is context-rich, and hence knowledge, there are increased benefits in increasing its longevity to reduce the contextual association burden for the same information in the future and also, because it can be used for decision making, keep it available for as long as it is relevant and decisions can be made from it. Because of this, it is important to look at storage requirements and patterns in standalone and distributed systems.

We can now place these definitions in the context of a knowledge representation for a distributed system. Bytes traversing the network are a valid component of the system, in potentia, but cannot be used to answer questions within the problem space. There are two reasons for this. The first is that the byte stream has not been decomposed to provide individual components. The second is that, until the bytes have completed transmission, there is no guarantee that the entire message has been transmitted. Since no questions can be answered, these network-traversing byte streams are classified as data in our model.
Once a complete network stream has been received, it can be decomposed to form a set of individual values. This assumes that the structure of the data stream is well-known between client and server, a basic assumption of any functioning distributed system. The encode/decode information is traditionally hard-coded into the system and, in many systems, the protocol used for transmission over the network (wire transmission) is dependent upon the network port used. For example, Hyper Text Transfer Protocol (HTTP) is expected on port 80, while Simple Mail Transfer Protocol (SMTP) is expected on port 25. Elements resulting from the decomposition of the data stream must have an associated type as, without this type information, the stream cannot be decomposed. This is due to the relationship between the structural size of the type and the number of bytes it will occupy in the data stream. At this point, typed values are available to the system so simple questions such as "What is the value of the third integer in the received stream" can be answered. However, despite being able to answer simple physical questions, it is still not possible to answer objective questions within problem space as these typed data, which are now referred to as information, have no context.

Recall that the programming language context of the typed decomposition of a data stream is given by the semantics of the programming language code and the underlying programming language semantics of the implementing language. At some point, if an integer has been received from the network it is placed into the knowledge representation context through the use of programming language operations. In order to determine if two items are contextually identical, or are in some form of logical relationship with each other, then the programming language constructs for both items must be correctly managed to ensure that, at the correct point in the code, the two integers are presented within the same context.

Once an informational element has been placed in context, and can be used to answer objective questions within the system, this final form is the knowledge contained in the original datum. Many systems do not achieve an explicit knowledge form for most of the data in the system, at least not in a way that allows arbitrary queries with the existing source code. While all data is usually used to form information, not all information goes on to become knowledge. Sometimes this is due to the information being ephemeral or overhead that is not related to core system functions, much as a large amount of information in the TCP/IP protocol is unnecessary in the context of the semantic footprint of a distributed system. However, large sections of information can be used without moving to an explicit knowledge form and still produce a final result that appears to represent knowledge. In situations such as this, the information is being implicitly transformed to a knowledge form in conjunction with other system elements and transformations but it does so with a loss of
The logical boundary of the ontologically-defined NBIS node

Three tier model to support ontological definition

Ontological Specification (Knowledge)

Data Model (Information and model grounding)

Programming Language Semantics (Model grounding)

Store (Knowledge)

Network Input/Output (Data)

Figure 4.1: Block structure of an ontologically-defined NBIS node

generality.

4.1.4 A three-tiered software architecture

We propose a three-tiered software architecture in order capture the data, information and knowledge in a system and also implement the system in such a way that the context of important system concepts is well-established. The three tiers of the model are: an ontological layer, the data model layer and the programming language semantics of the implementing language. While this model is used extensively in all programming activities, it is rarely referred to explicitly by programmers in any sphere; programmers implicitly use this model every time that they write a program. What is important here is which aspects of a knowledge-representation-based system are mapped into which tier of the system and why this is the best mapping. The model also clarifies the discussion on the representation of operating systems and distributed systems from Chapter 2, as well as providing necessary detail for the discussion of implementation. The outline of the model, developed as part of the research for this thesis, is shown as Figure 4.1.
4.1.4.1 Ontological Specification Tier

The ontological specification tier captures all information on fully-formed knowledge and is used to store to and retrieve from the store. The ontological specification tier is the only layer that deals with knowledge and, because it controls access to the store, effectively prohibits the use of data or information in store-based operations. It is, therefore, impossible to answer a query or store information unless it has been represented as knowledge. In this tier, information can be referred to only in a strict contextual framework. If the framework is shared, and the same human interpretation is applied to the framework, then meaning can be shared between client/server, client/client, server/server or between nodes of different systems that can use the same framework. The ontological specification contains the pure knowledge-domain based representation of the system. Ideally, a change in the knowledge of the system causes a corresponding change in this tier and this will change the logical relationship between well-defined elements, the data structures in use or the operational semantics of the system. The link between the ontology and the implementation of the system is a concrete realisation of the abstract ontology that defines what this system is.

4.1.4.2 Data Model Tier

The data model tier specifies the translations required to turn data into fully-formed knowledge by using data streams to move data into and out of the ontological model. Any mechanism that reads data for the server from the network is part of the data model - it uses ontologically defined protocol specifications, from the higher layer, to identify stream type and unpack the data into semantically-aligned components. The data model is the first half of the grounding of the ontological model.

Network packets are used to convey bytes from client to server and vice versa. It is impossible to determine the nature of a given stream of bytes until the complete message has been received and assembled. To allow for the explicit assembly of the network-originated byte stream prior to it being classified within the ontological hierarchy, the data and information handling facilities of the model are combined in a tier where data entering the system is decomposed to information and then represented as knowledge. This allows the formation of a single message from network bytes, which can then be decomposed to individual instances of types, and then associated with the ontological hierarchy.

The data model also describes the set of functions that provide the low level services required to make the ontologically-modelled system function correctly. OWL is not a programming language so it is not possible to write, for example, a UDP server in OWL. While the functions required to implement a UDP server could be encoded in $\lambda$-calculus
and embedded into an OWL representation, the overheads are significant. As no commercial processors exist that use the $\lambda$-calculus as their machine code, there must be a piece of software that can interpret the $\lambda$-calculus and develop the rest of the system from this level at some stage. The requirement to translate calculus to executable code leads to a boot-strapping issue where sufficient system definitions must exist to allow the lambda representation to be read in initially. As there must be a distinction between the ontology layer and the underlying grounding, the actual point of separation is irrelevant providing the fully-qualified knowledge is captured completely in the ontology.

The data model also handles information that never becomes knowledge in the sense of the local server. Such data and information include, for example, port numbers of querying hosts, timing information and any other ephemeral data. All data entered via the system’s defined access mechanisms is ephemeral until it has been strictly classified within the ontology - at the point at which it can be compared with the knowledge in the store or placed in the store it is semantically aligned. As mentioned previously, we can logically represent any form of query message to a system as a simple request to fill in missing form elements. Putting this into the current framework, there must exist a data model tier syntactic rewriting that maintains semantic sense but, by rewriting the information as knowledge and passing it to the ontological tier, is simply used to map knowledge into and out of the store.

### 4.1.4.3 Programming Language Semantics Tier

The second half of the grounding comes in the forms of the programming language semantics of the implementing language, the third and lowest tier of the software architecture. The implementing language is the (interpreted or compiled) language that directly addresses the operating system level functions. The grounding maps higher-level concepts into their programming language equivalents. These are, in turn, interpreted or compiled into a form that can then be executed by the processor, for example, the use of Python `int()` in place of OWL/XML `&xsd;unsignedShort`, and also the limited replacement of certain lambda expressions with Python lambda equivalents.\(^1\) An ontologically based representation is impossible without this explicit, pragmatic grounding as the stored knowledge cannot be realised or acted upon.

Compare this mapping with the previously discussed example of a programmer writing a program. The programmer will have a domain of interest that contains the problem to be solved or addressed by writing a program. At this point, there would usually be an informal representation of the knowledge of the system although, strictly, there would be

\(^1\)Python only allows simple lambda expressions as the indentation-based block statements prevent the single-line restricted lambda expressions from having a fully expressive vocabulary.
an associated ontological representation in the mind of the programmer. The grounding of the problem into a concrete realisation would take place predominantly at the data model and programming language semantics level as very few, if any, of the variables used in programming would have sufficient context to be arbitrarily re-used without having to establish semantic context by code examination. The code examination, in turn, depends on a thorough understanding of the mechanics of the programming language. For the programmer, the top tier exists but is never associated with the program in a machine-interpretable or exploitable manner. The lower two tiers make up the majority of most programming efforts.

4.1.4.4 Encapsulation in the software architecture

As a value passes through the software architecture, it is further defined and classified until it is accepted as a representation of knowledge. As can be seen in Figure 4.2, looking at the the three-tier software architecture in a ‘top-down’ fashion shows the encapsulation associated with each tier as it applies to values and as it applies to programming language semantics.

At the core are the programming language semantics of the implementation language. These provide ‘the toolbox’ from which all higher level operations are supported. There are no values entering the model at this point - values enter the model in the lower levels of the Data Model layer. Data entering the lower level of the data model is processed to become information and, as the data supports type and a context in the programming language semantics of the underlying system, the operations available for, say, determining equality move from a structurally naive byte-wise comparison to a structurally aware type-based equality mechanism.

An example of this, in the context of shoe sizes, would be to implement an equality relationship that could establish that the European size 45 was equivalent to the US size 11. Much as additional language tags can be added to metadata to provide tag names and descriptions in different languages, the labelling of data with type and context can clarify the relationships between two equivalent pieces of information that appear different at first inspection.

Once the information becomes knowledge, there is enough context that comparison and equality can be evaluated by looking at logical class membership, attributes and properties. The increasing context placed upon the value allows for a corresponding increase in the sophistication of the operators that can be used upon that value and also an increase in certainty that the correct operation is being applied to the correct value.
4.2 The Nature of the Knowledge Contained in a System

In this section, we use our understanding of the new software architecture as a background for a discussion of what happens in a traditional programming environment. We use this to illustrate how we would model a traditional system ontologically and motivate how we will apply our methodology to produce our implementation.

4.2.1 Contextual Inference in Traditional Programming Systems

The lack of context for individual informational elements in traditional programming forces programmers wishing to reuse or adapt existing software and systems to look elsewhere for the context. If the source code is available then comments, variable names and clear flow-of-control can be used to establish a system semantics-based context for elements. Code reuse is an increasingly important mechanism for reducing software development risks and
costs - it is exceedingly important in maintaining the viability of long-lived distributed systems as clever reuse can use the existing elements already deployed across the network to accommodate new scenarios unforeseen at the original point of deployment.

Often, contextual placement is established for larger code elements than single pieces of information. That is, functions and classes are given a context, rather than just individual elements. These larger aggregations often include elements that are small highly-contextualised informational elements, such as the value PI in the Java Math library, accessed as Math.PI from a constant static field. There is, however, no machine-interpretable aspect associated with these shared items, their context stems from human-readable documentation and a suitable choice of names.

Thus, to extend the capabilities of an existing widely-distributed network-based system, the programmer would have to either have access to the source code or would have to write additional modules using a black-box approach based on the advertised interfaces of modules. The first approach requires a detailed code inspection in order to establish the correct contextual framework for the program and then to determine that changes have to be made to accommodate the new requirements. The second approach requires that the existing interfaces be well documented, correctly implemented and provide the desired services. In both cases, there is no logical way to verify that changes made to the system will provide, at worst, the same system functionality as was originally developed. That is, that the set of correct system functions after the change are a non-strict superset of the set of correct system functions before the change. The term ‘correct system functions’ is used to refer to system capabilities that perform as intended, thereby excluding the need to preserve badly-behaved code between iterations.

For small-scale systems, and short-lived systems, the requirement that the next generation includes the correct functions of the previous iteration is important but it is not going to affect a large number of users if there is a problem. Large-scale, long-lived systems rely on a standards-based approach because it allows the deployment of updated components over time without compromising the function of the system. In order to correctly realise that standards-based approach, any changes that add features must not remove features that may be expected by other nodes until such time as all nodes have been advised that the feature is going to be removed. (As an aside, ontology languages can include deprecation advertisement features to include an explicit statement in the knowledge representation that would otherwise be a textual annotation in a man page.)

By explicitly requiring an ontology specification tier in the model, the implicit and ad hoc nature of contextual inference usually found in traditional programming systems is replaced with a formalised representation of the knowledge domain describing the system.
This can be used to clearly identify system components and, as will be shown, describe system operation in order to allow refactoring and alteration of system operation. However, by using ontological reasoning, the alterations can also be checked to ensure that they do not impair the core operation of the system.

4.2.2 What constitutes the knowledge of a target system?

As the ontological layer has the only access to the store, and hence only fully-qualified knowledge can be stored, we now explicitly specify what constitutes the knowledge of the system. We have discussed some aspects of the knowledge of a system earlier, in Section 4.1.1. A more complete presentation follows.

The knowledge of the system is defined, informally, as the following:

- The set of variables that correspond to the stored data of the system and persist from one transaction to the next.
- The logical relationships between these variables.
- The data representations of the variables.
- The relationships between data representations.
- The operational semantics detailing the legitimate transformations of data within the system.
- The relationships between semantic operations.
- The results of transformations that are required to persist. This spans the contextually-rich data that is generated from existing contextually-rich data.
- Those components of an externally-derived message that align with the set of variables outlined in the first point.

The majority of the definition is self-explanatory: the knowledge capture must include those elements of the system that persist and it must also clearly define potentially intangible elements so that they may be used. The requirement that at least portions of an externally-derived message must contain knowledge, or data that can become knowledge, reflects the requirement that only knowledge may be applied to the store. If an incoming message contains no knowledge, then it cannot be placed in or compared with the store as there is insufficient context to establish how this should occur.
Knowledge generated from the results of transformations is not always required to be stored. Consider a system where queries are made on a central data store and the responses are sent, in a format from the format used for the query, back to the requester. If such a response is stored, in its new form, at this site or at another remote site, the knowledge is being cached. It is not always necessary to cache the output of a system, especially if the resulting information is very short-lived or can be trivially regenerated internally. If the production mechanism has large overheads or if the response has a long valid lifetime then caching is a logical solution to minimise unnecessary computation.

If generated knowledge, with its rich contextual framework, is cached then the expiry of cache data can be enhanced by taking advantage of the framework. If a system node is refactored through ontological manipulation, the extent of the modification is immediately apparent on restart and only those items that are affected by the change need to be considered for expiry or update. For example, if a certain subclass of operations are modified, then only those cached responses that contain items denoting that subclass need to be examined to see if they are still valid. Such records could be dumped from the cache using a simple query or, alternatively, the incorrect item could be replaced with an up-to-date and accurate item.

The placement of cached elements within the model depends upon the level at which the elements are cached. There could be a data level cache that stores network data as it arrives, as complete messages. There could also be an information level cache in the upper tier of the data model that stores typed values, but without context. The assumption made in the remainder of this thesis is that the knowledge and ontology tier is the provider of cached information and, as a result, the knowledge store is the origin of any elements that enter a cache. As will be discussed, there are significant benefits to storing annotated knowledge-level elements for caching, despite the additional storage overheads.

4.2.3 Modelling a system ontologically

This section describes the methodology required to construct an ontological model of a network-based information system. We have already discussed the multiple tiers of the software architecture that are required to support an ontological definition, but now we discuss the nature of the ontology.

Any program can be regarded as a set of symbols, which have a syntactic and semantic relationship to the algorithm to which they correspond. However, there are implicit operational semantics, associated with the symbolic semantics for each element, which correspond to the execution of the program. When a traditional parse tree is created and
Figure 4.3: The three branch ontology

annotated, it is the way in which it is traversed that determines the final outcome of the translation of the syntactic symbols that comprise the program. Thus, a program can be viewed as a set of elements that have a strict representation, to allow for syntactic matching, and a valid set of logical compositions, to allow for semantic matching, and also a set of operations that determine the way in which these elements are composed to execute the program - a set of operational semantics. A complex system of programs can also be represented in this way by using an ontology that describes the representational, logical and operational semantic aspects of the system. This provides a three-branch ontology that is supported by the three-tier software architecture outlined previously. Section 7.1 develops these ideas further to provide the implementation of the ontology used in the final software.

The descriptive ontology is divided into separate, inter-related branches, as shown in Figure 4.3.

Thus, to capture a system ontologically, the knowledge elements of the system, along with their logical relationships and transformations, must be identified in order to be placed into the correct branch of the ontology and, as described in Section 7.1, then correctly classified, related to other entities and described within the operational semantics framework. However, if we consider the original program to represent a container for syntactic and semantic elements, this new knowledge-based representation also requires a logical container. In the abstract case, this motivates a form of store that can be used to store and manipulate knowledge elements. This corresponds to the knowledge store in the three-tier software architecture. The representations from the ontological branch are grounded through the knowledge store through the semantics of the data model and, in turn, the un-
derlying programmatic implementation. The operational semantics are fully described in the operational semantics branch of the ontology, which is interpreted from the store, but many aspects are implemented in the underlying data model as fundamental functions.

4.2.4 An example of ontological capture in the software architecture

We provide an example application of the architecture to foreshadow the development of concepts in this chapter, as well as Chapter 7, and to provide a conceptual framework for the reader.

As an example of the application of this technique, consider a business that uses temperature sensors to measure the temperature of key machines. There are also a set of responses that must be carried out in response to changes in temperature.

The concepts that define temperature and the actions taken are defined at the ontological level, as these have a knowledge value. The ontology contains elements that contain the values for temperature and actions, as well as the information describing the structure of these elements and the relationship between them in terms of predicates. The FunctionalBranch of the ontology also allows us to specify how these actions are taken and provide the control mechanism that allows the choice of actions at a given point.

The underlying three-tier software architecture utilises a base language that implements the data model providing the interfaces to the physical temperature measurement sensors and any output devices. Any actions that are regularly carried out by the system may also be encoded as executable code modules inside the data layer. However, these data level resources can only be used - and input be acted upon to produce output - through the ontological layer. There is, throughout this system, extensive use of metadata to describe values, and the subsequent association of metadata with other values and metadata. It is worth clarifying that the term metadata can refer to those additional annotations that can be associated with elements inside a system purely for the use of the system, and can also refer to the metadata used to annotate the information products of the system. Where confusion could arise, internal metadata is the term to describe metadata used by the system and external metadata is used to refer to the more traditional form of metadata, such as that which is associated with provenance, e-Science or Web Services.

Such a metadata-rich system, of either type, must have a very well-defined level of metadata and must be sufficient to represent the system without being so complex that it overwhelms an interpreter or lead to very bad performance. This is discussed in the next section.
4.2.5 Choosing the correct level of metadata

Large sections of the critical data in programs consist of numbers, which are relatively small in terms of byte storage. There are also strings, of varying lengths depending on the implementing system, and custom record and class formats that, when marshalled, can be much, much larger. The metadata required for any element to be considered knowledge must include a formal statement of the type of the value and a strict placement of the value within the contextual framework.

Modelling the knowledge-information-data relationships in terms of classes, every value must have membership in a class of items with its type and it must also be a member of a class that denotes its position in the hierarchy. The class of which it is a member will either be a subclass of a more general class or it will be the top of the ontology. Ultimately, all values, once qualified, can be referred to from the top of the ontology - the ontological root. The design of the ontology should provide transformation mechanisms, internally or externally, that allow the transformation of one value to another where a well-defined mapping exists between the two types. This does not guarantee that a transformation exists but, where it does, the ontology should provide the transformation.

The volume of metadata produced for a single value can be very large as a complete class framework has to be established, along with associated properties, in order to correctly express and define the value as data, information and knowledge. As an illustration, an early version of the DNS ontology used for early experiments was over a thousand lines long and defined twelve pieces of information. Approximately nine hundred and fifty lines were required to define the context of the data that was used. An individual data item can have up to ten lines for each entry, depending on how much of the information is new and which approach is taken in encoding. While this appears to be substantial overhead, the addition of more data does not require the addition of more context and the more data that is added, the more that this context is shared between items. In addition, many data items can be added with a relatively small number of lines. It is also important to remember that all of the context and associated metadata increases our ability to utilise and share the data. Rather than having a large and bulky way of representing the same service, we incur additional storage overheads in order to provide a better version of the system.

All of this metadata is strongly tied to the data in such a way that it can be used in queries and reasoning. We use an ontology language for the purposes of reasoning and a graph-based store will be used to store and provide the annotated data in a way that suits the model and does not compromise the knowledge-based approach.
4.2.5.1 The nature of the underlying store

The underlying store must support the storage of highly-annotated knowledge and, ideally, it should prevent access that would circumvent the model. Characteristics of such a store include facilities to strongly associate the elements with well-defined relationships as predicates to facilitate a graph-based conceptualisation of the stored knowledge. In the ideal case, there should be a mechanism for walking the store to extract information except by using the stored knowledge and the knowledge/context framework.

The knowledge stored can also refer to items of varying types. This prevents the direct use of a fixed-format table based approach, including those found in traditional RDBMS, unless the table can accommodate a variant record or untyped element column. Such a table-based system could still be used by employing generic type records that were sufficiently extensible to meet requirements but there would be an abstraction defined in the data model that mapped data in and out of the store, hiding the underlying implementation from the user. Such an approach would also potentially be very wasteful as the same size structure would be required for a long string as for a byte. Another issue with table-based systems is that a change to table structure may require the recreation of the new table with the re-insertion of all data. Thus, small-scale ontological changes may result in reduced performance due to unforeseen overheads in recreating the database. This could be reduced by only allowing structural changes at system start-up, absorbing such overhead into one-off start-up costs. This comes at the cost of continuity in the face of system evolution.

The next chapter assesses our methodology in the context of potential target systems and compares the approach taken in this thesis with other possible solutions. This leads to the identification of the shortfalls in existing approaches and provides us with an opportunity to identify better, or possible, solutions.
Chapter 5

Application of the Methodology

This chapter develops our methodology in the context of potential target systems, in order to identify where existing approaches do not meet, or do not properly meet, user and system requirements. We compare the software architecture proposed in Chapter 4 with several frequently used approaches and then identify how we can model systems so that we can provide the first steps of the application of our methodology. By identifying those aspects that are common to all distributed systems, rather than those that are more specialised, we can justify our ontologically-based approach in terms of both versatility and potential for code reuse. Finally, we discuss how we can fit systems into our architecture.

5.1 Target systems

The three-tier architecture and three-branch ontological model can be applied to existing legacy network-based information systems and modern distributed systems with equal validity. However, the major benefits can be realised when the system is of a size, longevity or utility that wide-spread participation in the system has established a large dependent user community. In Chapter 4, we established that there is significant overhead in explicitly implementing an ontological layer as the metadata burden increases in direct proportion to the details required in the contextual framework. The benefits must outweigh this additional overhead and, as a result, the systems targeted must have a sound purpose for the technology.

The target distributed system must have the majority of the following characteristics:

- Evolution - The system has evolved over time or has shown a need for short-timescale evolution. This may have previously been handled through central modification, committee-based control or a devolved development process.
• Data Exploitation - The data resources stored within the system, and made available through the system, are not being exploited to their fullest by the existing system in a way that meets user needs.

• Local Modification - A modification community has either arisen or is present in an informal manner to produce non-standard versions of the software that meet local requirements.

• Well-defined Purpose - The system has a well-defined purpose that, from a functional componentry perspective, can be integrated with other systems to produce useful super-servers.

• Metadata - Metadata is either part of the system or external metadata is a desirable addition to the system.

To explore these fully, each of the target characteristics will be discussed in detail. These characteristics will be discussed in the context of the Domain Name System (DNS) [3, 4]. We describe an implementation of the DNS, in order to demonstrate the application of the model to an existing network-based information system, in chapter 7. This implementation also demonstrates the capabilities of the approach.

An ontologically-based representation of a system is also regarded as a Knowledge-Representation-based (KR-based) system. As we develop this chapter, the focus on the system as a set of knowledge entities and relationships will be further developed, culminating in the conceptual model in Chapter 6.

5.1.1 Evolution

The addition of alternative ways of manipulating data is as much an evolution of a network-based information system as is the addition of new function to manipulate existing data. A system that does not evolve in a functional sense, a static system, can still benefit from data annotation as it allows for increased data manipulation and exploitation, whether inside or outside of the system. This is addressed in more detail in Section 5.1.2. While there will be overheads incurred through the knowledge-based descriptions that we use to provide this additional data-oriented benefit, these should be weighed against the immediate benefits of enhanced use of data.

System evolution differs from system maintenance in that it is a coarse-grained, high-level, structural form of change to the software system. This is in contrast with maintenance efforts that have little impact on structure and few economic or strategic benefits.
Where a system is evolving, in response to user demands, changes in technology or the influence of other systems, there must be some form of change management in use which prevents some portions of the distributed system from reaching a state where they become incapable of interoperation with other elements. Maintaining system unity and interoperation must always be a high priority when introducing new techniques, functions or capabilities to a distributed system. However, such change management has an administrative overhead and consumes resources in the form of time and produced reports to obtain a consensus. The programming overhead required to produce prototype systems is also significant and, depending on the system, may not necessarily be absorbed back into the deployment as a separate implementation may be used in the final system. The DNS uses an RFC-based approach to control the change that, if not controlled, would lead to the breakdown of an essential Internet service.

As discussed, a knowledge-based specification allows the ability to logically separate aspects of a system and, in conjunction with reasoning tools and external validation, allows a modification scheme that can admit local changes while still guaranteeing unchanged function from an external perspective. It also allows rapid prototyping for new features, where these are composites of other features, and a relatively rapid development process for new features that are being introduced separately by providing a well-described framework for programming and reducing confusion regarding the nature and use of system data.

Finally, if the evolutionary aspects of the system are logically contained in a separate subdomain to the core, then these can take advantage of a separate version control scheme and also support several different possible extensions to the core depending on local requirement. This can be contained within the three-branch ontological model by restricting certain operations over ontology elements based on node and server information passed up through the data model.

5.1.2 Data Exploitation

Data exploitation is effectively a subset of the evolutionary aspects of software but here the system does not change to provide data in a different way, a new need emerges for the same data to be provided in a different form. Consider an NBIS that functions as a distributed encyclopedia. If designed as human-readable initially, the development of a requirement for the same data to be machine-interpretable changes nothing about the NBIS’s function as an encyclopaedia. There must, however, be a change to the data to provide enough metadata, or context, for the data to be machine-interpretable. The change can be viewed as a 1-to-1 data transformation that is modelled as a transformational black box in line
between the output of the encyclopaedia and the input of the machine interpreter.

The final data forms that may be required by future applications are not part of the function of the data-providing system, and potentially never will be. However, the data stored within the system is always of potential interest to users beyond the current user domain but in a modified, or massaged, form. The restriction of evolution to data exploitation separates the function of a system from the final form of its data, a decoupling that we will discuss in more detail in Chapter 6.

As an illustration of restricting the evolution in system function, the Domain Name System has a Request for Comment (RFC) in place that is specifically aimed at preventing the development of the DNS into a general mapping store where the stored data is not relevant to the aims and goals of the DNS. This explicitly prohibits a development of further standards that would add extraneous, perceptibly non-core, mappings into the DNS. However, a local site may have a need for a mapping that would be considered non-core by other users - even though it leverages local data that is already stored. The request for the composition of the non-core record type would not necessarily make it to RFC stage, let alone be part of a final system.

The full exploitation of data may refer to the reuse of existing data in different contexts or to the restructuring of data into different types and formats in order to meet new requirements. There are many examples where scripting languages, such as Perl, awk or sed, are used to manipulate the raw bytes transmitted to and from a distributed system in order to allow inter-system communication or to allow the local extraction of additional, and currently unavailable, data.

5.1.3 Local Specialisation

The development of local specialisation communities for a widely-used distributed system does not always indicate that the system would benefit from knowledge-based representation. On occasion, there is a simple requirement to interact with locally-developed data formats and protocols that does not require a comprehensive overhaul of the NBIS in question. There are component-based systems where local modification schemes are encouraged and the unifying core of the system has a strong version control mechanism that can be used to integrate and control these diverse, locally-developed elements. The Linux operating system communities function in a similar way to this where the modification community is directed to certain safe areas for development, while the operating system kernel and the heart of the distributions are strongly controlled to prevent divergent development.

The benefits of a knowledge-based approach are strongly indicated when a local mod-
ification community, or the need for one, arises for local changes to a monolithic or non-component based system. The desired changes should fit with the overall purpose of the targeted system, should have a relatively long life-span and should significantly enhance the usefulness of the system or, alternatively, significantly reduce some manipulation operations outside of the system. Manipulation operations, in this context, are those operations required to align the products of the target system with the needs of the organisation employing the system. There must be a strong semantic linkage between the results of the manipulation operations and the knowledge contained in the target system in order for this to be a justification for local change. This linkage can be determined, in the context of the three-tier software architecture, by establishing how much of the operation is carried out at the knowledge level versus how much is carried out in the data model layer. Where a new operation is a pure knowledge manipulation or has a vertically-integrated relationship with the underlying model layers, there is likely to be a strong semantic linkage.

As a first example, a local change that changes the sequence of certain elements of data in a network transmission can be carried out purely in the knowledge layer. This is strongly knowledge-oriented. Similarly, the addition of a new feature in the knowledge level that requires only a new fundamental functional module to support it shows vertical integration through the model and is also likely to be more knowledge-oriented than not. However, a knowledge transformation that requires substantial rewriting in the data model is less likely to be able to be neatly contained and compartmentalised through ontological mechanisms - and backwards compatibility is likely to suffer.

For a final example, the ability to integrate local system documentation for nodes into the DNS is a form of mapping, is associated with host identification data and is, effectively, a formalisation of the ad hoc manner in which TXT, a text string associated with the node, and HINFO, hardware information for the node, records have been used in some circumstances [3, 4]. The overhead of providing programmer resources to rewrite BIND, or whichever DNS server is being used, is considerable and it is unlikely that a small, or resource-poor, site will pursue such an option unless there is a less arduous way of undertaking the changes. Despite this, a need still exists. Without a widespread need for such a change, this remains a local modification.

Changes that are trivial, short-lived or a poor fit for the system do not provide a sufficient sole motivation for moving to a knowledge-based approach, as the costs/benefits ratio is weighted too heavily towards costs.
5.1.4 Well-defined Purpose

The importance of the purpose of a system has already been outlined, using the DNS RFC that prevents the DNS growing in non-core ways and diffusing its current, strong focus. Given the large number of components available to assemble for computation and the possibilities for assembling data streams in ways that traverse several systems, as in workflows, it makes little sense to attempt to build a giant, monolithic distributed system that undertakes all functions.

More importantly, since the burden of annotation increases proportionally with increasing knowledge representation requirements, the strong purpose of a system limits the size of the knowledge domain to be implemented and this prevents an explosion in the number of metadata elements that have to be added. A generic server may be very useful but the requirement to encode all of the knowledge in the world, in addition to all of the computer science pragmatics and theory required to implement is a very large undertaking. With this in mind, single purpose servers provide an ideal basis for the development of a knowledge representation based model. A single purpose server can be mapped into a well-defined knowledge domain with clearly established boundaries and properties. Such systems should make up the bulk of knowledge-representation based systems.

Despite this, the ability to move data between systems is also of core importance in contemporary distributed computing. A translator node is often used in these circumstances, using technologies similar to the scripting languages already outlined and depending upon guarantees of the structural nature of the two data streams and structurally-defined (potentially information-level) rewritings between the two streams. The function of a translator node is, logically, simple. Data, information or knowledge from one system, depending on the type of system, is moved to and from another system. Most translator nodes do not take part in the systems for which they act as translators so their function is further simplified.

If two single-purpose systems have a requirement to share data then it is possible to construct a translator node that can move knowledge from one system to another, based on semantic alignment, but the node can also function as a member of both systems by using the descriptive representation that contains logic, structure and semantic operations. These boundary nodes are active participants in both systems but can also feed data from one to the other. Hence, there is a case for a super server that is a limited composition of well-defined single-purpose servers and this will be explored later. In Chapter 7, we focus on the development of adapter nodes that will provide standard and non-standard behaviour to existing legacy NBIS, as well as looking at extension systems that combine the information from two systems to provide a new synthetic data element that has no permanent representation in the stored data of either system.
5.1.5 Metadata

Unannotated data is the raw form of all values in the system. Initially, any data starts as a bit stream, whether derived from the network or other user input. After some manipulation, this becomes system data and, potentially, information. The classification of information requires the implicit context given by the programming language - beyond this framework there is nothing except bits and bytes. This changes if the data is annotated and the annotation is strongly associated with the data in that it is always found with the data or it must be examined in order to extract the data.

There are few systems that would not benefit from the addition of metadata but the benefit is always weighed against the burden of annotation and the space requirements of a useful markup scheme. In order to justify knowledge representation based on mark-up, the system capabilities must be significantly improved by the addition of metadata. Metadata can allow a large number of desirable characteristics if used in conjunction with tools and mechanisms that correctly exploit it.

The existence of a purely data-driven requirement for the addition of metadata acts as a strong indication that the system is a good fit for knowledge representation. If the data within a system will benefit greatly from metadata association, then there is an additional contextual framework for the data that is considered valuable. Any additional contextual framework is a stepping-stone to a completely knowledge-based representation. Any simple variables must be replaced with, at least, a class-based equivalent that provides the value and the metadata associated with the value - possibly including type information and other system knowledge.

The requirement to integrate external and internal metadata can be driven the previously mentioned need for provenance in e-Science and the integration of existing services with the semantic web.

We began this section by describing what our target systems must look like, with reference to five characteristics. Having now described these characteristics and identified what is important, we can now look at how other approaches have addressed the problems of implementing NBIS and satisfying the needs of both designers and users. We then identify circumstances in which our knowledge-based approach offers benefits.

5.2 Comparison with Existing Approaches

In the following sections, we compare the three-tier software architecture, supporting a three-branch ontological representation of the target system, with other leading mecha-
nisms for implementing complex process systems. We compare these systems in order to illustrate why, on a feature basis, the three-tier architecture/three-tier ontological model has sufficient capabilities to describe the system at least as well as existing representations. Here we focus on capabilities; performance issues are considered in Chapter 7.

5.2.1 Workflow-based approach

A workflow is an executable representation of a process [40]. This is consistent with a knowledge-based interpretation of data that captures logic, structure and operational semantics since, with the suitable interpreter, a KR-based system can be executed from its description. A KR-based system can be regarded as an ontological workflow connecting well-defined components corresponding to the knowledge-rich forms of system components, process information and data. Hence, in one sense, workflows are an intrinsic part of this technology.

When a workflow refers to the composition of existing elements from different systems, called here an external workflow to differentiate it from a KR-based workflow equivalent, this is a reference to a flow of data through processing and storage entities. The workflow uses a semantically rich reference mechanism to describe the data flow in terms of movement of the data through an abstract process representation. This allows the workflow to be interpreted in the frame of reference of the current implementation that underlies the workflow interpreter - the same workflow can be applied to different underlying hardware and middleware systems without loss of generality if it is sufficiently abstract in definition. Because of this, the two models, workflow and KR-based, are equivalent from a black-box perspective. The movement of data between processing components has the same level of error-checking, because of a deeply established contextual framework, and the higher-level, more sophisticated operations provided with the KR-approach can be overlaid with a small amount of work. When the workflow carries out transformations based on the informational level, it is not equivalent to the KR-based approach as transformations into and out of processing elements are based on structural context and an informal, programmer-stored, ontological representation that is not machine interpretable.

In the latter case, a node or system that implements the three-tier software architecture can be used in a workflow but its capabilities are not visible past the first step into the workflow. In a fully-qualified workflow, the metadata obtained from the KR-based system can be extracted by other systems with a shared context at a later stage in the workflow. In the situations where the component systems are completely unannotated and are, effectively, raw data sources and sinks, the KR-based approach can add metadata to raise the elements
to at least the informational level prior to handing them on to a workflow.

An external workflow can be used to leverage the capabilities of existing systems in order to derive a meaningful result from contextually poor components. The workflow provides an implicit semantic framework that then establishes context for the result. Rather than form a workflow to compose unannotated elements into a contextually rich entity, the KR-based approach allows the use of contextually rich entities at the component level as the ontological layer is explicitly encoded.

The three tier software architecture and KR-based approach are not a competing technology with workflows-based approaches but can augment a workflow model to allow a much greater range of potential operation and explicitly state the context implicitly contained within the workflow. This potential for interaction and operational enhancement provides justification for the approach discussed in this thesis, as well as the potential interaction with existing systems.

5.2.2 Orchestrative and Choreographic Approaches

Peltz [117] defines orchestration as a description of the message-level interactions of web services where the interactions include business logic and execution order. These interactions can be visualised as potentially long-lived workflows that span applications and businesses. Choreography is focussed on the sequences of messages transmitted between users involved with the transaction. Choreography is more usually associated with the exchanges between multiple distributed components. A process executed by a single user is more orchestrative in nature than choreographic. The purpose of orchestration is to provide open, standards-based mechanisms for composing existing distributed components in a way that provides higher-level processes.

Existing technologies such as WSBPEL [118] provide a way to co-ordinate and assemble existing services but do not, for a number of reasons, provide a way to decompose the component services into separable sub-component modules. Because of this, the granularity of computation is set at the level defined by the original program but it cannot easily be altered or reordered without rewriting a monolithic service. In this way they are similar to unannotated workflow systems as they do not provide, or allow, deep context exploitation. However, as for workflows, this does not prevent a more heavily contextualised system from being used in a WSBPEL environment.

The KR-based approach takes a lower-level view of a system, using a fine-grained component model, and provide services that can be used by WSBPEL and other business process modelling systems. The approach is orthogonal to orchestorative or choreographic
approaches. As both orchestrative and choreographic techniques are used extensively with Web Services, they are already applied to systems with a metadata-based environmental component. Again, this does not compete with these alternative composition mechanisms but, instead, can collaborate with and augment them by establishing a bridge between services provided by business process modelling systems. This can then allow their more widespread use with interoperation based on a consistent semantic model, rather than the aligning of syntactically-equivalent parameter lists and function names.

5.2.3 Service Oriented Architectures (SOA)

Service oriented architectures are a higher-level concept, building upon some of the technologies already discussed. An SOA comprises four components: transaction services that provide business processes, data services for the provision of data, a registry for locating and registering services and an orchestration mechanism. The business processes aspect is effectively modelled by web services, and orchestration mechanisms were discussed in the previous section. This leaves two aspects, namely the registry and data services, to explicitly compare with our KR-based approach to implementing NBIS.

Data services are often implicitly made available through transaction services, in which case the KR-based approach would be able to interact with this through an ontological description of the data model. This would allow automated annotation and the further use of the data as knowledge in the new system. Similarly, data from a KR-based system can be encoded to the target format required for the SOA data service and, despite the loss of contextual richness upon entering an implicit ontological environment, the data will be provided in the correct format for the SOA.

The registry services found in a SOA are more complex and a registry system would have to be supported within the data model and ontology layers of the architecture, with the final details specified by the ontology. The advertising of services can be derived directly from the knowledge-based specification of components but the underlying mechanism required for advertisement, registration and other registry features would have to be added to the supplied services in the data model. The next chapter discusses the supplied services in more detail.

An SOA can interoperate with the approach described in this chapter and can use as much or as little of the supplied context as is required. If the SOA comprises web services then type information, at least, is available for parameter matching and SAWSDL or OWL-S descriptions.

Important criteria for measuring the non-architectural capabilities of a service-oriented
architecture include data abstraction and integration and the protection of applications and data. Ideally, the complexity of data should be concealed from the user but the data should also be germane to the user’s interests. The KR-based approach can be used to synthesise new elements of data that show the required final knowledge, without having to provide a large array of semantically-connected values where the semantic connection must be applied by the user. Also, the KR-based approach supports the database notion of ‘views’, where the knowledge can be represented in a particular way in order to meet specific user requirements. The protection of data is more easily achieved if the data can be clearly identified in its correct context. The accidental mutation of data, and incorrect access to values, is much easier to prevent if system-level restrictions, based on a logical analysis of the domain, prevent certain activities from occurring. Again, the use of views can prevent certain users from certain perspectives on the data set, allowing for an additional layer of security.

In terms of these capability criteria, a service-oriented architecture can benefit significantly from the proposed approach.

5.2.4 ACID-based transaction model

Given that the model proposed in this thesis is for distributed systems that manipulate information, there is a parallel with transaction-based database systems. The ACID acronym stands for Atomicity, Consistency, Isolation, and Durability, the fundamental properties required for safe transaction processing in a database management system (DBMS). These fundamental properties can also be shown to be an assumed, if not formally required, characteristic of NBIS. This is logical as much of the underlying data organisation in an NBIS is carried out using database systems. Rather than subvert the underlying principles of their comprising databases, NBIS expose the data through network-based request and response handling functions that work with the underlying DBMS. Even legacy systems, such as the DNS, that may not have DBMS back-ends, provide a database-like model of integrity to insure that each request/response pair is handled as a safe transaction.

Atomicity refers to the guarantee that a transaction either succeeds or it fails in entirety. If it succeeds then the transaction has been successfully applied to the database. If it fails then the database is left unchanged. Consistency is a process invariant that states that the database will be in a legal state prior to and following the transaction. With isolation, any transaction appears to be isolated from any other transaction - while a transaction is in progress, no other transaction can see the intermediate states or make use of them. Finally, durability is the guarantee that a successful transaction will remain a successful transaction.
once the originating user has been notified that the transaction was successful.

This combination of facets provides a firm foundation for deploying transaction based systems as they prevent the accidental partial inclusion of data from a transaction failing halfway through update. If the transactions are atomic then the database cannot become inconsistent under normal operations. Other users cannot use inconsistent data by accessing a mid-transaction value that should not be visible and, the durability property allows the user to assume that a successful transaction will remain in the database. Once the values in a database are assigned to a given set through transactions, all reads on the data will return the same values regardless of how many reads occur (true idempotency). Starting from an empty data set, a complete replay of the transactions in the original sequence should produce the final state observed initially. Thus, the application of the transactions to a known starting point, if considered as a single operation, is not idempotent as the start state may change the outcome. If there are any transactions in the applied set that are dependent on values in the current state for their own values, then a different starting state will only result in the same outcome if, completely coincidentally, the values used for synthesis are the same as they would have been starting from the original, correct, starting state.

A number of solutions to database problems in single-node or tight cluster environments are not valid for network-wide deployment or use in situations where transactions are long-lived during their execution. For example, resource locking to guarantee isolation is of little use if a transaction takes 12 hours to complete - the locked node is effectively removed from the computation environment if the locks occur within a heavily used section.

Ultimately, however the system is represented, the successful management of distributed transactions requires that the system semantics are correct. However, there are some advantages of using a KR-based system that can be used to make it easier to support the ACID model in a distributed environment.

As the only section of the model that has access to the store, the ontological specification tier must have the knowledge-form representation of a transaction before it can be committed. This allows for well-controlled store access and, as the result of the transaction must also be in knowledge form prior to storage or decoding, provides a logical point for establishing the end of the transaction. The single point of access can be ordered in a serial access manner or with controlled parallel, multi-threaded, access providing that the knowledge expectations are met.

Consistency can be checked using external model checkers and reasoning engines to ensure that the old state and the new state both entail the original conceptual graph and have maintained their validity. The strict requirement for knowledge level encoding prevents simple malformation errors in the production of values as these would be rejected on
production, hence aborting the transaction.

The ability to annotate every significant value in a system provides a number of solutions that can be exploited within the KR-based system. One of these is that multiple copies of the same value can be generated within the framework, differentiated by an annotated timestamp. This allows for post-transaction reconciliation of dependencies, if such parallelism is required. This mechanism can provide the isolation requirement by checking for possible contamination of data after transactions have completed, rather than enforcing a complex, distributed locking protocol with the associated performance problems. This also addresses the problem, raised earlier, of the accidental use of an incorrect starting state for a transaction replay as the data dependency can be made upon a certain value that has a certain annotation value for its timestamp. The placement of such timechecking values can abort an incorrect reloading of a transaction set in the case of rollback or reload.

Finally, durability would be provided from within the store, with an operational semantic description of a log-based backup store maintained in order to add data back to a check-pointed store in the event of catastrophic failure. As the store design accommodates the insertion of data in strongly associated groups, the updates could conceivably be as straightforward as injecting all of the changed triples back into a triple store to simulate the modification to the graph. The transactions themselves do not have to be stored and reapplied as the result of the transactions can be determined through the comparison of timestamps between semantically aligned components. This is similar to the way that caches can be managed under the KR-based representation. Note that this is dissimilar to the approach used in Linda [119] as the triple store has no notion of an implicit semantic structure that can produce a safe parallel processing environment by association with the data store primitives. While a Linda-like system can be developed, there is no assumption of an implicit semantic structure of the triple store discussed in this thesis.

Interacting directly with an ACID-based system, or traditional DBMS, could be more difficult than simulating an ACID system across the network as a traditional database system has a large metadata-annotation burden and the operational semantics of the database are often optimised for performance, at the cost of some aspects of ACID. The performance requirements implicit in DBMSs with a large data population could conceivably limit the use of this approach. It should be noted that there is little requirement for adopting the KR-based approach with a DBMS unless it is going to be deployed across the network in order to share information with other entities. At this stage, the performance overheads of the WAN environment will dominate and the new data model for the system would be different to that of a single-node or tight cluster environment.
5.3 How the three-tier software architecture and KR-based approach meet the shortfalls

In this section, we motivate the decisions made that lead to our implementation. By drawing on the shortfalls identified in Chapter 2, the literature review, we will start a discussion of decoupling that will continue in Chapter 7 and forms one of the key characteristics of our approach to implementing NBIS. We will also discuss the impact of decoupling, in two subsections on optimised data handling and node complexity.

In Section 2.4.9, we identified a number of shortfalls in current approaches to developing and implementing NBIS. These are, in summary:

- Legacy systems need significant adaptation to work with modern systems employing annotation, such as the Semantic Web, despite the basic functional similarity of both legacy and modern systems.
- The focus on data presentation, with no context, prevents the informed reuse of data.
- All systems have an implicit knowledge-based view that is used in their construction and design - it is usually not machine-interpretable.
- The transformations on data and the data themselves are strongly coupled within individual systems. While this may give performance advantages it stifles reuse and re-exploration of alternative representations of the system knowledge domain.
- The nodes of a system are usually strongly bound to a single mode of operation. While this is often desirable, this lack of functional flexibility prevents agility and evolution.

All of the shortfalls identified centre around the necessity for storing knowledge in a way that the products of a system still have sufficient context to be useful if the system description was no longer available. Our question is: Can the data survive the loss of the program that created it? Our next question is: if not, why not? If we decouple the strong, programmed linkage between program and data then we will address most of the issues identified. Notably, this reduces the overall requirement for strongly controlled standards as the interaction between systems can take place in an ad hoc fashion by exploiting the deep context for data of the co-operating systems.

The knowledge-representation-based approach allows for the tight binding of establishing context to values and also allows for data reuse and system extension. The three-tier
software architecture supports this by defining the access points for data, information and knowledge - providing a solid grounding for the abstract knowledge representation. By using a KR-based representation, legacy systems and existing systems can transfer data by sharing contextual frameworks, there is no need to use a structurally-based wrapper to take data from one system to another.

The KR-approach is heavily tied to the annotation of values. The underlying store is designed to maintain this tight connection and all references to values are made through the knowledge-wrapper that contains the value. The view of data is not presentation-based - metadata describes the type of the data and its exact context, hence there is no need to depend on an assumption of shared structures between systems. Values entering an annotated system from a (traditionally unannotated) legacy system transfer as a strongly-contextualised entity. Even if the raw value is sent in a byte stream, if the request is made through consultation with a shared context document, such as a web-based ontology, then the value can immediately be re-associated with type and context through consultation with the document. However, by negotiation, a raw byte stream can be transformed to a much richer data stream that allows composite data entities to be transferred between two nodes, where traditionally only value-based streams would be used.

It has been noted several times in this chapter that there is always an ontological layer to a program or system as it is impossible to produce the system without at least the programmer having a view of the knowledge domains and their interactions. Usually these ontologies are implicitly defined and not available after system construction. The KR-based approach and the three-tier software architecture provide a place for an explicit model and then demand its construction as, without it, no storage or retrieval of information may take place.

5.3.1 Optimising data handling

A strong linkage between data formats and data processing allows a number of performance optimisations to be made in the handling of data. These are particularly visible when dealing with large datasets as it is much faster to extract data across well-defined byte boundaries that it is to parse data on the fly to determine data size and record ends. Parallel and high performance computational systems derive a significant portion of their performance enhancements from using optimised parallelisable operations on well-known data structures, especially for intensive computation such as large matrix operations. However, decoupling the data transformations from the data does not automatically demand that performance be lost. While the data model tier may not allow a direct code-based
linkage between a knowledge representation and incoming data, once the transformation has been looked up there is a structural equivalent that is used to extract data from the data stream and present it as information. Hence, there is a single additional operation that must be used at least once for any new element but, once read, this could conceivably be optimised in a similar way to traditional large-array structure handling. Once the ontology is reloaded, this association may change and this, in turn, allows the reuse of data and functions, without having to recode large sections of the underlying source code. Also, since all references will be to the heavily contextualised knowledge entity, all references are updated immediately when the knowledge entity changes.

5.3.2 Simple versus Complex System Nodes

If we are to break the strong link between program and data, potentially with the addition of intermediate handlers, we can start to represent a system as a set of interacting processes that have no strongly-defined order imposed upon them. The logical decomposition of a complex system is to a set of much simpler nodes. These nodes can, in turn, then be composed and arranged in such a way that they then present the appearance of the original complex system. There are some very good reasons for having system nodes that are simple and singular in their purpose. In a widely-distributed, network-based system, such nodes are often the workhorses as they are quick to set up, interchangeable in their function and, hence, are easily replicable and replaceable. The price of this simplicity is that such nodes cannot be reconfigured to perform other functions, unless they already carry, unused, the required functionality with them. For example, consider a Unix program to control a forking process. This is usually a piece of code that has two branches, one of which is taken if it is the parent and one of which is taken if it is the child. The code for both may be included in the program, if it can be enclosed in one program, or the branches may become other processes by using the `exec` call. A simple node that can become a more complex node must have, somewhere, a similar branching structure. The only way that it can act as both child and parent, conflating its possible functionally-defined behaviours, is if one of the branches contains all possible behaviours. This cannot be adjusted remotely unless a new code base is transferred to the node - that requires the node to be shutdown and restarted, providing a period of service failure. The three-tier software architecture allows the use of the ontological specification to use any or all of the functions defined in the data model without requiring reinstallation of the code base. It also allows for the development of ontological level operational semantics that allow the controlled introduction of new functions into the data model, from a remote source, which can then be used by the ontol-
ogy. The deep context, annotation and model tier controls allow a simplified development pathway as system variables can be exactly determined, simplifying parameter selection and return values. Combined, this allows for system evolution in place and also the ability for a system to react in an agile manner to changing requirements - even of policy rather than operational semantics.

In summary, although the overheads of implementing the three tier software architecture to support a knowledge representation based environment are significant, the problems present in existing systems are also significant. Through the careful use of constraints, annotation, reasoning and logic, it is possible to make existing network-based distributed information systems perform in a way that will extend their useful lifespans and allow increased interoperation with existing, legacy and future systems.

In the next section, internet services and operating systems are described within the the three-tier software architecture and knowledge-based representation to identify which aspects of the systems fit into the respective parts of the model. The section also details a set of fundamental functional units that provide a basis for all other functional expressions in the system.

5.4 Modelling Program Operation and Knowledge Domains

In this section, we move from discussing the background and problems faced in implementing an NBIS in a knowledge framework to discussing how an NBIS is modelled. Given that we have already introduced the notion of the three-branch ontology and a software architecture that can enable it, we now discuss how we use knowledge representation to model program operation.

With the definition of the syntax and denotational semantics of a program providing a machine-code representation, it is important to focus on the axiomatic semantics, as well as the operational semantics so that it is possible to model what a program is actually going to do. This is important for two reasons. The first is that a model of program operation, before any code is written, allows the production of the source code with less potential for the omission of key logical operations [120]. Testing can be conducted in an approximation of the system that is not the fully-coded system [121]. The second is that the model can be used to check that the program is performing correctly once it has been compiled [122].

At its core, a problem is a description of a particular universe of discourse that has a significant missing component. Truth cannot be asserted in this universe because the ambiguities in the missing data prevent all of the statements being made that must be made to answer the problem. A problem is expressed in order to determine the value, type,
structure or other nature of the missing component.

Although later chapters deal with the enabling technologies available to solve problems, these are all implementation issues rather than issues dealing directly with the nature of the problem or the information it contains. We begin with a discussion of those ways in which all distributed systems are similar and then discuss the ways in which they differ. This leads to a discussion of the modelling similarities between operating systems and distributed information systems, along with the development of core functionality that will be required for implementation. This chapter, and section, concludes with a discussion of how the implementation of NBIS will fit into the architecture that we wish to construct.

5.4.1 Aspects Common to All Distributed Systems

Here we discuss the aspects that may be found in any distributed system. We do this in order to establish the basic requirement for any architecture or model that will implement or represent a distributed system. Regardless of how a distributed information system is implemented, in order to be useful it must present a user interface or well-known interaction mechanism that allows widespread use. In this way, it provides an abstraction over a resource or set of resources and is functionally identical to that aspect of an operating system that handles the illusion of a hardware abstraction. The nature of the control exerted over resources that are controlled by a distributed information system or an operating system varies from resource to resource and system to system. The fact that an operating system is more likely to have write access to local disks, where a distributed information system is less unlikely to have local write privileges and will, more likely, not have any visibility of local storage possibilities, does not characterise a difference between the two systems as far as the abstraction is concerned. A request to a distributed information system will return an answer from storage located somewhere, as will a request made through the Unix Fast File System abstraction provided under the Unix operating system [30, 29].

As will be shown, the aspects where the Unix operating system and an NBIS differ is below the level of the abstraction. Consider the fundamental aspects of an operating system. There is some form of process management, a storage management system, a means of input/output (I/O) and some form of protection of the abstracted resources [12]. The role of a computing system is to execute correctly-encoded processes to meet the user’s task requirement - the underlying mechanisms, available through the abstraction, are all deployed to meet this requirement. Looking at all of the aspects of the interaction there is strong commonality between the aspects of storage management, as mentioned in the previous paragraph, and the protection of resources. Both of these may be implemented
inside the abstraction without the user being explicitly aware of their impact and are most likely to be implemented in this way to increase productivity.

5.4.1.1 Process Management

If process management, as the controller of executing programs, is seamlessly abstracted over the local and distributed resources then a user will not be able to tell which resource is being used unless performance data, response information or scheduling priorities are examined in the full context of the distributed system’s system knowledge framework. In the general case of a distributed system, the types of processes that can be sent out to the model are not heavily constrained and can be of a wide variety. Where the distributed system is restricted to a distributed information system, it is not available as an exploitable computational resource directly, although it may be indirectly tasked through the transmission of queries that cause processing within the distributed information system. In this latter model, a process that references an externally located DIS will do so in a way that executes a local client application of some sort in order to gain access - so there is no separate process management at the abstraction layer. The function call carried out to start the client process executing will appear identical to any other process - the fact that any results obtained will be derived from the network is not available to the user automatically.

5.4.1.2 Input/Output

I/O also falls into this category as, above the abstraction, I/O can be redirected through standard processes out to the network, and information returned, without any indication that network transfer has taken place, except for possible delay in replying that could as easily be attributed to local processing delays. Effectively, as will be discussed in the conceptual model, input/output exchanges with a process of any kind function as transformations of input to output - and a distributed information system transforms to output based on the composition of the knowledge in the query with the knowledge in the store, albeit implicitly for non-annotated systems. All local calls will, unless explicitly providing human-interpretable context that establishes the non-locality of their targets, appear to be executing on the current box. There is no machine-interpretable knowledge that can establish this unless the network interfaces are inspected and the routing topology is analysed to establish that information is leaving and returning to the system. Further discussion will extend this perspective-based argument for representing the relationships between systems as white-boxes to capture the system interface and in order to capture the knowledge contained within.
5.4.1.3 Concealment of implementation

It is common sense that the abstraction conceals the differences between the systems, unless the user’s attention is explicitly drawn to it. The underlying processor topology in a modern multi-processor computer, whether based on a bus, a star topology or a hypercube, is effectively concealed from the user and, often, the differences between systems cannot be exploited unless the user is aware that of the underlying processor topology. Simple extensions of the process model, such as client processes that conceal operations that traverse the external network, will also not be immediately abstraction-breaking.

In summary, it is those aspects that cannot be seamlessly integrated into the local resource management system and that must be explicitly stated in terms of distributed resources and their handling that break through the abstraction. Ideally, any such elements that should form part of the abstraction should be handled through seamless integration.

5.4.1.4 The use of knowledge in the system

The knowledge of the system user is then applied to that information to obtain semantically meaningful results from the data exposed by the hidden resources. There is rarely an explicitly stated knowledge layer in an operating system. As in traditional programming language use, meaning is often derived from human-readable filenames and labels rather than the explicit logical linkage of concepts.

There are, however, knowledge-based concepts in use at the high end of the abstraction. For example, user groupings in an operating system give a logical relationship between users that allows them to be grouped into a class for certain purposes, most usually for application access or security-based reasons. This is a weakly-defined class relationship as such classes are not fixed in an overall taxonomy or ontology and, often, the grouping systems do not support a hierarchical class structure but, instead, offer multiple class membership to support class subtleties through disjunction. So there is a simplistic knowledge model in use, one that is mostly concerned with information grouping and that does not contain the operational semantics, but it is a knowledge model.

It is here that the three distinct aspects of the ontological layer, logical, representational and functional, become apparent. Within the ontological layer, there is a choice as to how the knowledge is encoded and which aspects of the knowledge are encoded. The structures and simple nature of the informational layer can be encoded here, providing an ontological reference layer that can be used to further establish and verify context. However, it is also possible to encode the operational semantics of the system itself and the more complex logical relationships between aspects of the system.
Hence, while the ontological specification layer is still reserved for knowledge, it is possible to define the depth and nature of knowledge in such a way that the modelled system is a pure representation that adds no additional features. However, it is also possible to augment the captured knowledge in the system by adding further representations that extend functionality and, ultimately, decouple the key data of the system from fixed representations based on information-level understanding. By capturing the implicit ontological description of the system, a framework is provided for extending that description and, as a result, potentially extending the function of the system. Figure 5.1 shows an elementary mapping of an operating system into our new model.

In this diagram, the raw I/O is associated with the data model layer in the model, as the programming language semantics that ground the model are more accurately tied to the underlying hardware. Thus, the kernel level is logically associated with the bottom half of the data model - the data level. The more abstract view of the resources managed by the kernel, and those with which users normally interact, are represented at the informational...
level and are found in the top layer of the data model. Finally, those representations of
available resources that support taxonomic or hierarchical groupings can be placed into the
knowledge level as there is sufficient information to provide a grouping of information. We
consider this simple grouping of knowledge to be the lowest form of knowledge. What is
missing, however, are the operational semantics of the system that are implicitly defined
outside of this model and the ability to infer additional relationships between entities.

For example, a physical disk drive and the kernel’s interface to it reside at the lower half
of the data model. The mounted abstract disk-based entity, with i-nodes, resides in the top
level of the data model. The ontological layer has a view of a file-system and can use this
to classify hierarchical relationships within the file system. There is not sufficient infor-
mation to infer new relationships between files (or their underlying i-nodes) beyond what
is explicitly stated in the file system, nor can we change the way in which the file system
works because we do not have access to a mutable metadescription of the file system.

5.4.2 Distinct Aspects of Different Distributed Systems

The level at which the true nature of the system can be discerned exists below the conve-
nience of the hardware abstraction. Once the overall representation of resources that are
accessible through use of a system-defined naming convention is penetrated, the location
of such resources becomes an issue.

The key difference between the resource control exercised over local resources by an
operating system and those managed by a distributed system, of any kind, is that the dis-
tributed system faces an increased range of challenges, as described in Section 2.2.5. These
challenges are heterogeneity, public network, non-ownership of resources, non-local stor-
age and the requirements for standard protocols and policies. Ways of addressing these
challenges have already been discussed in Section 3.1.

The non-local nature of distributed resources poses problems because of the inability to
make assumptions that are stable in the face of the actions of unknown entities. Requesting
a lock on an object to which continued access is not guaranteed is neither advisable nor
guaranteed to succeed. If such a lock was granted, for how long should it remain active?
Does intervening network failure constitute lock failure? These questions have been dis-
cussed at length in the literature and will not be answered here but are raised to illustrate
the potential problems for the reality of the system that underlies the abstraction.

The data that is obtained from resources, local and distributed, is provided in a variety
of forms. The information from a disk drive may arrive as 512-byte blocks where the inform-
ination stream from a remote service could be a packet of 20, 129 or 1024 bytes, depending
on the protocol and the payload of the message. Structure is an essential aspect to consider when dealing with the output of such systems. The high-level abstraction of a character stream with neatly-defined terminating characters does not apply at this level, especially if any detailed addressing is required of the underlying resources. Consider the Unix file system, the abstract unity of the file is, in reality, a connected set of i-nodes with a metadata annotation giving the collection of i-nodes a human-readable name [30]. The fixed nature of the interpretation of these i-node relationships limits which aspects of metadata are available for mutation and what can be stored in them. This illustrates a fixed mode for operational semantics that underlies many aspects in a distributed or local operating system - if the description of how a system operates can be encoded and evolved then there are many useful alterations that can be made below the abstraction layer. These can provide increased efficiency in performance or storage, or can provide legacy functions in conjunction with newer function, without the user having to modify their usage of elements above the abstraction layer. It also provides the system with the capability to expose additional functionality above the representation in order to update outmoded system behaviours or to increase system agility in the face of evolution.

5.4.3 Modelling operating systems and distributed information systems

Whether above or below the abstraction of resources, the benefit of a knowledge-based approach is clear: an informed model that captures system knowledge in a way that allows it to be interpreted allows system modification through the modification of the model. There are also efficiencies that can be achieved through the use of increased contextual and hierarchical placement of information, leading to trees of knowledge that can be used throughout the systems.

Figure 5.1 shows what is missing from a direct translation of an operating system into a knowledge-based representation based on the new model. A naive capture of the operating system provides information of content and structure but requires an externally defined operational semantic model to impose an operational control sequence and ordering onto that content. We could, instead, take advantage of the ability to augment simple structural and logical relationships with captured operational semantics and determine operation by providing detailed context for the logical relationships. Consider the Unix concept of a group, discussed earlier. If user x is to belong to the systems group and to the javaprog group, then two entries must be made to reflect their membership in both. If files are to be accessed by both groups then either the files must be set up to be readable to everyone
or an Access Control List (ACL) has to be employed to allow the layering of multiple group access rights onto a single file. An ACL is a metadata extension for files that was developed to address the inability to express group membership except by combination of groups and the exclusion of non-member groups. Now consider an alternative where users who belong to systems automatically belong to the javaprog group. If \( x \) belongs to the class systems and this is a subclass of javaprog then every member of systems is also a member of javaprog. Such relationships can be verified by reasoning engines or, in simpler cases, through the introduction of short-cut rules in the data model. While this requires the careful construction of the group tree, it removes the necessity for complex additional mechanisms to deal with the problem.

The automatic group membership approach outlined above, employing subclasses and supported ontologically, extends the operational semantics and metadata limits on files and allowing for multiple group checking, without having to implement the orthogonal structures of the ACLs. More importantly, constructed properly, the change in the file system is then immediately visible to every application that deals with it rather than having to recompile every file system application to incorporate the new ACL mechanism.

The example above starts to illustrate the power of our proposed software architecture and ontological structure, but it also illustrates the requirements for careful planning in the design of aspects of the system and the distinct aspects of the design. A system that is designed without a concept of evolution or data exploitation can still be exploited and evolved but will require a large additional framework. In addition, the design decisions made may be difficult to work with. A system that is developed with future extensibility in mind will be far easier to work with as some accommodation for extensibility mechanisms will already be included. As can be seen from the similarities between systems outlined in the first section of this chapter, most similarities between different systems are found above the abstraction layer. By extending this argument, there is a level at which the model can provide an interface that conceals the modelled elements below and still appear very similar to other systems.

In our model, within the ontological layer, we make an explicit statement of the value of the contained knowledge (encapsulating the data), the type of the value (that captures the information) and then a statement of class membership and associated properties that capture the context. This is captured with a statement of logic and structure, but a missing aspect is statement of the context in which the value can be used. To capture this, and all possible transformations of the data, we now show how to state the operational semantics in a way that shows the use of data within the model. This final aspect, along with inferred logical relationships, are the the items missing from the model shown in Figure 5.1.
The detail with which operational semantics are specified is a more complex problem than the specification of the structural and logical relationships. With a well-established type system, there is a point at which the classification of the value terminates and a result is obtained. Similarly, a logical framework for context can be contained within a strictly-defined environment. While the operational semantics of a distributed or operating system ultimately stops at the Turing machine representation, or at a \( \lambda \)-calculus representation if a Turing-complete functional representation is preferred, there are obvious points in these systems, outlined above, where there is a grouping of operational semantics into larger blocks that have distinct functions.

Unless there exists an underlying hardware interface that directly supports the interpretation of a knowledge-based representation, at some stage all of the knowledge encoded in the ontological specification must be interpreted by the data model (grounding it to an implementing language) and then through the programming language semantics to the underlying machine code. This then presents an obvious question: where is the line drawn between those functions provided by the data model, and grounded by the semantics of the implementing language, and those functions that can be defined within the ontological model? We propose the use of a \( \lambda \)-calculus representation for the definition of functions for use in the ontological layer of the model. While there are many possible languages that we could use here, including Scheme, Haskell or even Java, the \( \lambda \)-calculus has a simple grammar that is Turing-complete without having the overheads of a large Application Programming Interface (API) or a large set of pre-defined operations. This makes it light-weight, very versatile and easily adaptable to whichever form is required to correctly represent the operational semantics of the implemented system. By using the \( \lambda \)-calculus, we can provide access to the functions supported in the data model layer and allow their reuse and composition in new and different ways. This wrapper also maintains the consistent approach to the use of information from the data model layer - it must be wrapped in order to be used within the ontological layer. Rather than placing the data model function in strict context and calling it knowledge, it is wrapped inside a \( \lambda \)-program and thus approved for operation at the ontological level. A \( \lambda \)-calculus program can be interpreted, and compiled if required, by a \( \lambda \) interpreter supported in the implementation of the data model layer. However, this only works effectively providing that the interpreter has enough defined primitives to allow access to all resources available to the operating system. Once again, however, the stumbling block is upon which and how many of these ‘primitive’ functions are defined.

From Section 5.4.1, that discusses aspects common to all distributed systems, an implementation of a distributed system should have a notion of process management, storage control, input/output and a protection mechanism available, whether completely defined
locally or available from a primitive function library. Given that the implementing hardware, in almost all cases, does not support direct interpretation of ontological specifications and that a data model is necessary, we include in our model a set of fundamental functions available to the ontological specification layer, from the data model, for use in statements of operational semantics. These fundamental functions provide the basic functionality required to manipulate resources in such a way to bring information in from the data model and use it as information. This allows the simplification of the of the λ parser, as well as a reduction in the complexity of the λ programs required to capture the operational semantics. Thus, there are three types of operational semantics definitions in the ontological specification layer for distributed and local operating systems. These are:

- References to fundamental functions only, with local data references. (Provided from the data model)
- References to λ-defined functions only, with local data references. (Interpreted by the data model but provided by the ontology.)
- Hybrid references, incorporating both. (A combination mechanism with parsing carried out in the data model.)

As the fundamental functions are not the only functions that can be used in the operational semantic aspect of the ontological specification, this set of possible functions is referred to as the native functions of the ontological model. As will be seen in Chapter 7, these native functions have a specific access mechanism. They can also be further extended using the Remotely Installed Native Function (RINF) model outlined in Section 5.4.5. The detail and nature of fundamental functions is also shown in the next section.

### 5.4.4 Fundamental Functions

Fundamental functions are, within our model, a set of frequently used code modules that can be composed to provide all of the operational requirements of the implemented system. Fundamental functions exist because of the common requirements found in network-based information systems. All NBIS must support network-based activity, over defined transports, as well as supporting any of a number of distributed system architectures. While the fine details of the requirements for a code module may vary, there is a core level of operation that we have identified to produce a set of functions that provide essential services and can also be composed with each other to provide more services.

Rather than force any deployment of the model to include a λ-calculus definition of a high-level functional aggregate, such as a UDP server or file system interface, these are
included with the data model layer of the architecture to enhance usability and address some basic pragmatic issues. These stem from the similarity of the systems for which this model is intended. An operating system requires basic functional modules in order to function correctly. A distributed system requires a similar abstraction but with slightly different underlying modules, which are network aware.

The fundamental functions defined for the model are listed below, with a description of their basic function. Figure 5.2 shows how the requirements of operating systems and distributed systems overlap in their requirements for fundamental functions - and also where they differ.

There is a requirement for input and output, especially as the focus of this model is on network-based widely-distributed information systems. Without input and output there is no transfer of data, information or knowledge. In the local operating system, I/O can be handled through access to a local keyboard, monitor and disks. In a distributed system,
especially a distributed information system, a great deal of information comes from the network in the form of traffic across the UDP and TCP protocols, as well as remote file access. Because of the time delays in waiting for human response, and the large interrupt burden on a process, it is rare for a distributed system’s execution to wait on a key being typed.

The fundamental functions are designed to be used in order to establish, store and act upon the knowledge explicitly encoded from the implicit and explicit contexts of the data. Common to both applications of the model are the requirements for the ability to decompose the input, which is data, via a knowledge-specified protocol in order to establish type and context, and hence derive knowledge from data. Similarly, there is no point in storing information unless it can be retrieved. The *store query* fundamental function, which provides the ability to make queries on the store, is a crucial part of a knowledge-based system that is designed to provide answers and, in the current definition, also provides a controlled mechanism for adding information to the store.

The presence of the *null* and *identity* functions may appear confusing at first. These exist so that the operational semantics of all data transformations within the model can be captured. For certain applications, the system exists to provide unfiltered access to a knowledge store. In this case the operation of the system is, implicitly, an identity transform composed with a store query return result. Why is this composition necessary? The information contained within the store, while in context if read correctly, is strictly information, if the data representation is abstract, until it has been placed into the correct context by the ontological specification layer. We add a specification that we compose all results from the store with at least one native function. This specified behaviour ensures that, despite the underlying data representation, the result of the composition is knowledge and, hence, only knowledge can be derived from the store - not raw data. This prevents processes and users from bypassing our strong requirement, from Section 4.2.5.1, that only the ontological specification layer can manage the store. The identity function meets this requirement for untransformed store queries. Similarly, if a transformation returns no data, the null function guarantees that the correct knowledge corresponding to a contextually rich ‘nothing’ are returned. The next question that arises is: why can the store query function not just return knowledge? Because of the nature of the store query, there is a possibility that data model information will have to be passed back in order to address local issues, such as exceptions caused by store filling or removal operations on an empty store, that have no knowledge representation within the current frame of reference. These transitory references are still legal within the model, since they are never committed to the store, but cannot be passed back to a user as there is no valid explicit context. If the entire operating
system was encoded, rather than those aspects required for the operation of a distributed information system, then any information handed back could be resolved as no information would be out-of-context. However, the issue of how to deal with transitory non-knowledge entities illustrates one of the key separation points between the operating system that supports the three-tier architecture and the model of the operating system that is implemented within the three-tier architecture - there is a limit to what is implemented and, because of that, certain design decisions have been made to limit the impact of the problems that arise from this.

Similarly, UDP and TCP traffic arriving from the network must pass through the protocol-based decomposer that determines which protocol is to be used to turn data into information and hence onto knowledge. The decomposer also handles bad behaviour and provides knowledge or null to the calling entity, preventing the propagation of data model-based exceptions and problems through the ontological specification layer. Due to this, there is no exception handling model at the ontological layer as all that takes place is the assignment, insertion, deletion and querying of well-defined, highly-contextualised elements in the store. The well-behaved nature of the context-establishing systems are provided by the ontology language chosen and the way in which it is implemented. The nature of the chosen implementation language for the data model provides the exception-handling behaviour for the data model itself - and, as such, motivates the choice of a language with excellent exception-handling routines to prevent the escalation of exceptions causing faults in the ontological layer. This is, in addition, prevented by the interpretation of the ontological layer as a knowledge-layer with operational semantics extensions, rather than as a programming language.

The decisions as to what is and what is not a fundamental function may change depending upon the time at which the decision is made and by whom the decision is made. Rather than force the ontological definition of the system to be grounded in a three-tier software architecture that does not meet current requirements, we propose the use of remotely installable native functions to allow the introduction of new fundamental functions at the data model level. This allows the ongoing evolution of the system to occur without forcing an infeasible level of complexity into the $\lambda$-calculus specifications that control the operational semantics.

### 5.4.5 Remotely Installable Native Functions

Operating systems and distributed operating systems are designed to manage resources for users but they are intended to do so for as long as the user wishes - i.e. reliability is im-
Important [15]. Long-lived systems often have additional burdens with reference to reliability because of outmoded protocol or standards implementations but, at the same time, may continue to be used and required for the future. The ability to evolve the system to meet new requirements, new standards and to fix old problems is an essential point to consider when applying the model to these systems. For example, the older implementations of the Berkeley Internet Name Demon (BIND), that provides DNS services, do not have many of the secure mechanisms that prevent a range of attacks or potential compromises of their stored data. However, it was necessary to support backwards compatibility for these servers for some time and, to an extent, they are still supported today despite being hopelessly insecure by contemporary DNS server standards.

Modifying the ontological description of operational semantics can add references to new native functions that were not defined at original time of writing. Hence, while they can be defined in $\lambda$-calculus, this requires that native functions can be added after original system design. While the core functions are already provided, there may be a local need based on performance or expedience that requires an existing function to be streamlined. It also allows new versions of old functions to be installed side-by-side with existing versions, rather than in-place redefinition, to allow parallel testing. This allows for changes such as the introduction of a new filesystem type for a file-reading module or changes to a TCP module to support a change in the standard.

Any functions defined in the data model will be written in the data model’s implementing language. Thus, they require support in the bottom layer of the model. As will be discussed in the Chapter 7, there is a naming convention to distinguish those elements that are defined locally to allow for different code repositories. Remotely Installable Native Functions (RINFs) require that these new native functions conform to the native function interface standard, are written in the data model’s implementing language, have no side effects and have strong exception handling.

In this way, RINFs show the model for all of the fundamental functions that are defined for the system. The guarantees must be the same for both RINF and Fundamental Functions, with an additional guarantee that a RINF must be installed from a trusted location. RINFs allow in-place evolution with continued operation, as the new function can be accessed immediately once the reference has been added to the ontology and the ontology has been reread and verified.

A RINF can be seen as a fundamental function that was not originally identified as fundamental but will now form part of the building blocks for operational semantic manipulation in the ontological layer via the $\lambda$-calculus specification.
5.4.6 Fitting into the model

In this chapter, it has been necessary to start discussing some of the pragmatic issues with fitting a resource-oriented system, with all of the problems that physical resources can present, into a theoretical model that holds a high-level knowledge-based representation as the only form that is valid for information exchange. It is essential to remember the focus of this project: widely-distributed, network-based information systems with long-lived and complex memberships of participating nodes. It is only those aspects of the operating system that are needed that must be encoded into the knowledge form and the vast majority of these are made available to the ontological layer as fundamental functions and, thus, do not need to be re-implemented unless the language that implements the data model is changed. In other words, although a fully-knowledge oriented operating system is a tractable problem, the overheads are significant and would require either a processor that directly interpreted the ontological specification or had a large interpreter layer, in a microkernel environment, which dealt as directly as possible with the available resources. Rather than focus on this, the model has been described in conjunction with an operating system to show that it is possible - not that it is necessarily desirable.

It is the representation of the network-based information system that is of most interest since, with the resource handling abstracted, the representation can be far more easily described as a straight-forward set of data transformations. Input is composed with stored data to produce output. The model guarantees that whatever is returned to the user has as much context as possible so that global reference can be made in a shared context and, with human interpretation, this allows shared meaning.

Within the ontological specification, there must be enough framework to support the data and information representations, as well as the knowledge-specific aspects. The basis for these representations are the syntax, semantics and logic of the system being modelled. These are all discussed in the following section, with discussion of the importance of metadata and ontologies as the basis for semantic annotation technologies. This leads directly to the complex conceptual model that allows us to correctly derive the structure and elements of the final ontology used in implementation.

In the next chapter, we develop the conceptual model in order to provide the foundations for the production of our ontology. Our approach will not work without a sound model, upon which we can build our architecture and thus implement the target systems.
Chapter 6

Conceptual Model

This chapter contains the conceptual model for the environment of the problem domain, and a detailed discussion of the contained concepts, which ultimately leads to the production of the detailed ontology of the system as an abstract system with real-world constraints applied. This chapter concludes with a discussion of operational semantics and is followed by a brief discussion of the formal and theoretical basis of RDF and OWL in order to clarify what can and cannot be assumed about the modelling of such a system.

It is essential to establish the conceptual structure of the system as this is required to ensure that each tier of the model meets the conceptual frameworks required for implementation and, thus, can provide a true model of the target system. The high-level conceptual representations of both servers and clients are examined, with a decomposition of the server-side operations required to implement the three-tiered KR-based model. This section also provides more detailed motivation and design requirements for the underlying storage mechanisms, including the simple nature of store operations.

There is then a discussion of the operational semantics capture for a system and the requirement to provide a lower-level tier than the ontology to provide the implementation of a system’s operational semantics. Finally, the way in which knowledge objects use self-reference to provide a set of locally significant functions, in conjunction with store operations, to provide knowledge from and store knowledge to the store.

6.1 The Problem Domain

The problem domain for widely-distributed, network-based information systems is the internet. A diagram showing the possible composition of such a system, indicating the geographical and organisational boundaries that can be crossed by such a system, is found at Figure 6.1. The data points labelled Server indicate a node that can respond to client
requests. The other icons in the image are all clients. Note the arrow between servers that shows that there is either a server-to-server exchange protocol or the possibility of a server also acting as a client.

As shown, the logical entity that, for convenience, is referred to as a single system comprises storage nodes, access nodes and interconnections. In a widespread implementation of a network-based information system, such systems have components within many organisations and geographical locations. When a service requires active community participation to provide the information, such as the DNS, then there may be multiple nodes and
data stores in a single organisation to provide local reliability, hierarchical partitioning and load-balancing. There may also be additional replication and load-balancing mechanisms at a system level so that information provided by one source is cached at, and available from other entities in the system. Figure 6.2 shows the layout of a simple network-based information system with the geographical and business overlays removed. This illustrates the underlying complexity of the system structure, even without any geographical or business-related structural requirements.

### 6.2 High-level Conceptual Model

From the previous discussion of operating systems, and distributed systems, the abstract conceptualisation of such systems can be captured in a model with abstract representations of process management, input/output, resource access and protection, where each of these may be applied to local or remote resources. In order to remove any dependency on the location of the resources and to provide a standard interface, all requests to resources are made through a proxy abstraction that, at this level, provides a uniform access mechanism for established resources. Obviously, these resource links must be established but the immediate abstraction is of a system that can schedule processes, deal with I/O, retrieve information from and store to storage resources and control access to all established resources without concerns over location. Precisely, this is a functional conceptual model, not an architectural conceptual model.

This immediately leads into the conceptual representation of the servers and clients in an arbitrary network-based distributed system for which information storage and retrieval is a focus. Figure 6.3 shows the high-level conceptual representation of a server for such a system and figure 6.4 shows the client. The dashed lines in the client diagram show that, while a client may make a query and then not commit the new information locally, it remains a possibility. However, if it does so, then it is described by the same mechanisms as the server.

The input/output proxies in each diagram show an abstraction away from the physical medium that supports input/output. Thus, what is behind the proxy could be a network cable, a tape or disk system that is providing information stored from elsewhere or, even, a keyboard. Similarly, the protection proxies represent the presence of a protection service that limits access to or from the data store, that is, in turn, abstracted away by a proxy. The protection proxy could perform no function but it is include in the diagram to show that, conceptually, the decode/act mechanism does not have unfettered access to the storage system.
Figure 6.3: The high-level functional conceptual representation of a network-based information system server

Figure 6.4: The high-level functional conceptual representation of a network-based information system client
The server model must be further developed to show the role of the standardised format. In order to do this, two key concepts must be introduced. The first is that of an established context. Establishing the context of a value that has a typed representation allows the location of that value as an entity in the data model. All values in the system have an implicit context, even if an explicit one is lacking, in order to produce a real-world value to which meaning can be applied. In terms of programming language concepts, our program logic must be semantically grounded in a real-world context in a consistent manner or the execution of the program, which is otherwise syntactically and program-semantics correct, will produce a useless result. The second key concept is that of ephemeral values, values that have a local meaning (and may be capable of having an external context established) but that are not stored for the long term. These ephemeral values cannot have a lasting impact upon the program as they are never associated with an established context and, as a result, are only visible inside the modules in which they are used.

Figure 6.5 shows a diagram of the server conceptual model, with the addition of a process that can be used to establish the context of a value. This is the first part of the conceptual protection model, as only values with context can be retrieved from or stored to the data store - as outlined in the new approach. Both the encode and decode processes must establish which protocol is in use prior to decoding the message - this may be determined in a number of ways but, for the time being, the exact process is not specified. It is assumed that there exists a common communication protocol. The linkage of the context to the decoded elements from the input/output stream allows the immediate determination of contextual validity - which is the first protection mechanism preventing access to the store. There is a branch statement controlling access to the store and providing for error feedback, in any of the valid forms from simple reporting to exception raising.

However, as outlined in previous chapters, only representing the data in the system in this way is a severe limitation of the potential of this representation. The operational semantics are used to determine what happens within the server, whether it is handling i/o, preparing responses or establishing context. One problem with specifying the operational semantics is ambiguity of reference, where the programmer defining the semantics makes reference to program critical variables in an ambiguous way. A large portion of this stems from the requirement to establish the contextual framework of the variable before use. Ultimately, for a network-based information system, there is a requirement for i/o, query handling and returning the result - providing that the query abstraction can be adequately defined, most of these queries can be resolved using existing information in the store or information about information in the store. Figure 6.6 shows a more detailed conceptualisation of the ‘Act on Input’ and ‘Act on Result’ processes from the previous figure to show
Figure 6.5: The high-level functional conceptual representation with context establishment

the role of context aware subprocesses in this conceptual representation. These are now represented as a single process with multiple parts to capture all of the behaviour.

What is immediately apparent from the diagram is the potential for encapsulation. The context-rich knowledge from the input decoding is passed to the store for a query operation. Knowledge is then passed back to the answer handler, which then passes that on to a filter. The filter is defined as a process that, in conjunction with other knowledge in the store, filters the returned results and may return all, some or none of the returned values. It may also not return information directly returned from the store, it may instead substitute metadata such as the number of records returned or even the set of records that were not returned. The filter will be discussed in more detail in Section 6.2.2. The knowledge returned from the filter process is then composed as a reply, with any additional knowledge added that will be needed in the final representation, although it is not yet rendered into the final delivery format. At this point, the reply may be stored as either a record of what has been produced or for caching purposes. Finally, the knowledge is assembled into a protocol specified byte stream and presented to the I/O proxy for delivery. Note that, throughout this process, there is no need for ephemeral data to be passed from one sub-process to another as the knowledge capture contains all of the necessary data.

Before the final introduction of operational semantics into the model, it is important
to describe the conceptual nature of queries and the store to show why the operational semantic approach is valid.

### 6.2.1 Query formats and the store

For the purposes of this discussion, the store is an abstract storage resource that has a well-defined input and output mechanism and has infinite capacity. Since machine-interpretable context is being automatically attached to stored values to produce knowledge, and this knowledge is being stored ontologically to allow reasoning and inference, there is a corresponding storage format abstraction that can be employed to capture all of the knowledge and make it available in a machine-interpretable manner. An ontology may be defined in the form of a directed graph where nodes define concepts, classes, properties or individuals, and arcs denote relationships between nodes. (There are alternative representations, including the location of predicates on the arcs as in RDF but this notation will be used for this chapter for diagrammatic clarity.) As each concept has a corresponding grounding in the ontological definition, for example, a class is defined as a class with a name and any additional restrictions or properties, all graphical relationships are expressed using subject,
**predicate, object** format where a predicate gives the well-defined relationship between sub-
ject and object. These 3-tuples are combined through a query mechanism to identify the context of data. Obviously, if this is to be unambiguous, there is only one object for a given subject and predicate. Given that the entire relationship between subject and object hinges on the nature of the predicate, this is not a serious limitation. A knowledge entity either exists or it does not. If it does exist, then multiple entries of the same subject, predicate and object format merely confirm the existence of the entity - no information is added to the system. Henceforth, this notation will be referred to as \((s, p, o)\) format.

As the store is defined as tuple storage, then a simple conceptual query mechanism is defined as follows:

For any tuple \((x_0, x_1, x_2, ..., x_n, y_0, y_1, ..., y_m)\), where \((x_0, x_1, x_2, ..., x_n)\) are bound literals and \((y_0, y_1, ..., y_m)\) are unbound variables, the result(s) returned must match each of the \(x\) bindings and provide a binding for each of the \(y\) variables. Hence \(\text{('A', 'B', y)}\) can only return a value if a tuple such as \(\text{('A', 'B', 'C')}\) exists in the store.

Similarly, a tuple may be added to the store if the tuple is presented such that the number of bound literals equals the size of the tuple - there are no unbound variables. This statement is copied into the store. (This is why the store size was set to be unbounded as this allows for infinite expansion of the store.)

### 6.2.1.1 Terminal and non-terminal graph nodes

In \((s, p, o)\) format, any graph may be encoded as a set of tuples where \((A, x, y)\) is the start of the graph and \((w, z, Z)\) is the final tuple in the sequence. All nodes for which \((s, p, o)\) exists, but \((o, x, y)\) contain an object that is never used as a subject, are referred to as terminal nodes.

More formally, a count operation is defined that provides the number of tuples returned by a query. If \(\text{count}((A, B, C)) = 1\) and there does not exist \(((C, X, Y))\) then \((A, B, C)\) is a terminal tuple. That is, no predicate \(X\) exists that has its subject the object of predicate \(C\).

No assumptions can be made about the terminal node apart from the fact that it is terminal. It cannot be determined to be a literal value or a resource, in the RDF sense, until such time as the predicate is identified. In the conceptual model, nodes are not distinguished by being literals or resources so no further distinction is possible.
6.2.1.2 Summary of query operations

As mentioned previously in Subsection 4.1.1.2 and Section 5.4.4, we can logically represent any form of query message to a system as a simple request to fill in missing form elements - that is, there exists a syntactic rewriting for the input and output that maintains semantic sense but is simply to map into and out of the store. Any knowledge encoding that is used must have a transformation mechanism that maps its formats into either a graph-based format, which then has a tuple representation, or must have a direct tuple representation in order to be used with the model defined.

This mechanism was developed to be interoperable with existing description frameworks and ontology languages such as RDF, OWL, N3 and also Prolog, providing that the Prolog atom specification is encoded in a particular way.

The query mechanism described here is not a concrete implementation: there are no details on what is returned if there is no result, nor what happens if incomplete tuples exist in the store. The query, as discussed here, provides the abstract framework required to inform the implementation and allow for the final discussion of operational semantics.

6.2.2 Operational Semantics

The most complex components of the ‘Act on’ super-process, Figure 6.6, is the sub-process labelled ‘Filter Results’; because it allows the introduction and substitution of values, and hence knowledge, into the results by combing some aspect of the knowledge in the query with the results returned from that query. Logically, there must be semantic agreement between the query and the information that is in, or can be derived from the contents of, the store - or there is no point in making the query.

By definition, ‘Filter Results’, Figure 6.6, works with knowledge, therefore all of the values with which it works have established context and are not ephemeral. The context of the query, the query itself and any other pertinent knowledge is passed to this process in some form that strongly associates context with value. As the type definition of any defined piece of knowledge must be in the store, as a tuple of the form \((\text{knowledge}, hasType, k\text{Type})\) for a predicate \(hasType\), the value associated with knowledge can be determined with a query of the form \((\text{knowledge}, hasValue, k\text{Value})\) and the type can be established with a separate query. Thus a 2-tuple of \((\text{knowledge}, k\text{Value})\) is sufficient to establish the linkage between the name of a piece of knowledge and its value, although it would not be sufficiently unambiguous for a store query. This immediately allows the use of an associative array to store knowledge and its corresponding value for use in any non-store based operations.
This ties directly to the operational semantics, which can express manipulation of an associative array in order to make changes to the knowledge that was passed in through the I/O system. There are only two references that need to be passed from sub-process to sub-process to ensure that all knowledge and store access is carried out successfully - an associative array that contains key:value pairs where the key is the context of the value and a reference to some access mechanism for the store. Logically, this should be an abstraction for the store that implements the protection proxy and the store proxy.

As has already been discussed, it is not possible to write a high-level specification that is directly executed - there must be an intermediate layer, which is referred to here as the data model, introduced in Section 4.1.4.2. Contained within that model are fundamental functions that have a well-defined behaviour. If these fundamental functions are defined in such a way that they pass the associative array and the store reference, then this provides a basis for a simple functional composition mechanism that allows the capture of operational semantics.

However, what is required is a mechanism to allow the limited alteration of the associative array to reflect changes in the values associated with knowledge. There also exists the need to be able to perform branch operations based on values stored in the associative array. These two concepts are defined as the ReplaceAnswer function and the ValueQuery function respectively, the latter in conjunction with a boolean branch facility.

The store can be used to hold well-defined, and context-rich, references to combinations of fundamental functions that can then be evaluated by the data model to provide the operational semantics for the system. These can then be passed around, as can any other knowledge, to allow programs to contain their own instructions and modify them in response to changing requirements. The inclusions of the ReplaceAnswer and ValueQuery functions allows the representation of the ‘Filter Answer’ process as a set of ValueQuery and ReplaceAnswer compositions but this imposes a hard constraint upon the possible answers that can be obtained from the system.

Since ValueQuery and ReplaceAnswer must work with knowledge, and all knowledge either exists in the store or is concerned with the quantity or nature of what is in the store, the only results that can be returned are also limited to what is in the store or what can be described about the store through the use of queries on the store itself. This is not a significant limitation as the nature of the distributed system is such that the values contained within the store are the target of the queries and the additional capability to also provide metadata from the store in a generic, programmable way is actually a significant improvement upon what can be achieved through a hard-coded simple, non-semantically aligned query mechanism.
6.2.3 Object Conceptualisation

The overall concept for this system is that a query exists as a bundled set of knowledge, with some missing elements. While queries can be referred to as instances of some query-class, the knowledge linkage requires a new term. We will refer to bundled knowledge sets to as knowledge objects. As the knowledge object is processed, some values are changed and, ideally, if the query can be resolved, the knowledge object is returned to the caller containing no unbound variables. It contains as much knowledge for that query as the system contains and is permitted to return to that user under current conditions.

Thus, each query contains a description of a partially-complete piece of knowledge. Logically, we can then see the presentation of a query that is a complete knowledge entity as adding, or possible confirming, knowledge to the system. Where a query is incomplete, the partially-complete query will have completed sections that can be used to establish context for those elements that have not been quantified and this then allows the binding of knowledge from the store into the unfilled query elements. If there are query elements for which a binding does not exist, then the query cannot be completely satisfied but, instead, is only capable of being partially satisfied at best.

All of the processing can be regarded as a set of methods that act upon this object data to provide a final form. Ultimately, this leads to a conceptual representation of each query as a single instance of a query class with a locally significant set of functions that are applied by the data model to the encapsulated data. This is shown as a UML sequence diagram at Figure 6.7.

This clearly shows the self-calls on the knowledge object, as well as the subordinate queries to the store to make the initial query, to find the required operational semantics and to potentially place information back into the store at completion of the operation. Note that all calls to the store must do so through the protection proxy.

In the next section, some of the formalisms used to this point are further defined and extended, along with the presentation of the formal elements of the target implementation languages for this project. These extend the concepts outlined here and provide the basis for the final implementation.

6.3 Formal and Theoretical Basis: RDF and OWL

Over the rest of this chapter, a more formal representation for RDF and OWL is presented in order to provide a stronger basis for the concepts and ideas expressed so far and to also provide some formal grounding that will be used in the implementation of the data
model. The goal is to provide evidence to support the conjecture that the approach taken is a good fit to existing languages and approaches and is at least as capable as these models of expressing the problem domain.

At this stage, some decisions must be made as to which technologies will be used in the implementation and, hence, which formal semantics are to be unified with our model. As wish to use a graph-based representation, RDF [57] is a logical choice for the representational framework and was originally discussed in the Information section of Chapter 2. RDF is a good choice for a potential implementation as it has well-defined semantics [123] as well as a number of representations that implement the model. These include the XML [55] representation commonly used in Semantic Web applications as well as other repre-
sentations, including the N3 format [57].

Following on from this, the ontological language choice must be a technology that works with RDF/XML and also provides the full ontological vocabulary required. OWL [124], the Web Ontology Language, is the obvious choice to fill this role as it is built upon RDF/XML and has a model theoretic semantics basis that is an extension of the RDF semantics [95]. OWL was discussed in the Knowledge section of Chapter 2, as part of our discussion of ontologies.

Because some aspects of RDF semantics, and OWL semantics, are used to provide the self-healing properties discussed in the implementation, the semantics of RDF will be discussed in brief outline. A full discussion of the semantics of RDF can be found in the semantics document [123] and is not reproduced here.

6.3.1 RDF

From the semantics document [123], RDF is ‘an assertional language that is intended to be used to express propositions using precise formal vocabularies’. The semantics of RDF are expressed using model theory, which describes the minimal conditions that must be satisfied for every expression in the language to be assigned an appropriate meaning. In this way, the semantic specification of RDF is used to determine the validity of inference processes and, hence, identify when truth is preserved and, importantly, when truth is potentially not preserved.

An RDF graph is defined as a set of RDF triples and a subgraph of an RDF graph is a subset of the triples contained in the graph. Each triple is considered to be a subgraph in its own right. An RDF graph consists of set of nodes and arcs, where the connection pattern is node-arc-node. There are three types of nodes, urirefs, blank nodes and literals. An example RDF graph is shown in Figure 6.8. This demonstrates the relationship between the graph representation of RDF and the XML representation of RDF, shown in Figure 6.9.

For comparison, the equivalent RDF/XML is shown in Figure 6.9.

The namespace definitions at the beginning of the XML provide a shared vocabulary that can be utilised to ensure understanding between users of the XML and, in the case of machine interpretation, a shared context for statements made in different RDF files. The tags and attributes combine to place the required information into a form that can then be represented with an RDF graph.

A uriref node provides a unique identifier in the form of a Uniform Resource Identifier (URI). Blank nodes do not have URLs and are used where the identification of the resource does not exist at time of assignment or is not considered meaningful. Literals
Figure 6.8: An RDF Graph showing urirefs and literals, connected by arcs.

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <contact:Person rdf:about="http://www.cs.adelaide.edu.au/People/jnick/contact#me">
    <contact:fullName>Nick Falkner</contact:fullName>
    <contact:personalTitle>Mr.</contact:personalTitle>
  </contact:Person>
</rdf:RDF>
```

Figure 6.9: The RDF Graph from Figure 6.8 shown in RDF/XML

have three parts, a character string, an optional language tag and a data type. Blank nodes are sometimes referred to as bnodes or anonymous nodes, although the latter form is now deprecated usage. These three node types are sufficient to represent the RDF triple, which is the subject, predicate, object form previously discussed in the Section 6.2.

A name may be a URI reference or a literal. Note that, because the literal may contain a URI-based type reference, a typed literal comprises two names. A vocabulary is a set of names and the vocabulary of a graph $G$ is the set of names that occur in either subject, predicate or object of any triple in $G$. The one exception to this is that URIs that only occur inside typed literals are not required to be in the vocabulary.

If the set of blank nodes in a graph can be replaced by a mapping to URI references, blank nodes and literals then any mapping from $G$ that replaces all or some of the blank nodes is referred to as an instance. Every graph is an instance of itself. An RDF graph is considered to be ground if it contains no blank nodes. Finally, a merge of a set of RDF graphs is, in the absence of blank nodes, the union of graphs. If blank nodes are present, the merge is the union of a set of graphs where each member of the original set is replaced
by an equivalent graph to itself with no blank nodes.

6.3.1.1 Blank Nodes and Interpretations

In order to discuss the significance of blank nodes, it is necessary to introduce the notion of an interpretation [123]. Since making an assertion is effectively making a claim about the universe of discourse, which we will define here as the world, an assertion is a constraint upon the possible nature of the world. An interpretation provides sufficient information about a possible world in order to establish whether any ground RDF triple is true or false - whether it is valid within the world.

Blank nodes indicate that a given something exists without providing any name information. This identifies the key difference between a simple union of graphs and a merge, as defined above. If two graphs are from the same information source and use a blank node in the same way in both graphs, then a simple union may be made where references to the blank node are made across both graphs. It is possible to refer to the blank node from both graphs in the union and identify the same ‘something’. However, if the graphs are merged then the blank nodes in each graph will be replaced with distinct named nodes and the blank node identity between the two graphs will not be maintained. A reference to the previously blank node from one graph will not refer to the same, previously blank, node in the other graph - the merge has forced the removal of the blank nodes and their replacement nodes are not identified as being the same node. The second behaviour is appropriate when the nature of the information sources for graphs in a set is either unknown or cannot be relied upon. As can be seen from the definition of an interpretation, the interpretation of an RDF graph $E$ that contains blank nodes can only be true if there is some mapping from the blank nodes of $E$ to the resources that make up the domain of the interpretation. In other words, blank nodes must be capable of being mapped into the set of resources before the truth value can be true, otherwise it is false. Hence, a graph with blank nodes can be true but only if such a mapping exists - there is no requirement to carry out the mapping operation.

6.3.1.2 Entailment

For two graphs $A$ and $B$, if $A$ entails $B$ then any assertion made about $A$ can also be made about $B$ [125, 123]. This is an important concept because it allows us to start making assumptions about the similarity of two graphs, and hence two graph representations of ontologies, based on the assertions that can be made in both graphs. There are several important lemmas which are used to provide further information about entailment and the
real world interpretation of entailment. The subgraph lemma states that a graph entails all of its subgraphs. The instance lemma states that all instances of a graph are entailed by the graph. The merging lemma states that the merged graph entails all of the graphs used in its construction. The monotonicity lemma states that if a subgraph of a graph entails another graph, then the original graph entails the second.

This leads to the most important lemma, from the perspective of the implementation in this thesis - the interpolation lemma. A graph $S$ entails a graph $E$ if and only if a subgraph of the merge of $S$ is an instance of $E$. In order to detect whether one set of RDF graphs entails another, take a subgraph of the merge of the graphs, allow the blank nodes in $E$ to be treated as variables and bind these to corresponding names in the graphs in $S$. If such a binding exists, or, in the event of ground graph, there is a correspondence between the two graphs which does not require blank node binding, then entailment has been established. This is only valid when applied to the conclusion of a proposed entailment - effectively using the RDF document as a query rather than an assertion. RDF entailment is also decidable as complete subgraph checking algorithms exist.

Entailment is an important concept as it is used within the implementation stage to determine whether two graphs can still have the same assertions made about them. There are corresponding arguments made for RDF-entailment within the vocabulary of RDF but these are not discussed further here. They may be found in [123].

6.3.2 OWL

OWL may be mapped from abstract syntax to the exchange syntax of RDF/XML and a transformation table exists which may be used to map OWL to RDF graphs [95]. However, to prevent the incorrect augmentation of the information contained in the original OWL, certain constraints must be applied to an OWL DL or OWL Lite representation as an RDF graph. Any RDF graph is an OWL Full graph but, as mentioned previously, OWL Full is not guaranteed decidable and description logic (DL) reasoning cannot be applied to it. The constraints on OWL DL and Lite take the form of a disallowed vocabulary which cannot be used in their expression and, as a result, greatly reduces the expressive power of each form.

6.3.2.1 Entailment in OWL and Decidability

From the previous statement of the importance of entailment, there is an obvious dilemma. The only way to guarantee that an ontology specified in OWL will be able to be tested for rdf-entailment is if it can be represented as an RDF graph. The only way that this can be
guaranteed is either by using a very low level of expression or by using OWL Full. However, if OWL Full is used, there is no decidable reasoning support available. For example, in OWL Full there is no restriction on the use of transitive properties and, as a result, the termination of analysis of such a structure is not guaranteed [126]. OWL DL imposes limits on expressivity, such as the prohibiting certain syntactic cycles and requiring that classes, properties and individuals be disjoint. OWL Lite removes further constructors from OWL DL, limits cardinality expressions to 0 or 1 and places other limits on expressiveness. Despite these restrictions, entailment calculations in OWL Lite require exponential time for computation [126] and, if the ontology is large, may require more time than a system can allocate to it. OWL DL requires NEXPTIME for the computation of entailment, where the complexity class NEXPTIME is the set of decision problems that can be solved by a non-deterministic Turing machine using time $O(2^{p(n)})$ for some polynomial $p(n)$, and unlimited space. Despite the three sub-languages of OWL the majority of ontologies available are in OWL Full format, although a significant portion of these are OWL Full due to errors of construction or omission in design [127].

Using a DL-compatible ontology language, such as OWL DL, allows the use of reasoning to infer additional relationships which are not explicitly stated in the ontology, except as inference relationships. OWL Full is not guaranteed to complete the computations required to be decidable and, hence, is not guaranteed to provide sufficient information to show entailment. However, if such a state is reached then it is accurate - the risk here is not of producing bad information but in not completing the computations in bounded time. As will be discussed in Chapter 7, very few network-based services operate in an unconstrained time window due to the inbuilt restrictions of network traffic and required performance characteristics. A number of the ‘black holes’ in OWL Full can be removed and limit the possibility of unbounded completion, although completion can still not be guaranteed.

The time complexity bounds on OWL Lite and OWL DL also indicate that the computations should be carried out on the smallest set which spans the knowledge as, in EXPTIME and NEXPTIME, the effect of a small increase in element size can have a dramatic effect on the possible time to guaranteed completion.

### 6.3.2.2 Significance

Importantly, if OWL DL entails an RDF graph then OWL Full will also entail the same RDF graph. Hence, if we can produce an OWL DL subgraph of an OWL Full graph, it is already known that any subgraph of a graph is entailed by the graph, hence anything entailed by the OWL DL subgraph can be entailed by the OWL Full graph. We assume that
the graph being operated on meets certain criteria regarding separated vocabularies but, having done so, can now provide the basis for one of the major implementation decisions taken in the implementation of SAMPAN - to use a heuristically driven approach to produce OWL DL subgraphs that can then be used to determine entailment.

6.3.3 Ontological grounding in RDF and OWL semantics

The requirements already stated for literal values, in Chapter 5, include the requirement to store the value (data), with its type (information) and its context (knowledge). The structure of literals in RDF and OWL, where there is a name, a value and an associated type allows us to map the knowledge atom from the three-tier software architecture into the semantics of both RDF and OWL as a typed literal. Similarly, the relationships between entities (nodes) in the system all map into at least one part of the interpretation of $V$ given a datatype mapping $D$ that includes at least rdf:XMLLiteral, xsd:integer and xsd:string.

The datatype mapping $D$ is supported by the underlying implementation language, making up the lower-half of the data model tier, supported by the semantics of the implementation language. This does place a restriction upon the underlying language, namely that it is capable of providing the required grounding for the datatype mapping. As almost all contemporary languages can provide support for these three datatypes, as a minimum, this is not a restrictive constraint.

6.3.3.1 Entailment of operational semantics

Ideally, if operational semantics can be encoded in an RDF form, then entailment could be used to determine if the same assertions can be made about two differently represented forms of the same semantics. The general case is impossible but a restricted case may be possible if the correct representation and a limited representative form is used. If the RDF node-arc-node representation is used to store the operational semantics then a similar entailment argument could conceivably be made.

Assume that a $\lambda$-calculus representation is used to provide the operational semantics. The graph representation of such an expression is traditionally as found at Figure 6.10, nodes are meaningful but the arcs are empty.

Mapping such a graph into the RDF graph model requires finding a value to place into the arc to preserve the semantics of RDF, namely that node-arc-node is an equivalent representation for (subject, predicate, object). Even using a graph representation constrained into a head-normal form does not allow an easy mapping into RDF form. A possible mapping
would involve a ‘Prolog-ised’ version of the operational semantics, using Prolog’s notion of atoms and predicate relationships to build up a large and complex set of relationships where the predicates sit on arcs and an atom is defined as `<name, hasValue, value>` so that the monadic atomic declaration becomes an (s,p,o) predicate-dependent tuple.

If such a mapping is available, is it possible to prove that the λ-representation in RDF form:

1. is entailed by the RDF graph to which it belongs?
2. is therefore entailed by an OWL DL ontology built on top?
3. does not affect the decidability of the OWL DL?

As we will discuss in the following section, we will see that entailment is not necessary for a system that has pragmatic, rather than theoretical, execution bounds. However, we will explain our answers here, before moving to that discussion.

The answer to point 1 is clearly ‘yes’ as any RDF subgraph of an RDF graph is entailed by its parent from the subgraph lemma. The answer to ‘2’ is not known unless the the OWL DL ontology is shown to entail the RDF and, given that there are forms of RDF that cannot be entailed by OWL DL (as they are OWL Full), this is not a guaranteed ‘yes’. If, however, an OWL DL ontology entails the RDF graph then it will retain its decidability, if the same assertions can be made about entailed graphs, as all of the assertions that can be made about the OWL DL ontology do not affect its decidability. However, answering this point relies upon establishing that the RDF graph is entailed by the ontology and that the operational semantics graph can be represented in a λ-form. Of significant concern is the ability to introduce loops into a program specification so that the operational semantics representation could never have entailment established, or any additional information inferred, because of a simple control loop forcing the reasoner/entailment analyser back to
the same point in the tree. While heuristics could be used to ‘DLise’ such a situation, as in the Pellet reasoner [128], it is first worth asking if such an approach is necessary in this context.

6.3.3.2 Is entailment required?

The discussion of operational semantics and formal semantics in this chapter has illustrated the complexity of computing entailment in the ‘lightweight’ versions of the OWL ontology language. OWL Lite has exponential time complexity so even if the ability to determine entailment were a highly desirable characteristic in a large distributed system, it is not guaranteed that the answer would be given in a time that met performance requirements.

Reasoning support is of assistance where knowledge classification can be carried out in an environment where real-time response is not necessarily required. In these environments, we will wait for the final knowledge produced as long as completion is guaranteed within some time. Our assumption is that such knowledge will be worth the wait, as we do not have competing real-time or short-term operational goals with which waiting interferes. While there are systems that make such information available in a wide-spread network environment, these are more of the nature of archival systems that present knowledge in a fixed form, rather than the more agile and temporary nature of active network-based information systems with a strong service role.

There are also some components of a network-based system that cannot be inferred, they must be asserted. One of these is network protocols. While it may be possible through inspection to determine a probably byte/bit boundary scheme for a given protocol, this is not sufficiently definite to provide a definite inference that every message that comes afterwards will conform to such a scheme. It is the equivalent of assuming that a lifeguard can stand on a beach for a day and, if they observe no problems, can be removed because no problems will ever occur. If all messages up to a timepoint \( t \) have been, apparently, of the same format, this does not guarantee that every message after that point will conform. Therefore, the protocol specification must be asserted and if a message arrives that does not conform to it then it is assumed that the message was not of the target protocol. Whether the message was not in the target protocol due to in-transit damage, incorrect addressing or a misunderstanding of the protocol cannot be known with certainty unless the precise effects of the possible interference are known and are unique. In this case, encoding the protocol specification in the ontology still allows all of the elements to be dealt with as knowledge but entailment is of little practical use here. Therefore, any entailment checking should disregard protocol checking.

The notion that we can limit our entailment checking extends the idea, expressed in
Section 4.2, that as little as possible of the ontology should be analysed if reasoning or entailment checks are to be made upon it. The first supporting argument is on the grounds of complexity and the explosion in time required. The second supporting argument is based on the previous paragraph and is summarised as: limit analysis for maximum practical benefit.

In a distributed system, each node has finite resources. Devoting storage space and computation time to unnecessary, and potentially unbounded, computations is both impractical and ill-advised - especially in systems where large groups of users effectively donate resources. Combined with the requirements that network operations time out after a given period of inactivity, even in the case of incomplete message exchanges, to prevent deadlock, this provides an upper bound on the time $v$ can be used for reasoning support and entailment detection.

While there is no formal complexity class for such a bound, the default timeout periods of existing systems can be observed as indicative of acceptable times. The default NFS timeout for RPC operations is 300 seconds (5 minutes). The default timeout for DNS queries is 10 seconds. In the DNS, this timeout is only reached if no servers can be reached that can provide an answer. If an authoritative server can be reached in this time, and does not have the required information, then it will provide a response in this time to the effect of ‘information not found’. In the NFS example, there is a much stronger relationship between client and server as there is an established share and mount relationship versus the intermittent ‘query-response’ mechanism of DNS. Because of this longer-lived relationship, the time that is considered to be a reasonable waiting time is significantly longer - by an order of magnitude. This provides a guideline to the acceptable times for query-based and established-relationship systems due to the widespread acceptance of both DNS and NFS. Query-based systems should respond in a time of approximately under a minute and longer-lived relationship-based systems should respond in under 10 minutes.

The short response time for query systems obviously limits either the degree of checking carried out or the size of graph over which analysis is performed. Either the ontology is very simple, in which case it will not be very useful for classifying knowledge, or a more complex ontology is not checked often or, alternatively, only a subgraph is analysed. Again, the question arises - is it valuable to carry out entailment checking of operational semantics if such semantics may be expressed in the ontology? The response is based on another question, which is ‘what is the benefit of doing so?’ Since there does not exist a general mechanism for establishing if two programs are the same as each other and implement the same algorithm, there is no way of using entailment in a context-free environment to provide a ‘program checker’. However, such a checking mechanism could be used to de-
termine if the program is valid before the operational semantics are applied to the system. There are existing mechanisms, such as parsers and compilers, which perform the same activity in a well-defined manner without the additional complexity of forcing the semantics representation into RDF graphs.

There is still value in placing a reference to the Literal representation of the operational semantics encapsulation (as a text string representing a program) as this may be referred to as a resource within the ontology and associated with other entities to provide the knowledge that this resource name corresponds to this semantic operation. Providing that two systems or programs share the ontology and the ontological context is maintained within the same real-world context, the operational semantics have the same context and should result in the same results on application to similar inputs. This literal value should be in a form that can be interpreted within the data model to provide the data model grounding for the externally-semantically-rich string, with strong context, corresponding to a program. The literal does not have to be in the form of the underlying implementation programming language but there must exist a mechanism in the data model for translating the literal form to a form that can be enacted as the operational semantics of the system.

The pragmatic requirements of the implementation of a network-based information system, especially the time and other resource constraints explicit in such a system, preclude the widespread use of entailment checking. Entailment use is further restricted by the number of elements that must exist through assertion, rather than inference, such as network protocols. In this environment, there is a limited place for entailment checking but widespread use of this technique will quickly make an NBIS implementation slow and unwieldy. However, sub-graphs can be checked if there is a strong benefit in determining entailment between two sub-graph representations of operational semantics from two different systems.

6.4 Summary of Methodology

In the last three chapters, we introduced and developed our new software architecture, based on the classification of the contents of a system as data, information and knowledge, incorporating an ontological interpreter to manipulate system entities as pieces of system knowledge. We also discussed which systems could benefit from this approach and compared the new architecture with the most important existing alternative approaches to illustrate how our approach is different but interoperable with existing standards. Several areas were explored in detail, including the importance of controlled access to a knowledge store to prevent non-knowledge operations from compromising operational integrity. Finally, we
discussed the role of entailment within the framework of the formal and theoretical basis for RDF and OWL and what this would mean to a potential implementation.

These ideas are extended in the next chapter, which details the implementation of the SAMPAN system. The chapter provides details of the development of the three-tier software architecture, the construction of the three-branch ontology and the pragmatic issues that were addressed in producing two different implementation versions in different implementing languages. The chapter contains a detailed discussion of the Domain Name Service servers and clients that are re-implemented in SAMPAN, as well as discussions of the practical applications of the theoretical aspects discussed in this chapter.
Chapter 7

Implementation

The implementation of the new model for a widely-distributed, heavily-used network-based information system required a series of steps to choose the correct target system and to map it into the new model, arriving at a completed set of software elements. Our development process followed the Rational Unified Process in that it was use-case driven, architecture-centric and iterative and incremental [129]. The use cases of the target system were used to capture requirements and were then linked to the analysis, design and testing classes to produce the final software. The implementation is architecture-centric because the software environment was taken into account early on in the design process, resulting in a design process that was not strictly top-down. Finally, a prototype system was built in stages of increasing complexity, with small increases of functionality compared to previous version, which reduced the impact of software errors on previously defined, and correctly functioning, modules. This then was developed into the final proof-of-concept system with reuse of proven design classes and algorithms. This slow increase in incremental functionality reduced the risk of complete system failure in case of error introduction and assisted in debugging and code verification.

The implementation also required the introduction of pragmatic considerations into the high-level conceptual model in order to ground it in a physical domain upon which implementation could be applied. The first sections of this chapter introduce the required pragmatics, the choice of target system and those steps that were required to map the chosen system into the new model. The actual implementation is then discussed, with attention paid to interesting and complicated issues that arose during the implementation, and the chapter concludes with a discussion of the test cases and experimental frameworks that are considered to be a sufficient test of operation.

This implementation was developed in order to demonstrate that the new approach allows a superset of the behaviour of the previous approaches. Our primary goal was to
demonstrate the added functionality and benefits of our approach to implementing NBIS. A secondary goal was to develop the system and still achieve a reasonable standard of performance, in comparison to the existing approaches. The additional overhead required to add ontological definition and management into the architecture adds additional complexity that comes at a performance cost but clearly demonstrates the advantages given by the benefits of new operations and new operational modes. As the demonstration of benefits is our primary goal, we can discuss performance issues within this thesis but are not required to make performance our most important concern.

7.1 Pragmatic Requirements of the Model

The UML sequence diagram in Chapter 4, Figure 6.7, shows how the processing of data, to information and then knowledge, is handled with the overall operations viewed as methods on a given object within a class. What it does not show is the impact of pragmatic restrictions. As this system must function across a network there are certain constraints that must apply, regardless of target system or model design. These include network packet transmission limits, whether they are hard packet-size limits as found in UDP or quantisation implication for protocols such as TCP/IP. As has been discussed, data in the form of byte streams must be decomposed into syntactically distinct entities before the context application can take place and it can be regarded as knowledge. With the UDP constraints in mind, the byte stream cannot be longer than 512 bytes. Thus, if there is knowledge in the system concerning the nature of the protocol used in the incoming message this can be used to put a limitation on the amount of processing required for a message in order to place it in context and either enter it into the store or use it query the store.

More importantly, the nature of a fragmentation-tolerant and retransmission capable network transmission protocol such as TCP/IP imposes additional constraints on the nature of the data readers used to provide a physical implementation for the model. Since the byte streams can be formed, by the underlying TCP reader, out of byte sequences arriving in badly ordered and disjoint packets, these cannot treat the incoming byte streams with any regard to their future knowledge status. There must be a complete separation between the assembly of complete messages and the processing of these messages inside the new system. Existing service programs for UDP and TCP protocols are readily available and can provide a single assembled message from the incoming packet or set of packet. It is a pragmatic requirement of the model that these be used in an almost knowledge free environment to avoid serialisation problems in decoding. However, the incoming message handling servers can provide some information in a strong context as certain information
is guaranteed to be useful as knowledge when derived from a connection receiver. This information includes the protocol that is used to read the message, the system port (if ports are used) at which the message arrived and the IP address from which the message is received. This data can quickly be promoted to knowledge as all of the data formats for this data are well known and its context is very small but well understood. In terms of the model, the data formats form a context for a given message and can be associated with that message as well-defined values.

The conceptual model is a very general model and, while it details which elements are responsible for the various components of the mapping for implementation, it does not directly address the minimum level of predefinition required. The implementation is designed to maintain the generality required by the conceptual model but, at the same time, it must avoid the boot-strapping issue implicit in any self-defining system. In particular, as there is now capture of the operational semantics, there must be a well-defined division between the fundamental functions and those operational semantics that are encoded through the use of the $\lambda$-calculus specification language used. There must also be a basic specification for the ontology, the minimal satisfying ontology, so that a general analysis of the ontology can provide sufficient distinction to identify the logical relationships between entities, the representation of those entities and the operational semantics captured within the description of the knowledge domain. In order to allow a system to move beyond these basic definitions, the basic definitions must be present in all ontologies that can be used in the ontology layer of a server that implements our three-tier architecture. This then provides that assumptions can be made in the data model to encode the implicit semantic grounding that, in turn, allow the explicit specification of the further semantic grounding of the system.

The ontology must contain a concept of a class that contains everything else, to allow subsumption and subsequent entailment if used. In our mode, this is referred to as the OntologyParent. Its children are the LogicalBranch, RepresentationalBranch and FunctionalBranch. The minimal base ontology structure is shown at Figure 7.1.

### 7.1.1 Top level structure of the ontology

The LogicalBranch contains subclasses that detail the logical relationships between concepts and individuals within the model. As these relationships are defined as being well-formed OWL and RDF constructs, depending on precisely which OWL sublanguage is in use, this branch provides a convenient containing structure for an already well-defined and context rich set of operations. The structure underneath this point does not need any more
specification as the relationships can be determined if any of a wide number of potential configurations are used. To be more precise, the data model can determine relationships sufficiently well to carry out its supporting operations - the only impact of poor structure underneath this point is that the ontology may not perform its task correctly but due to mis-specification of the implemented system, rather than the misinterpretation of a correct specification.

Similarly, for the RepresentationalBranch, the data representations in an OWL ontology are well-defined as reference is made to existing datatype specifications, either locally produced or within the XML Schema Datatypes (XSD) definition document. The XML Schema Datatypes definition document defines facilities for defining datatypes to be used in XML Schemas as well as other XML specifications [56]. This allows for an explicit statement of type, and hence allows type checking, in XML and XML-based documents. To be meaningful and to be aligned with the conceptually defined primitive types upon which XSD is based, a representational (type) definition must confirm to the XSD definition document. This provides a widely-agreed upon contextual framework that does not require additional explicit specification in its structure here in order to provide a strong context for data formats. If an error is made here, the underlying data model may not correctly align XSD types with implementing language types. However, this is, again, a
mis-specification, rather than a misinterpretation of a correct specification.

Finally, the branch that requires the most specification is the FunctionalBranch. This contains the operational semantics and, for reasons already discussed, requires sufficient definition that the data model can locate and correctly identify operational elements, place them in context, and execute them in the correct sequence. Not shown is the individual of the Class lambdaFunction which is the start symbol for all operational semantics, however the framework for it is shown and, from the diagrammatic representation of the ontological structure, it can be located by the data model. Locating the start symbol by analysis of the class structure, with an assumed knowledge of the format of program strings in the system, adds unnecessary overhead and complexity to the system. Start symbol determination by analysis is also not robust in the event of two distinct operational modes. With two modes, which are defined as part of a conceptual hierarchy and are in the ascendancy at different points over the lifetime of the program, either could provide a different start symbol and both could be right or wrong depending on the state of the system.

7.1.2 Lower level structure of the Functional Branch

Delving more deeply into the ontological structure allows the introduction of the key classes and facilities for supporting the operational semantics of an implemented system.

The classHook class provides well-defined membership and non-membership operations for any class defined in the ontology as class hook targets. This provides an implicit function that may be called to determine if a given individual is or is not a member of a class by using the name of the class concept. This removes the need to generate a new membership operation that would require the transfer of ephemeral data, the name of the class of interest, in as a parameter. Using this mechanism allows the use of the standard store and associative array parameters defined earlier although classHook has a special return type in that it returns a boolean value. Because of this it can only be used as the test component of a \(\lambda\)-expression in the form of a conditional. The values \texttt{true} or \texttt{false} cannot be associated with a given individual to permanently reflect that individual’s membership status of a given class because the knowledge value is not fixed. At the time of determination, the status is correct but, unless the status is guaranteed to be updated on change of membership, the knowledge that \(x\) is a member of Class \(Y\) is intrinsically ephemeral.

The dashed box labelled \textit{Local Semantics Class} illustrates where the system can support multiple operational semantics systems for two different modes of operation. Thus if a system is performing two different services with the same data, there would be two different semantics classes with a different set of subordinate functions and, of course, distinct start
functions.

This high-level distinction between different modes is only truly critical in the operational semantics branch. This is because the internal representation of data is going to be the same regardless of the number of semantic modes in use, and the logical relationships between data will also have a high level of overlap. If this were not the case, then we would be modelling a different data set, rather than applying a different set of operational semantics to the same data. The distinction is only critical when the data model is attempting to identify those operational semantics that are unique to a given mode as the differing operational semantics could, if not identified, lead to an incorrect comprehension of system data.

### 7.1.3 Data Model Issues

Some of the problems that were overcome in implementing the new model have already been addressed. Pragmatic requirements had a profound impact upon the final grounding of the model and, as the part of the system that is immediately exposed to the ontology, the data model was developed to address these issues and provide a smooth transition from knowledge in the ontology to byte handling in the underlying programming language implementation.

#### 7.1.3.1 Type Information

A crucial component of the data model is the unification of ontologically-defined type information, most commonly from XML Schema definitions, with underlying types in the implementing programming language (IPL). For example, while XML Schema provides a byte type, it provides no bit type. We can produce a bit type by restricting the range and number of digits normally available for a byte and naming this new structure as a bit, maintaining the logical hierarchical relationship that a bit is a subclass of a byte. This could then be associated with an appropriate basic type or class-based type in the underlying language.

It is possible to define the type system so that facet and restriction information from XSD are used to define pseudo-typed variables that are never formally introduced into the pre-defined type system of the IPL. This approach allows for strict checking in the ontological model but the XSD type is then coerced to a different type in the IPL. From the previous example, rather than represent a bit correctly, it would be possible to assign the value to an int or a boolean instead. The manipulation and coercion requires substantial overhead, as well as the translation of all data into this format. Rather than follow this
path, the XSD types were tabulated, with a corresponding type in the IPL. Where an IPL type did not exist, either the XSD specification was used to create classes conforming to the new type or, if it was a frequently occurring case, a new type was added to the IPL support object library. An example of such a type is that of an IP address - four octets that are logically associated with other in a strict sequence. It is usually not found in the type system of a programming language, although it is often defined as a utility abstract data type, but is a basic concept in the universe of discourse used to ground network-based information systems. An XSD specification of IP address was created and the underlying integer type in the IPL is used to complete the mapping from high-level representation to the implementation layer. Facet information associated with the XSD specification restricted the pattern of valid integers so that type checking could be carried out. The pathway from top-level to implementation-level provides the semantic grounding for the IP address type.

By using pattern specifications in XML Schema Datatypes, existing IPL types can be used to assemble new type-defining classes in a well-defined way. One of the XML Schema requirements [56] is that the XML Schema language must allow the creation of user-defined datatypes, such as datatypes that are derived from existing datatypes and that may constrain certain of their properties. These derived types include list datatypes and union datatypes. List datatypes are a finite-length, possibly zero-length, sequence of values of an atomic datatype or union datatype. A union datatype is a union of one or more other datatypes, the member types, or a restriction of another union datatype.

### 7.1.3.2 Derived Datatypes

A derived datatype may be created by using a pattern-based restriction where the pattern specifies a regular expression that must be matched to the value to ensure that the type is correct. Because the regular expression contains shorthand references to predefined types, such as ‘\d’ for integers, simple equivalence between IPL types and the components of the pattern can be established and used to produce type-defining classes. Similarly, the union statement then provides a list of the simple types that make up a union and these can also be used to determine the valid values and type representations from the underlying IPL.

The most commonly occurring problem in mapping target system types to IPL types occurred with variable length datatypes as these did not have simple byte representations. If an integer is defined as occupying 8 bytes then, with information as to whether the integer is signed or unsigned, gives a well-defined range of represented values but also gives an immutable value for the number of bytes that have to be read from an incoming byte stream to obtain the value of the integer. Where a string value is expected, there is a lot more uncertainty as to when the bytes have been read that contain the string. The string may
be terminated by a special character (such as null or line feed) or it may follow the Pascal string convention where a leading byte gives the length of the following string and a series of such strings can be chained together until a zero length is encountered. While the latter string is, technically, null-terminated, this does not contain the full semantics of the string operation. The Pascal string may be encoded as a list of a union of byte and unterminated string datatypes, where the string in each list element has a length defined to be the value specified by the byte. The adjustment of the type to exactly match the string length to the byte contents is left to the data model as such an approach requires XML rewriting. This will be discussed further later in this chapter.

There is one significant type that is not provided by the XML Schema, as it is byte oriented, and that we developed to support the implementation, already discussed in Section 7.1.3.1. This is the type for a bit. XSD provides a byte datatype but many distributed systems use bits in their wire transmission schemes to encode flags and useful information that are critical to the successful decoding of the message. If such data is going to become knowledge then there must exist a type to represent it, which can then be placed into the correct local context. The local bit datatype has the following facet-based restrictions on Byte: a totalDigits of 1, a maxInclusive of 1 and a minInclusive of 0.

7.1.3.3 Introducing new types

Introducing new types does introduce problems as there are possible problems with two users approaching the same data from different type bases. However, as any user of an ontologically-augmented system must use the ontology to guarantee that they are using the enhanced features of the system, their use of any new-type-system specific information is contingent upon their reading of the ontology to determine that such types are available. The only potential problem is that the version of the ontology that they are using is different from the current version of the ontology and may not contain the current version of the new type. This can be addressed through the use of an ontological version number to allow users to check that their used ontologies, or ontological fragments, are up to date.

Another type, to which we have referred earlier, that is commonly used to represent network information is that of an IP address. An IP address can be represented as four bytes assembled as a list with a set order. The use of this type prevents the use of non-knowledge level aggregator types in the data model that store the IP address as concatenated or conjoined octets. Through the use of an IP address class in the data model, it is also possible to provide each octet, if required, through an array-like reference mechanism based on integer.

The new types described belong to two different classes of derived types in the ontolog-
ical framework as one, the string types, is used for parsing data into and out of the model and the knowledge-based conceptual notion of a string is insufficient to capture enough syntactic information. The second, the IP address, is a union of simple types to produce a type that has a one-to-one correspondence with a piece of knowledge. In the second case, the data becomes information and is then aggregated before context is applied in the form of a well-located name in the ontology.

The underlying atomic types of XML Schema are grounded in an assumed conceptual framework where it is assumed that by providing the lexical mapping, value space and facets of a type, the semantic grounding of any associated operations take place inside the user’s understanding of the type’s domain. For example, the use of the float type assumes that the user can define and correctly use floating point operations as no such operations are defined in the specification. All derived types inherit the associated grounding from the composing primitive types. This provides the justification for the ‘short tree’ definition of the representational branch as any types in use in the system can be traced back to the semantic grounding of XSD.

7.2 Choosing a target system

In choosing a target system to be a good demonstration of our new approach, we adopted the criteria that the system be:

- widely-used,
- widely-distributed,
- network-based and
- have a strong informational focus.

The informational focus requires that system be primarily a data-based system rather than providing dedicated remote processing facilities. It is acceptable that a system combines data and processing capabilities, but we require a strong focus on the data side. More importantly, such a system has to be standards-based, which is not a restrictive criterion given the widely-distributed requirement, but capable of benefiting from local modifications in parallel with the standards. With this in mind, the first system chosen as a target is the Domain Name System (DNS).

The Domain Name System (DNS) [3, 4] is a service that stores and provides mappings as a hierarchical system, employing distributed management, extensive use of caching and
a strong standards-based model for making changes to the system. The most commonly used of these mappings are the mapping from IP address to computer name and vice versa. There are many others available but, whenever one is added, the defining standards must be altered and the new change implemented across all of the deployed servers and clients. Not only is the standards-based model slow to provide changes but the final implementation can take considerable time, especially where errors are introduced or existing function is compromised in an unforseen way by the new feature.

This is in contrast with the mechanism used to store, modify and provide the mappings, which is much faster and far easier to fix if problems occur. Any centrally-managed modification to the client and server functionality is currently made at a global level as any changes are applied to the DNS as a whole. While some aspects of the DNS are (and should be) global, some aspects could safely be defined at the local level providing global operation is not impaired. Non-global modifications of the DNS are desirable because these allow virtual organisations and small sites to control how their DNS data is used in their local applications. For example, Semantic Web users could use an annotated DNS data stream to provide a service that can be directly integrated with existing Semantic Web services and add additional metadata describing machines or existing services on a host. Local modification, and output modification based on target address, could also allow the same server to provide answers for standard DNS queries and Semantic Web queries. This facility extends the usable life-span of systems that, in turn, makes better use of the large set of resources that have gone into the production and maintenance of large-scale NBIS. A system that supports standards-based behaviour and well-controlled evolution is far more likely to succeed, or even thrive, over time than a monolithic and stagnating system.

Our goal is to provide a mechanism that allows this local modification without compromising the ability of a DNS server or client to function in the standard environment. More importantly, and as we discuss later, this leads directly onto the notion of knowledge-domain-defined reconfigurable servers that can provide a wide variety of services without requiring the re-engineering of the underlying server.

### 7.3 The Domain Name System (DNS)

We introduced the DNS in Section 2.4.8.1 in order to discuss its basic characteristics and also to introduce a legacy system that is ubiquitous, firmly entrenched as a network service and has a rigid standards-based development mechanism.

We will now discuss some of the existing problems within the DNS to show why our approach is a successful alternative.
7.3.1 Operational Issues in the DNS

DNS is predominantly implemented over Unreliable Datagram Protocol (UDP). Although this does not have the overheads of TCP/IP it also does not have the retransmission or fault detection facilities. If a user fails to get a response to a query then it may be because the server is not there or it may be because the question got lost. To deal with this, DNS requests are retransmitted, usually to a set of servers in turn, at intervals to attempt to get some sort of response.

TCP is used for zone transfer and also where a piece of information is too large to be contained in a single UDP datagram. TCP is then used as a fall-back mechanism since it can provide facilities for handling a larger message broken into several pieces and then rearranged at the receiver’s end.

The most extensive assessment of the performance of the DNS, prior to 2000, was carried out by Danzig et al in 1992 [130]. Although their paper did not give fine-grained results, down to the level of which RR was being used when, they did show that implementation problems were causing the DNS to consume up to twenty times more bandwidth than was strictly required. This was caused predominantly by questionable implementation decisions made in the construction of DNS servers and resolver libraries, rather than by hosts requesting information for which there was no answer.

Later analysis of the DNS, in the post-Web era [131, 132], showed that although the amount of DNS-specific traffic had dropped as a percentage of overall traffic, this traffic was still significant and that key functions of the DNS, such as caching behaviour and resilience to misconfiguration, were neither sufficiently agile nor adaptive to user behaviour or changing environments. The separate work of several authors [133, 134, 135] have all illustrated problems where the sheer success of DNS, with its very wide deployment, makes it hard to resolve problems with authorisation, illegitimate queries and poor caching strategies. Adding servers does not improve matters if the system as a whole does not respond to changing conditions.

Danzig et al [130] noted many examples of circumstances where the misbehaviour of the server or querying agent was at the implementation level, rather than from the misconfiguration of the server. Querying all of the servers in a list of servers is an example of poor design - misconfiguration would be supplying an incorrect list of servers. Analysis also revealed that caching mechanisms, while sufficient, were as prone to the same misconfiguration and design flaws as the servers were themselves [133]. Adding new features or modifying existing features only increases the potential for misconfiguration or poor design - especially in a widely distributed and decentralised implementation.
7.3.2 The effect of strict control on system modification

A key characteristic of the distributed systems of the late 20th century is that, having reached a sufficient level of distribution and adoption in the community, it takes longer to make changes because the change process has to be controlled to ensure that future systems interoperate with existing systems.

There is a base level of function for widely distributed information systems that transcends the requirement for such systems to be efficient - providing that they can still operate. Historically, the acceptance of this relationship stems from the ad hoc introduction of these systems to manage important tasks that are devolved from the highly specialised creators to the general systems community and then to the general public. In the absence of specialised knowledge, the overall performance of the system relies upon two key factors: correct server design and limiting the effect of misconfiguration.

Ideally, misconfiguration on servers should only directly affect the site that it describes. Network-wide poor performance should not be caused through the grossly inefficient network usage of a small site’s DNS server on a fat pipe. A poorly designed query agent can cause a great deal of additional traffic through querying all possible responders simultaneously or repeating queries to servers that have already indicated that they do not have the answer.

The strict centralised control of the specification of the DNS provides a model that can be used to detect errors in implementation, and local modifications to correct this, but it does not allow for localised extensions that is outside of the specification as these developments would, correctly, be identified as errors against the central description and ignored by the majority of servers.

7.3.3 Enforced configuration versus evolutionary configuration

In this context, an enforced configuration is one where a system is defined in such a way that it cannot be changed without threatening its continued ability to perform as expected. Wherever widespread interoperability is required, particularly in a distributed environment, all parties involved must be able to send and receive the key data for interaction and have a well-defined model for handling data that does not meet the requirements/expectations of other side.

Within the DNS, this behaviour takes the format of a well-defined syntax for exchanges between client and server, the behaviour when resolution can and cannot take place and the line format (or packaging) of the message itself. This is also used in the exchange of resource records, as discussed previously, where resource record type numbers are used to
identify the payload type, rather than a textual name for the resource record. The RR type number mappings are an example of the enforced configuration as ad-hoc alterations to these mappings will result in incorrect answers to questions or, possibly, no answers at all.

For example, an A record returns a 32-bit IP address. If a local modification carelessly associates the RR Type number of an A record, 1, with that of a TXT record then the corrupted server could return a TXT record instead. TXT records return strings and are not limited to 32 bits so an attempt to read the TXT string as an IP address will, at worst, give a nonsense reply or, at best, not pass unpacking checks at the other end and be discarded as useless.

Enforced configuration also applies when a user is attempting to expand into currently unclaimed space within a system. If an RR Type number is, as of today, not in use then a local implementation of server and query agent could use this to add an additional query type. For example, if 202 is currently free then a local modification could assign this to their new, locally defined extension, DTXT RR. This will work until such time as there is a legitimate expansion of the overall system into this space and the locally defined type is now at odds with the globally defined type.

This contrasts with evolutionary configuration where the overall function of the system can survive local changes through the use of adaptive mechanisms specifically designed to prevent customisation from having an impact upon overall function. Strictly, this is limited evolutionary configuration as there is an implicit fixed point in the system that local changes cannot move beyond. The fixed point is defined as the level of function that cannot be reconfigured without impairing the function of the system; the concept will be discussed in more detail in Section 7.5. In the case of traditional Java applications this fixed point exists just above the Java Virtual Machine itself - although these programs can change many aspects of themselves they cannot change the way that the JVM works. They can use system-defined functions to modify some behaviour but they cannot, for example, rewrite the JVM as a C compiler using a Java application running above it.

Where a system is replaced by one that can employ adaptive or customisation mechanisms, the fixed point should be placed at a level where the basic function of the system cannot be impaired. In the case of the DNS, the existing function of the DNS should be captured and should be protected from user activity that would prevent normal behaviour of a traditional DNS server.

So the two limits on the adaptive nature of a DNS server that can evolve is limited to the key points mentioned earlier, namely that such adaptation cannot stop the server from fulfilling the same role as a traditionally implemented server and that the behaviour of the server in the face of misconfiguration is predictable and well-defined. However, this
misconfiguration is not necessarily purely in the data level, it may be misconfiguration in
the behaviour or structure of the server.

Here we see the first indication of the true nature of the system. What could be consid-
ered poor design in another system could conceivably be a misconfiguration in this system.
For example, a DNS server could be misconfigured to the point that it can still read in zone
files but reads A records as TXT records. In this case, there is no way for the traditional
data files to provide the correct behaviour as the underlying system cannot read them in
correctly and represent the data accurately.

7.3.4 The DNS and semantic enhancement technologies

Semantic annotation is the addition of metadata that describes the meaning, purpose or
function of data in a system, where the metadata is specifically associated with data that
already exists in, or is being added to the system. It is of most benefit when new system
development techniques can take advantage of pre-existing systems without having to re-
engineer the legacy code.

A significant advantage of extending the DNS with semantic annotation is that a large
number of applications across the Internet could use this metadata to store or derive in-
formation that would normally be stored out-of band, as text files associated with DNS
stored records, or that may not be obviously related to information that already exists in the
DNS. This would allow the integration of the DNS with other systems that can use or add
annotation to provide additional services to their user community.

Such annotation could allow the integration of the DNS into the framework of the Se-
matic Web. A semantic web user may require more provenance information than is nor-
mally transmitted in answer to a DNS query. Metadata adding such provenance information
at the server itself would allow the providing server to define the provenance information
rather than require a downstream service to add provenance metadata. If the server can
annotate the data then local data modification tools can be produced to automate the anno-
tation process based on user id and timestamps recorded during the editing process. This
is an example of the type of comprehensive data capture that can be required when at-
ttempting to capture the data and the state and reliability of that data during the execution
of distributed applications. Our proposed approach deals with the knowledge that is em-
odied, and realised, by the data while allowing an overall comprehensive data capture
model.

There is currently no mechanism that allows DNS behaviour to be altered on a site-
by-site basis without extensive local re-coding of the source that requires expertise in both
programming and in knowledge of the DNS. In this context, a site is a logical entity that could contain one machine, several machines in one location or the machines of an entire company, spread nationally or globally. It is a close approximation to the notion of a virtual organisation (VO) referred to in Grid computing [69] discussed in subsection 2.3.6.4. The site has a problem-based focus, in that there is the requirement to answer questions within the problem domain of the DNS site’s information resources, and this fits into the problem-focused model of VOs.

Although there has been some work on the domain name system with ontologies, it has focused on providing an ontology for the management of DNS [136] rather than the information DNS contains, which is stored in zone files. There has also been work [137] to provide a virtualisation of DNS to allow the use of RDBMS for storing this data, and hence support queries over this space. The mapping between the DNS and an RDBMS representation, while a possible outcome of what is proposed here, is not the focus of this thesis.

7.4 The two implementations of the DNS in a partial and more developed KR framework

Two distinct versions of the ontologically based approach were produced during the research undertaken for this thesis. The first prototype system [138, 139] used ontologies to provide a means of modelling and representing knowledge domains, with respect to the logical and representational aspects of the DNS. This captured the relationships between data elements and the type and data structures of those elements. It did not provide a user-definable subset of the operational semantics of the DNS to the ontology that would lead towards a less rigid description of server behaviour. These operational semantics describe DNS operation - how is the data to be manipulated, under what circumstances and, in addition, do we add or manipulate metadata? This software is referred to as the $\alpha$-implementation. The second implementation contained the mechanisms required to implement the operational semantics facilities. The second implementation is referred to as the $\beta$-implementation.

7.5 Problems in self-definition

The ability to alter the logical, representational and semantic structure of a running system allows the examination of the benefits and problems of allowing dynamic reconfiguration
and controlled evolution within a server. This leads directly into the question of where, in the system, is safe from its self-modification or its fixed point. As mentioned earlier, the fixed point in a system is the point at which self-modification or evolution is prevented to protect the core function of the system [140], based on earlier work carried out on definitional interpreters [141]. This is similar to the definition of a red line defined in [142] but is not restricted to a strict separation between application and operating system. However, this is a different definition of a fixed point than is used in functional programming [143] and, despite the discussion of the λ-calculus in this thesis, this term is not used with the traditional functional meaning.

The closer the fixed point is to the executing environment, the more the behaviour of a server can be changed although the more risk there is that such modification can lead to system failure. As had been discussed, the fixed point of the system described here sits below the level of a traditional DNS server but, in the model, within the data model section of the mapping. The underlying server may be a simple server that can read configuration files that demonstrate how to become a DNS server and, in turn, then reads server-specific configuration files to describe how to become the DNS server for a particular domain. This discussion of a generic super-server, that is almost completely user-defined while still allowing wide interoperation, is briefly covered in the future work section.

7.6 Developing a semantic model of the DNS

The concept for the semantic model is a separation of the DNS into three distinct aspects: the logical aspect, the representational aspect and the operational semantics of the service. The initial model was a sketched outline of the DNS as a purely logical structure representing the DNS zone file data, with the describing ontology stored in OWL/XML format. Individual types of resource record were sub-classes of the overall concept of a resource record and simple relationships were established representing the restrictions of cardinality found in zone files. Simple logical relationships were explored to define the relationships between domains, zones and resource records. This was never implemented as this it would not provide an extensible platform for further exploration without the design work used for later stages.

The intermediate model built upon this by adding more logical structure, refining the existing logical structure to describe record relationships in more detail and adding a representational aspect to the ontology that captured how data is being structured, stored and communicated between clients and servers. The computational operations required to present this data to clients and read this data from files is deemed to be within the server and
immutable. In other words, this model allowed alterations to data representation and association but made no fundamental changes to the way in which the DNS server presented itself to the world. This was implemented as a Java-based server, the $\alpha$-implementation, that functioned as a basic DNS server with user-extensible data structures, but did not allow user-defined reconfiguration of the operational semantics [144, 138, 139]. This constituted the $\alpha$-implementation.

Our final semantic model added the concept of operational semantics to the tree, along with the functional sub-languages. The simple final model, supporting basic functions and no cryptographic support, was initially implemented as an extension to the previous Java software but was restarted as a new project based in Python to take advantage of the rapid prototyping features and simpler thread model. This implementation, the $\beta$-implementation, allowed the development of DNS servers that could provide novel DNS functions and, more importantly, allowed the development of mutable DNS servers with significantly lowered fixed points.

### 7.7 Mapping the DNS into the model

We describe here the $\alpha$-implementation, as the basic developmental model for our implementation, starting with a description of the record handling requirements of the DNS. RFC 1035 [4] defines the RDATA component of resource records, the component of the record that is returned in response to a query. This is, in turn, embedded in a strongly defined format. All RRs have the same top level format with varying RDATA elements. Resource records are, in turn, embedded in the DNS message format [4] once the knowledge that they contain is projected outside of the system.

Such records have a logical relationship between themselves and the parent notion of a resource record. They also have a structural representation that defines how the value stored in the mapping is to be sent to a resolver or a server. In the case of an A record, this is a 32-bit internet address. In the case of a TXT record, this is a set of one or more `<character-string>`s, where a `<character-string>` is defined as a contiguous set of characters without interior spaces, or as a string enclosed in double-quotes. The behaviour of the server when a request is sent is defined by its operational semantics - how it handles the incoming byte stream and produces a locally coherent request that it can then answer, recode and then send back to the requester. The logical aspects of the DNS map into the logical section of our model, issues with syntax and type map into the structural and representational section and, finally, operational semantics are dealt with in the operational semantics section. Since most of the operational semantics are implemented using the composition of user-defined
and primitive functions this sub-branch is interchangeably referred to as the functional sub-
branch.

7.7.1 Mapping into the Ontology and Data Model

Wherever an ontological specification is made as part of the ‘three branch’ base ontology, a corresponding definition is made in the data model to accommodate the implied knowledge associated with the data. For the most part, this requires the existence of a corresponding type, for representational issues, or an interpreter for any statement that has operational significance. There are two classes of statements that can have an effect upon operational concerns: one is the ontological relationships that link ontologically defined concepts together and the other are instructions that are directed to the data model although they are defined in the ontology tier. The ontological relationships are grounded in the OWL definition and are not further interpreted by the data model, although these relationships may be used in conjunction with other knowledge to affect interpretation. The instructions can change the order in which operations are carried out or can change the context of operations through the composition of different functional blocks. As the ontology specification has no mechanism for interpreting these, all of these must be passed to the data model for interpretation.

Reviewing Figure 6.7, which shows the UML sequence of general operations in the system, we can now produce a new diagram, Figure 7.2, which shows the DNS operations that provide the distributed mapping service mapped into this framework.

What should be immediately clear from the diagram is that the traditional model of DNS mapping queries does not have a one-to-one relationship with the capabilities of the sequence model. There is no correspondence to two of the operations that we have specified as a part of NBIS: the filtering step and the optional store-back step, first identified in Figure 6.6. In the first case, some DNS implementations do offer a limited, system-specific means of filtering output depending on where queries originate from but this is defined in a strictly informational sense. In addition, and as discussed in Chapter 7, the filter module in the UML sequence diagram is capable of providing metadata, as well as data, based on any of the knowledge in or about the store in an agile and flexible manner. The store-back operation is often implemented through the use of external caches, as discussed in earlier sections.
7.8 Describing the operational semantics

The DNS is not purely a simple mapping service. It can function as a distributed associative array, where a query returns the same answer idempotently or as a stateful server that tracks authorisation and access privileges for clients that attempt to download zone files or carry out privileged transfers. In addition, the server can provide alternative servers to query to provide resolvers with more information than can be obtained at the originally queried server. Recursive querying relies upon the server that takes the initial query sending back a list of other servers to query or querying other servers on the behalf of the original resolver. This more complex behaviour is firmly fixed in the existing implementation of the DNS and can be represented as two distinct modes. The first mode presents the information to any and all resolvers, if the information is available. This is the ‘Always return data’ mode. The second mode is more complex and provides a simple choice mechanism. If a certain
condition is satisfied then provide the information. Satisfying the guard statement may also cause a state change inside the server so that it can recognise authentication for a fixed period.

While the server and configuration files can be altered by the administrator they cannot change the basic operation of these two modes unless they change the server code. These are referred to as the Implicit and Explicit Behavioural division of operational activity inside the DNS server. The operational semantics of the server are hidden inside the implementation and cannot be changed without re-engineering. The ontological model allows the explicit statement of operational semantics and allows the explicit, user-driven statement of functions that supercede or add to existing DNS functions. Importantly, these are specified by the user by changing the statement of how the DNS is supposed to function. The initial design of the ontological structure of DNS is shown in Figure 7.3 representing the different behaviours and responsibilities found in each branch. This figure is not an extension of the previous minimal-ontology class diagram and should not be overlaid onto it. This was modified during the implementation phase, leading to the final design shown in Figure 7.8.

Figure 7.3 shows the three branches of the ontology, logical, representational and functional. Each sub-branch shows the subclass relationships between the components of the ontology. For example, the logical sub-branch supports a DNS record type TXT and can provide a record type based on TXT. The representational sub-branch stores the type information for A and TXT records to allow data to be stored and transferred appropriately.
Similarly, the functional sub-branch captures the operational semantics of the DNS server, separating implicit and explicit behaviour, with a further separation between the two return modes identified previously. The box labelled "Doesn't Exist" refers to the fact that there is no mechanism in the current DNS standards that define an explicit behavioural mode for returning one form of data to one client and a different one to another, barring cryptographically signed exchanges. Figure 7.4 shows the operational semantics branch in Figure 7.3 overlaid with Figure 7.1.

The implicit behaviour section uses the composition of fundamental functions, located within the native functions class, with λ expressions to provide the straight-forward mapping functionality as seen in the UML sequence diagram. To capture the more complex operations requires the use of a larger set of the user defined functions and the class hooks to provide better programmatic access to operational semantics, data and metadata.
7.8.1 A language for capturing operational semantics

OWL-DL [124] was the original choice when developing the ontology as this is a ontology language that can be shared across the Internet with guaranteed decidability properties. However, as mentioned earlier, the guarantee of decidability is of little use if the time required to compute entailment regularly results in timeouts and unsatisfied user requests. When developing the model of the operation of a DNS server it quickly became apparent that there were several problems that had to be overcome in the implementation of the operational semantics as an ontology. The over-arching concern was that decidability is an unattainable goal given the need to introduce data model specific elements. Although these could be described in RDF and RDFS, neither of these have bounded computational time for entailment calculations.

Following the development of the intermediate model, the proof-of-concept server implementing the final model was designed with the following points in mind:

1. To reduce problems with computation time, the embedded language elements do not allow recursion. The embedded language loop elements are explicitly unrolled prior to evaluation. If the element could not be unrolled successfully then the entire code fragment is replaced with a null return. This limits the impact of allowing loop structures.

2. The fixed point is set above the cryptographic functions and the strict interpretation of the OWL ontologies describing logical, structural and functional aspects. This is shown at Figure 7.5 and allows experimentation with user-defined operational semantics without having to provide a complete definition - taking advantage of existing cryptographic services.

3. Lazy evaluation is used because of operational concerns and the guarantee of a result. In the operational code, expressions are evaluated only when needed and are paired with a custodian thread to ensure that the expression is evaluated before a defined time or the evaluation is halted and an exception propagated back through the chain. However, at the low level, there is strict evaluation of data streams as the lower half of the data model takes in entire messages before passing them up to the lazily evaluated upper half of the data model.

4. The underlying semantics should be declarative as the entire focus of this project is on removing an imperative focus, of design, of implementation and of operation. The removal of the imperative focus on design allows us to concentrate on what is to be achieved, rather than providing a detailed support mechanism to achieve a fixed
goal. By providing a mechanism to allow for the decomposition of the system into functional elements, and allowing a recomposition mechanism, we gain far more flexibility than we could achieve with an equivalent imperative approach. Although the final system is functional with side-effects, the consideration of the isolation and limitation of these side-effects is essential and is largely achieved through the requirement for any information to be placed in context, and thus grounded, before it can be used.

The underlying technologies referenced in Figure 7.5 have already been introduced. An RDF/OWL parser parses an ontology defined in OWL-DL, interprets the ontology and links the ontology with the user-defined functions and system-defined functions to produce the executing server. The server can receive queries or send requests by using encapsulation of its stored data into the appropriate wire format for the target user. The encapsulator uses a description of the system behaviour to manipulate the data into, and from, the correct format. The role of the ontological specification and data model tiers are clearly labelled, showing the boundary between them is defined as the fixed point of the system.

### 7.8.2 Language Choice

As previously outlined, the operational semantic language to be embedded within the ontology is a modified form of the λ-calculus with no recursion and an encapsulating flow-of-control block structure. We chose this form because it is well-understood, Turing complete and many techniques exist for parsing and compiling the language. The applied λ-calculus allows block control statements as constants and also has a notion of built-in functions. Rewriting rules can be provided to substitute ‘pure’ λ-expressions for these constants if required [52]. The λ-calculus expressions function as anonymous functions that rewrite the DNS data to produce a new byte stream, encapsulated in the established context of a named piece of knowledge that is part of the operational semantics branch of the ontology.

### 7.9 Ontological development of the Operational Semantics sub-branch

The classes and sub-classes were developed based on Figure 7.3. As the DNS must return sets of bytes it makes sense to apply the composition of the modification functions to the byte stream to modify the stream. Although a compositional model is strictly the composition of functions with functions, the initial step of application
Figure 7.5: The location of the fixed point.

is normally subsumed in the new model by the series of functional compositions required to process the stream once it has been rendered into knowledge. $\lambda x.x$ takes the knowledge as a parameter and returns it unmodified. $\lambda x.0$ is the null function, with the understanding that the server would interpret this as the ‘null response’ and encapsulate it for transmission to the client. The logical flow of the encapsulation process is shown in Figure 7.6. There is no set number of encapsulations that are carried out - this is user-defined when the encapsulations are defined and can be modified by redefining information contained in the knowledge domain. Figure 7.2 shows the application of the logical model shown in this diagram to the UML Sequence diagram.

So far, functions have demonstrated that are composed with RR data and then encapsulated for transport. The definition of what happens to data from the point of reading to the point of hand-off to a local area network can be completely defined by the user using knowledge domains. However, the more that the server can be reconfigured in this way, the
lower the fixed point is. Eventually the fixed point sits just above those system primitives that are provided from the operating system and cannot be redefined since the server cannot currently pierce the operating system it runs on.

In this proof-of-concept model the cryptographic functions of the target system are provided by the server itself, written in Python. The decision to leave these functions in the underlying server was made to maintain DNS security and allow the deployment of a test server without adding potential security issues to the list of possible problems.

An example of a filter operation based on source IP address is shown at Figure 7.7. In the example, the record is returned without modification to a requester that is a member of the Internal class but alternative information is returned to a requester that is not a member of the class. Internal is defined as a *classHook* class and, hence, can be called as a function with a query parameter explicitly stated (the store and associative array parameters are implicitly added at the data model level). *ValueQuery* itself has a query parameter as well. The condition is true if *&dns;CurrQHost* has a value and that value can be located as a name of a member of the Internal class. If that is the case then the knowledge is passed through the identity function and is returned unchanged - the querying host is an internal, and hence apparently trusted host, which can receive unfiltered information. If the condition is not true then the value of *&dns;AdvertisedIP* is used to replace the value of *&dns;IPAddress* in the current knowledge fragment being passed around. If there is no value for *&dns;AdvertisedIP* then the *&dns;IPAddress* value will be assigned a null value.
if Internal(ValueQuery(&dns;CurrQHost))
  (Identity)
else
  (Compose ReplaceAnswer(&dns;IPAddress) &dns;AdvertisedIP)

Figure 7.7: An example of the applied \(\lambda\)-calculus.

7.10 Native Functions

The set of fundamental functions provides the core functionality of the system. In the case of a simple and unmodifiable implementation of a target system, the fundamental functions can be tailored to that implementation and not necessarily the small recomposable units found elsewhere. This, however, merely produces a relatively inefficient interpreted implementation of an existing system with a large amount of overhead in ontological processing for no real benefit.

By keeping the fundamental functions small and single purpose, with a set of operations for composing them, redundancy in code is reduced and the possibilities for recomposition increase dramatically. On occasion, certain special purpose functions must be added to the system as the overhead in coding these functions into the ontological layer is significant or the function is likely to be ubiquitously deployed. In the case of the DNS, the cryptological functions that support certain cryptographic transfers were encoded as fundamental functions.

Remotely Installable Native Functions (RINF), discussed in Section 5.4.5, support was added to the implementation to allow the expansion of the native functions found in the ontology so that trusted parties could add new software without needing access to the source directory. This demonstrated two points. Firstly, that the server could support a maintenance protocol in addition to its target protocols, and, secondly, that such a server could be used in a distributed environment without having to provide direct file-level access to systems. A maintenance protocol is, in this thesis, defined as a protocol supported by a system that is only used to modify or support the system infrastructure - it is never a provider or carrier of the information that the system provides to other users. Thus the existence of a trusted channel for adding new software, and changing the way in that the server operates, constitutes a maintenance protocol in addition to any other protocols are supported.

The naming scheme used to identify functions provides context for native functions and is also the first level of a simple protection mechanism. Rather than the entity-based ‘&something’ notation used for establishing context for values in the ontology, functions
are defined as individuals of the nativeFunction class where the name of the function is the name of the individual and the individual has a property hasNativeName where the value of the property gives both the location of the function and its Data Model name. The format of this value is “location::dataModelName” and is deliberately not a legal XML reference to prevent accidental use of native function identifier strings as variables. The location can be any value providing that there exists a corresponding Data Model class name that provides the named function as a method. The name used for all pre-defined native functions for the DNS is dnsServices and, to prevent overwriting of existing method calls, the RINFs were located in a separate class named rinfServices. The RINF class provides an abstraction that gives the appearance of a single class of methods, although these are stored as separate files and are not recompiled together.

These mechanisms provided the basis required to model the DNS in terms of its logical relationships, data structures and (a subset of) its operational semantics within an ontology. The data model interprets the ontology and provides those elements missing from the ontology using native functions. Native functions referred to in the ontology only deal with knowledge, they do not handle data or incomplete information. The low-level bottom-tier data model functions that deal with data cannot be referred to within the ontology as this would breach the abstraction.

### 7.11 Application: Extending the functionality of Internet services

This section has already discussed the underlying technologies implementing the knowledge domain based server, and can now discuss how these can be used to extend the function of an existing server.

#### 7.11.1 Extending an existing service

The DNS, like most widely-used distributed systems, is under a strong modification control mechanism through the use of RFCs and derived standards. This scheme controls which resource records may be defined and how they are defined. This strong control ensures interoperability between different servers and clients. However, under the existing DNS modification scheme, this control also means that an RFC process has to be followed to effect global change and this is a slow process due the requirement to reach a majority agreement within a large, well-represented user base. With a knowledge domain encoding of the DNS system DNS modification need no longer a centralised, rigidly controlled and
slow process while, at the same time, it can still provide a core functionality based upon the RFCs if desired. Two sites can collaborate without having to carry out detailed re-coding and testing and, importantly, completely outside of the RFC mechanism. There is no reason why more resource record types could not be encoded in OWL and used to extend local functionality. An ontologically enhanced server only needs a new ontology to provide additional services to its user base. Potentially, the use of semantically enhanced systems can interact with the ontologically enhanced DNS to make ad-hoc changes as required by the system agents without having to involve any human agents at all. Such a system must employ strict controls and security to avoid compromise or exploitation.

An ontology also provides an excellent mechanism for advertising the structure and relationships of such new resource record types. There is already a defined DNS mechanism for advertising services, DNS SRV [7], and this mechanism provides a means for advertising which new services are available. Since SRV records are already used in the existing DNS, records can be added for servers that can provide ontologies and also advertise new resource record types. This allows advertisements of the new services to be provided in the existing DNS without requiring all servers to be globally updated to handle a new ontological advertisement protocol. The ontologically-based system with local extensions can coexist with traditional DNS services because of the transparency requirements described in RFC 3597 [9]. This RFC removes the requirement for nameservers to have the same software version, or capabilities, as the servers that they are exchanging data with. More importantly, it allows servers to store record data that does not match any resource record types known to them providing that it comes from a known source.

With the introduction of extensions, it is important to maintain the core functionality of the DNS in order to allow interoperability and the continued functioning of one the world’s critical distributed information systems. Our vision, firstly, is of a core ontology that encodes the RFC-prescribed DNS elements. Then, any extension ontologies start from that point and extend the core ontology. Thus two enhanced servers, even with different extensions, can still use the standard elements of the DNS even if they cannot understand each other’s extension model. This is because the server will still respond as if it were an unmodified server and can serve all traditional requests, while being ready to handle requests that only make sense coming from or going to an enhanced server. In terms of the model already shown at Figure 7.3, there is no Choose branch implementation in the explicit server behaviour and, as a result, behaviour does not deviate from the standard. The resource record mappings are also limited to those defined by the standards.

A possible use for such an extension is the provision of additional knowledge channels for local sites. The ITXT resource record is a TXT record that is only viewable from
within a given site, where the concept of *within* is given by membership of a set of trusted IP addresses rather than by using a strict geographical or geometrical interpretation. ITXT records can only be viewed from hosts within the domain for which the nameserver is an authority. The IP address of the host is resolved to a name and checked for spoofing. It is the role of network security to ensure that only hosts that are from within the network appear to be from within the network to the nameserver. Some network administrators already hand-annotate the zone files to include additional textual data but, for security reasons, this information is generally stored as comments in the file rather than data that can be accessed via the DNS. Such information could include the hardware address or office location of a given machine. This type of information could be useless to the rest of the world or a potential security threat if widely available so should not be available outside of a controlled area. However, the use of such control mechanisms are is often governed by local policy and a global decision cannot be made as to whether such a feature should be a required part of the DNS standards. Some implementations of DNS provide such a division of access for a set of resource record types but this then leaves the local providers who do not wish to use this feature with unnecessary, and misconfigurable, complexity in the code base. By unifying the knowledge domain of DNS with the local requirements for that knowledge, the decision is left in the hands of the site managers and is not limited by what a given software provider chooses to implement.

As an example, one existing DNS server implementation, the BIND server, can also provide different responses to a query based on different client source IP addresses. This is not part of the DNS specifications, it is a widespread BIND-implemented feature to address a requirement that is not part of DNS. To provide similar functionality on a server that, otherwise, is completely true to the RFCs would require additional coding. The ITXT record is an example of a simple configuration change that could be implemented at a site without server recoding or the implicit support of one of the larger server providers.

Similar information-associated uses include MTXT for metadata-related text, such as would be used for annotation in the Semantic Web, and CTXT for encrypted text. All three of these new TXT types are immediately useful in any large-machine environment and especially one where Web Services were deployed. MTXT messages are used to provide out-of-band (OOB) metadata to describe an otherwise standards-based DNS message format response and also can deliver the current ontology, if authorised to do so. CTXT allows the delivery of simply encrypted messages, where keys have already been agreed upon between the two participants, and effectively allow ITXT messages to be sent beyond the limiting boundary as a CTXT relationship extends the trust boundary of the system.

The relationships between data can also be exploited in order to automatically extract
information that is not explicitly stated if using a decidable OWL dialect or if working with a decidable subtree of an OWL file. For example, an A record mapping a name to an IP address can have an inverse relationship, using an existing OWL language construct, so that the IP to name mapping (PTR record) can be generated automatically. This is not always a desirable behaviour for multi-hosted machines but, within a site, an administrator could choose to use this OWL-derived functionality. Even with decidable dialects of OWL the more of these that are introduced, the longer it takes to determine the entailment and, even if this is only carried out at startup, the longer the time it takes to deploy the knowledge.

The ontology that was developed through the steps of the implementation is attached as Appendix A. While this ontology cannot, for reasons of space, provide the entire dataset used, it does provide all of the necessary framework and example data to illustrate the final ontology.

### 7.12 Implementation of an ontology based DNS server

The $\alpha$-implementation is seen as a proof-of-concept system and comprised an existing DNS server, adapted to parse XML-based data files and use the existing internal data structures to store the parsed data. This prototype system was developed in Java. In the second phase, implementation of explicit user functions, as much or as little of the existing DNS implementation were used to allow the testing of user functions without crippling the underlying server due to non-existent or incomplete features. The goals of the $\alpha$-implementation were:

- Provide a DNS server that operated from an RDF store.
- Develop an ontology to use with the store data to encode representations and logical relationships between data.
- Provide the ability to add resource record types to a DNS server.
- Provide a matching client that can use the ontology to retrieve the new record types.
- Provide experimental data for the performance overheads of this approach.

Although the Berkeley Internet Name Daemon (BIND) was originally considered at the start of implementation planning, the method it uses for encoding resource record types depends upon compile time pre-processor statements and would have required major modifications to change. Instead, for the prototype, the ‘dnsjava’ Java-based DNS implementation, produced by Brian Wellington [145], was chosen as a basis. This system is a straightforward implementation of the DNS RFCs in Java, originally carried out as a programming
exercise by its author. dnsjava is not a commercial-grade DNS server but, because of the
programming approach taken, it is easy to modify while still maintaining the required DNS
functionality.

The key to the modifications for the $\alpha$-implementation is that dnsjava uses an abstract
Record class that all of the RR types extend. In order to remove the dependency on compile-
time definition of RR types, the server was altered to use a GenericRecord class that ex-
tended Record, and could thus be implemented, but is effectively a blank canvas. After
instantiation, such a record is then modified based on information stored in the representa-
tional ontology to store the correct components of the correct type. Methods were provided
to transfer data to and from wire format, and to and from String format, as per the original
Record. These operated on the assembled component datatype to provide the illusion that
these GenericRecords were identical to the pre-defined records that they replaced.

We modified the storage of RR type numbers so that these were read from the ontol-
ogy, rather than being predefined in a static structure. Type numbers are usually statically
defined but this is unsuitable for a dynamic approach, where RR numbers can be defined
locally and extend the space occupied by the globally defined resource records.

The stages of the implementation for the initial prototype were:

- Produce the OWL files describing the knowledge domains for the DNS and the in-
  stances of the classes giving the zone data for a test domain.

- Obtain an existing DNS server that could be modified legitimately.

- Implement, or adapt, an RDF/XML parser to read in the OWL files.

- Demonstrate correct operation of the DNS using ontological configuration files.

- Collect performance data.

Once the prototype had been used to verify that the $\alpha$-implementation approach was
sound, a new design was produced to include the full range of operational semantic is-

sues raised in previous chapters and to provide the explicit distinction between underlying
programming language semantics, data model and ontological tier. This $\beta$-implementation
guaranteed the separation of data, information and knowledge as detailed in Subsection 2.1.1.
The final ontological representation of the DNS that was implemented can be seen in Fig-
ure 7.8.

This provided simple guard statements, conditional containers and the anonymous func-
tions to support simple operation and still use the cryptographic mechanisms of underlying
server shown at Figure 7.9. This cryptographic inclusion allowed TSIG authentication and
Figure 7.8: The post-implementation model of the DNS with operational semantics

the partial use of DNSSIG verification without having to recode the cryptographic modules from scratch.

The $\beta$-implementation server is written in Python 2.4 and uses the rdflib package [146], version 2.3.3, to provide RDF manipulation functionality and the python-dns DNS library, version 2.3.0, to provide those functions that are not explicitly defined within the ontology or the data model to reduce development time. The stages of the Python implementation were:

- Implement, or adapt, an RDF/XML parser to read in the OWL files.
- Provide a robust analogue for the query operation defined in the model.
- Demonstrate correct operation of the DNS using ontological configuration files.
- Implement the operational semantics knowledge domain in an OWL file.
- Develop the simple command parser and $\lambda$-calculus parser to read in the functions.
- Demonstrate new and explicitly defined functional behaviour.
Figure 7.9: The operational semantics component, showing cryptographic functions

- Add new RR Types for a limited domain and test that they work correctly.
- Develop a limited set of ontologically-enhanced tools that can use the ontologies for increased efficiency.

The simple command parser was initially designed to handle conditional structures of the form ‘if { conditional } then { λ-expression} else {λ-expression}’. The λ-expressions were limited to simple statements on one variable in the first instance to map the entire expression, rather than allow it to be broken into subparts. Using the applied λ-calculus if, then, else were defined as constants with λ-calculus definitions, as shown in Figure 7.7. In addition to the ‘if’ conditional, an ‘if’ statement was also provided using the presentation of three statements, individually enclosed in parentheses, in sequence where the first would be evaluated and the second would be evaluated if the first were true, otherwise the third was evaluated. In this case, the truth value was take to be True or non-Null.

In order to work efficiently with the RDF triple store provided by rdflib, the SPARQL RDF query language is used [147] to provide query functionality over the RDF store. SPARQL uses triple patterns, conjunctions, disjunctions, and optional patterns to support querying and also supports constraining queries by source RDF graph. Results of SPARQL
queries can be ordered, limited and offset in number, and presented in several different forms [147]. SPARQL is currently a W3C working draft but is sufficiently well-defined to be in use in experimental RDF handling implementations. The syntax and semantics of SPARQL allow for mandatory and optional components, that has a substantial impact on the evaluation of queries. If only the AND and FILTER operators are used then evaluation is bounded by \(O(|P|.|D|)\), where \(D\) is an RDF dataset and \(P\) is a graph pattern. However, if the UNION operator is used, then the evaluation is NP-complete [148]. For this reason, queries are restricted to the operators AND and FILTER wherever possible, with a small \(P\).

SPARQL provided the data model with a mechanism to support all of the query operations outlined in the conceptual model.

SPARQL queries can be formed using a combination of variable and literal place holders to limit the queries returned. For example, the query string (‘?a’,”rdf:hasType’,’?c’), (‘?c’,”rdf:hasType’,’owl:Class’) will return values for variables \(a\) and \(c\) where \(a\) has the type of an owl:Class with identifier \(c\). This allows the formation of queries that can locate a node in the knowledge hierarchy in an explicit context - this guarantees that the correct triple is being used for value determination once the value node has been located if the preceding triples are chosen correctly.

We modified the server to check for the existence of functional definitions at start-up. Where the resulting expression can be easily parsed, it is pre-reduced, for efficiency, and the result stored. Thus, if a record always returns data, the implicit Identity function, then this is replaced with the Identity and no further parsing takes place in response to additional queries. Similarly, a record that never returns data has an associated shortcut that ensures that the server never parses it or sends data. Where an expression can have variable outcomes, these are always parsed based on the nature, identity and location of the querying agent - as far as can be determined.

A service resource record type was implemented that, if from the authorised entity, would cause the server to reread its configuration files. This can be used remotely and allows for ease of testing but is also an example of the type of meta-operation that can be captured with the operational semantics specification.

7.13 Ontological Reasoning and Entailment

As the target system had to act as a well-behaved member of a larger system, it is important to ensure that any additional functionality is invisible to an unenhanced system member using the implementation. To that end, any reasoning support or entailment determination has to be carried out with an upper time bound of \(T_{TO}\). \(T_{TO}\) is the longest time that a query
or other operation can remain unresolved before the requester will terminate the query.

Although portions of the ontology can be represented as an OWL DL ontology, there is the potential for the introduction of non-DL elements through simple misconfiguration (in the event of a missing type declaration) or through the use of features that are only available in OWL Full. As has been discussed, regardless of which approach is employed, the time to complete entailment computations is still significant and, in the case of OWL Full, potentially unbounded.

The management of ontological reasoning and entailment is handled with two mechanisms. The first of these is the species identification of the OWL dialect in use through submission of the ontology to an external species checker. The external species checker initially used was available via the WWW at The WonderWeb OWL Validator [149]. This required additional network traffic to carry out classification and, during network outages or when the network was under high load, had an unacceptable failure rate due to communication problems.

In order to maintain the species classification feature, the location or development of a local species checker became a requirement of the project. Due to the time required to write a suitable species classifier, we decided to use an externally available piece of software instead. The Pellet classifier [150] was integrated into the project and the Pellet classifier was used from that point on for species classification. This classification verified that the XML for the document was well-formed and also placed the ontology into one of the three categories - OWL Lite, OWL DL or OWL Full. The second mechanism was a computation termination control that would allow a maximum execution time of $T_{TO}$ before terminating the computation. In the event of an OWL Lite or OWL DL classified graph, or subgraph, it is possible to complete the entailment calculation in a ‘background’ mode although contact with a requester would still be severed. In the event of a repeated query, the information may then have been added to the knowledge base through the ‘background’ completion of the computation. Species identification is carried out prior to system startup, with a knowledge element containing the current ontology’s classification added to the ontology. Removing the need for a local species classifier reduced system complexity dramatically. An implementation of the $SHOIN$ tableaux algorithm [150] was used initially but, in order to employ the clashing axiom detection aspects, the Pellet CLI was used to determine entailment for OWL DL fragments of the ontology in later versions of the $\beta$-implementation [128].

Subgraphs are also derived from the overall Full ontology, using import statements to build from a DL ontology core and building Full extensions. This allows limited reasoning support within the subgraphs although the results of entailment calculations in a subgraph
applied back to the whole ontology could not be guaranteed to be valid without rerunning
the entailment computations and, potentially, never terminating. A heuristic approach pro-
vides some basic means for DLizing the ontology if it carried a Full classification but these
are not guaranteed to produce a DL ontology.

The heuristics employed are designed to detect for, and correct, simple accidental OWL
Full statements that have entered an otherwise DL ontology, as well as the removal of cer-
tain aspects that can not be expressed in OWL DL. The heuristics were developed in part
by consultation with the OWL test cases [151] and Appendix E of the OWL Web Ontol-
gy Language Reference [152]. These heuristics form an important part of the iterative
design process adopted as these address our requirement to simplify OWL-Full ontologies
to a recognisably OWL-DL format where such a transformation could be carried out au-
tomatically and without changing the truth value of the ontology. This ‘DLization’ then
allowed the use of automated tools for entailment checking and the inference of implicit
information.

The correction heuristics are:

- Check that the document does not use any URI references starting with the prefix
  http://www.w3.org/2002/07/owl# except those found in the RDF Schema for OWL.
  This corrects for misspellings in the namespace that, while trivial, would stop formal
classification to OWL DL.

- Requiring explicit type information for all URI references that are owl:Class or
  owl:Restriction and parsing the ontology to provide the correct type. While all of
  the SAMPAN aspects of the system have a strongly typed nature, because of the in-
formation layer requirement for type, some of the supporting OWL structure could
conceivably be untyped and hence prevent correct classification.

- Errors in vocabulary were corrected from rdf: to owl: where such vocabulary existed.
  Some vocabulary elements exist in both RDF and OWL but, in OWL DL, the use of
  RDF constructs is heavily restricted.

- The detection and correction of malformed expressions based on a template set of
  ‘good’ OWL structures. This corrected for misshapen classes and incorrect relation-
ships between entities.

The ‘DLization’ heuristics were targeted at OWL Full structures, such as the use of
classes as instances, and aspects of the ontology were either removed or simplified to pro-
vide a DL fragment in the area of interest.

These heuristics are:
• Localization, in the terminology of the OWL abstract syntax and semantics document, by providing assertions that then allow OWL DL entailment to be established.

• Removing any OWL Full only vocabulary and, if possible, substituting an OWL DL equivalent based on a structural lookup table but, far more frequently, removing the Full element from the ontology, as well as any references to it.

• Removing an OWL DL legal vocabulary that was being employed in a way that should only be used in OWL Full. (For example, using the inverse of property characteristic on a datatype property.)

7.14 Policy specifications

The operation of any service is governed by more than its performance characteristics and transformations. Almost all systems have an additional, often implicit, policy layer that dictates how such a system provides its functions. Such a policy layer dictates how the service is provided, rather than what it actually does. In the case of an SMTP-based mail server, it is a local policy that dictates the provision of six to ten concurrent service threads, rather than the specification of SMTP. Each of the threads is the realisation of the SMTP standard but the server will still be an SMTP server with a single active thread. The policy layer controls the operation of the service or system but is orthogonal to the service being implemented.

We placed additional knowledge into the system regarding the operation of the system itself. This included simple policy information such as the number of servers that should be waiting on a port and, hence, the number of concurrent processes that could be served. In order to check this, DL subgraphs were produced specifying the conceptual framework for policies and a running copy of the ontology contained instances of these concepts. As these are DL, policy conformance could be determined through entailment checking.

The underlying threads that implement servers are all provided as entities in the data model. This means that, as they cannot be classified as knowledge, they are not visible in the ontology except as implicit analogues to the operational semantics strings that handle input. Despite this, the knowledge controlling them is contained in the ontology and may be used in other places in the system, including in the provision of system metadata. It is, therefore, important that these data model level entities maintain the separation of data, information and knowledge used elsewhere in the system.
7.15 Aggregating metadata and data

The primary advantage of increasing the interoperability of NBIS is that new responses can be assembled from multiple sources, based on a single query. In this section, we discuss the benefits of doing this and some examples to illustrate how our implementation provides this advantage.

The server can assemble data from multiple sources if the knowledge required is either contained in the store or a native function is provided that can deliver the knowledge from that source. This allows the assembly of multiple data streams, and potentially metadata streams, into the one reply to a query. Figure 7.10 shows a simple DNS query, with the division between user and system space, as implemented in standard DNS systems and the standards-based mode of the new approach.

Using different protocol specifications, defined in the ontology, the DNS data stored in the store can be delivered using DNS Message Format or other formats, such as XHTML or SOAP, for example. Figure 7.11 shows an ontological definition of the information being used as the source for responses to a standard client and to a Semantic Web client.

E-Science places great weight on the provenance of information - where it came from, the date and time at which it was obtained and other metadata associated with the knowledge itself. These epistemological concerns can be met by sending additional data, along with the query response, that provides this missing data. Figure 7.12 shows an aggregated data stream response that uses knowledge about the system to augment the query. The knowledge of which version of the zone file is being used is stored as knowledge as part of the start of authority (SOA) record associated with every DNS zone file. An e-Science-oriented client can request this metadata as part of a query operation so that the user then
has a record of the version of the ontology that was in use at a given point. To eliminate issues in clock synchronisation, a native function can also provide ephemeral knowledge in the form of an NTP timestamp that is fed back to the client.

If the information on what has been sent to the client is being stored, then the ephemeral timestamp, which has no true meaning in the system, can then be stored as knowledge because, in the context of the cached response, it has a knowledge value as the time that was passed to the client as the current time. Thus, in a very limited context, it has a knowledge value and so can be stored but, if no caching is being used, this value would be disposed of as, without the context of the cached data, it is meaningless. From this, we can conclude that even ephemeral data is worth rendering into knowledge providing that we establish a context within which it has worthwhile knowledge value.
7.16 Ontology support in the client

In order to use ontological concepts that are defined for the server, the client must also use the ontology in some way. A client does not necessarily have to use the entire ontology in order to have sufficient context to ensure conceptual alignment. As we will discuss, a DNS client only needs to know resource record numbers to access the correct RRs. It has no need for a detailed comprehension of the underlying ontology that allows the manipulation of these numbers.

In the case of the DNS, a simple client just needs the mappings for the new resource record types and the bare minimum of ontology that defines the basis for those resource record definitions. In more formal terms, the client requires the smallest set of ontology elements that entail the concepts of the resource records. Given that the reading in of an entire ontology and its assembly into the store format will take increasing time with increasing ontology size, shrinking the ontology to the smallest possible set reduces the startup time for the client. This, however, assumes that the client does not have any operational semantics specified inside the ontology as this dramatically increases both the size and complexity of the ontology, hence affecting loading time.

Where a client does have operational semantics requirements, the target defining ontology is imported and is then extended with local extensions to define local start symbols, operational semantics and native functions. Local native functions defined in the client must use a name other than those that are defined in the imported ontology, to prevent accidental overwriting. The parent ontology can also label classes as immutable to prevent them being accidentally overwritten at any point in the tree below the labelled point. Immutability is to be avoided in general, as it encourages users to develop their own local copies of an ontology that are not linked to the parent and may diverge over time. Immutable points should only be placed on tree points that are standards-locked and, hence, slow to change.

It is not essential that the client have any knowledge of an ontology. A standard client for any of the protocols supported by the server can continue to exchange messages with the server but, without the ontology, it cannot use any of the extended functionality. Regardless of how a client uses the ontological approach, the standard behaviour should always be supported as the standards-based mechanism must always be guaranteed to succeed if the system is to accurately model the standard.

The tolerance for computational overheads is significantly lower in the client than the server, as the server can absorb start-up costs if the time between restarts is large. In the case of a client, the start-up costs are incurred every time the client program is executed,
as with the server, but the mean lifetime of clients is less than that of the server and, as a result, the time spent in pre- and post-activity processing becomes significant.

Two mechanisms were used to cut down on client processing costs. The first is, as described above, reducing the size of the ontology. As will be seen in the results chapter, the smaller the ontology, the less time it takes to load the RDF store, as well as seeing a decrease in overall search time. The second approach employed a client store that was loaded on first execution by a client and remained in operation, although quiescent, for a defined period that is far longer than the usual client execution time. This second approach required that the persisting store poll the server before executing a user query to ensure that the current ontology is still valid, based on timestamp data. If the ontology had changed, then the user incurred the longer start-up time as the ontology was downloaded to the client and reloaded. As will be seen in the results chapter, this incurred an increased cost in the event of an ontology change as the original store had to be deleted and then reloaded from the new ontology. The final version of the persisting store client supported a periodic poll that was carried out even when users were not using the store so that any upgrade could take place while users were not actively connected.

7.17 Summary

The $\beta$-implementation server implements the three-tier software architecture and, using the skeletal base ontology as a starting point, provides an implementation of a simple DNS server that is capable of performing the usual DNS operations but, at the same time, implements additional record types that are not in the same standard. Through the use of a shared ontology, whether as a whole or as a fragment, between client and server, it is possible to have a dynamic and evolving system that is a superset of standards-based behaviours.

In the next chapter, the results of operations carried out on the server will be discussed, showing the experimental composition for both prototype and proof-of-concept servers. Complexity analysis will be used to show the effect of various sections of the code and why the expected results and obtained results are within expected parameters.
Chapter 8

Results

This chapter provides the details of the experiments undertaken with both of the developed servers, the $\alpha$-implementation in Java and the $\beta$-implementation Python-based server. The nature of the experiments carried out differ in detail and approach between the two servers because the $\beta$-implementation has a far greater development of the ideas contained in this thesis. As a result, the experiments that were valid for the $\alpha$-server tests could be carried out on the $\beta$-server but the majority of tests and, in particular, the details of the tests carried out on the $\beta$-server could not be applied to the $\alpha$-server.

The chapter opens with a discussion of the experimental background for the two systems and then provides a detailed discussion of the experimental approach, with the results of experimentation. We develop a complexity based framework within which we analyse and justify the results.

8.1 Experimental Outline

There are two major reasons for undertaking a new approach to solving an existing problem. The first is to provide a mechanism to solve the problem more efficiently or in a shorter time. The second is to provide an alternative mechanism that provides all of the functionality of the original approach but provides additional benefits. As discussed in the Chapter 7, this thesis has taken the second approach.

In order to validate our new approach, the following steps were taken. We:

- Show that the server can function as a member of the target system.
- Show that the additional functionality does not impair the core functions.
- Show that the additional functionality works.
• Provide information as to the scale of computational overhead required for the additional functionality.

There were two distinct systems produced. The $\alpha$-implementation provided some of the functionality of the new model, as discussed in Chapter 7, and is implemented in Java. The $\beta$-implementation implemented all of the model and is implemented in Python. There is, therefore, no simple way of comparing the performance of the servers with each other as they implement different models in different programming languages. Hence, we show that they can be analysed within an underlying complexity structure and compared in terms of their complexity. We also compare their performance against a target implementation of the standards-based NBIS that does not use any of the new techniques or ontological representation. However, such comparison is, by definition, limited as certain features only exist in the ontologically extended systems and cannot be compared with an un-extended system.

As has been discussed, we used the Domain Name System as the NBIS that provided our target for the implementations. Our earlier experiments with the $\alpha$-implementation were compared against a Java-based DNS client and server combination that implemented none of the model. Similarly, the $\beta$-implementation is based upon a Python-based DNS client and server combination that implemented the standards without any aspects of the model. Different, language-specific, DNS libraries were used for each implementation but the comparison clients and servers for each implementation type used the same set of libraries as the enhanced clients and servers. Thus, each implementation could be compared to the performance of an unenhanced system while taking into account language-specific timing and overhead issues, as well as any implicit timing and overhead issues in the libraries. DNS system utilities from the test platform operating system could also be used against either enhanced or un-enhanced servers. We developed a common DNS test structure for use in both sets of experiments. This is discussed in the next section.

8.1.1 DNS structure

The overall test structure used for testing SAMPAN nodes in a DNS environment is shown at Figure 8.1. DNS employs a system of primary and secondary nodes, replicating the information contained inside a domain, with a set of referring nodes that know the location of the authoritative hosts for all of the nodes which they contain. To provide a starting point for searching there are a number of root nodes that provide the information on where to find the top-level domain (TLD) servers so that further queries can be made. The breakdown of a DNS name into the components and the servers that provide the information is found at
Figure 8.1: An example DNS structure showing the location of ontologically enhanced nodes

Figure 8.2.

8.2 Testbed Environment

The experiments were carried out using the following platforms:

- Apple iBook G4 1GHz PowerPC processor, 640MB of RAM, 512KB L2 Cache, OS X 10.4.8
- Apple Desktop G4 800MHz PowerPC processor, 512MB of RAM, 256KB L2 Cache, OS X 10.4.8
- PC 2.4GHz processor, 1GB of RAM, 512KB L2 Cache, Linux 2.6.9-42.0.2.ELsmp
- Apple MacBook Pro, Intel Core2Duo 2.33GHz, 2GB of RAM, 4MB L2 Cache (per processor), OS X 10.4.8.
Figure 8.2: The delegation of name domains to servers in the DNS

The first three machines were used to determine interoperation and to provide the DNS model from the previous section. All timing information was collected from the iBook G4 and, as a result, all queries were initiated from this host. The fourth machine was used towards the end of the project for the ‘faster machine’ testing discussed in Section 8.5.3.

The early experiments were initially carried out on the iBook and Apple Desktop under OS X 10.3.9. All $\alpha$-implementation experiments were carried out under OS X 10.3.9. All other experiments were run under OS X 10.4.8. A test rerun of the $\alpha$-implementation experiments under OS X 10.4.8 did not show a significant improvement in performance - the resulting improvement was of the order of 1 millisecond, which is the quantum of measurement. Because of this, the experiments were not rerun in entirety.

### 8.3 $\alpha$-system experiments

The $\alpha$-implementation dealt primarily with the logical and representational aspects of the DNS. However, we incorporated some small aspects of the operational semantics mechanism to allow the definition of new resource record aspects, which could then be supported
In order to test the client-side aspects of the approach, we developed an ontologically based version of the domain information groper (dig) tool. dig is used to provide a mechanism for querying and testing DNS entries. Our dig variant derives its knowledge of the DNS from ontologies, and can access shared ontologies to determine the DNS extensions in use by an enhanced server. Performance information was collected by using the dig tool provided with OS X 10.3.9, also the ontologically enhanced dig tools, and performing queries against an enhanced server and a standard server. dig performs DNS lookups and displays the answers that are returned. The operational semantics associated with certain requests were changed to demonstrate the impact of local-site customisation for clients that were aware of the change and for clients that were unaware of the change. The transformation between a recognised DNS request and the returned response is referred to as the functional mapping.

The test environment consisted of a mix of standard and enhanced servers and clients. The goal was to compare the performance of the interoperation of the two types of software, and demonstrate their successful interoperation. We carried out four experiments, with experiments 1 and 2 separated into two further divisions. Functional mappings could not be changed on a standard server, which is why experiments 3 and 4 do not have the ‘a’ and ‘b’ variants. Each experiment was conducted 1000 times and the mean and standard deviation were calculated. Table 8.1 shows the collected performance measurements.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Average Time (s)</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>10.04</td>
<td>0.067</td>
</tr>
<tr>
<td>1b</td>
<td>0.085</td>
<td>0.017</td>
</tr>
<tr>
<td>2a</td>
<td>0.922</td>
<td>0.027</td>
</tr>
<tr>
<td>2b</td>
<td>1.124</td>
<td>0.046</td>
</tr>
<tr>
<td>3</td>
<td>1.092</td>
<td>0.024</td>
</tr>
<tr>
<td>4 (baseline)</td>
<td>0.089</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 8.1: Performance measurements for testing. The implemented system is designed to show that our approach can maintain the defined DNS functionality while allowing ease of extension in a way that does not break the centrally defined authoritative model.
had been instructed to return a malformed response and, after 10 seconds, the query times out, as no servers have given a response. Experiment 1b shows the successful interaction of modified and unmodified systems using a standard dig tool with an enhanced server and requesting an A record for which records existed. The functional mapping was set to identity.

Experiments 2a and 2b show the function of the completely enhanced system, with both client and server having full access to the functional specifications of the stored records. In 2a, the client does not send a request because the dig tool can read the server’s ontology, determine that the server will not send an answer and, hence, not send a query that will receive no answer. In 2b, it can send a request and does receive an answer, as the functional mapping reverts to the identity function. It takes approximately 0.6 seconds for the ontologically-enhanced, Java-based, dig tool to start, read in the ontology and then start processing the user request and, taking this into account, it is clear that choosing to not send the request improves the efficiency of the client/server communication by approximately 100%. In both experiments, an ontologically enhanced dig tool was used with an enhanced server. In 2a, the functional mapping was set to null and in 2b the functional mapping was set to identity.

Experiment 3 shows the overheads of using the enhanced dig tool with a standard server. An A record was requested for which records existed.

Finally, experiment 4 is a traditional DNS client communicating with a traditional DNS server. A standard dig tool was used with a standard server and an A record was requested for which records existed. Performance is comparable with the performance of a standard tool with an enhanced server, as in experiment 1b.

From this, it is clear that the choice of client and the implementation of that client may have some effect upon the overall time taken to resolve a query but problems introduced by misconfigured servers are far more significant.

Correcting such a misconfiguration in a standard server would take considerable time. Using the enhanced server, a problem causing a 1000-fold delay can be resolved in a matter of seconds without significant performance loss.

We will now discuss the time complexity of queries in the system in order to provide a basis for discussing the results obtained from the α-implementation and to lay the foundation for a comparison of these results with results from other experiments. We will discuss the contributors to complexity inside the server, identify what these contributions are and then build to a description of the dominating complexity factors inside the server.
### 8.3.1 Time Complexity of Queries

The complexity issues for the $\alpha$-server are divided into two groups: the overheads of queries into the RDF store and the complexity of the general operations. In the $\alpha$-server, the RDF store is not a separate entity and is logically represented by data structures associated with DNS elements. Once the RDF structured configuration files had been parsed in, these are stored in a tree-based representation for ease of searching. We use an AVL binary search tree storing multivalues to assist in locating entries. Two distinct trees, one indexing on name and one on IP address, are used to store the data. Because an AVL tree is employed, lookup, insertion and deletion all take $O(\log n)$ time where $n$ is the number of nodes in the system. In this case, the number of nodes is directly equivalent to the number of records that are being provided within the representation of the DNS domain.

This then leaves the overheads of the complexity introduced by general operations. Most of the operations are $O(n)$ where $n$ is the number of possible operations or mappings requested by a query. However, the number of mappings is at worst the same magnitude as the number of records stored and, in large domains, the number of records is far greater than the number of operations. Cryptographic operations are the obvious exception as they are far more complex. We do not provide complexity analysis for these, as these cryptographic functions are defined externally. This separation of functionality in the implementation is discussed in Subsection 7.8.1.

The major source of introduced complexity arises from:

- The XML parser that passes information into the AVL tree. The parser is LALR(1) with a potentially exponential complexity in generation but, due to the very small size of both grammar and input sequence, this is a very small component of overall time. To allow for the greatest flexibility in changing the operational semantics of the implemented system, the grammar is regenerated at each server initiation. The parsing time of a simple grammar such as this is far less complex and takes place in linear time as it relies on table look-up. Measurements of parsing time show that the sequences are being decoded in linear time based on the number of elements in the string.

- The transformations to wire format. These are also small due to the average size of internet messages in the DNS - the vast majority take place over UDP so this limits the maximum message to 512 bytes. The wire transform is at least an $O(n \log n)$ operation as it conducts a total of $n$ searches over the AVL tree to locate the correct element to assemble the output message. In this case, the number of items in the message and the number of records are at least of the same magnitude.
Thus, it is only when the server is under extreme load that anything other than the number of records has a significant impact upon the performance of the server. However, when the number of messages required simultaneously starts to approach the number of records in the database, the complexity tends asymptotically to $O(n^2)$, where $n$ is the number of records. If a large number of parser operations is needed then the entire computation will be dominated by the asymptotic $O(2^m)$ complexity of the LALR parser stage, if the grammar is re-generated on each execution and $m$ is the number of elements of the language. However, if parsing takes place without parser generation, then experimental data shows that the parser is bounded in $O(m)$. It is for this reason that the XML is only parsed once at the start of server execution and, also, the $\lambda$-based mode in this server is pre-compiled to a simple execution form that could be looked up at a linear complexity cost.

This leaves the overall complexity of the $\alpha$-implementation server as $O(n^2)$ in the worst case scenario, with a general operational complexity of $O(\log n)$ where there are a large number of stored records compared to incoming requests. In both cases, $n$ is the number of stored records in the system.

### 8.4 $\beta$-system experiments

The $\beta$-implementation employed a significantly more complex ontology than that used for the prototype, as the operational semantics specification became more complex and more explicitly located in the ontological tier rather than in the data model. While this introduced increased overheads, it also provided more functionality and flexibility. This section is divided between a discussion of the results of operations carried out to show continued DNS emulation in the new system and a more detailed discussion of the individual contributions of system components to the performance figures.

The same experiments were run as for the $\alpha$-system, with a new version of the dig tool written in Python and the new server providing a far more complex model of the operational semantics. The results of the experiments are shown in Tables 8.2 and 8.3. The Average Time figure measures time from the start of the query to the processing of the response by the client for presentation to the user.

#### 8.4.1 Multi-protocol server

The $\beta$-implementation provided a far more detailed set of operational semantic manipulation tools and, as a result, additional tests were employed to take advantage of this.

Experiment 5 consisted of two clients: a DNS client that has ontology parsing and
interpreting and can access the defining ontology and a SOAP-based client that requests
DNS messages inside SOAP wrappers to pass between Web Services. The SOAP format
was added after the initial work had been done on the system and required minor additions
to the format ontology and a link to the internal format. No server recoding is required. A
third data set is included, which shows the performance of unenhanced client and server
performing the same operations, for comparison. The third data set measures time spent to
respond to DNS message format requests on the standard port, as it could not respond to
SOAP format messages or messages on other ports.

In the first experiment, DNS message format is available on the standard DNS port (53)
and an XML 1.0/SOAP 1.2 format is available on an otherwise unassigned low port (60).
Clients can request data stored in the DNS server through either DNS message format or
SOAP format. Since the requests are being served on separate ports there is no overhead
in determining which message format is being used. Testing was conducted of the perfor-
mance with 1000 trials and the mean and standard deviation were calculated. The clients
and the server are all Python-based and executing on the same machine so there are no vari-
ations due to network traffic. Figure 8.3 shows the collected performance measurements
for the time taken to process the client request and return a result to the client. Client pro-
cessing times are not reflected in this figure. All results are within one millisecond at the
95% confidence level. To clarify the differences between results, the time taken to load the
ontology for the server was removed from the final figures. All other ontologically depen-

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Average Time (s)</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>10.08</td>
<td>0.042</td>
</tr>
<tr>
<td>1b</td>
<td>0.517</td>
<td>0.021</td>
</tr>
<tr>
<td>2a</td>
<td>8.901</td>
<td>0.044</td>
</tr>
<tr>
<td>2b</td>
<td>9.430</td>
<td>0.032</td>
</tr>
<tr>
<td>3</td>
<td>8.957</td>
<td>0.082</td>
</tr>
<tr>
<td>4 (baseline)</td>
<td>0.089</td>
<td>0.007</td>
</tr>
</tbody>
</table>

Table 8.2: Client-side Performance measurements with ontology load

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Average Time (s)</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>0.426</td>
<td>0.019</td>
</tr>
<tr>
<td>1b</td>
<td>0.425</td>
<td>0.020</td>
</tr>
<tr>
<td>2a</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>2b</td>
<td>0.426</td>
<td>0.021</td>
</tr>
<tr>
<td>3</td>
<td>Not available</td>
<td>Not available</td>
</tr>
<tr>
<td>4 (baseline)</td>
<td>Not available</td>
<td>Not available</td>
</tr>
</tbody>
</table>

Table 8.3: Server-side Performance measurements without ontology load
<table>
<thead>
<tr>
<th>Client type</th>
<th>Mean time to process query (s)</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard DNS</td>
<td>0.425</td>
<td>0.018</td>
</tr>
<tr>
<td>Enhanced DNS</td>
<td>0.426</td>
<td>0.020</td>
</tr>
<tr>
<td>SOAP</td>
<td>0.497</td>
<td>0.012</td>
</tr>
</tbody>
</table>

Figure 8.3: Multi-format service on separate ports.

<table>
<thead>
<tr>
<th>Client type</th>
<th>Mean time to process query (s)</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard DNS</td>
<td>0.438</td>
<td>0.015</td>
</tr>
<tr>
<td>Enhanced DNS</td>
<td>0.440</td>
<td>0.021</td>
</tr>
<tr>
<td>SOAP</td>
<td>0.508</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Figure 8.4: Multi-format service on a shared port.

dent elements, such as interpreting the ontology and retrieving items from the store, were included. The average time to load the test ontology once is 8.891 seconds.

Requests are serviced in very similar times since both requests require translation from wire format to internal query format and then, once a result has been obtained, translation of the answer back to wire format. The slightly greater time for the SOAP requests arises from the requirement to embed a well-formed DNS message inside a SOAP wrapper.

In the second experiment, both wire protocols are made available on the same network port - DNS (53). The same requests are sent for all three clients over 1000 trials under the same conditions as the first experiment. Figure 8.4 shows the collected performance measurements.

A shared-port multi-format system must detect the correct format as the message arrives, rather than depending on the incoming port for format information. The testing code adds a small amount of overhead and these overhead increases apply to all queries since now everything must be tested prior to conversion to internal format.

The more complex the shared-port specification is, the longer it will take to service requests. Wire protocols that are similar will make this procedure harder and, in some cases, distinction between alternative protocols will not be possible and separate ports may have to be used. For example, suppose format TestA is defined so that four data values are represented as four bytes where the bytes can have any values. Further, suppose TestB is defined with two data values represented as two two-byte groups where the bytes can have any value. There is no test to determine whether a four byte message is of TestA or TestB format with reliability.
8.5 Breakdown of performance components in the $\beta$-implementation

From implementation section 7.4, it can be seen that a SAMPAN client and server perform more operations than a traditional client or server for the DNS. For the client and server, the complete set of operations are shown in Table 8.4.

<table>
<thead>
<tr>
<th>Client Operations</th>
<th>Server Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load ontological framework</td>
<td>Load ontological framework</td>
</tr>
<tr>
<td>Receive a byte stream from the user</td>
<td>Read DNS data files and populate store</td>
</tr>
<tr>
<td>Decode byte stream</td>
<td>Wait for client requests</td>
</tr>
<tr>
<td>Determine transmission format</td>
<td></td>
</tr>
<tr>
<td>Encode and send</td>
<td></td>
</tr>
<tr>
<td>Wait for reply</td>
<td></td>
</tr>
<tr>
<td>Decode reply and return the result</td>
<td>Transmit result to client in correct format.</td>
</tr>
</tbody>
</table>

Table 8.4: Client and server operations.

The additional overhead for the client consists of the loading and interpretation of the ontology to transform user input into knowledge for transmission and, when receiving data, transforming knowledge to the final delivery format.

While most of these steps exist in a traditional server, with the exception of the ontological bootstrapping, it is the amount of overhead caused by the ontological interpretation that has the most significant impact on performance.

The following names are used to identify key points of the operation of client and server. These will be used in the detailed performance figures and graphs to show the contribution of each sub-operation.

- LOAD: Operations required to load the ontology into the store. In the case of the DNS server, this load time will also include the zone data for the DNS zone in use and
these zones are loaded once for server instantiation. Traditionally, LOAD operations are incurred once for every server startup or data file refresh.

- **STREAMREAD**: Operations required to receive or transmit a stream of data. This does not include the time taken across the network.

- **DECOMPQ**: Operations required to encode or decode a query.

- **DECOMPUQ**: Operations required to encode or decode user input at the client. This is a client only component.

- **RESOLVEQ**: Operations required to carry out a DNS resolution based on a query. This effectively represents the overheads of consulting the stored operational semantics.

- **DECOMPA**: Operations required to encode or decode an answer packet.

- **IDPROT**: Operations required to identify the protocol.

- **ANSWERQ**: This is the time taken to retrieve the correct answer once query decomposition had taken place.

- **MESSAGEADD**: Operations required to add a message, in knowledge format, to the store. This applies to the server only. MESSAGEADD operations simulate caching in the DNS environment and provides a component of provenance establishment for external clients. This is also applicable to any internet-based system that has a read/write mechanism implemented for its data store and allows client-based update.

- **FILTER**: Operations required to carry out the post-store actions check.

Thus the server operation can be represented as LOAD, STREAMREAD, IDPROT, DECOMPQ, RESOLVEQ, ANSWERQ, MESSAGEADD, FILTER, DECOMPA, STREAMREAD.

The performance figures listed also show two other operations, TREE and DESTROY. TREE represents the operations required to locate elements in the store-based tree representation. TREE times refer to the time taken to walk the entire tree and represent a worst case time. DESTROY operations remove all triples from the underlying store. This simulates the effect of a destructive reload on the database and is added to show the increasing impact of triple-store size on performance.
The client, which is logically simpler, can be represented as LOAD, STREAMREAD, DECOMPUQ, IDPROT, DECOMPQ, STREAMREAD, STREAMREAD, DECOMPA, DECOMPUQ. Where the user format is simple or is a fixed format, the DECOMPUQ items can be ignored and the operations become LOAD, STREAMREAD, IDPROT, DECOMPQ, STREAMREAD, STREAMREAD, DECOMPA.

A key assumption is that the client that requests information in a given transmission format will receive the answer in the same format. The rationale for this is that the transformation from user data to knowledge, via wire format, is established for the outbound leg. As the client will have to decode the response, there is nothing to be gained from requesting the information be transferred in a different format unless it is to be immediately passed to another client or system that requires a different format. In the latter case, it is more sensible to request that the response, in an alternative format, be sent to the third party directly. The default behaviour of this system is to respond to clients in the format in which the original query was transmitted.

Two experiments were carried out to determine the impact of each of the identified aspects and their contribution to overall execution time. In order to do this, a simulated query and response were delivered to the client and the server and the software was instrumented to provide detailed timing information. In order to differentiate these experiments from the previous set, they will be referred with a leading 2. Hence, the first experiment of the second set is 2-1.

### 8.5.1 Experiment 2-1

Experiment 2-1 measured the individual contribution of each aspect of the ontological mechanism and its impact on the operational test ontology. The operational test ontology contains all of the necessary concepts to provide the functions of a DNS server but contains a subset of the instance data used. The test ontology can answer the minimal set of user queries and tests logical, representational and operational semantic capacity. As store size has a dramatic effect on performance, this ontology provides the best case performance for an ontology that provides the implementation of the model with the smallest possible data set. In this experiment, the following measurements were made:

- LOAD
- TREE
- DECOMPA and MESSAGEADD
• DECOMPQ, RESOLVEQ and ANSWERQ
• STREAMREAD
• IDPROT
• FILTER
• DESTROY

The operations were applied in a set order, with a fresh instantiation of client and server each time for worst-case store load performance. Where the server persists between operations, there is no LOAD overhead but the test data changes to ensure that a change takes place in the knowledge store. The set order had to be followed as a DESTROY operation renders the store inoperable until a LOAD operation refreshes the contents. This set was tested 1000 times and the results averaged. The resulting values are shown in Table 8.5.

<table>
<thead>
<tr>
<th>Aspect name</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>8.891</td>
</tr>
<tr>
<td>TREE</td>
<td>1.342</td>
</tr>
<tr>
<td>DECOMPA</td>
<td>0.086</td>
</tr>
<tr>
<td>MESSAGEADD</td>
<td>0.196</td>
</tr>
<tr>
<td>DECOMPQ</td>
<td>0.068</td>
</tr>
<tr>
<td>RESOLVEQ</td>
<td>0.023</td>
</tr>
<tr>
<td>ANSWERQ</td>
<td>0.020</td>
</tr>
<tr>
<td>STREAMREAD</td>
<td>0.001</td>
</tr>
<tr>
<td>IDPROT</td>
<td>0.012</td>
</tr>
<tr>
<td>FILTER</td>
<td>0.005</td>
</tr>
<tr>
<td>DESTROY</td>
<td>5.253</td>
</tr>
</tbody>
</table>

Table 8.5: Results of experiment 2-1, showing time spent in each activity.

8.5.1.1 Analysis of Experiment 2-1 results

The most time-consuming operations are those that focus on store access to add items to the store, remove items from the store or carry out an extensive survey of the store contents. Thus, LOAD, DESTROY, TREE and MESSAGEADD have the most significant overheads. DECOMPA is larger than DECOMPQ because DNS query packets are almost always smaller than DNS response packets and, in this experiment, the DNS query packets were approximately one third the size of the response packets.
The FILTER and IDPROT operations have very small overheads because, once read at load time, the actions taken when filtering or protocol switching are required are efficient. Similarly, STREAMREAD is a minor contributor to the performance measurements as the underlying function is written in Python, grounded in the data model layer, and has no interaction with the store. The results of the experiments involving multiple protocols can now be compared to the IDPROT value given here. Tables 8.3 and 8.4 provide an indication of the overheads of protocol determination with a value in the range of 0.011 to 0.013 seconds for the time taken to establish which protocol has been employed. The time taken for this additional protocol determination has a standard deviation in the order of 0.010 seconds. This agrees with the IDPROT value determined here of 0.012.

The total average time to complete a single test was 14.729 seconds. Given that the server follows the sequence of operations shown above, consulting Table 8.5 shows that the total expected time is 14.556 seconds that is within 1% of the expected result. The missing time is accounted for in the time spent in the instrumentation and measurement of the other results.

Removing the, artificial, overheads of LOAD and DESTROY for every server operation gives an average server response time of 0.412 seconds. However, the average client time is expected to be 9.06 seconds. The actual average time is 9.19 seconds, again due to instrumentation and measurement. This is very close to the DNS timeout period and, as discussed before, at the edge of acceptable time limits. As most of the time is spent loading the ontology, rather than issuing a query and then waiting for 9 seconds, any queries issues can be responded to well within the normal DNS timeout period. The actual time spent handling the query and response is 0.169 seconds, less than 2% of the total time including load time.

The ‘complete’ test ontology provided a large and complex structure that provided all of the elements required to run a DNS server on top of our software architecture, using the ontology and a well-defined set of fundamental functions. As already stated, the requirements for the client are far less rigerous. The client requires enough logical and representational information to successfully encode and decode the messages. The essential requirements for the client are the description of the types and relationships of the knowledge aspects it will deal with - namely, those specified in the DNS RFCs [3, 4]. We produced a simplified version of the main ontology, only using essential elements for the client, which substantially reduced ontologically load time while still allowing almost all of the benefits of the new approach. As can be seen in the next experiment, a careful reduction in ontological complexity can reduce the load time to under 0.5 seconds. Combining this with the existing time spent in query handling gives a significant improvement in response time (13.5 times
faster) and reduces the client response time to be in line with the server response time.

8.5.2 Experiment 2-2

Experiment 2-2 measured the impact of ontology size and complexity on the LOAD and TREE times. Concepts and instances were removed from the operational test ontology to produce smaller ontologies with reduced functionality. Due to the nature of the RDF produced from an ontology, the removal of a single statement can remove an entire section of the final tree if the validity of the ontology is checked during construction. Thus, removal was carried out with care to test the impact of the removal of components of concepts, entire concepts and instances.

As the number of RDF triples produced from an ontology do not immediately correspond to a complexity estimate, we developed a complexity metric that provided a weighting for more detailed concepts and those that were deeply nested inside other concepts.

8.5.2.1 Complexity rating for ontologies

The complexity metric provides a comparison between less complex ontologies formed from a more complex ‘parent’ ontology. This allows for the comparison of, for example, a DLised ontology formed from an OWL Full ontology. Two sub-ontologies could have similar sizes in terms of number of concepts but have very different load and traversal times due to the structure of the elements in the RDF graph derived from the ontological specification. The complexity metric takes into account the depth of definition associated with ontological concepts, weights these depths to account for the time spent in search operations and also takes account of the instance implications by compensating for the number of individuals defined at a given depth.

If the depth, $d$, of a given element can range from 2 to a maximum depth, $D$, the number of concepts at that depth is given by $c_d$, the depth compensation figure is given by $dc_d$ and the number of individuals at a given depth is given by $n_d$ then the formula for the raw complexity rating is:

$$\sum_{d=2}^{D} c_d \times dc_d \times n_d$$ (8.1)

As the complexity of AVL tree search is $O(n \log n)$, as previously discussed, the values $dc_d$ were set so that they doubled at every increase in depth: $d \geq 2 : dc_d = O(d-2)$. This is $O(n^2)$ bound and, as $n$ becomes increasingly large, provides a complexity estimate that
dominates the $O(n \log n)$ contribution and, hence, does not underestimate the contribution of searching the AVL tree. Substituting this into the previous formula gives:

$$\sum_{d=2}^{D} c_d \ast 2^{(d-1)} \ast n_d$$  \hspace{1cm} (8.2)

All values are normalised against the smallest ontology that represents the basic logical, representational and operational semantic structure, an ontology that provides three level-2 concepts and no individuals. We refer to this normalised value as the **relative store size**. The smallest ontology represents the ontological equivalent of the minimal ontological graph specified in Figure 4.3. The complexity metric of this ontology has a raw value of 3 and thus all ontological complexity values are divided by 3.

Two subordinate experiments were carried out to measure depth and instance relationships across the test ontologies. Experiment 2-2a determined $D$ and the number of concepts at a depth $d$ throughout the ontologies. Experiment 2-2b determined the number of individuals at each depth. The results of both experiments are used to determine the complexity value used in the results for Experiment 2-2. As the experimental results for 2-2a and 2-2b have no utility outside of the normalised result shown in Experiment 2-2, they are not shown here as they are subsumed by the results of Experiment 2-2.

### 8.5.2.2 Test Ontologies for Experiment 2-2

Table 8.6 shows the names and normalised complexity figures for the test ontologies used. The LOAD and TREE times for the ontologies are shown at Table 8.7. Recall that the LOAD time is the taken to perform the operations that load the ontology into the store and prepare the store for queries and access. The second table also shows the number of items, relative to the smallest ontology, produced in the RDF store by the ontology. This is introduced to illustrate why a pure measure of RDF store occupancy does not give a clear indication of possible performance as there is a significant difference in complexity measurement between ontologies that have a very small difference in relative size.

### 8.5.2.3 Analysis of Experiment 2-2 results

We validate the use of a complexity measure instead of a straight measure of the number of items in the store by showing that the complexity measurement is a better fit to performance time than a simple measure of the size of the ontology. Figure 8.5 graphs the relationship between them. The small increase in relative size between ontologies 4, 5, 6, 7 and 8 spans a relatively large leap in complexity metric. Thus, as expected, the depth and complexity
<table>
<thead>
<tr>
<th>Ontology number</th>
<th>Complexity Metric</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>521</td>
<td>non-DL test ontology</td>
</tr>
<tr>
<td>1</td>
<td>478</td>
<td>DL test ontology</td>
</tr>
<tr>
<td>2</td>
<td>424</td>
<td>Ontology 1 without internal testing classes for protocols</td>
</tr>
<tr>
<td>3</td>
<td>321</td>
<td>Ontology 3 was used for calibration of the test suite. It was not used for measurement testing.</td>
</tr>
<tr>
<td>4</td>
<td>108</td>
<td>Ontology 2 without operational semantics support.</td>
</tr>
<tr>
<td>5</td>
<td>108</td>
<td>Ontology 4 with a reduced A-Box.</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>Ontology 5 without support for ordered IP address representations.</td>
</tr>
<tr>
<td>7</td>
<td>87</td>
<td>Ontology 5 with reduced complexity in response handling.</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
<td>Ontology 7 with minimal complexity in response handling.</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>Smallest ontology</td>
</tr>
</tbody>
</table>

Table 8.6: Test ontologies for experiment 2-2 showing complexity metric

<table>
<thead>
<tr>
<th>Ontology</th>
<th>LOAD</th>
<th>TREE</th>
<th>Complexity</th>
<th>Relative store size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.665</td>
<td>1.453</td>
<td>521</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>8.512</td>
<td>1.230</td>
<td>478</td>
<td>79</td>
</tr>
<tr>
<td>2</td>
<td>7.158</td>
<td>1.222</td>
<td>424</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>7.169</td>
<td>1.075</td>
<td>321</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>5.334</td>
<td>0.632</td>
<td>108</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>5.314</td>
<td>0.631</td>
<td>108</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>2.584</td>
<td>0.493</td>
<td>87</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>0.779</td>
<td>0.251</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>0.193</td>
<td>0.008</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.7: LOAD and TREE times for the ontologies in experiment 2-2.
Figure 8.5: The graph of complexity metric versus relative store size

of an ontology is not purely represented by the number of RDF triples generated from it. However, with a purposeful addition of features around a well-defined central structure, as in ontologies 0, 1, 2 and 4, the increase in ontology elements has a more linear relationship to the complexity of the ontology. This is due to the reduced addition of large, deep conceptual structures with the accompanying groups of individuals.

Figures 8.6 and 8.7 clearly illustrate the difficulty in estimating performance based on a straight measure of store size. Figure 8.6 displays a disproportionate jump in loading time over a very small change in size, with a matching clustering in the time taken to traverse the tree. This is not surprising as the size of a store is not a clear indicator of complexity. While it may indicate a high potential complexity, a large store could as easily be a large number of shallow trees rather than a small number of deep and complex trees.

Figure 8.7 shows the relationship of the complexity metric to loading and tree traversal time. There is much less clustering in the data, although there is a large increase in loading time between ontologies 6, 7 and 8. As discussed in the descriptions of the ontology, the major difference between these ontologies is the reduction in complexity from 6 to
8. Thus, while a small number of elements are being removed, the impact of this on the loading time is far more significant than the impact on the traversal time. Traversal time is bounded by $O(n \log n)$ where $n$ is the number of elements but the introduction of more complex elements into the ontology places heavier demands on the store in a much more significant way. Our choice of complexity metric is designed to allow the estimate of an approximately linear relationship between the timing requirements of ontologies that share a similar purpose, albeit with a differing internal structure.

### 8.5.3 Subsequent Experiments with Faster Equipment

Towards the end of experimental work, new hardware was made available that was significantly more powerful than the original experimental platform. The new experimental server platform was, for the purposes of these experiments only, an Intel Core 2 Duo 2.33 GHz server with 2GB of RAM under OS X 10.4.8.

Experiments 2-1 and 2-2 were re-run with the new server hardware and performance
measurements captured at the server, as in the previous executions. All library and Python versions were identical to those used in the previous experiments. The new experiments are referred to as 2-1-N and 2-2-N.

8.5.3.1 Experiment 2-1-N results

This set was tested 1000 times and the results averaged. The resulting values are shown in Table 8.8.

8.5.3.2 Experiment 2-2-N results

Table 8.9 shows the results for LOAD and TREE times for the ontologies listed in Table 8.6 using the new experimental hardware for the server.
<table>
<thead>
<tr>
<th>Aspect name</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>1.044</td>
</tr>
<tr>
<td>TREE</td>
<td>0.343</td>
</tr>
<tr>
<td>DECOMPA</td>
<td>0.018</td>
</tr>
<tr>
<td>MESSAGEADD</td>
<td>0.040</td>
</tr>
<tr>
<td>DECOMPQ</td>
<td>0.016</td>
</tr>
<tr>
<td>RESOLVEQ</td>
<td>0.005</td>
</tr>
<tr>
<td>ANSWERQ</td>
<td>0.004</td>
</tr>
<tr>
<td>STREAMREAD</td>
<td>0.001</td>
</tr>
<tr>
<td>IDPROT</td>
<td>0.003</td>
</tr>
<tr>
<td>FILTER</td>
<td>0.001</td>
</tr>
<tr>
<td>DESTROY</td>
<td>0.578</td>
</tr>
</tbody>
</table>

Table 8.8: Results of experiment 2-1-N, showing time spent in each activity.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>LOAD</th>
<th>TREE</th>
<th>Complexity</th>
<th>Relative store size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.021</td>
<td>0.369</td>
<td>521</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>1.021</td>
<td>0.313</td>
<td>478</td>
<td>79</td>
</tr>
<tr>
<td>2</td>
<td>0.859</td>
<td>0.314</td>
<td>424</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>0.849</td>
<td>0.279</td>
<td>321</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>0.661</td>
<td>0.161</td>
<td>108</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>0.656</td>
<td>0.160</td>
<td>108</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>0.368</td>
<td>0.128</td>
<td>87</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>0.168</td>
<td>0.064</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>0.062</td>
<td>0.002</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.9: LOAD and TREE times for the ontologies in experiment 2-2-N.
8.5.3.3 Analysis of new performance results

Overall there is a marked improvement in performance that, importantly, allows the ontology to become much larger before the waiting time for the user becomes as obvious as it was in the previous experiments. The performance of the current hardware is 5-10 times faster, depending on the specific application. Compared with table 8.7, table 8.9 shows that STORE LOAD time is, on average, 12% of what is was previously, and TREE traversal time is consistently 25% of the previous time. Most other operations execute in approximately 20% of the previous execution time although performance savings diminish as the ontology becomes much smaller, due to the required overheads of starting the Python interpreter. At a very small ontology size, these overheads dominate.

The major improvements to LOAD time stem from the use of a dual core processor with a much larger L2 cache. With all other system activity centred on one processor, context switching due to I/O issues and GUI-related overheads are significantly diminished on the single processor that is dedicated to executing the Python code that ultimately provides the store. The available L2 cache has also increased from 512KB total to 4MB per processor, an 8-fold increase for a single processor. This also significantly reduces the impact of file I/O and page faulting, greatly increasing the overall performance of the disk-intensive, memory-consuming LOAD operation. The TREE time shows less overall improvement as it works on the in-memory store, rather than from the disk-based ontology, and only derives benefits from the reduced page-faulting and faster memory access.

The original experiments were performed on a variety of machines, ranging in age from 3 to 6 years. This set of experiments shows how the system performs on contemporary commercial hardware, running standard operating systems and freely available software. The small ontologies now perform at a level that is a far closer approximation to the performance figures of standard DNS clients and servers, and the largest ontologies have a small waiting time that is far easier to accept at the user level.

8.5.4 Further optimisation

A further optimisation was carried out to test the ‘best performance’ case for the new hardware and to reduce STORE LOAD overhead, that is by far the largest contributor to execution time if LOAD occurs on every activity. The modification removes the membership test carried out on the store, prior to the assertion of a new triple. Rather than carry out redundancy checks on loading every time, the ontology allows the assertion of duplicate triples, with the underlying store modified so that this does not create two elements but overwrites the first occurrence.
8.5.4.1 Experiment 2-1-N-C results

Experiment 2-1-N-C was run with the new code in place, on the new hardware, for 1000 iterations. The results are shown at Table 8.10.

<table>
<thead>
<tr>
<th>Aspect name</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0.778</td>
</tr>
<tr>
<td>TREE</td>
<td>0.343</td>
</tr>
<tr>
<td>DECOMPA</td>
<td>0.018</td>
</tr>
<tr>
<td>MESSAGEADD</td>
<td>0.038</td>
</tr>
<tr>
<td>DECOMPQ</td>
<td>0.015</td>
</tr>
<tr>
<td>RESOLVEQ</td>
<td>0.005</td>
</tr>
<tr>
<td>ANSWERQ</td>
<td>0.004</td>
</tr>
<tr>
<td>STREAMREAD</td>
<td>0.001</td>
</tr>
<tr>
<td>IDPROT</td>
<td>0.03</td>
</tr>
<tr>
<td>FILTER</td>
<td>0.001</td>
</tr>
<tr>
<td>DESTROY</td>
<td>0.578</td>
</tr>
</tbody>
</table>

Table 8.10: Results of experiment 2-1-N-C, showing time spent in each activity.

8.5.4.2 Experiment 2-2-N-C results

Table 8.11 shows the results for LOAD and TREE times for the ontologies listed in Table 8.6 using the new experimental hardware for the server and the new code.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>LOAD</th>
<th>TREE</th>
<th>Complexity</th>
<th>Relative store size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.765</td>
<td>0.368</td>
<td>521</td>
<td>81</td>
</tr>
<tr>
<td>1</td>
<td>0.763</td>
<td>0.315</td>
<td>478</td>
<td>79</td>
</tr>
<tr>
<td>2</td>
<td>0.688</td>
<td>0.312</td>
<td>424</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>0.688</td>
<td>0.279</td>
<td>321</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>0.503</td>
<td>0.160</td>
<td>108</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>0.504</td>
<td>0.160</td>
<td>108</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>0.304</td>
<td>0.128</td>
<td>87</td>
<td>55</td>
</tr>
<tr>
<td>8</td>
<td>0.168</td>
<td>0.063</td>
<td>65</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>0.073</td>
<td>0.002</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.11: LOAD and TREE times for the ontologies in experiment 2-2-N-C.

8.5.4.3 Analysis of results of the optimised code experiments

As the STORE LOAD operation is the only operation that references the add code [146], it should be the only operation affected by the change in code. The results of both exper-
ments confirm this. Although there is minor experimental variation for other operations, ranging from 98 to 102% of the original values found in table 8.9, the STORE LOAD operation is reduced in time by between 20-25%, depending on the size of the ontology. The final ontology records a larger time than the original, due to the variations in overhead time swamping the miniscule difference measurable between the loading time of the small ontology under original and new code.

The most important result is that the overall performance time has dropped significantly from the original worst case of over 9 seconds. Recall that the server operation can be represented as LOAD, STREAMREAD, IDPROT, DECOMPQ, RESOLVEQ, ANSWERQ, MESSAGEADD, FILTER, DECOMPA, STREAMREAD. The new value for this total is 0.891, less than 10% of the original worst-case value.

The worst-case time to execute a standard DNS query on the new hardware is 0.524 seconds, with an average time of 0.021 seconds, both figures discounting network delays. Removing the STORE LOAD operation for each query gives an approximate response time of 0.113 seconds. This gives a comparable performance ranging from 5.4 times slower to 1.7 time slower but this is a marked improvement over the initial results with old hardware and prior to optimisation.

However, as ontologies become more complex, this value will grow again so, despite this being a workable value for the current implementation, it is not a guarantee of sub-second performance for all possible ontologies associated with this problem domain. The complexity of the ontologies increase as further structure and conceptual complexity is added to the ontology. The most significant jump in loading time is seen, as expected, when the initial conceptual framework is added.

8.5.5 Complexity of queries

Within the RDF triple store, entire records cannot be retrieved with one query. The depth of a value is defined in this context as the number of \((s, p, o)\) triples required to trace from the contextual root as the leading subject to the value as the terminating object. Depending on the depth of the value that is being retrieved, there can be up to 10 triples that must be retrieved in order to absolutely determine contextual veracity. Thus, even a single DNS entry that logically consists of a set of subordinate values can require approximately 100 store queries to assemble the entire value. All operations that manipulate data in the ontologically-implemented server make store queries to establish context and validate assignment so every operation can incur store query costs. The structure of the graph also has an impact on store performance, thus the relationship between store size and operational
performance cannot be quantified in terms of the number of triples in the store.

However, the underlying store in the rdflib library has an $O(n)$ cost for operations, where $n$ is the number of triples in the store, rather than the $\log n$ seen for the previous AVL tree. For large $n$, where the depth is much less than $n$, the overall complexity is $O(n)$ for store operations. The introduction of the SPARQL query language [148] for assembling complex queries increases the worst case complexity to $O(n^2)$ for large graph patterns, where a large graph pattern is defined as one with a large depth. Hence, overall store queries have a complexity approaching $O(n^2)$.

However, the underlying store in the rdflib library has an $O(n)$ cost for operations, where $n$ is the number of triples in the store. For large $n$, where the depth is much less than $n$, the overall complexity is $O(n)$ for store operations. The introduction of the SPARQL query language for assembling multi-element queries increases the worst case complexity to $O(n^2)$ for large graph patterns, where a large graph pattern is defined as one with a large depth. Hence, overall store queries have a complexity approaching $O(n^2)$. Any query to the store that requires the completion of a DNS query, such as ANSWERQ, incurs the SPARQL query cost but internal caching, and subsequent hits on the cache, can amortise this cost over the life of the server. Determining the operational semantic and functional composition of an operation, such as DECOMPQ, is carried out once for every server instantiation and this SPARQL query overhead is amortised over the life of the server. Thus, most store operations still tend to $O(n)$ overall rather than $O(n^2)$. Overheads associated with the expressions capturing operational semantics are small as the transformations defined are kept simple.

### 8.5.6 Overhead due to multi-protocol issues

The time taken to determine which protocol is being employed, in a multi-protocol single port system, is directly proportional to the number of bits that must be read in order to identify the message. In the performance figures this is measured as IDPROT. In the case of a UDP message, each message must be considered individually. In the case of a TCP message, those messages received in a given set can be assumed to be of the same protocol as the first packet in the sequence. The data model passes entire messages up to be processed so determination can be carried out over the entire message. In the vast majority of cases protocol determination can be made within the first 512 bytes as legacy protocol design tended to focus on a strictly defined structure that would fit into a 512-byte payload size.

The time taken to search $n$ elements is $O(n)$, as the worst case is that all elements must
be searched, and it is only the final element that is the discriminator. A port will support a subset of known protocols so analysis must proceed until the discriminator has been read and a protocol has been chosen. If there are \( m \) protocols then the comparison element must be compared \( m \) times, with a complexity of \( O(mn) \). The worst case for this is that the number of bytes and the number of protocols are of roughly the same order of magnitude, in which case the determination operation itself is \( O(n^2) \).

However, heuristic intervention can radically reduce this due to the removal of the need to search all elements. For example, the third and fourth bytes of the DNS message format header contain flag values. The original specifications in RFC 1035 restrict the flag settings so that the fourth byte of the header can have only 18 different values, 9 for an incoming query and 9 for an outgoing response. So, given that the lower level of the data model passes up a single vector of bytes, the fourth byte can be inspected to see if the byte corresponds to one of the legal values for a query. If the multiple protocols supported on the port do not have an overlap for the values found in the fourth byte, then some protocols can be eliminated very quickly. In this case, protocol determination is \( O(1) \).

However, where multiple XML-based formats are in use, especially where HTTP wrappers are in use, it is still necessary to read the entire message until the appropriate components are reached, as wrapped XML formats use delimiters to indicate the component separation. Older formats used more fixed-width formatting as the formats were more restrictive and application specific. The more generic nature of XML-ised, wrapped formats leads, once again, to a potential complexity of \( O(n^2) \).

This not as bad as it initially seems due to the limitation on message size. A UDP packet can only have, as a maximum, 512 bytes of payload so \( n \) cannot become arbitrarily large. Based on this, what initially appeared to be \( O(n^2) \) is \( O(1) \), especially when compared to the impact of operations in the data store. This can also be justified for TCP packets by looking at the time taken to transmit an arbitrarily large sequence of data. As TCP packets must take some time to be transmitted, there is an imposed boundary on the number of messages that can be put together to compose a message as the query cannot be so large that it cannot be transmitted in finite time. Hence, \( n \) cannot be arbitrarily large for TCP messages either.

In this case, the impact of messages will only be felt if the number of messages becomes large and then will approach \( O(n) \) based purely on the number of messages rather than because of the underlying analysis of the message sequence.

Overall, the store dominates again with an \( O(n^2) \) complexity.
8.5.7 Performance bottlenecks

The time taken to load the ontology initially is by far the most significant contributor to processing overheads. The graph store, as it is used frequently, is the most obvious performance bottleneck. From experiment 1, where a LOAD was triggered on every query, the ontological loading time effectively restricted the implementation to a proof-of-concept curiosity. On the faster hardware, the implementation became usable but, with increasing ontological size and structural complexity, the worst-case performance can still lead to unacceptable performance. LOAD times may also be reduced through two approaches. The first approach is to maintain the graph store as a remotely accessible persistent store that is accessed by the client or the server, however this incurs potential delays in the propagation of changes and also introduces the potential for a single point of failure to an otherwise decentralised system. The second approach is to optimise the graph store loading activity by reducing the complexity of graph operations. We discuss a hybrid of these approaches in the next section.

8.5.8 Optimisation points

One point of optimisation is in the store itself. If we optimise the store then we will reduce all access times, including LOAD. To prevent the addition of redundant triples, the store is checked prior to adding a new RDF triple. Removing this check, and recoding the store so that duplicate RDF triples overwrite the old ones, reducing store loading time, LOAD, by up to 25%. A database-backed version of the store is an obvious optimisation, utilising the data access and query optimisations of modern RDBMS, potentially yielding $O(n \log n)$ or even $O(n)$ performance.

Changing the query response mechanism can also greatly enhance performance if, instead of adding messages to the store as part of standard operations to provide query caching and provenance information, the store is run in a read-only mode once it has been populated. IDPROT can also be eliminated if the server runs a single protocol on one port.

8.6 Summary

As has been shown in this chapter, the overheads of implementation are not trivial but they can be managed to provide a service that, while slower than the unenhanced implementation, has far more functionality. Also, the development of faster commodity processor technology has provided evidence that a task that was a challenging computation six years ago can be performed in time that is far more acceptable to users.
The caveat is that there are large complexity pitfalls within the current implementation. If the LALR grammar expanded to too great a degree, it would no longer be possible to regenerate the grammar at every server reboot. Similarly, the complexity of the underlying ontology has a large impact on the time taken to assemble the ontology in memory as a graph based on the OWL description. A thoughtful analysis of the possible overheads should deal with the majority of implementation problems but a large and complex ontology will still have an underlying complexity that may not be able to be ameliorated through implementation-specific manipulations.
Chapter 9

Future Work and Applications

The implementation demonstrates the utility of the new approach as a successful use of ontological specification and semantic annotation to enhance the operation of network-based information systems. However, there are still areas where further work can be done. Also, we can apply this work to other existing technologies to provide increased benefits.

This chapter contains an initial discussion of possible extensions of this work and then discusses other technologies and how they could benefit from this approach. It concludes with a discussion of the general advantages that can be gained from converting any distributed system to the new model.

9.1 Extensions

During the development of the system, some aspects were not developed further as we chose to develop a single proof-of-concept server that delivered as much fully-formed functionality as possible. This section describes those areas where the system could have been, or still could be, extended to provide new or novel functionality. There is often significant impact in implementing a given extension, largely due to the required additions to the operational semantics or those parts of the data model that support the operational semantics. The resulting increase in complexity would need to be assessed against the benefit in each case.

9.1.1 Operational Semantics

In Section 7.8, we discussed the development of operational semantics support in our implementation. The most obvious extension to the existing work is the further development of the operational semantics. Unfortunately, as outlined in the results chapter, the more
complex the underlying grammar becomes and the longer the constructed sentences, the more that the complexity of grammar parsing can dominate computation. This suggests the use of a simpler grammar for specifying the operational semantics and, potentially, a language that is not Turing complete, rather than the applied $\lambda$-calculus. The main advantage of using a language such as the $\lambda$-calculus is that the entire system could be written in $\lambda$-calculus, although it would prove both slow and unwieldy, or we can use this language in conjunction with other, pre-written, code modules. If we employed a language that was not Turing complete, we risk reducing our operational semantics language to merely a glue language that composes together modules written in a more complete language. This would greatly reduce the flexibility we enjoy with a language that can be used to assemble other functional modules but can also be used to write the same modules. There are no constraints on how or when we employ the $\lambda$-calculus, other than those imposed by performance issues or execution-based pragmatics.

The applied $\lambda$-calculus was chosen because of its flexibility but, as has been discussed in Chapter 8, if native functions are used extensively then better performance can be obtained by using a combination of native functions composed together using $\lambda$-calculus, rather than writing large subroutines in the $\lambda$-calculus. This is for two reasons, the first is that the amount of time spent in interpretation is reduced and the second is that any time spent in producing the parsing table will be reduced. However, this reduces the role of $\lambda$-calculus to a ‘glue’ language. Further work in this area might include the development of LALR(1), SLR and LL(1) non-$\lambda$-based languages using careful table construction to reduce the potential explosion of states. Using this can reduce the complexity for grammar generation from exponential class to $O(n^3)$ [153]. With an underlying store mechanism with either $O(\log n)$ or $O(n^2)$ performance, this would still dominate but at a much slower growth rate than for the exponential case. However, as discussed in the Results chapter, there is not necessarily a requirement to regenerate the grammar at every step and, if this is the case, then the parsing step can be completed in linear time.

### 9.1.2 Representation of native functions

Instead of placing more code into the native functions, an obvious alternative is to place less code into the native functions. Using the implementation of the DNS as an example, one place where the native code could be reduced is in the encoding of the cryptographic functions. These functions could be encoded in operational semantics description language - the $\lambda$-calculus. However, this would place a much greater emphasis on the efficiency and nature of the underlying parser mechanisms and would, for a reasonable speed of operation,
require the pre-compilation of the operational semantics that comprise the cryptographic
systems. This then also motivates a change of underlying language to one that can be com-
piled and heavily optimised, rather than the Python language used in the \( \beta \)-implementation.

The introduction of a full compiler into the system significantly increases the time taken
to start the system and requires the addition of a phase to the operation of the system: on-
tological parsing. This parsing is a separate activity with a partially annotated and compiled
ontology being read at start-up. The major benefit, however, is that the compiled instruc-
tions should execute significantly more quickly than the interpreted version. Again this
raises issues of decidability with the potential introduction of OWL Full elements if the on-
tology is automatically annotated and altered on-the-fly. However, we believe that careful
use of grammatical rewriting can prevent this from occurring.

9.1.3 Self-healing mechanisms

The implemented entailment checks described in Section 7.13 allow for a policy-based
recovery mechanism if the executing system diverges too far from the conceptual model
of system operation, but this healing mechanism is limited as it only applies to a coarse-
grained representation of program execution. The system operation is identified as large,
logical and functional blocks - it is not possible to carry out healing on individual knowl-
dge entities.

The transition from on-the-wire data to knowledge is, in the current implementation, a
strictly controlled process where any missing mapping, from data to information or from
information to knowledge, will prevent the use of the data. This implementation decision
provides a protection and control mechanism to prevent uncontextualised access to the
knowledge store and, thus, any possible misuse of the stored knowledge. However, there
are times when we may be happy with ‘good enough’ rather than ‘guaranteed knowledge’.

Depending on how data enters the system, we can specify an alternative mechanism
for producing a class of untrusted knowledge entities that can be used in limited ways
within the system. Thus, if only some of a query can be decoded, we may not have to
reject the entire query. Instead, if the undecoded elements are either known, from previous
exchanges, or have no direct bearing on the query then an untrusted query can be made on
the store. This requires the additional overhead of an annotation to the returned data stream
to indicate that this is an untrusted result due to the inability to successfully encode a set of
data elements. Rather than request a retransmission or wait for timeout to occur, the sender
and receiver can look at the query and result, mark the components that could be encoded
and then leave it up to the final recipient of the result to determine if the result can be used
without the additional information.

Such a solution requires the addition of new out-of-band data channels to take the additional metadata and, as a result, this cannot be applied to clients and servers using legacy protocols where protocol variation is not allowed. However, those enhanced nodes within a system can apply this strategy in order to reduce retransmission or timeout overhead.

9.1.4 Confidence-based protocol determination

The development of our methodology and its application, in Chapters 4 and 5, were based on an assumption that all knowledge entities in use were fully defined. The notion of potentially untrusted responses in an otherwise strictly-controlled environment introduces the possibility of other less strict approaches to data handling. Automated protocol determination was excluded as a possibility in Section 5.4.2 due to the inability to guarantee that a protocol was correct. In cases where the components of a message in a given protocol are identified with delimiters rather than bit positions, a less strict approach can be taken with a reduction in the potential problems associated with unconstrained automated protocol determination.

In a positionally-dependent format, such as DNS Message Format, byte position is significant and the misinterpretation of byte boundaries and structures can lead to a complete alteration of the message’s meaning. For example, reading bytes 3 and 4 of a DNSMF stream as a single value will still allow the extraction of the flag values that are contained therein, but reading bytes 4 and 5 as the flag values will provide an 8-bit shift for one set of flags and completely ignore the remainder. Worse still, the 5th byte will be read as if it were flags, rather than an integer. In a string-based approach, such as the predominance of HTTP transported protocols with an XML focus, the use of delimited strings provides a much less structurally bound approach.

What is required is an overarching ontology that can be used to place the detected string information into the correct context. In the case where a knowledge pattern is not matched by the decomposed input, the current behaviour is to reject the decomposition and ignore the message. However, if previous messages have arrived that match the original format, these can be compared on a component-by-component basis to the previous message to see if the contents are similar. If there is similarity then, rather than rejection, the received message can be placed into an untrusted namespace and then linked into the main RDF store through references. Thus, the information is still available but is clearly part of another namespace, preventing it from occluding any information that it could affect in the primary namespace. This could then be unified later manually or through the receipt of a correctly
formatted message to verify the original, although an obvious question in this context is ‘if such a message were possible why was it not sent first’?

This approach requires the correct use of subtypes to assist the process as the XML Schema definitions for datatypes provide enough information that, if an entity is typed as IDREF as a subtype of string, then the tested entity must match the IDREF type to be deemed similar, not just the string. Also, the possible messages that can be successfully captured are those where the record components are in the correct place with the exception of missing or added terminal elements. If a leading element is missing then a successful decoding, even into the untrusted namespace, can cause problems at later rationalisations of the store.

Consider a message that consists of four delimited strings. The first is an IP address and can then be pattern matched to four octets, separated by periods and terminated with the delimiter. The remaining elements are all simple strings. If five strings are sent and the first four decode correctly, then this can be added to the untrusted namespace and stored. This is obviously untrustworthy as the additional element can have been inserted anywhere, providing that it was of the form of an IP address since that would match both string and IP address lexical specifications. So, to limit the impact of possible misinterpretation, any matching should take place for the more restrictive types as well if there are more or less elements than should be present. In the case of a repeated first element, the match to the IP address in two places indicates repetition and can then either be discarded as too untrustworthy or the first element disregarded.

This extension requires an extensive amount of redevelopment of the ontological specification to accommodate a fuzzy approach to trust values and the setting of threshold values to allow the admission of trust-classified data to the system. The addition of these trust classifications can be undertaken in OWL but currently requires the use of discrete separation of values rather than a true continuum.

9.1.5 Ontology repair for DL features

We have already discussed heuristics that are used to address commonly occurring mistakes in Section 7.13 but we believe that these can be extended without necessarily causing a significant amount of additional computation. Suppose that we inadvertently produce an OWL Full ontology to control the system, when an OWL DL specification would suffice. This can remove the DL termination guarantees that are used to provide reasoning support through entailment. Further work in resolving the accidental introduction of non-decidability through the use of heuristics to eliminate common mistakes can be integrated
as part of the analysis of the ontology on start-up. While this would add overhead during start-up, the time saving of restricting computation time from unbounded to an exponential class can be substantial, especially as we moved to guaranteed termination, if reasoning support is essential in the problem domain that is implemented.

### 9.1.6 Name abstraction

With the use of URIs and XML namespaces, a large amount of the context in the store is tied to machine names and, particularly in a system such as the DNS, the agility and potential stability of such a mechanism is open to question. Although the namespaces can be arbitrary, the locations of resources are important, especially if they are being obtained from import statements or used as locations somewhere within the operational semantics.

A name abstraction scheme can prevent references to external names from becoming stale if there is an internal naming convention that is translated on demand to the external format when shared. As the name is updated whenever an external reference is required, this abstracted reference is always fresh at the time of determination. In particular, the on-demand translation can take advantage of enhanced systems such as SAMPAN-DNS to update references to inactive nodes and prevent or limit timeout issues in the system. Such an abstraction can also be used to load balance resource requests across a family of nodes or provide another layer of reliability.

### 9.1.7 Denotational Semantics

Another useful extension would be the development of denotational semantics to formally specify the semantics of the system. Given that the system introduces new mechanisms that combine and extend existing, well-defined, semantic elements, this specification would remove any ambiguity and allow the formal verification of the system. However, this would be a very large undertaking and was not pursued in this thesis due to the sheer size of the project as it would effectively correspond to another thesis, constructed in parallel.

Denotational semantics can also provide a stronger basis for our type system with a defined mapping between the XSD type definitions and the implementing language type definitions. The possible issues of concurrent access to the store, currently managed through the use of protected access but not verified, can also be determined using this approach to verify that the knowledge development process and subsequent limitation of store access does not automatically introduce deadlock or livelock conditions.
9.2 Applications

The following sections detail potential applications of the system in conjunction with, and occasionally replacing, existing technologies. The key argument for using SAMPAN nodes in the place of existing nodes is generally based on the fact that a knowledge-based approach provides far more context for producing an alternative representation. It is easier to derive a representation from existing knowledge than it is to derive new knowledge from an existing representation as the representation may not have a sufficiently explicit statement of the semantics of the data to correctly define the encapsulated knowledge. The existence of a floating point field in a protocol template does not provide enough information to determine the nature of the stored value, but if we work the other way, then the semantics of the stored value can be analysed to provide a suitable representation for that value with far greater confidence.

9.2.1 Interoperation with WSDL

The Web Services Definition Language (WSDL), introduced in Section 2.4.5, provides the protocol bindings and messages formats that are required to interact with web services. Abstract public specifications are bound to network protocols and message formats. In this way, a client wishing to use a web service can look at a directory to discover which functions are available, if any special datatypes are in use and may then employ a protocol such as SOAP to call the function.

The SAMPAN nodes are designed to decouple their internal data storage format from the external representation, as has been demonstrated with multi-protocol support on individual nodes. Given that the knowledge level encoding of all data will provide a translation mechanism to an external wire format, producing a wire format to meet the requirements of a WSDL specification or providing a WSDL specification from the knowledge encoding of the format are both equally feasible. Given that WSDL specifications use XML specifications for additional types, the inclusion of types other than those found in the XML Schema document can be performed by placing the new type specification into the internal type hierarchy.

However, what cannot be derived from a WSDL specification is the context of the data elements. In SAMPAN, nodes exchange data for which they share knowledge. So deriving a meaningful WSDL specification from a SAMPAN node is straight-forward but producing knowledge from a WSDL specification is not because there is no reference ontology to provide semantic grounding.

A SAMPAN system can be integrated with a WSDL-based system and provide at least
the same level of information available to standard WSDL clients but, in the presence of SAMPAN-enhanced servers and clients, can also provide a semantically-backed WSDL specification where co-operating nodes can agree to exchange certain knowledge elements in certain parts of the WSDL specification.

9.2.2 Interoperation with SAWSDL

By extension, SAWSDL integrates very well with a SAMPAN system as the SAMPAN system provides the defining ontology that is then used by SAWSDL to provide semantic grounding. SAWSDL descriptions can reference the defining ontology to provide semantic grounding for operations conducted on nodes that are not full SAMPAN nodes.

The metadata annotations that provide the model reference will deliver information to SAMPAN ports in the defined format but the actual tie between the physical representation encoded in the WSDL and the model reference attached as SAWSDL must be maintained by a human programmer. It is possible for a model reference to become stale over time and, as the WSDL is not generated from the SAWSDL model reference, the WSDL must be updated accordingly.

9.2.3 Interoperation with OWL-S

We have previously discussed OWL-S as a mechanism for providing an OWL services ontology, in Section 2.4.5.3, but we discuss it here in order to illustrate how we can integrate OWL-S with our approach.

OWL-S provides a mechanism for specifying service representation through a service profile, but it does not require this representation and can use specialised representations. This allows the specification approach used in SAMPAN to be integrated with an OWL-S description of a service and make the ontologically-backed, and semantically rich, service representation available to be shared between nodes that are or are not SAMPAN enhanced. Also, a SAMPAN ontology, or a component of one, can be used as a reference taxonomy for specifying categories and classifications of services.

This allows the integration of the knowledge-based approach of SAMPAN within the framework of OWL-S, bringing the benefits of the shared context for entities to a standards-based mechanism for the semantic mark-up of web services. As has already been discussed for WSDL, the production of OWL-S specifications from a SAMPAN knowledge base is straightforward, as the SAMPAN knowledge base contains more than enough information to produce the OWL-S description, but the reverse requires the injection of additional information. This is particularly true of the operational semantics of the service, although
the correct use of KIF-expressions in OWL-S, discussed in subsection 2.4.5.3, would allow some of the operational semantics to be captured for individual operations through the specification of preconditions and effects. The KIF-expressions embedded in OWL-S require the use of field engineering to substitute variables into the correct places in the expression, rather than using centrally-defined knowledge components where the name is a transparent reference to the value [98].

What is, however, lacking is a context for the KIF-expressions. A large portion of OWL-S is used to describe the structural relationships of the described entities to their parameters and outputs. However, the knowledge-centric approach focusses on describing the entity so that it becomes knowledge and then relates knowledge elements to each other. A significant amount of the data model in SAMPAN is devoted to rendering such relationships implicit to provide the knowledge-driven mechanisms, rather than requiring them to be explicitly stated as found in the OWL-S ontology using KIF.

9.2.4 Semantic Web

The goal of the semantic web, initially discussed in 2.4.5.3, is to facilitate information exchange by using machine-interpretable, rather than human-interpretable/machine-mineable, documents. To this end, the markup of documents is extended from layout and simple formatting to describe elements as resources, using RDF, and then providing ontological context and reasoning overlays with the OWL family of languages. This is designed to enhanced the usefulness of shared data repositories, such as the World Wide Web.

As a SAMPAN node can provide data in whichever format is required, providing a transformation from knowledge to wire-format has been provided, such nodes can be accessed from whichever service is in use on the Semantic Web. Also, with the ability to annotate data streams with additional information, data sources can be composed to produce annotation overlaid on top of existing, unannotated data sources. This allows the combination of unannotated legacy systems with enough information to be useful to other nodes in a semantic web environment. This leads to applications within the E-Science environment, as discussed earlier.

Metadata vocabularies can also be shared to allow the use of contextually-rich items in the shared environment. Depending on the complexity of these vocabularies, these can be classified as either taxonomies or ontologies. In either case, a SAMPAN ontology can be shared to establish shared context across the semantic web, including providing a common contextual framework for operational semantics in complex systems as well as the more commonly shared data representations and hierarchical contexts. The nodes can be refac-
stored to provide as much, or as little, additional markup as is needed to integrate the node into the desired machine interpretable framework.

9.2.5 General distributed systems

In this section, we discuss the role of SAMPAN nodes in general distributed systems. We begin with the discussion of using SAMPAN nodes as adapter nodes and then discuss SAMPAN in terms of the Grid. These final examples and applications illustrate the versatility and potential for our approach to implementing NBIS.

The obvious role of adapter nodes in a distributed system is to bring together several different systems to allow the knowledge in one to be used in another without requiring syntactically defined translation mechanisms between them. This allows the production of much larger virtual knowledge-based systems where the component systems are used to produce a greater level of knowledge in combination than could be achieved individually. In this context, the knowledge level consists of both the amount of context that can be established for a value and the metadata associated with the knowledge. The context and metadata ground the value inside a system in a way that dramatically increases its utility.

One of the key results of the adapter concept is that multiple versions of the same system can coexist, using an adapter to support interoperation and present a uniform interface to the user community. Even if the latest version of system software changes through the inclusion of new protocols, or complete transformation of input/output, an adapter node can support multiple protocol versions in the same distributed system at the same time.

Adapter nodes can encapsulate legacy system nodes to provide the rich metadata annotation that can be required by modern applications using newer protocols. They can also be used in situations where upgrading components of an architecture cannot be carried out, due to legacy machine issues. Adapter nodes can also be of benefit when introducing new protocols or replacing older protocols in an existing system. These nodes can provide an adaptation layer between those elements of a system that are not supporting new protocols and those that are. In this case they are functioning occlusively rather than as boundary nodes. The difference between the two mechanisms is shown at Figure 9.1 and Figure 9.2.

In Figure 9.1, the adapter node provides a knowledge-driven mechanism for both systems to share knowledge from each other. However, a syntactically-coordinated translation mechanism could allow nodes from each system to directly transfer data without the knowledge framework.

In Figure 9.2, the first distributed system is only available through the adapter node and, as a result, direct communication between nodes in this system and the other system is
Figure 9.1: Adapter nodes on the boundary displayed against adapter nodes without occlusion

Figure 9.2: Adapter nodes on the boundary displayed against adapter nodes as occluders
impossible. In this way, Distributed System A can continue to operate in any manner providing that the adapter node can perform the alignment between the knowledge in System A and that of System B.

9.2.6 Grid computing

We now discuss two applications that are commonly used in the Grid in order to show less obvious potential applications for SAMPAN. This example application illustrates a use of semantic grounding that is not strictly related to computational issues. Instead, we discuss service level agreements (SLAs). SLAs are used to agree upon the resources that can be used to solve a problem and under what conditions these resources are to be used. This is, in effect, a meta-computing or pre-computing issue that also requires strong semantic grounding to be useful to all parties.

The provision of service level agreements in the Grid is an important part of Grid computing. Service level agreements define the agreement between service provider and service consumer as to what level of service is expected. SLAs also determine the definition of common terms where confusion could arise between the two parties. Determining an SLA without a contextual framework to determine component replacement or alternative mechanisms for meeting the agreement forces those users developing such an agreement to fall back to human-specified mechanisms. While an SLA can be machine-interpretable, it is not very flexible until the components can be factored as knowledge - that requires a strong, semantically-aligned mechanism to be able to place all sources, data, representations and semantics into an interpretable framework. Hence, a SAMPAN adapter node or set of nodes could be used to encapsulate parts of a system to allow the management and interpretation of the SLA.

However, as discussed in the previous section, we can also utilise SAMPAN to support multiple protocol versions in the same system. Since we can occlude or enhance some parts of a system, from the view of a given consumer, we can construct a Grid within a Grid to provide two distinct versions of the same Grid infrastructure. This approach allows the effective deployment of a versioned distributed system as some components of a larger structure can be used without necessarily having to bring them into step with the rest of the structure. Thus a larger Grid can be formed out of many disparate smaller Grids without necessarily requiring them all to be version identical. In a computationally intensive environment, a knowledge-based representation may not be desirable due to the overheads of processing in the store and the operational semantics. Multiple versioning can, of course, be applied to any distributed system in order to provide the features of
several different versions of the distributed system at the same time.

9.3 Summary

In this chapter, we discussed possible extensions to the work developed in the methodology and implementation sections and also described a range of applications that used existing technology in combination with SAMPAN. The applications sections demonstrated where SAMPAN could be used and how it could be used in conjunction with existing technologies in order to enhance NBIS operation. The next chapter deals with our conclusions based on the theoretical and pragmatic bases of the system.
Chapter 10

Conclusions

In this chapter, we provide our conclusions based on the development and implementation of our ontology-based approach to implementing network-based information systems with a three-tier software architecture. In this thesis, we have motivated the requirement for our approach in Chapter 1 and provided our evidence for the absence of a similar approach through the literature review in Chapter 2. Having then identified the challenges that face the NBIS developer in Chapter 3, we then developed our methodology over Chapters 4, 5 and 6. This leads to the details of implementation in Chapter 7 and, from our operational system, the results in Chapter 8. The previous chapter, Chapter 9, explores what our approach could do, with future extension, or as an application in conjunction with existing technology. We will now summarise the key conclusions and contributions of our work in semantic specification, increasing the ability of distributed information systems to meet user expectations and the advantages of a knowledge-based approach.

The development of network-based systems has led to a need to provide a standards-based framework to allow interoperation in the presence of heterogeneous underlying networks and systems. This standards-based approach has been very successful but is only required because existing mechanisms for discourse cannot guarantee that the contextual framework is the same for an arbitrary number of participants. Thus, we can have globally defined systems but at the cost of local variation, in most cases. If participants can agree upon a local exchange mechanism based on their mutual understanding of the knowledge contained in such a system, then there is scope for the development of locally available aspects to globally distributed systems or variation in the versions of the systems that are supported.
10.1 Specifying Semantics

In this section, we discuss the importance of semantic specification and how a good semantic specification does not rely on the happy accident of aligned syntax conveying semantic equivalence. This has important implications on the use of standards as many standards have a syntactic specification that is backed by a non-machine-interpretable written set of semantics. If semantic specification is sound, then standards can be written at the semantic level and interpreted at that level by the implementing machines.

Syntactic specification often takes the place of semantic alignment, especially where knowledge-based frameworks are not in use. Using syntax as an approximation for a semantic framework only works through the use of implicit semantic overlays based on a common, non-machine-interpretable understanding of what is supposed to be happening. Although approaches such as OWL-S provide mechanisms to specify the details of computation, and hence reduce ambiguity, the majority of systems specify structures for interchange and then proceed on the assumption that structural matching will sufficiently narrow the possibility for semantic mismatch. The majority of existing systems use the implicit ontological specification of the problem domain that is found in the heads of the participants to apply a layer of implicit meaning to well-defined structural representations.

A system may be defined through the definition of the types of messages that travel between its nodes and the role of each node, the ways that business logic is applied to each message and in the actions that are taken upon the application of such logic. Such a definition does not necessarily provide enough information to be able to determine what would happen if a different logic, or a slightly different message protocol, were used. This deviation of definition does not occur when a standards-based system is in use, as non-conforming elements are rejected based on the standards-based acceptance template. However, this rejection is rarely a problem for the majority of users as the standard is often highly specified to meet the perceived user needs. In other words, bad messages are rejected but that’s acceptable behaviour as no-one wishes to intentionally exchange bad messages. The only problem with this is that any variation from a standard, however beneficial, will be seen as a bad message until the change becomes part of, or tolerated by, the standard.

While standards are highly advantageous they can also stifle creativity and novelty in the development of new ways of dealing with the data that is contained within a system. The benefits of a standards-based approach outweigh the costs if it is the case that any working global system, even if it is not a good local fit, is preferable to no system. Ultimately, a standards-based approach is an inflexible approach to the provision of the knowledge represented by the system data as the knowledge cannot be used to directly evolve the
system without potentially violating the standard. Not only must the system function be capable of evolution but so, also, must the way in which knowledge can be described. Any knowledge that cannot be acted upon is only accessible as implicit knowledge in the system and it is the implicit knowledge that cannot be captured - this prevents a system from being able to use that knowledge in a machine-interpretable environment to alter its operations in a more agile way.

10.2 Contribution

This thesis has documented the development of a new approach to distributed systems in general, and network-based information systems specifically, where the explicit specification of the structures, relationships and operational semantics of the values in the system provide an explicit representation of the ontological intent of the designer and user. It is impossible to misuse values outside of their knowledge framework because this would require ignoring either the logical constraints, or the correct representation, or the legitimate operations allowed upon the value. We cannot ignore the system semantics and, as a result, restrict the possible set of operations to those that are acceptable to our system specification. Additionally, we can provide different ways of using the same stored data, a set of different views, in order to allow standards-based behaviour at the same time as new locally-desirable activities.

This works in conjunction with large, heavily-standardised systems as it can be used to provide nodes that give all of the appearances of conforming to the standard while, internally, remaining capable of using this knowledge in ways that do not depend upon fixed, structural encodings. Users of such a system may choose to have a local interpretation of some data or operations that, providing it does not intrude upon the system’s core operations, may operate in parallel with the global users completely unaware of their actions. This provides the missing localisation that, in the increasing demand for decentralised management of resources and agile, user-responsive systems, is vital if the system is to meet all needs - not just the needs of a majority.

Standardisation can be used to manage transitions between versions of a system, guaranteeing data exchange between different versions, while also allowing a series of synchronisation points for major revisions or global updates. However, it is rare that a system will completely change the knowledge that it was designed to represent as this would imply that the new system had no features in common with the old system and this is not an evolution but a complete replacement at the expense of the original knowledge domain. By employing knowledge domains to maintain separate but equally accessible views of a system over
the lifespan of several versions, multiple versions of a system’s standards can still inter-
operate for as long as the definition of the previous version is allowed to exist. This is a
far more lightweight way to provide multiple versions of the same system as a knowledge-
based approach removes the need to leave an old server configured under the old system
version. Instead, a SAMPAN server that can work in multiple versions can support legacy
clients without requiring a different server version or additional hardware.

We have demonstrated that a knowledge-based approach will allow local variations to
continue, if they are useful to the local community, providing that they are not inimical to
the operation of the core group of behaviours. Where a feature is deprecated and eventually
removed from the standard, this knowledge can be retained in the local variations and
continue to be used. The ontological mechanism shown in this thesis provides a straight-
forward approach to allow the migration of knowledge from standard to local and can
be as simple as editing a single file to move the relationships between entities or change
namespace specifications.

10.3 Meeting User Expectations of a Distributed Informa-
tion System

Our knowledge-based approach allows the mapping of the system into a set of components
that can then be used by the ontological tier, supported by the data model, to allow ab-
straction away from the underlying formats and mechanisms. A user’s expectations of a
system may not always be in agreement with what the system actually provides - especially
if a very large number of users participate in a widely-distributed system where each adds
some subtlety of interpretation to a centrally-defined system. The local providers of an
information source may also wish to be able to manipulate it for their own requirements or
provide the information in a set way for storage, legal or other reasons. Without a means
for providing extended functionality or alternative ways of viewing the data, the users must
use another system, which may have to be developed in-house, in order to get the function-
ality that they expected to get from the original. Where the system behaviours are specified
in a visible, shareable and verifiable manner, user expectations cannot reasonably diverge
too far from the system specifications as they are clearly presented.

Our approach allows users to focus on the manipulation of knowledge, rather than the
details of byte alignment. This is not just a functional focus but it is also a philosophical
focus. Without a well-defined representation, with the relationships to other values and
a list of well-defined operational semantics, a value is not knowledge - it is merely data.
An extension mechanism that allows knowledge to be transformed for purposes of interchange, rather than requiring a strict data translation through shared syntax and implied semantics, is going to be easier to program, have well-defined failure modes and also allow local variation to co-exist with global standards providing that the specification is correctly partitioned. The approach described in this thesis meets all of these criteria and provides a firm basis for implementing a system that meets local and global need simultaneously, while retaining a straight-forward approach to manipulating knowledge.

10.3.1 Meeting Global and Local Requirements

We demonstrated the ability to introduce local changes and keep global standards in the SAMPAN server through the introduction of different operational semantics for Internal and External node in our target DNS server, combined with the introduction of non-standard resource records which were available only for enhanced servers. We can generalise this example to demonstrate the separation of a network environment into a class of machines internal to the domain and a class for those machines that reside without. This separation can also provide a clear line of demarcation between the global version of a standardised service and the local, potentially modified, version of the same service. The ability to handle global and local needs at the same time, but without introducing conflicts, is a distinct advantage in the implementation of widely-distributed systems. This does not make the use of standards undesirable, but a standards-based approach can put more distance between user expectations and system operation.

When the information and functionality of two information systems are joined together, the union of the data sets must be determined in a way that reflects both the original systems and the requirements that led to the production of the union. We demonstrated this with the NTP/DNS/DNS metadata union answer produced for Semantic Web applications in Section 7.15. For example, there is little point in linking NTP data to DNS data if the required NTP data are omitted from the final union. The association of two different data sets requires a common knowledge framework - there must be some informal ontology that describes the union of the two systems as, otherwise, why is this data being associated? The explicit statement of the ontology of both systems, with a common knowledge framework joining them, allows a much simpler mechanism for joining the data from two systems and for presenting the information back to the network. By specifying the two systems in terms of their knowledge and then unifying that knowledge, the existing mechanisms for translating the individual knowledge items can be applied to each item and the final result passed out of the system in the required protocol specification. As the specification gives
order and type information, it is only the arrangement of the values that is of concern and this can be managed by using the well-defined, context-rich names of the ontologically based system.

10.3.2 Performance Implications

The overheads of our knowledge-based approach can be significant, as was shown in Chapter 8. Our complexity analysis clearly indicates that careful attention must be paid to the benefits gained from the use of such a system. While any protocol that requires parsing can introduce high-complexity aspects, the known impact of using an information storage mechanism with high time complexity will increase the weight of clients and servers substantially. Clients have to be kept relatively small and agile as they may be used from anywhere in the environment. Servers can have larger overheads but keep them better concealed because of the tendency to start up, operate for a longer time and then be shut down or restarted. Clients may have a far shorter time between start up and shut down so a long start up time is far more noticeable.

This leads to questions of the overall performance of such a system and why these servers are often referred to as boundary nodes. Where translation is required to bridge the gap between systems, and it is useful that the systems be joined, performance is often less of an issue as the translation is required to provide the useful system that results from the union. Providing that it occurs in reasonable time, some performance overhead is acceptable. Rather than implementing an entire system in these highly configurable nodes, the overall layout is more likely to consist of standardised nodes that perform basic function and adapter nodes that then join these with other systems or provide additional function for a small group of users. This, in turn, leads to the question of replication and recovery in such a system as the different nodes may need to have their own replicas as they cannot depend upon the usual peer relationship for failover as their peers may not have the required functionality. Figure 10.1 shows the expected deployment pattern for the nodes, emphasising the boundary location.

In Chapters 7 and 8, we emphasised the cost/benefits ratio to demonstrate the feasibility of our approach. Because of the overheads of the additional processing, that may be ameliorated by careful choice of underlying technologies but never fully diminished, the benefits must be significant. The proof-of-concept system showed that the addition of new functionality could, with a good underlying design, take place in a straightforward but abstract manner. This removed the requirement for the user to provide structural manipulations and allowed them to focus on the the way that they wished to project the information
Figure 10.1: The expected deployment of adapter nodes versus standard nodes contained in a controlled repository.

10.4 The Advantages of a Knowledge-Based Approach

The benefits obtained from such a mechanism must compensate the developer for the time taken to develop the ontology, the storage overheads and the requirement to run a node that is unlike its peers from an internal perspective, despite its outward similarity. The major benefits that have been shown in this thesis will now be discussed individually to show why such an approach is worth the effort and time.

10.4.1 Explicit Ontological Representation Minimises Confusion

The first advantage is that the explicit ontological capture requires all users to agree upon an explicit ontological representation of the system. This may seem obvious but the implicit ontological agreement which system participants share, for any system, is a wellspring of misunderstanding, comprehension problems and potential errors. Through explicit representation, the system is laid bare and this, in turn, facilitates understanding of, and potential extension and development in, the system. We cannot define functions within the ontology below the boundary between the ontological tier and the data model, especially for the
native functions that are ontologically opaque.

### 10.4.1.1 Shared Concepts Allows Much Wider Reuse of Data

The agreement between users as to how a system is composed allows the users of a shared ontology to make references to concepts and any shared individuals without having to establish an ad-hoc semantic framework. This then allows the formation of new protocols for exchange that are optimised for the purposes of the store owner and the store user. Large systems with distributed storage of information can place a storage burden on the component systems without necessarily meeting the needs of all of the component node providers. While most of their needs may be met, it is still possible that the data that the system stores is never made available to those users storing it in the form that would be the most use to them. A shared ontology, based on a separation of the ontological representation into logical, structural and operational requirements, allows the knowledge in the system to be manipulated into any desired format and joined with other data sources as required because it does not affect the standards-based operation.

The RDF form of the data provides a common basis for any transformation operations as the value, the type of the value and the context of the value are stored. In this format, the value can easily be extracted into a variable of the determined type and any context-based relationships can be exploited to make better use of the value. This provides, in effect, a normal form for the value as, once defined, any further references to this value can be made through a reference to its ontologically-defined value. This reduces redundancy issues and dependence of the updating of redundant entries in a database.

### 10.4.2 Reduced Double-Handling of Data

The normal-form representation of the system reduces redundancy throughout the entire system as the contextually derived name for a value is accessible throughout the entire system and is not limited in scope to a functional element. This is due to the global nature of the system knowledge and the requirement that, to be committed to the store, the knowledge must be a fact and not a time-dependent piece of knowledge. If a set of values have been sent to a user then the knowledge may be stored that, at a certain time, a certain message was sent to a user and these were the values used. This is highly-specialised knowledge as some of the message elements may reflect facts in the system that have changed.

As discussed in caching possibilities in Chapters 7 and 8, the use of the strong context for naming values allows a partial cache expiry, or even update, in the face of local knowledge changes. This provides a time-based classification of the knowledge in the system to
things that are always the same, things that change slowly and things that change quickly. The structural nature of most systems is, at its core, fixed. Even if some elements are added, there is a core structure that is unchanging. This is knowledge. There are then a set of data values that are the deliverable elements of the system and, depending on the system, these can be slowly or rapidly changing. Since these are the deliverables of the system then these also constitute knowledge as the client is using the system to provide this knowledge at a certain time.

Finally, there is ephemeral knowledge that is of such a short lifespan that it cannot be provided externally because it is not guaranteed to be correct after the completion of the current operation. Separating these classes of knowledge, keeping them separated and preventing ephemeral data from accidentally becoming knowledge would be much harder without a clear distinction between the knowledge and data layers. This is a major benefit of the approach taken in this thesis.

10.4.3 Increased Opportunities for Collaboration

Remote participants can form a collaborative environment for developing new aspects for a system without having to share file system access, using ontological alteration and remotely installable native functions. This provides an equivalent development model to the web-based check-in systems that can be used for distributed collaboration. However, the use of an ontology allows the use of validation tools to verify that the ontology has been extended correctly. In cases where description logic graphs or subgraphs can be produced, structural and logical relationships can be checked to ensure that new changes do not compromise the standards core or prevent the operation of other key functions. The use of ontologies also allows the core ontology to be separated and imported into another ontology that then builds the extensions on top of the core.

The separation of the ontology into separate structures also suggests patterns for subgraph production where clients do not need all of the available functionality. If a client only needs structural or logical information and not the details of underlying server semantics, then this is a much smaller ontology to work with, reducing load time and overall time to execute. This can facilitate the development of agile, specialised clients rather than having to use a large client that can meet every need but is very large and very slow to operate due to a large number of possible operations encoded in guard statements.
10.4.4 Decoupling of Stored Data from External Representations

The most significant benefit of our approach is the explicit and complete decoupling of the stored data from the external representations that are used to exchange that data. The transformation mechanism takes knowledge, from the store abstraction, and produces data in wire format. The nature, efficiency and design of the store is not prescribed so the store can be replaced with any technology that can store the triples and make them available on demand. This allows the use of technologies such as fixed-format databases for their speed, without enforcing an SQL based query mechanism on top, although such a mechanism could be used through the SPARQL abstraction with an appropriate library. This enforces the separation of the aspects of the program that are concerned with efficiency and those aspects of the program that are concerned with knowledge management.

The decoupling allows the presentation of knowledge in the most useful form to the consumer. The decoupling process requires that the data in the system be capable of becoming knowledge and this, in turn, requires that a user describes the system in sufficient detail to allow this process to happen in a machine-controlled and reliable manner. This allows users to take advantage of all the possible benefits that can be realised from their data, rather than forcing them to consume their own data in a form that is potentially not that useful to them. Many decoupled data stores can have their knowledge combined to provide composite answers in response to complex queries. The ability to assemble complex responses through the association of knowledge reduces the need to have fixed-format translator nodes between systems in order to facilitate knowledge fusion. Since protocols can be agreed upon between SAMPAN systems in a very flexible and extensible way, these complex responses can rapidly find their way into the day-to-day traffic between enhanced servers and clients.

Protocol evolution and alteration can also significantly reduce network traffic. Combined with flexible operational semantics, servers can be altered to reach the best network usage pattern for the network upon which they are deployed. The use of UDP packets in DNS, the default behaviour, will only lead to a large number of long timeouts in a network with large packet loss. Switching the operational mode to TCP will reduce the timeout period at the expense of more packets transmitted. Alternatively, the timeout period can be dropped based on knowledge specified in the shared ontology to change the length of time over which systems will wait. All of these can be achieved more simply by making changes in the descriptive ontology than would be possible if source recoding were required for client and server. Even if some data model tier encoding is required, the level of abstraction already present in the data model simplifies coding changes as all knowledge can only be dealt with through the associative array holding the knowledge, with references
back to the store. There are no context-free global variables that can be misinterpreted to cause additional programming problems.

In conclusion, this thesis has shown that the use of widely-distributed network-based systems with an information focus can benefit from an ontological representation of their operations, employing a multi-tier model to separate data from information and information from knowledge while allowing controlled transformations between the different stages. Despite the performance overheads that are inevitably incurred by adding additional layers of function, this approach allows users to manipulate data in new ways without compromising the core functionality of a standards-based system.
Appendix A

The core ontology for DNS operations

This appendix contains the core ontology used for the α- and β-implementations and discussed in Chapter 7. The results of the experimentation carried out, based on this ontology, may be found in Chapter 8. As the ontology evolved over time, there are some redundant aspects that remain in the ontology despite no longer being necessary for the β-implementation. They are, however, required for the correct operation of the α-implementation.

This ontology only contains a small amount of the actual data used for implementation testing, and is restricted to the core of the implemented DNS-compliant systems, due to space constraints in this thesis.

Where necessary, line breaks have been inserted to force the lines to remain on the page. Other formatting changes have been made in the transition to this form, for readability.

```xml
<rdf:RDF
 xmlns="#dns;#" xmlns:dns="#dns;#" xmlns:owl="#owl;#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:xsd="#xsd;#">
<owl:Ontology rdf:about="#">
  <rdfs:comment>An example DNS zone file ontology in OWL. This one uses datatype properties for field values.</rdfs:comment>
  <rdfs:label>DNS Zone File Ontology</rdfs:label>
</owl:Ontology>
<owl:Class rdf:ID="OntologyParent"/>
<owl:Class rdf:ID="LogicalBranch">
  <rdfs:subClassOf rdf:resource="#dns;OntologyParent"/>
</owl:Class>
<owl:Class rdf:ID="RepresentationalBranch">
  <rdfs:subClassOf rdf:resource="#dns;OntologyParent"/>
</owl:Class>
</rdf:RDF>
```
<owl:Class rdf:ID="FunctionalBranch">
    <rdfs:subClassOf rdf:resource="&dns;OntologyParent"/>
</owl:Class>

<owl:Class rdf:ID="Global">
    <rdfs:subClassOf rdf:resource="&dns;LogicalBranch"/>
</owl:Class>

<owl:Class rdf:ID="Domain">
    <rdfs:subClassOf rdf:resource="&dns;Global"/>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasDomainName"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasDomainName">
    <rdfs:domain rdf:resource="#Domain"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="RepresentationOfRR">
    <rdfs:subClassOf rdf:resource="&dns;RepresentationalBranch"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasRRName"/>
            <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasActualRepresentation"/>
            <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasRRName">
    <rdfs:domain rdf:resource="#RepresentationOfRR"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="wireTransformation">
    <rdfs:subClassOf rdf:resource="&dns;actualRepresentation"/>
</owl:Class>

<owl:Class rdf:ID="actualRepresentation">
    <rdfs:subClassOf rdf:resource="&dns;representationOfRR"/>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasWireTransformation"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasWireTransformation">
    <rdfs:domain rdf:resource="#actualRepresentation"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="ipOctet">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasNumericValue"/>
            <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasNumericValue">
    <rdfs:domain rdf:resource="#representationOfRR"/>
    <rdfs:range rdf:resource="&xsd;positiveInteger"/>
</owl:DatatypeProperty>

<owl:FunctionalProperty rdf:ID="has_contents"/>
<owl:FunctionalProperty rdf:ID="Structural_Sequence">
  <rdfs:range rdf:resource="#Argument_list"/>
  <rdfs:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain rdf:resource="#actualRepresentation"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="Rest_of_List">
  <rdfs:range rdf:resource="#Argument_list"/>
  <rdfs:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain rdf:resource="#Argument_list"/>
</owl:FunctionalProperty>

<ipOctet rdf:ID="singleOctet"/>

<ipOctet rdf:ID="#_192">
  <hasNumericValue rdf:datatype="&xsd;positiveInteger">192</hasNumericValue>
</ipOctet>

<ipOctet rdf:ID="#_168">
  <hasNumericValue rdf:datatype="&xsd;positiveInteger">168</hasNumericValue>
</ipOctet>

<ipOctet rdf:ID="#_001">
  <hasNumericValue rdf:datatype="&xsd;positiveInteger">1</hasNumericValue>
</ipOctet>

<Argument_list rdf:ID="Empty_List"/>

<Argument_list rdf:ID="#Octet_4">
  <has_contents rdf:resource="#singleOctet"/>
  <rest_of_list rdf:resource="#Empty_List"/>
</Argument_list>

<Argument_list rdf:ID="#Octet_3">
  <has_contents rdf:resource="#singleOctet"/>
  <rest_of_list rdf:resource="#Octet_4"/>
</Argument_list>

<actualRepresentation rdf:ID="AR_A">
  <Structural_Sequence>
    <Argument_list rdf:ID="#Octet_1">
      <rest_of_list>
        <Argument_list rdf:ID="#Octet_2">
          <has_contents rdf:resource="#singleOctet"/>
          <rest_of_list rdf:resource="#Octet_3"/>
        </Argument_list>
      </rest_of_list>
    </Argument_list>
  </Structural_Sequence>
</actualRepresentation>

<owl:Class rdf:ID="stringObject">
  <rdfs:subClassOf rdf:resource="&dns;RepresentationalBranch"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasStringValue"/>
      <owl:maxCardinality rdf:datatype="&xsd;string">1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasStringValue">
  <rdfs:domain rdf:resource="&dns;OntologyParent"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<actualRepresentation rdf:ID="AR_TXT">
  <Structural_Sequence>
    <Argument_list rdf:ID="#TXTString"/>
  </Structural_Sequence>
</actualRepresentation>
<owl:DatatypeProperty rdf:ID="Hostname">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="A">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="CNAME">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="TXT">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="ITXT">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="AdvertisedIP">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="TTL">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="&xsd;nonNegativeInteger"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="hasTTL">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="#TTL"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasIClass">
    <rdfs:domain rdf:resource="#ResourceRecord"/>
    <rdfs:range rdf:resource="#IClass"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#ResourceRecord">
    <rdfs:label xml:lang="en">Resource Record</rdfs:label>
    <rdfs:comment xml:lang="en">This class provides a means of describing the set of DNS resource records associated with a given hostname.</rdfs:comment>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#TTL"/>
        <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
    </owl:Restriction>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasIClass"/>
        <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
    </owl:Restriction>
</owl:Class>

<RepresentationOfRR rdf:ID="RepOfA">
    <hasName rdf:datatype="&xsd;string">A</hasName>
    <hasActualRepresentation rdf:resource="#AAA"/>
</RepresentationOfRR>
<owl:ObjectProperty rdf:ID="hasMXName">
  <rdfs:domain rdf:resource="#MailHandler"/>
  <rdfs:range rdf:resource="#MXName"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPreference">
  <rdfs:domain rdf:resource="#MailHandler"/>
  <rdfs:range rdf:resource="#Preference"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasMailHandler">
  <rdfs:domain rdf:resource="#ResourceRecord"/>
  <rdfs:domain rdf:domain="#MailHandler"/>
</owl:ObjectProperty>

<MXName rdf:ID="mail.testdomain.org"/>

<MXName rdf:ID="smtp0.cs.adelaide.edu.au"/>

<Preference rdf:ID="pref10"/>

<MailHandler rdf:ID="_10_mail.testdomain.org">
  <hasMXName rdf:resource="#mail.testdomain.org"/>
  <hasPreference rdf:resource="#pref10"/>
</MailHandler>

<MailHandler rdf:ID="_10_smtp0.cs.adelaide.edu.au">
  <hasMXName rdf:resource="#smtp0.cs.adelaide.edu.au"/>
  <hasPreference rdf:resource="#pref10"/>
</MailHandler>

<owl:Class rdf:ID="ProtocolSpecification">
  <rdfs:subClassOf rdf:resource="#dns:RepresentationOfRR"/>
</owl:Class>

<owl:Class rdf:ID="Preamble">
  <rdfs:subClassOf rdf:resource="#dns:ProtocolSpecification"/>
</owl:Class>

<owl:Class rdf:ID="Postamble">
  <rdfs:subClassOf rdf:resource="#dns:ProtocolSpecification"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasPreamble">
  <rdfs:domain rdf:resource="#ProtocolSpecification"/>
  <rdfs:range rdf:resource="#Preamble"/>
</owl:ObjectProperty>
<ProtocolSpecification>
  <Order rdf:datatype="&xsd;string">MsgID,Flags,NumQueries,NumReplies,NumAuthRecs,NumAddRecs,QName,QType,QClass</Order>
</ProtocolSpecification>

<owl:Class rdf:ID="RROutputClass">
  <rdfs:subClassOf rdf:resource="#ResourceRecord"/>
</owl:Class>

<owl:DatatypeProperty rdf:ID="Name">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="Type">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="&xsd;unsignedShort"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="Klass">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="RDLength">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="&xsd;unsignedShort"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="RDATA">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="hasRRDataModelOptions">
  <rdfs:domain rdf:resource="#RROutputClass"/>
  <rdfs:range rdf:resource="#DataModelOptions"/>
</owl:ObjectProperty>

<DataModelOptions rdf:ID="Contiguous"/>
<DataModelOptions rdf:ID="Multiple"/>

<RROutputClass rdf:ID="Answer">
  <Name rdf:datatype="&xsd;string"/>
  <Type rdf:datatype="&xsd;unsignedShort"/>
  <Klass rdf:datatype="&xsd;string"/>
  <TTL rdf:datatype="&xsd;nonNegativeInteger"/>
  <RDLength rdf:datatype="&xsd;unsignedShort"/>
  <RDATA rdf:datatype="&xsd;string"/>
  <Order rdf:datatype="&xsd;string">Name,Type,Klass,TTL,RDLength,RDATA</Order>
  <associatedCounter rdf:datatype="&xsd;string">NumReplies</associatedCounter>
  <hasRRDataModelOptions rdf:resource="#Contiguous"/>
  <hasRRDataModelOptions rdf:resource="#Multiple"/>
</RROutputClass>

<RROutputClass rdf:ID="Authority">
  <Name rdf:datatype="&xsd;string"/>
  <Type rdf:datatype="&xsd;unsignedShort"/>
  <Klass rdf:datatype="&xsd;unsignedShort"/>
  <TTL rdf:datatype="&xsd;nonNegativeInteger"/>
  <RDLength rdf:datatype="&xsd;unsignedShort"/>
  <RDATA rdf:datatype="&xsd;string"/>
  <Order rdf:datatype="&xsd;string">Name,Type,Klass,TTL,RDLength,RDATA</Order>
  <associatedCounter rdf:datatype="&xsd;string">NumAuthRecs</associatedCounter>
  <hasRRDataModelOptions rdf:resource="#Contiguous"/>
  <hasRRDataModelOptions rdf:resource="#Multiple"/>
</RROutputClass>
<RROutputClass rdf:ID="Additional">
  <Name rdf:datatype="&xsd;string"/>
  <Type rdf:datatype="&xsd;unsignedShort"/>
  <Klass rdf:datatype="&xsd;unsignedShort"/>
  <TTL rdf:datatype="&xsd;nonNegativeInteger"/>
  <RDLength rdf:datatype="&xsd;unsignedShort"/>
  <RDATA rdf:datatype="&xsd;string"/>
  <Order rdf:datatype="&xsd;string">Name,Type,Klass,TTL,RDLength,RDATA</Order>
  <associatedCounter rdf:datatype="&xsd;string">NumAddRecs</associatedCounter>
  <hasRRDataModelOptions rdf:resource="#Contiguous"/>
  <hasRRDataModelOptions rdf:resource="#Multiple"/>
</RROutputClass>

<owl:ObjectProperty rdf:ID="hasAnswer">
  <rdfs:domain rdf:resource="#ProtocolSpecification"/>
  <rdfs:range rdf:resource="#Answer"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasAuthority">
  <rdfs:domain rdf:resource="#ProtocolSpecification"/>
  <rdfs:range rdf:resource="#Authority"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasAdditional">
  <rdfs:domain rdf:resource="#ProtocolSpecification"/>
  <rdfs:range rdf:resource="#Answer"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasDiffItem">
  <rdfs:domain rdf:resource="#ProtocolSpecification"/>
  <rdfs:range rdf:resource="#differentiatingItem"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="differentiatingItem">
  <rdfs:subClassOf rdf:resource="#RepresentationalBranch"/>
</owl:Class>

<owl:DatatypeProperty rdf:ID="streamByte">
  <rdfs:domain rdf:resource="#differentiatingItem"/>
  <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="dealBreaker">
  <rdfs:domain rdf:resource="#differentiatingItem"/>
  <rdfs:range rdf:resource="&xsd;byte"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="confirmingCharacter">
  <rdfs:domain rdf:resource="#differentiatingItem"/>
  <rdfs:range rdf:resource="&xsd;byte"/>
</owl:DatatypeProperty>

<differentiatingItem rdf:ID="DNSM">
  <streamByte rdf:datatype="&xsd;integer">1</streamByte>
  <dealBreaker rdf:datatype="&xsd;byte">80</dealBreaker>
  <confirmingCharacter rdf:datatype="&xsd;byte">85</confirmingCharacter>
</differentiatingItem>

<differentiatingItem rdf:ID="DHSM">
  <streamByte rdf:datatype="&xsd;integer">1</streamByte>
  <dealBreaker rdf:datatype="&xsd;byte">109</dealBreaker>
  <confirmingCharacter rdf:datatype="&xsd;byte">85</confirmingCharacter>
</differentiatingItem>

<differentiatingItem rdf:ID="HTTPM">
  <streamByte rdf:datatype="&xsd;integer">1</streamByte>
  <dealBreaker rdf:datatype="&xsd;byte">80</dealBreaker>
  <confirmingCharacter rdf:datatype="&xsd;byte">85</confirmingCharacter>
</differentiatingItem>

<ProtocolSpecification rdf:ID="HTTPPacket">
  <hasDiffItem rdf:resource="#HTTP"/>
</ProtocolSpecification>
</CompositeRecord>
<CompositeRecord rdf:ID="ns2.adelaide.edu.au."/>
<A rdf:datatype="&xsd;string">129.127.41.3</A>
</CompositeRecord>

</rdf:RDF>
Bibliography


