Name: Gu Tao
Degree: Doctor of Philosophy
Dept: Department of Computer Science
Thesis Title: A Semantic Approach for Scalable and Self-organized Context-aware Systems

## Abstract

In this thesis, we investigate and study on a set of core infrastructure services to simplify the task of building reliable and scalable context-aware applications in pervasive computing environments. Particularly, we focus on two problems: Providing a scalable and robust context lookup service in wide-area networks, and an efficient context processing mechanism. To tackle these problems, we propose a semantic P2P overlay network for efficient context lookup, and a distributed logical reasoning mechanism for context interpretation. We carry out comprehensive simulations to evaluate the performance of the proposed overlay network. The results show that our system possesses good scalability, better search efficiency and low overlay maintenance overhead. We also develop a working prototype system to demonstrate how our proposed techniques work in a practical way, and build several typical context-aware applications on top of our prototype. Our experiences show that our system works effectively in a real-world setting and the application development process is greatly simplified.

**Keywords**: *Context-aware computing, Context model, Context ontologies, Semantic P2P lookup, Query routing, Subscription, Context reasoning*

# A SEMANTIC APPROACH FOR SCALABLE AND SELF-ORGANIZED CONTEXT-AWARE SYSTEMS

GU TAO

NATIONAL UNIVERSITY OF SINGAPORE

2005

# A SEMANTIC APPROACH FOR SCALABLE AND SELF-ORGANIZED CONTEXT-AWARE SYSTEMS

GU TAO
*(B.Eng., HUST and M.S., NTU)*

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2005

# Acknowledgements

I would like to thank my supervisors, Dr. Pung Hung Keng (Associate Professor, School of Computing, National University of Singapore), Dr. Andreas Fasbender (Director of Ericsson Cyberlab Singapore) and Dr. Zhang Daqing (Institute of Infocomm Research). Their guidance, experience and encouragement have been invaluable for my research. I appreciate all of their support, suggestions and comments throughout the work. I also want to thank Dr. Dong Jin Song and Dr. Ooi Wei Tsang for their many useful comments and their time in reviewing this thesis.

I would like to thank all members in the Networks Systems and Services Lab (previously named Center for Internet Research) including He Jun, Peng Bin, Zhou Lifeng, Feng Yuan, Qian Haichun and Yao Jiankang. In particular, I would like to thank Edmond Tan who has been spending his time discussing with me some research and implementation issues related with this work.

Last but not least, I would like to thank my parents, Gu Weiyi and Wang Meiyun, my sister Gu Qi for their unwavering support and encouragement. My special thanks go to my wife Yang Yanfang. I am indebted to my wife Yanfang and my two lovely kids, for their love, and for more than I will ever be able to express.

# Table of Contents

Acknowledgments ..... i

Table of contents ..... ii

Summary ..... vii

Publications ..... ix

Papers submitted for review ..... xi

List of tables ..... xii

List of figures ..... xiii

1 Introduction ..... 1

1.1 Understanding context in pervasive computing ..... 2

1.1.1 Context definition ..... 2

1.1.2 Characteristics of context information ..... 3

1.2 Functional requirements of system's infrastructure for context-aware computing ..... 6

1.3 Problem statement ..... 10

1.4 Our Approach ..... 12

1.4.1 Ontology-based context modeling ..... 12

1.4.2 A semantic Peer-to-Peer overlay ..... 12

1.4.3 Distributed context reasoning ..... 15

1.4.4 Research methodology ..... 15

1.5 Thesis contributions and outline ..... 16

2 Related work ..... 18

2.1 Context-aware systems ..... 18

2.1.1 Existing infrastructure-based systems ..... 19

2.1.2 Summary ..... 24

2.2 Information retrieval ..... 25

2.2.1 The centralized approach ..... 25

2.2.2 The P2P approach ..... 25
2.2.2.1 Unstructured P2P systems ..... 26
2.2.2.2 Structured P2P systems ..... 28
2.2.2.3 Semantic P2P systems ..... 29
2.2.3 Summary ..... 32
3 Context modeling and reasoning ..... 33
3.1 Motivation and our context modeling approach ..... 33
3.2 An ontology-based context model ..... 34
3.3 Context ontologies ..... 35
3.4 Context reasoning ..... 37
3.4.1 Ontology reasoning ..... 37
3.4.2 User-defined rule-based reasoning ..... 38
3.4.3 Reasoning performance and discussion ..... 39
3.5 Summary ..... 41
4 P2P context lookup ..... 43
4.1 Architecture overview ..... 43
4.2 Ontology-based semantic clustering ..... 46
4.3 ContextBus ..... 49
4.3.1 Bootstrapping ..... 51
4.3.2 Routing ..... 52
4.3.3 Discussion ..... 52
4.4 Semantic Context Space ..... 53
4.4.1 Peer placement ..... 53
4.4.2 Cluster naming scheme ..... 55
4.4.3 Ring construction ..... 55
4.4.4 Cluster splitting and merging ..... 56
4.4.5 The routing algorithm ..... 58
4.4.6 Subscription ..... 63
4.4.7 Peer dynamics and failure ..... 66
4.5 Performance evaluation and comparison ..... 67
4.5.1 Simulation model ..... 68
4.5.2 Performance metrics ..... 70

4.5.3 Simulation results ..... 70
4.5.3.1 Search efficiency ..... 70
4.5.3.2 Overheads ..... 73
4.5.3.3 Clustering effects ..... 75
4.5.3.4 Selection of shortcuts ..... 78
4.5.3.5 Load balancing ..... 79
4.6 Summary ..... 79

5 Cost-aware selective flooding ..... 81

5.1 Related work ..... 83
5.2 Neighborhood link cost measurement ..... 86
5.3 Basic routing algorithm ..... 87
5.4 Routing decision mediation ..... 91
5.5 The main algorithm ..... 92
5.6 A case study ..... 95
5.7 Performance evaluation ..... 98
5.7.1 Simulation model and metrics ..... 98
5.7.2 Simulation results ..... 99
5.8 Summary ..... 101

6 Prototype implementation ..... 102

6.1 Overview ..... 103
6.2 Bootstrapping ..... 104
6.2.1 Semantic cluster mapping ..... 105
6.2.2 SWebCache ..... 107
6.2.3 Connection ..... 109
6.2.4 Reference registration ..... 113
6.3 Message receivers ..... 114
6.4 Message forwarding and processing ..... 115
6.4.1 Message forwarding ..... 116
6.4.2 Message processing ..... 117
6.5 Search and subscription ..... 118
6.5.1 Query types ..... 119
6.5.2 Query messages ..... 119

6.5.3 QueryHit messages ..... 120
6.5.4 Subscriptions ..... 121
6.6 LookupClient ..... 122
6.6.1 Initiating queries ..... 123
6.6.2 Receiving query responses ..... 123
6.7 Context Producer ..... 123
6.7.1 Initiating queries ..... 124
6.7.2 Sensor management ..... 125
6.7.3 Context data management ..... 127
6.7.4 Query management ..... 128
6.7.4.1 Local context lookup ..... 128
6.7.4.2 Subscription acceptance ..... 129
6.7.4.3 Subscription response ..... 130
6.8 Context Interpreter ..... 131
6.8.1 Deduced query registration ..... 132
6.8.2 Rule management ..... 133
6.8.2.1 Determination of matching rules ..... 134
6.8.2.2 Rule instantiation ..... 134
6.8.3 Deduced query management ..... 135
6.8.4 Context data reasoning ..... 136
6.8.4.1 Internal query generation ..... 136
6.8.4.2 High-level context data derivation ..... 136
6.9 Development of context-aware applications ..... 137
6.9.1 SCS APIs ..... 138
6.9.2 Sample context-aware applications ..... 138
6.9.2.1 SmartHome application ..... 139
6.9.2.2 ShoppingAssistant application ..... 145
6.9.3 Other scenarios and ongoing work ..... 147
6.10 Prototype evaluation ..... 148
6.10.1 The prototype testbed ..... 148
6.10.2 Bootstrapping ..... 151
6.10.3 Dynamic characteristic ..... 153
6.10.4 Query response time ..... 154
6.10.5 Query processing capability ..... 157

6.10.6 Improving deduced query processing ..... 158
6.10.7 Memory consumption for deduced query processing ..... 162
6.10.8 Validation of our simulation model ..... 163
6.11 Summary ..... 164
7 Conclusion and future work ..... 165
7.1 Summary ..... 165
7.2 Future work ..... 166
Bibliography ..... 170
Appendix A – The upper ontology and a set of domain-specific ontologies ..... 180
Appendix B – User-defined rules in the SmartHome application ..... 190
Appendix C – An example of domain-specific context ontologies such as grocery store, book store and child care center used in the ShoppingAssistant application ..... 192
Appendix D – Sample queries used in prototype evaluations ..... 197

# Summary

The advancement of context-aware computing allows users, devices and services to be aware of and automatically adapt to their physical and computational environments. In recent years, many context-aware systems have been built to meet the required levels of autonomy and flexibility for advanced applications. Context information plays a key role in proliferating and enmeshing computation into our lives.

In this thesis, we aim to provide infrastructure support for designing scalable and self-organized context-aware systems, and easing the development of context-aware applications over multiple context spaces. We identify a set of core services – context lookup and context processing, coupled with a context representation model, and propose solutions for each of them. For context modeling, we propose an ontology-based model to represent context information in a machine-understandable and machine-processable fashion. For context lookup, we propose a semantic P2P overlay network to provide users and applications with an efficient lookup service. We develop various techniques to meet scalability and dynamicity requirements such as an ontology-based semantic clustering scheme for fast semantic abstraction, a one-dimensional ring space for reducing overlay maintenance cost and enabling efficient routing, cluster splitting and merging for self-scaling to number of context producer peers, a cost-aware selective flooding algorithm for minimizing redundant query messages, and a context push service for notifying context consumers about changes quickly. For context processing, we propose a distributed logical reasoning approach to interpret various contexts. Through logical reasoning, we are able to raise the level of context abstraction based on users' or applications' needs. Context reasoning is

done in a distributed fashion because a centralized reasoning engine may not be scalable (due to the single processing bottleneck and the single point of failure).

Comprehensive simulations show that our proposed lookup system offers better search efficiency and incurs low overlay maintenance overhead when comparing with other similar systems. It has good scalability and load balancing characteristics, and is self-organized in nature. We develop a working prototype system to demonstrate how our proposed techniques work practically. The evaluation results of the prototype show that our system works effectively in a real-world setting. We also develop several typical context-aware applications to illustrate the development process. Our experiences show that the application development process is greatly simplified with our approach. This is because the application developers need only focus on application-level tasks without wasting time and efforts on low-level details.

# Publications


[1] J. K. Yao, T. Gu, and H. K. Pung. A Jini-based Service Location Manager in OCTOPUS. In Proceedings of the 7th IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA 2003), Honolulu, Hawaii, August 2003.

[2] T. Gu, H. C. Qian, J. K.Yao, and H. K. Pung. An Architecture for Flexible Service Discovery in OCTOPUS. In Proceedings of the 12th IEEE International Conference on Computer Communications and Networks (ICCCN 2003), Dallas, Texas, October 2003.

[3] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang. An Ontology-based Context Model in Intelligent Environments. In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004), San Diego, California, January 2004.

[4] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung. Ontology Based Context Modeling and Reasoning using OWL. In Proceedings of Workshop on Context Modeling and Reasoning (CoMoRea 2004), in conjunction with the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), Orlando, Florida, March 2004.

[5] T. Gu, H. K. Pung, and D. Q. Zhang. A Bayesian Approach for Dealing with Uncertain Contexts. In Proceedings of the Second International Conference on Pervasive Computing (Pervasive 2004), in the book "Advances in Pervasive Computing" published by the Austrian Computer Society, vol. 176, ISBN 3-85403-176-9, Vienna, Austria, April 2004.

[6] T. Gu, H. K. Pung, and D. Q. Zhang. A Middleware for Context-Aware Mobile Services. In Proceedings of IEEE Vehicular Technology Conference (VTC 2004), Milan, Italy, May 2004.

[7] T. Gu, H. K. Pung, and D. Q. Zhang. Towards an OSGi-Based Infrastructure for Context-Aware Applications in Smart Homes. IEEE Pervasive Computing, Vol. 3, Issue 4, 2004.

[8] T. Gu, H. K. Pung, and D. Q. Zhang. A Service-Oriented Middleware for Building Context-Aware Services. Elsevier Journal of Network and Computer Applications (JNCA), Vol. 28, Issue 1, pp. 1-18, January 2005.

[9] T. Gu, H. K. Pung, and J. K. Yao. Towards a Flexible Service Discovery. Elsevier Journal of Network and Computer Applications (JNCA), Vol. 28, Issue 3, pp. 233-248, May 2005.

[10] T. Gu, E. Tan, H. K. Pung, and D. Zhang. ContextPeers: Scalable Peer-to-Peer Search for Context Information. In Proceedings of the International Workshop on Innovations in Web Infrastructure (IWI 2005), in conjunction with the 14th World Wide Web Conference (WWW 2005), Japan, May 2005.

[11] T. Gu, E. Tan, H. K. Pung, and D. Zhang. A Peer-to-Peer Architecture for Context Lookup. In Proceedings of the International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2005), San Diego, California, July 2005.

[12] T. Gu, H. K. Pung, and D. Zhang. A Peer-to-Peer Overlay for Context Information Search. In Proceedings of the 14th IEEE International Conference on Computer Communications and Networks (ICCCN 2005), San Diego, California, October 2005.

[13] T. Gu, H. K. Pung, and D. Zhang. A P2P Context Lookup Service for Multiple Smart Spaces, In Proceedings of the International Conference on Mobile Systems, Applications, and Services (Mobisys 2006), Poster paper. Uppsala, Sweden, June 2006.

[14] T. Gu, H. K. Pung, and D. Zhang. A Hierarchical Semantic Overlay for P2P Search. In Proceedings of the INFOCOM 2006, Poster paper. Barcelona, Spain, April 2006.

# Papers Submitted for Review

[1] T. Gu, H. K. Pung, and D. Zhang. Information Retrieval in Schema-Based P2P Systems. Submitted to a journal.

[2] T. Gu, H. K. Pung, and D. Zhang. A Peer-to-Peer Approach to Context Interpretation in Pervasive Computing Environments. Submitted to a conference.

[3] T. Gu, H. K. Pung, and D. Zhang. ContextPeers: a Distributed Context Lookup System in Pervasive Computing. Submitted to a journal.

[4] T. Gu, H. K. Pung, and D. Zhang. Minimizing Search Cost in Unstructured P2P Systems. Under preparation for a journal.

# List of Tables

Table 3.1: A partial RDFS and OWL rule set ..... 38

Table 3.2: A partial user-defined rule set ..... 39

Table 5.1: Node *x*'s routing table ..... 87

Table 6.1: Various connection flags ..... 112

Table 6.2: The results for the bootstrap process ..... 152

Table 6.3: Results on time-to-stability (without backup links) ..... 154

Table 6.4: Different methods for deduced query processing ..... 159

# List of Figures

Figure 1.1: Overview of a typical context-aware infrastructure ..... 7

Figure 3.1: A partial context ontology written in OWL ..... 35

Figure 3.2: A two-tier approach to context ontologies ..... 36

Figure 3.3: A partial definition of domain-specific ontologies ..... 37

Figure 3.4: Performance of context reasoning ..... 40

Figure 3.5: Reasoning comparison ..... 41

Figure 4.1: An example of semantic cluster mapping ..... 48

Figure 4.2: Overview of the ContextBus architecture ..... 49

Figure 4.3: One-dimensional ring structure ..... 54

Figure 4.4: Query routing ..... 60

Figure 4.5: Pseudocode of the search algorithm ..... 61

Figure 4.6: Subscription acceptance policy ..... 64

Figure 4.7: Fraction of nodes contacted per query ..... 71

Figure 4.8: Search path length ..... 72

Figure 4.9: The effect of parallel search in SCS ..... 73

Figure 4.10: Search cost ..... 74

Figure 4.11: Maintenance cost ..... 75

Figure 4.12: Search path length vs. cluster size *M* ..... 76

Figure 4.13: Search cost vs. cluster size *M* ..... 76

Figure 4.14: Costs of node joining/leaving and cluster splitting/merging vs. cluster size *M* ..... 77

Figure 4.15: Shortcuts ..... 78

Figure 4.16: Routing load ..... 79

Figure 5.1: Unnecessary query messages in a Gnutella-like network ..... 82

Figure 5.2: Link cost measurement and exchange messages..... 86

Figure 5.3: Optimized paths for a *3-loop* with source node *x*..... 88

Figure 5.4: Optimized paths for a *4-loop* with source node *x*..... 90

Figure 5.5: Optimized paths for an *n-loop* where $n = 5$..... 91

Figure 5.6: Main CASF algorithm..... 94

Figure 5.7: A case study..... 96

Figure 5.8: Effectiveness of the CASF algorithm..... 99

Figure 5.9: Bandwidth consumption..... 101

Figure 6.1: Classes responsible for connecting to the SCS network ..... 105

Figure 6.2: Screen shot of an SWebCache..... 109

Figure 6.3: Structures of *Join* and *JoinReply* messages..... 110

Figure 6.4: Screen shot of connections..... 113

Figure 6.5: Classes responsible for message forwarding and processing..... 116

Figure 6.6: Structure of *Query* message ..... 120

Figure 6.7: Structure of *QueryHit* message ..... 120

Figure 6.8: Class diagram of LookupClient..... 122

Figure 6.9: Classes responsible for context data and query management ..... 124

Figure 6.10: GUI of Context Producer for searching and subscribing context data... 125

Figure 6.11: GUI of Context Producer for sensor management ..... 126

Figure 6.12: GUI of Context Producer for sensor value selection..... 127

Figure 6.13: Screen shot of a subscription response..... 131

Figure 6.14: Classes responsible for context data reasoning ..... 132

Figure 6.15: Screen shot of incoming subscriptions for all deduced queries in a Context Interpreter ..... 133

Figure 6.16: Screen shot of the SmartHome application ..... 141

Figure 6.17: A sample rule for context reasoning in the SmartHome application ..... 143

Figure 6.18: Skeleton of the code in the SmartHome application ..... 144

Figure 6.19: Scenario of the ShoppingAssistant application ..... 145

Figure 6.20: The physical layout of our prototype testbed ..... 149

Figure 6.21: An example of the ring space constructed during the evaluations ..... 150

Figure 6.22: Response time for non-deduced queries ..... 156

Figure 6.23: Response time for deduced queries ..... 156

Figure 6.24: ContextPeer query processing capability ..... 158

Figure 6.25: Deduced query processing time ..... 160

Figure 6.26: Memory consumption for the different methods ..... 162

# CHAPTER 1

# INTRODUCTION

Emerging pervasive computing technologies provide "anytime, anywhere" computing by decoupling users from devices [1]. They enable applications to perform tasks on behalf of users. To allow the user to concentrate on his/her tasks, applications must be capable of operating in highly dynamic environments. Therefore, all entities (such as devices, services and agents) in a pervasive environment must be aware of their contexts, and automatically adapt to changing contexts. This is known as context-aware computing. Context information is a key for propagating and enmeshing computation into our lives, and exhibiting the required levels of autonomy and flexibility in context-aware computing.

The concept of context-aware computing has been around for many years; many researchers have studied this topic and developed various context-aware applications to demonstrate their benefits in different aspects to human livings. For examples, a context-aware mobile phone should automatically into a silence mode when entering a live concert hall; a context-aware message forwarding application could selectively display instant messages based on the sender and the nature of the message; in a smart home environment, a wall-mounted display could turn on and display relevant information to an approaching user; in health and elderly care applications, an alert to hospital emergency could be triggered whenever the blood pressure of a patient being monitored exceeds a certain threshold, or a reminder message could be sent to a patient at home to remind him taking medicine according to a doctor's e-prescription. Other examples of context-aware applications include conference assistants, shopping assistants, context-aware tour guides and community applications.

Early context-aware system prototypes such as Active Badges [2] and Cyberguide [3] demonstrated the benefits of combining sensing technologies with computational power in provisioning context-awareness to various applications. However, these systems have also shown that it is still extremely difficult to design, develop and maintain robust context-aware applications [4]. The difficulties are primarily due to the lack of adequate infrastructure support [5]. There are a number of issues that must be resolved in a context-aware infrastructure, including handling diverse and potentially unreliable sensor data, dealing with context acquisition and representation, maintaining system interoperability, and resolving the basic difficulties involved in building a reliable distributed system, etc.

## 1.1 Understanding context in pervasive computing

The term context is widely used with a variety of meanings. In this section, we define the meaning of "context" and "context information" to be used throughout this thesis. We also identify some characteristics of context information that are important to the design of context-aware systems.

### 1.1.1 Context definition

Context has commonly been characterized as an application's environment or situation [6][7]; and as a combination of features of the execution environment, including computing, user and physical features [8]. Dey provides the following definition [9], which is perhaps now the most widely accepted definition:

*Context is any information that can be used to characterize the situation of an entity. An entity is a person, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves*.

In this thesis, we follow the basic definition of context information proposed by Dey. Particularly, we view context information as a set of data, which can be acquired directly from sensors and users, or be derived through some appropriate means. Examples of context information may include user information (name, address, role, etc.), location (coordinate, temperature, etc.), computational entity (device, network, application, etc.), and user activity (scheduled activities, deduced activities, etc.)

### 1.1.2 Characteristics of context information

Appropriate exploitations of the characteristics of context information can lead to a better management and use of context information. In the following, we discuss a number of key characteristics of context information and their implications to the design of context-aware systems.

*i. Context information is widely distributed and highly heterogeneous.*

Context information is typically spread over a wide-area network and across different application domains. They may include a wide range of information resources of which only a small subset (such as sensed context information) are used in earlier context-aware applications. More recent applications usually combine multiple types of context information in their design, such as sensed context information with non-sensed context information. Non-sensed context information can be classified into user-defined context information and derived context information. User-defined context information is often obtained directly from users or applications. Derived context information is obtained through derivation mechanisms, such as an aggregation of multiple sources of context data or interpretation of low-level explicit contexts to obtain a higher level of abstraction. The integration of context information

from such diverse sources will naturally lead to extreme heterogeneity, in terms of the type of context data and application domains.

To have heterogeneous context information spreading over a network, an appropriate context model should be used to represent different types of context information so as to achieve better interpretability and information sharing across different application domains.

*ii. Context information exhibits a range of temporal characteristics*

Context information exhibits a range of temporal characteristics, as pointed out by Henricksen in [10]. It can be classified into static and dynamic information. Intuitively, static information describes persistent properties and usually remains unchanged in its lifetime, such as a person's date of birth or the type of a computing device. In contrast, dynamic information can be highly volatile; for example, relationships between colleagues typically endure for months or years while a person's location and activity often change from one minute to the next. Sensed context information is often dynamic, and is usually updated frequently in response to continuous or periodic sensor output.

The dynamicity of context information implies that context-aware systems need to detect and react quickly to context changes. Traditional pull-based context acquisition techniques may not scale well in the presence of frequent changes of context; push-based techniques are preferred instead. Moreover, the latter also incurs less overhead to the system; and how to minimize these overhead is critical to the design of context-aware systems.

*iii. Context information is interrelated and has different levels of precisions*

Any context information derived from a source is likely to be related to information originated from that source. The original information is usually low-level, explicit context information. For example, a derived person's current activity may depend on his/her current location, time and date, and his/her surrounding environmental contexts; such as lighting, noise level, and etc. The interrelationship of context information suggests that we can derive context information based on low-level and explicit context information through derivation mechanisms such as logic reasoning.

Context information is imprecise [10]; different types of context information exhibit different levels of precision. Static context information is usually assigned with a higher degree of precision, whereas dynamic context information may become staled if not updated frequently. For example, sensed context information is prone to inaccuracies as a result of sensing errors, network failures or limitations inherent within the sensing technology. In addition, when context data changes rapidly, the delays introduced by the distribution processes and the interpretation processes (that transforms sensor output into high-level context information) can lead to loss of accuracy. Derived context information is largely determined by the properties of input context data; it usually inherits most inaccuracies of its origins. Additionally, the use of brittle heuristics or the reliance on crude sensor inputs for inferring high-level context information would lead to further errors. Therefore, it is inevitable that a well-design context-aware system must also deal with uncertainties of context information.

*iv. Sensed context information is normally bound to its producer*

Context information can be viewed as a general network resource. However, some restrictions arise when context information is stored in a network. Sensed context

information is usually bound to its providers [11]. This is because sensed data is tightly controlled by the type and location of a physical sensor. For example, we can attach an RFID receiver which is located in a meeting room to a PC and connect this PC to the Internet. Since the RFID receiver can keep track of any RFID-tagged object (e.g., a person wearing a RFID tag), this PC can be viewed as a context producer node which is capable of providing the location context of users (such as John is located in this room). Subsequently, this location context (i.e., someone is in this room) can be stored in the node. It is also possible to store this location context in other nodes in the network; however, the cost of updating user's location context can be very high especially when the mobility of the person is higher. Hence, an appropriate context storage model needs to store context data close to where it is generated in the network [4] (i.e., near the source node).

## 1.2 Functional requirements of system's infrastructure for context-aware computing

Research in context-aware computing faces many challenges due to increasing autonomy of the services, dynamic computing environment, variety of user requirements and various resource limitations. Over time, the research approach has been shifted from being application-centric to being infrastructure-centric. The former is characterized by a horizontal software architecture in which the functions of context-awareness are tightly coupled with a specific set of applications [2][3][12][13]; the latter offers context-aware functions as horizontal common infrastructure services (also known as 'context-aware middleware') [14][15][16][17][4][18][19], including our earlier work – SOCAM [20]. In this thesis work, we adopt the infrastructure-based approach to design context-aware systems.

We illustrate the layered structure of a typical context-aware infrastructure in Figure 1.1, and briefly discuss the functionalities of a context-aware infrastructure and the research challenges they pose.

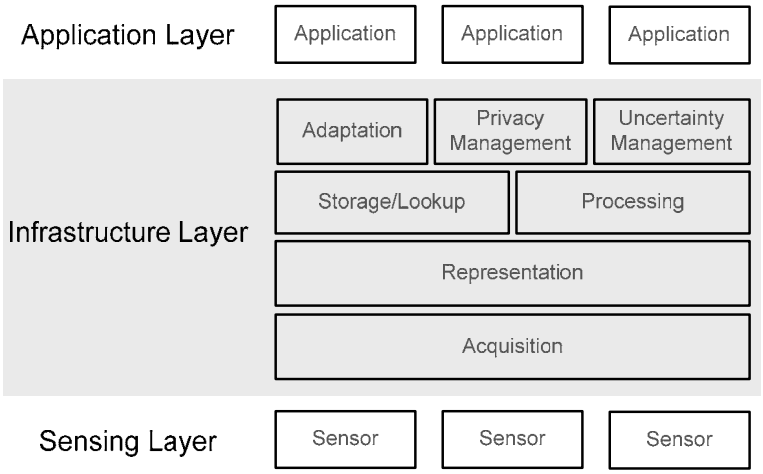



Figure 1.1: Overview of a typical context-aware infrastructure

*i. Context representation*

A common model for representing context information is the foundation of any context-aware system. A well-defined context model should have the ability to represent and capture different characteristics of context information such as uncertainty, and provide a common platform for sharing and processing context information across different context-aware systems and domains. As we will survey in Section 2.2, many existing context models lack the above features.

*ii. Context acquisition*

Context acquisition is a mechanism to acquire context data from various sources, including physical sensors, database servers or web services. As we will discuss in Section 2.1, compared to earlier context-aware systems [2][3][12][13][14][15],

current context-aware systems [16][17][4][18][19] including our earlier architecture [20], possess the capability to obtain contexts from heterogeneous sources by decoupling low-level sensings with high-level context usages. This approach typically deploys a component called a widget or a context producer to acquire context data.

*iii. Context processing*

Context processing is to manipulate and process context information. The challenge of context processing is how to manipulate context information at different levels (from simple manipulation to sophisticated manipulation) to better meet application requirements. As we will discuss in Section 2.1, most existing systems manipulate context information in a simple way, such as aggregating or merging interrelated context information and transforming different context information. Recent context-aware applications tend to deal with high-level contexts such as recognition of human activities. Hence, some advanced interpretation techniques are required to process related context information to derive high-level and implicit contexts.

*iv. Context storage and lookup*

Context storage and lookup are mechanisms through which context producers store their contexts in a network, so that both users and applications can subsequently locate them across the network. Context lookup typically disseminates context information using synchronous queries and asynchronous notifications. For any context-aware application, context lookup (also known as context discovery) is usually the first step to be taken before users and applications can utilize context information. Most current context-aware infrastructures either do not support context lookup or relying on other existing general techniques for locating context, as we will discuss in Section 2.1. While general discovery techniques offer the basic support for

discovering context, they have not taken into account of the characteristics of context information (discussed in Section 1.1.2) in their designs; hence, the resulting system performance will be compromised. The issues we need to consider for context lookup include: How does the lookup system scale well with large numbers of entities in wide-area networks? How to minimize the overhead of the lookup system in the presence of dynamic joining and leaving of context producer nodes? Furthermore, context lookup is tightly coupled with how context data is stored in the network, which is either centrally or distributedly. Clearly, the architecture and context storage model will affect the context lookup.

*v. Uncertainty management*

As we have discussed in Section 1.1.2, context information is imperfect due to the limitation of sensing technology, the dynamics of context information, and the accuracy of context processing. How to handle uncertain contexts and solve context conflict has been addressed in many context-aware infrastructures [21][22][23][24]. The reliability and usability of context-aware solutions depends partly on how well uncertainties in context information could be handled satisfactorily. Many research have been initiated to investigate this problem [21][22][23][24], which is beyond the scope of this thesis. However, we will have a provision in our context model [25] for representing uncertainty of contexts, which may be useful for future work.

*vi. Context adaptation*

Since context information is dynamic, context-aware applications consuming such information are expected to be able to adapt or response to their changes. Little work has been done to incorporate mechanisms for supporting adaptation. We do not elaborate this issue further as it is beyond the scope of this thesis.

*vii. Privacy management*

Privacy is an important design issue in pervasive computing. A common approach toward privacy is to provide anonymity or to keep personal information secret from others. The challenge here is that, from a computer science perspective, privacy is not a purely technical issue, but also involves aspects of legislation, corporate policy, and social norms [17]. Furthermore, privacy is a malleable concept in practice, based on individual perceptions of risks and benefits.

## 1.3 Problem statement

This thesis addresses the problem of the provision of context-aware infrastructure support for collaborative context-aware applications over multiple context spaces. It provides a set of core infrastructure services – wide-area context lookup and distributed context reasoning, coupled with a context representation model. The specific problems we seek to address may be briefly summarized as follows:

- How to provide a scalable context lookup service in multiple context spaces.
- How to provide a distributed context reasoning service that is feasible to be applied in multiple context spaces.
- How to provide a common context model that supports the two core services (wide-area context lookup and distributed context reasoning) and enables context sharing between context-aware applications over multiple context spaces.

The reasons that we focus on the two core services are two folds. Firstly, the goal of context-aware computing is to acquire and utilize context information to build applications that are appropriate to people, place, time, events, etc [27]. Context data

required by collaborative context-aware application may be spread across multiple domains in a wide-area network. Hence, wide-area context lookup is the primary task to accomplish for any context-aware infrastructures that aims for simplifying the building of collaborative context-aware applications in multiple context spaces. In addition, advanced context reasoning techniques are necessary to increase the level of flexibility of such applications. Secondly, as surveyed in Section 2.1, little work has been done in addressing the above problems in multiple context spaces. Most of existing context-aware infrastructures adopt a centralized approach for context lookup. This approach works efficiently in a single context space; however, it may not scale well in multiple context spaces. Many decentralized lookup systems in peer-to-peer (P2P) computing, as surveyed in Section 2.2.2, could be applied in wide-area context lookup. However, in the design of context lookup, we should take into account of the impacts of different characteristics of context information. For example, frequent changes of context producer nodes and their context data, or storing sensed context data away from its source producer node may incur a large amount of communication overhead, etc. We will further elaborate on these issues related to wide-area context lookup in Section 1.4.2. Logic reasoning has been proposed to use in context-aware computing such as in [20][18]; and has shown its usefulness and flexibility to derive high-level contexts from low-level contexts. However, as surveyed in Section 2.1, existing centralized or server-based reasoning systems may not scale well in multiple context spaces. Considering the dynamicity of context information, distributed context reasoning can be more challenging. For example, since the reasoning engines can be embedded into the nodes in multiple domains, how to design both pull and push-based mechanisms to support the reasoning task. Prior to the above two core services, a common context model has to be established. Our context model aims to

provide supports for wide-area context lookup and distributed context reasoning. In the next section, we will outline our approaches and highlight the research issues with respect to each of these problems

## 1.4 Our Approach

In this section, we present our approach to address each of the problems mentioned above; a more detailed discussion of our approaches and a comparison to other approaches will be presented in the related work section.

### 1.4.1 Ontology-based context modeling

We propose an ontology-based context model [28] in which contexts are represented as RDF triples. We leverage the advantage of RDF-based data model, for example, RDF data is machine-understandable and machine-processable. The main benefits of this model are: First, it is based on an open standard, and hence, not proprietary to any particular system or platform. Second, it provides the fundamental model for interpreting context data using logical reasoning. We also propose a hierarchical design for context ontology, which is essential for both context lookup and reasoning in multiple context spaces.

### 1.4.2 A semantic peer-to-peer overlay

To provide a scalable context lookup service over multiple context spaces, we propose Semantic Context Space (SCS) [29][30], a semantic P2P overlay network in which context data is organized and retrieved according to their semantics. The basic idea is to cluster peers based on their data semantics and organize them in a structured P2P overlay network for efficient routing. In SCS, context data is represented by a collection of RDF triples based on a set of schemas (i.e., context ontologies). These

triples, in various domains, logically represent the semantics of the context data. Each context data can be viewed as a point in a multi-dimensional Semantic Context Space. Context data is stored in a distributed manner in various context producer nodes where the data is generated. Pieces of context data which are semantically similar are "tied" together in SCS so that they can be retrieved by a context query which has the same semantics. As a result, the system is able to forward a query to nodes, which are likely to contain the relevant context data. This allows for a lower network load and better search performance.

While the basic idea may appear simple, there are several critical issues that have to be considered in order to make our proposed scheme work effectively in multiple context spaces. First, with the increasing use of large amounts of context data by various applications in pervasive computing, scalability is the most important issue to consider in the design of any context-aware system. A well-designed overlay network needs to scale and adapt to the growth of context data. Second, as context data exhibits a range of temporal characteristics, overlay maintenance cost may rise due to the frequent changes of peers and their data. How to minimize overlay maintenance cost is a challenge in the design of the navigation and search mechanisms. Third, due to the dynamic nature of peers in context-aware systems, mapping context data and queries to semantic clusters may incur large overheads for the network. How to extract and obtain the semantics from both context data and queries efficiently and precisely with less overheads is critical. Fourth, as context data exhibits the characteristics of heterogeneity, the number of domains to group various context data and specify queries can be potentially large in real-life applications. As a result, the number of semantic clusters in SCS could be large. Thus, a well-designed overlay network needs to be able to facilitate efficient search in a high-dimensional context

space without incurring large overheads. Finally, as context data may change rapidly in context-aware environments, it is important to notify context consumers automatically whenever changes occur. Hence, it is important to facilitate an event notification mechanism to adapt to changes in SCS.

To address these issues, we propose the following techniques in SCS:

- Upon joining the system, peers are grouped and arranged into a one-dimensional ring space where various semantic clusters are organized and interconnected. The ring structure enables the mapping of clusters in a k-dimensional semantic space to a one-dimensional semantic space[1], and hence reduces overlay maintenance overhead.

- We propose a cluster encoding scheme that enables the system to adapt to the number of peers by splitting or merging clusters. This scheme provides for a system of good scalability and load balancing characteristics. It also enables the use of parallelism in our system when searching for data within a semantic cluster.

- We use ontology-based metadata to extract the semantics of data and queries, and group peers into various semantic clusters. This technique can map data and queries to the appropriate semantic cluster(s) with minimum computational overhead despite peers joining/leaving the network frequently and the data changing often.

- We deploy both pull and push services in SCS. Context consumers can submit either search requests or subscription requests. The latter allows context consumers to be notified whenever data changes occur.

[1] A semantic space refers to a network in which data are orgnanized based on their semantics.

- Aiming to minimize unnecessary query messages caused by the blinding flooding mechanism used by nodes within a cluster, we propose a Cost-Aware Selective Flooding (CASF) technique [31] to reduce redundant query messages. This technique makes use of two-hop neighborhood and link cost information to ensure that only necessary messages are flooded across the network. With less query messages generated, system scalability can be further improved.

### 1.4.3 Distributed context reasoning

In SCS, we propose a logical reasoning approach to interpret various types of context data and their properties, and derive high-level and implicit contexts from low-level and explicit contexts. Using the rule-based logical reasoning, we are able to raise the level of context abstraction according to users' or applications' requirements. Our earlier results [20] show that logical reasoning is a computationally intensive process. Hence, a centralized reasoning engine may not scale up well because of the processing bottleneck and the single point of failure. Therefore, we adopt a distributed approach to context reasoning. The reasoning engines can be embedded into various nodes across different domains with each reasoning engine performing reasoning tasks using a subset of logical rules and context data in a subset of domains.

### 1.4.4 Research methodology

In this thesis, we adopt an experimental research method – both simulation and prototype – to evaluate our system. We use simulation to evaluate the routing and clustering techniques of SCS on a large scale and compare the performance with other approaches. To assess practical issues in a real-world setting, we build a prototype system to demonstrate the working principles of our proposed techniques such as ontology-based semantic mapping, SCS overlay construction, routing, and push and

pull services. We conduct performance measurements over the prototype system and use the results to calibrate our simulation models. To validate our infrastructure-based system, we also develop several typical context-aware applications both in a single domain and across multiple domains. We show how our system eases the development process and enables the fast prototyping of various context-aware applications.

## 1.5 Thesis contributions and outline

In summary, this thesis makes the following key contributions:

- We propose a set of core infrastructure services to support and simply the building and maintaining of collaborative context-aware applications in multiple context spaces. The core services are wide-area context lookup and distributed context reasoning, coupled with an appropriate context model.

- We propose an ontology-based context model that provides an open platform for sharing context information across different domains and enables interoperability of context information exchange. We also propose a hierarchical design of context ontology for enabling semantic context lookup and logical reasoning in multiple context spaces.

- We propose a semantic P2P overlay network named SCS to provide users and applications with an efficient context lookup service. We design various techniques to meet the requirements of scalability and dynamicity, such as an ontology-based semantic mapping scheme for fast semantic abstraction, one-dimensional ring space for reducing overlay maintenance cost and efficient routing, cluster splitting and merging for self-scaling to number of context

producer peers, cost-aware selective flooding for minimizing redundant query messages, a context push service for notifying context consumers about changes quickly.

- We demonstrate the practicality of our system by developing a working prototype system and implementing various techniques, including distributed context reasoning. The evaluation results of the prototype show that our system works effectively in a real-world setting.

- Based on our prototype system, we design and build several typical context-aware applications in both a single context space and multiple context spaces to validate our infrastructure-based system. Our experiences show that the development process is greatly simplified. Application developers need only focus on application-level tasks without wasting time and efforts on low-level details.

The rest of the thesis is organized as follows. Chapter 2 surveys relevant context-aware systems and information retrieval systems, and discusses how our approach differs from others with respect to context modeling, lookup and processing. Chapter 3 describes our ontology-based context model and presents our earlier experimental results for context reasoning. Chapter 4 describes our semantic P2P overlay network in detail and presents the simulation results from a range of experiments. Chapter 5 describes the Cost-Aware Selective Flooding technique and presents the simulation results. In Chapter 6, we describe our prototype implementation of SCS and evaluate the prototype in close-to-real scenarios. We also demonstrate a number of context-aware applications in the chapter. Chapter 7 concludes the thesis and identifies possible future work. Supplemental materials are contained in the appendices.

# CHAPTER 2

# RELATED WORK

This chapter surveys and discusses existing context-aware systems developed for building various context-aware applications. We evaluate each of them and show how our work differs from previous work. We discuss the three aspects (context modeling, lookup and processing) of the systems surveyed, and compare them with that of our approach. We also discuss the existing information retrieval techniques which has inspired us to the design of SCS – a semantic P2P context lookup system.

## 2.1 Context-aware systems

There are generally two approaches to build context-aware systems: the application-specific vertical approach and the infrastructure horizontal approach. Many earlier context-aware systems focused on building specific context-aware applications in a particular domain by using the former approach. Although these systems provide real application examples to demonstrate the usefulness and potential benefits of context-aware systems, they are difficult to develop and maintain. Recent work in context-aware systems shifts many of the complex functionalities from applications to infrastructures, thereby simplifying the construction of robust applications. The infrastructures perform tasks such as context acquisition, context representation, persistent storage of context information within servers, context interpretation, dissemination to applications using synchronous queries and asynchronous notifications, adaptation, and context privacy control. In the next section, we survey existing infrastructure-based systems and discuss their strengths and weaknesses. We

summarize our findings and discuss how our approach is related to and different from them in Section 2.1.2.

### 2.1.1 Existing infrastructure-based systems

Schilit et al. [14] pioneered the development of infrastructure support for context-aware computing, by proposing an architecture comprising distributed context servers called environment servers and user agents. The architecture partitions the context description among the servers and agents, which stores context information in the form of simple environment variables. Environment severs maintain information related to domains such as rooms, project groups or other logical or physical entities while user agents record context information for each user. In this work, a small set of context information, i.e., environmental contexts, is used and represented by a simple context model represented in the form of text variables. The context information is stored in a centralized server. For context lookup service, the user or application simply query the centralized server. However, the simple name-value pairs for representing context data may not meet the level of expressiveness as required by users and applications. In addition, a centralized server may become a bottleneck when scaling to a large number of users and applications in multiple context spaces.

In Cooltown [13], a web-based system for context-awareness was proposed. Cooltown embeds context information within a web-based framework, associating each entity (a people, place, and thing) with a description retrievable via a URL. A simple location-based discovery mechanism is used for context lookup, which involves the use of beacons to transmit the URL of the local environment wirelessly. Cooltown also provides data transfer between entities so that a user could discover devices and objects present in the environment. Its context model is informal as

arbitrary information can be embedded in the web pages. This feature, together with the restrictive discovery mechanism (which is based on the assumption that only information about the local environment is required at any time), limits the utility of the model [26]. Rather than presenting HTML-based context data for people, SCS uses RDF-based data for processing by machines. We also focus on context lookup issues, aiming to provide a scalable lookup service over multiple context spaces in a wide-area network.

The Context Toolkit, developed by Dey et al. [15], provides a software framework and a number of reusable components to support rapid prototyping of sensor-based context-aware applications. The toolkit defines the following abstract component types: widgets, which function as software wrappers for sensors; interpreters, which raise the level of abstraction of context information to better match application requirements; and aggregators, which collect different types of context information related to a single entity. Widgets incorporate context information using the persistent storage of a relational database, and implement an information model based on simple attributes. By drawing upon standard libraries of reusable components that instantiate these three abstract types, programmers can easily build applications to enable context-ware behavior. In their work, Dey et al. identified several important functionalities that should be supported by any context-aware infrastructure include context acquisition, context interpretation, and context aggregation. The context interpreter is able to raise the abstract level of context information, for example, transforming raw location coordinates to a building and room number. The Context Toolkit focuses less on the issues of context lookup by assuming the priori knowledge about the presence of a widget or a context broker. Similar to the concept of context interpreter in the Context Toolkit, SCS uses logical reasoning to derive high-level

contexts for applications. In addition, we have identified that context lookup is one of the core services in any context-aware infrastructure, and propose various techniques to address the issues of context lookup in the presence of multiple context spaces.

Chen et al. proposed a platform, named Solar [16], to support context acquisition, aggregation and dissemination. This infrastructure is based on the use of a graph abstraction to specify the structure of their context framework. The graph components are sources, representing sensors and operators, representing processing components that perform interpretation and aggregation. Context information traverses the graph in the form of event streams. Applications produce textual specification of their context requirements in the form of graphs. In response, Solar creates the required operators and event subscriptions. Solar also provides a policy driven data dissemination service based on a multicast tree. Context events are pushed to users and applications through an application-level multicast tree. A policy propagates in the overlay with the receiver's subscription request so the policy embeds in every node of the dissemination path, and multiple receivers' requests incrementally construct the multicast tree. While Solar focuses more on a context notification service, SCS provides both pull and push services to better meeting application requirements. SCS takes a P2P approach by semantically clustering context producers into a structured P2P overlay for efficient lookup. For context interpretation, Solar introduces an operator which performs processing functions over incoming context events, such as converting GPS coordinates in location events into ZIP codes. In contrast, SCS uses a more expressive context model based on RDF and a logical reasoning approach for context interpretation.

Hong et al. [4] proposed the Confab infrastructure, which includes the following three features to simplify the task of building context-aware applications: (i) a flexible and distributed data store to make it easy to model, store and disseminate context data; (ii) a context specification language for declaratively stating and processing context needs; and (iii) reasonable and customizable privacy mechanisms to help protecting context data of end-users. The context storage consists of a logical context data model, which provides a logical representation of context information and a physical data store where the context data is actually stored. While the context service in SCS shares the similar idea of distributed context storage of Confab in which the context data is kept close to where it was generated and where it is likely to be used [4], our emphasis is more on how to provide a scalable P2P lookup service for users and applications over a possible wide area involving multiple context spaces. Our P2P model also takes into account the characteristics of context information (e.g., dynamic, sensed context is bound with its producer, etc.). In addition, the RDF-based context model in SCS has additional advantages such as basing on an open standard specification language platform, whereas the proprietary context specification language in Confab may limit the utility of the model.

Chen et al. [19] proposed the CoBrA infrastructure for context representation, knowledge sharing and user's privacy control. CoBrA provides a centralized model where context information is shared by all devices, services and agents in a smart space. A set of ontologies written in OWL have been developed for an intelligent meeting room. CoBrA also defines different access control models for protecting the privacy of users. An access control model consists of a set of inference rules that CoBrA uses to grant permission for revealing a user's contextual information. Recently, Chen et al. have also initiated an effort to define standard ontologies for

ubiquitous and pervasive applications [32]. Although many issues are to be resolved before standardization of ontologies is fessible, we believe their efforts will lead to wider use of ontologies in context-aware computing. The key difference between the ontological model in SCS and the one in CoBrA is that we use a two-tier design, leading to a more flexible use of context ontologies. CoBrA uses a centralized server called Context Broker to store context information for a single context space; context lookup is done by querying the server. In contrast, SCS takes a decentralized approach to context lookup in multiple context spaces.

Ranganathan et al. [18] developed a middleware infrastructure to enable context awareness in ubiquitous computing environments. They proposed a context model which is based on first-order logical predicate and marked up in DAML. DAML encoded context ontologies are used to ensure semantic interoperability between different agents, as well as between different ubiquitous computing environments. They used logical reasoning and machine learning techniques to decide application behavior. The middleware is implemented on top of CORBA [33], and uses CORBA Naming Service and CORBA Trading Service for context discovery. This requires the advertisement and request to be constructed in the form of a CORBA object. SCS also adopts an ontology-based approach for context modeling; in addition, the context model in SCS addresses issues such as context classification, dependency, quality and uncertainty. The ontological model of SCS is designed such that it can be easily extended. Rather than looking at context discovery issues in an application domain, SCS emphasizes more on cross-domain issues and proposes a fully de-centralized lookup service. Further more, while the reasoning approach in SCS is based on first-order logic approach similar to that of [18][19], we study the feasibility of applying context reasoning in pervasive computing, and propose distributed context reasoning

where each context interpreter only maintains a subset of user-defined rules based on its capability in an application domain.

Wang et al. [80] proposed a pervasive computing infrastructure named Semanitc Space that exploits Semantic Web technologies to support explicit representation, expressive querying, and flexible reasoning of contexts in smart spaces. Their infrastructure facilitates pervasive computing applications with context-awareness in a single smart space. While SCS also exploits Semantic Web technologies for context model and reasoning [28], we aim at providing efficient lookup and distributed context reasoning in multiple smart spaces. The key difference is that we take a decentralized approach to context-awareness while Semanitc Space is basically a centralized infrastructure.

### 2.1.2 Summary

Context lookup has not been adequately addressed in all the works surveyed so far, issues related to cross-domains lookup have not been treated at all. This is not surprising as most context-aware infrastructures have been proposed for single context space, and have adopted a centralized approach for context lookup, such as in [18][19]. This approach has limitations such as a single processing bottleneck, which ultimately leads to a scalability problem; and a single point of failure, which undermines system robustness. This approach also requires system administration at the centralized server. In contrast, the emerging P2P computing model seems to provide a more effective approach for overcoming the limitations we have just noted, and may potentially offer many advantages to be mentioned later. Any node of a P2P system can be both a client and a server. Though nodes may associate with different administrative domains, they are usually not centrally managed and administered.

Nodes may also join and leave the system dynamically. Motivated by the advantages of P2P systems, we propose a semantic P2P model for context lookup in multiple context spaces. In the next section, we survey some important lookup related techniques in the information retrieval literature, and focus our discussion on P2P information retrieval.

## 2.2 Information retrieval

In this section, we survey and discuss the related work on information retrieval systems. We discuss the pros and cons of each system, and highlight how our approach is different from existing P2P systems.

### 2.2.1 The centralized approach

Many centralized repositories and lookup systems have been implemented to support storing, indexing and querying RDF documents, such as RDFDB [34], RDFStore [35], Jena [36] and Sesame [37]. These centralized RDF repositories typically use in-memory or database-supported processing, and files or a relational database as the back-end RDF triple store. RDFDB supports an SQL-like query language while RDFStore and Jena support SquishQL-style RDF query languages. These systems are simpler to design and work reasonably fast for low to moderate number of RDF triples. However, their centralized processing approach is not appropriate for context lookup in multiple context spaces as we have discussed in previous sections.

### 2.2.2 The P2P approach

P2P approaches have been proposed to overcome some of the limitations of centralized approach; they have gained popularity due to their better scalability, fault-tolerance and self-organizing characteristics. Hence, P2P is a suitable technology for

the development of lookup systems. P2P systems can be generally categorized into *unstructured*, *structured* and *semantic* P2P systems; they are discussed in the following sub-sections.

*2.2.2.1 Unstructured P2P systems*

*Unstructured* P2P systems can be further classified into hybrid systems, pure systems and super-peer systems.

Hybrid P2P systems such as Napster [38] rely on centralized index servers for searching, but its information transfers are still conducted in a P2P fashion. Hence, peers are equal in downloading information only. While centralized search is generally more efficient than distributed search, the cost incurred on the single node housing the centralized index is very high, in addition to the potential shortcomings of performance bottleneck and single point of failure.

Pure P2P systems such as Gnutella [39] and Freenet [40] do not impose any constraint on data placement and network topology. These systems rely on some form of message flooding to search for resources. For example, Gnutella adopts a breath-first approach to flood requests while Freenet uses a depth-first approach. To prevent the high cost of flooding the entire network, both systems use a time-to-live (TTL) mechanism to limit the scope of a search. However, they tend to be inefficient as query messages are still flooded indiscriminately (i.e., blind-flooding) to the network. The network traffic generated by complete flooding is making the system unscalable. Another important source of inefficiency is the bottlenecks caused by the very limited capabilities of some peers. However, these systems have been widely deployed in real life because of their flexibility, simplicity, and that they require no complex state information at each node.

Super-peer systems such as KaZaA [41] consist of super-peers and their clients. A super-peer is a node that acts as a centralized server to a subset of clients. Clients submit queries to their super-peers and receive results from it. Super-peers are also connected to each other as peers in a pure P2P system; they route messages over this P2P overlay network, and submit and answer queries among themselves or on behalf of their respective clients. Super-peers are equal in terms of search, and all peers (including clients) are equal in terms of information download. Super-peer systems have many advantages over hybrid and pure P2P systems. First, the search is much faster in super-peer networks since the search of information is now done at a smaller set of super-peers, each of which has indexed information for its set of peers. For example, a search which takes *O(N)* time on a pure/hybrid P2P network, takes *O(N/M)* time on a super-peer network (where *M* is the average number of peers connected to a single super-peer) [41]. This partly eliminates the problem of message flooding typically associated with a pure P2P system. Also, super-peers, which are designed to be more reliable and trustworthy, can monitor the client activities of all peers connected to them. This ensures that malicious activities can be controlled across the network. In a *pure* P2P system, every peer is given equal responsibility irrespective of its computing/network capabilities. This can quickly lead to deterioration of performance due to network fragmentation caused by less capable nodes being added to the network. This problem is alleviated in a super-peer system, as only relatively powerful computers with sufficient network bandwidth are assigned the status of super-peers. This ensures that the super-peer network divides loads according to the capability of peers, leading to overall better performance.

In summary, *unstructured* P2P systems allow peers to interconnect freely, making it easy to handle the dynamic changes of peers and their data. These systems do not

impose any structure on the managed resources, and hence have low overlay maintenance overhead. However, a query has to be flooded to all nodes in the network including those nodes that do not have relevant data. The blind flooding mechanism used without any restriction on the scope of flooding can become very inefficient because of excessive redundant messages. Above all, the fundamental problem that makes search in these systems difficult is that data is distributed randomly in the overlay network with respect to its semantics. Given a search request, the system either has to search all the nodes or run a risk of missing relevant data.

*2.2.2.2 Structured P2P systems*

*Structured* P2P systems such as Chord [42], Content-Addressable Networks (CAN) [43], Tapestry [44] and Pastry [45] typically implement distributed hash tables (DHTs) and use hashed keys to direct a lookup request to specific nodes by leveraging a structured overlay network among peers. In these systems, objects are associated with a key that can be produced by hashing the object name. Nodes have identifiers which share the same space as the keys. Each node is responsible for storing a range of keys and corresponding objects. The nodes maintain an overlay network, with each node having several other nodes as neighbors. When a lookup (key) request is issued from one node, the lookup message is routed through the overlay network to the node responsible for the key. Different DHT-based systems construct different overlay networks and employ different routing algorithms. They can guarantee completing lookup in a logarithmic number of steps – *O(logN)* or *O(dN1/d)* hops and each node only maintains the information of *O(logN)* or *d* neighbors for a network of size *N,* where *d* is the dimension of the hypercube organization of the network. Therefore, they provide very good scalability.

However, the placement of data in these systems is tightly controlled based on distributed hash functions, and the overlay maintenance overhead is high in more dynamic networks. As context data is mobile and dynamic due to frequent joining to or leaving from the system by peers and changes of context, a higher maintenance overhead for updating relevant information in DHT-based overlay networks is inevitable. Moreover, as sensed context data is usually bound with its producer, it may not be desirable to place/store context data to a particular node based on the hash value. For example, a user's current location data (e.g., at home) should be stored in a node at/near his/her home rather than in a node which may be far away based on the hash value of the data.

Some RDF lookup systems are based on *structured* DHT-based P2P systems such as RDFPeers [46] – a scalable and distributed RDF repository proposed by Cai et al. RDFPeers is organized into a multi-attribute addressable network (MAAN) [47] which extends Chord to efficiently answer multi-attribute and range queries. When an RDF triple is inserted into the network, it is stored three times, as a globally-known hash function is applied to its subject, predicate, and object. Queries can then efficiently be routed to those nodes in the network where the triples in question are known to be stored if they exist. However, the overlay maintenance cost is high in this system in the presence of dynamic peer joining and leaving. In addition, storing each RDF triple multiple times in the network increases storage cost.

*2.2.2.3 Semantic-based P2P systems*

Piazza [48] is a P2P data management system that supports interoperation of both XML and RDF data sources. It addresses the issue of heterogeneity in P2P systems at the schema level, and allows for information sharing with different schemas relying

on local mappings between schemas. Piazza is based on the pure P2P architecture. Nejdl et al. proposed Edutella [49][50] which provides an RDF-based metadata infrastructure for P2P applications. The system builds upon peers that use explicit schemas to describe their contents. They use super-peer based topologies, in which peers are organized in hypercubes to route queries. The hypercube topology is similar to a Gnutella-like *unstructured* P2P network, with the key advantage that each node in the hypercube topology only receives a query once. However, in these two systems, queries have to be flooded to every node, making the system difficult to scale. Therefore, intelligent routing and network organization strategies are needed in such networks, to enable queries be routed to a semantically chosen subset of peers.

Crespo et al. [51] proposed the concept of Semantic Overlay Networks (SONs) in which peers are grouped by the semantic relationships of documents they store. Each peer stores additional information about content classification and route queries to the appropriate SONs, increasing the chances that matching objects will be found quickly and reducing the search load on nodes that have unrelated content. The study in [52] uses probabilistic analysis to show that multiple overlays, with each devoted to a particular kind of objects, can improve search performance considerably. However, the study does not address the routing issue as it ignores the link structure within an overlay network and represents an overlay network simply by the set of nodes in it. The maintenance cost in SONs may become more expensive when the number of SONs increases. While we adopt the basic idea of semantic clustering, we impose certain link structures on semantic clusters to facilitate both intra-cluster and inter-cluster routing. We also aim to reduce the overlay maintenance cost incurred by using high-dimensional overlays.

Tang et al. [53] applied classical Information Retrieval techniques to P2P systems and built a decentralized P2P information retrieval system called pSearch. The system makes use of a variant of CAN to build the semantic overlay and uses Latent Semantic Indexing (LSI) [54] – an extension of Vector Space Model (VSM) [65], to map documents into term vectors in the space. Li et al. [55][79] built a semantic small world (SSW) network in which peers with semantically close data are clustered based on term vectors computed using LSI. They proposed an adaptive space linearization technique, and constructed link structures based on small world network theory. The small world network model was originally introduced by Kleinberg [56]. He proposed a two-dimensional grid where every node maintains four links to each of its closest neighbors and one long distance link to a node chosen from a probability function. He showed that a query can be routed to any node in $O(log^2 n)$ hops, where $n$ is the total number of nodes in the network. Manku et al. [57] extended Kleinberg's small world construction by applying distributed hash tables and showed that with $s$ ($k > 1$) links per node, the routing latency reduces to $O(\frac{1}{k} log^2 n)$.

Our work is inspired by the small world network model. The SCS overlay network not only maps a $k$-dimensional semantic space to a one-dimensional semantic space (through the ring structure), but also allows peers to be grouped into sub-clusters in a semantic cluster (through the cluster encoding scheme) to better meet the scalability requirement and to facilitate parallel search. To route queries across clusters in SCS, we select two long distance links, which are located at certain positions of the ring space instead of choosing one randomly, as in the small world network model. Through simulation, we show how these two long distance links improve search efficiency with a varying number of semantic clusters. Furthermore, we propose the use of schema-based metadata to extract data semantics, which incurs a lower

overhead than LSI does. We show how these ideas can be applied in a semantic-based P2P lookup system for locating context information in multiple context spaces.

### 2.2.3 Summary

We have discussed previously a number of P2P information retrieval approaches. In these systems, semantic-based P2P systems allow users and applications to retrieve information that are semantically close to the query. Efficient semantic-based search is also a key determinant to system scalability.

We use a semantic P2P overlay as our basic design approach, and aim to provide a scalable, self-organized context lookup system in multiple context spaces. Our proposed ontology-based semantic clustering technique has several advantages as compared to other semantic extraction techniques such as VSM and LSI (used in [53][55][79]). The formal design of ontologies minimizes the problems of synonyms and polysemy incurred by VSM. Based on ontologies, data and queries can be mapped to appropriate semantic clusters directly without costly computation as in LSI, yet the same precision is retained. While we share the similar idea of semantic clustering as in SONs, we impose certain link structures on semantic clusters for routing queries efficiently and reducing overlay maintenance cost. We will describe and evaluate our proposed techniques in detail and compare the performance of our system to SONs in Chapter 4.

# CHAPTER 3

# CONTEXT MODELING AND REASONING[2]

A model for context representation model should be well established to support context lookup and reasoning services. In this chapter, we first describe the motivation of our modeling approach and propose an ontology-based context model, and then we present results of our earlier study on context reasoning based on our context model.

## 3.1 Motivation and our context modeling approach

In existing context-aware systems, contexts are often described as strings in documentations or modeled as software (e.g., Java) objects. This representation model is not expressive enough, and may depend on its software platform. We propose an ontology-based context model, which allows us to represent context data and its semantics independent of programming language, underlying operating system or application domains. The main benefit of this model is that it enables formal analysis of domain knowledge for context reasoning using first-order logic, temporal logic, and other methods.

In our context model, context data is described by ontologies written in OWL [59] – the web ontology language proposed by W3C's Web Ontology Working Group for the Semantic Web [58]. We have chosen OWL to realize our context model and to define our context ontologies for three reasons. First, it is more expressive compared to other

[2] The contents of this chapter have been presented in Papers 3 and 8 in the author's publication list.

ontology languages such as RDFS [60]. Second, it has the capability of supporting semantic interoperability for the exchange and sharing of context knowledge between different systems; it also enables automated reasoning be used by automated processes. Third, DAML+OIL [61] has been merged into OWL to become an open W3C standard.

## 3.2 An ontology-based context model

In our model, contexts are represented as first-order predicate calculus. The basic model has the form of a RDF triple, *Predicate(subject, value)*, in which:

- *subject* $\in$ S*: set of subject names, e.g., a person, a location or an object.
- *Predicate* $\in$ V*: set of predicate names, e.g., is located in, has status, and etc.
- *value* $\in$ O*: set of all values of subjects in S*, e.g., the living room, open, close, empty, and etc.

For example, *Location(John, bathroom)* represents John is located in the bathroom, *Temperature(kitchen, 120)* represents the temperature of the kitchen is 120ºF, and *Status(door, open)* represents the door's status is open, etc.

The basic context model can be extended to form a complex context or a set of contexts by combining the predicate and Boolean operations (union, intersection and complement). For example, *FoodPreference(familyMembers, foodItems)*, i.e., all family members' food preferences are a list of food items, can be represented as *FoodPreference(John, FoodList_1)* $\cup$ *FoodPreference(Alice, FoodList_2)* $\cup$ *FoodPreference(Tom, FoodList_3)*.

The structures and properties of context predicates are described in an ontology which may include descriptions of classes, properties and their instances. The ontology is written in OWL as a collection of RDF triples, each statement being in the form (*subject, predicate, object*), where subject and object are the ontology's objects or individuals, and predicate is a property relation defined by the ontology. An example is shown in Figure 3.1.

```
<rdf:RDF xmlsn:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlsn:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlsn:owl="http://www.w3.org/2002/07/owl#"
         xmlsn:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
>
. . .
<owl:Class rdf:ID="Adult">
   <rdfs:subClassOf rdf:resource="#Person"/>
   <owl:disjointWith rdfs:resource="#Child"/>
</owl:Class>

<socam:Adult rdf:ID="John">
   <socam:locatedIn rdf:resource="#bedroom"/>
</socam:Adult>
. . .
</rdf:RDF>
```

Figure 3.1: A partial context ontology written in OWL

## 3.3 Context ontologies

In this section, we present our design of context ontologies. We adopt a two-tier approach to design our context ontologies. This approach is based on the design principle of easy extensibility of context ontologies. In addition, the two-tier approach allows us to perform context reasoning in a distributed fashion. Distributed context reasoning can overcome the major obstacles of centralized reasoning as we will discuss in the next section. Our context ontologies are divided into the common upper ontology for the general concepts, and domain-specific ontologies for different sub-domains. The common upper ontology is a high-level ontology which captures

general contexts of the physical world in pervasive computing environments. The domain-specific ontologies are a set of low-level ontologies which define the details of general concepts and their properties.

For example in Figure 3.2, the upper ontology defines the basic concepts of person, location, activity, device, network and application. The class *ContextEntity* provides an entry point of reference for declaring the upper ontology. One instance of *ContextEntity* exists for each distinct user or service. Each instance of *ContextEntity* has a set of descendant classes of *Person, Location, Activity, Device, Network* and *Application*. The details of these basic concepts and their properties are defined in the domain-specific ontologies, which are partially shown in Figure 3.3. The definitions of the common upper ontology and a set of domain-specific ontologies can be found in Appendix A.

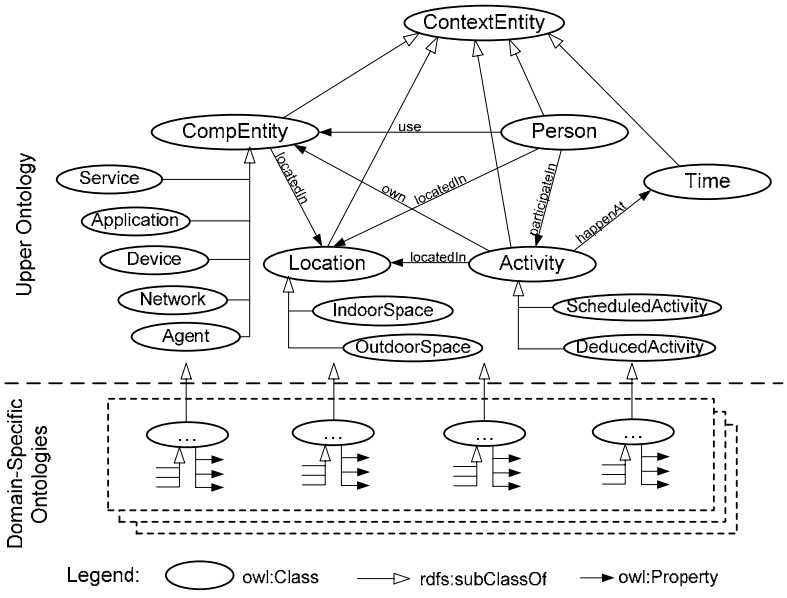



Figure 3.2: A two-tier approach to context ontologies

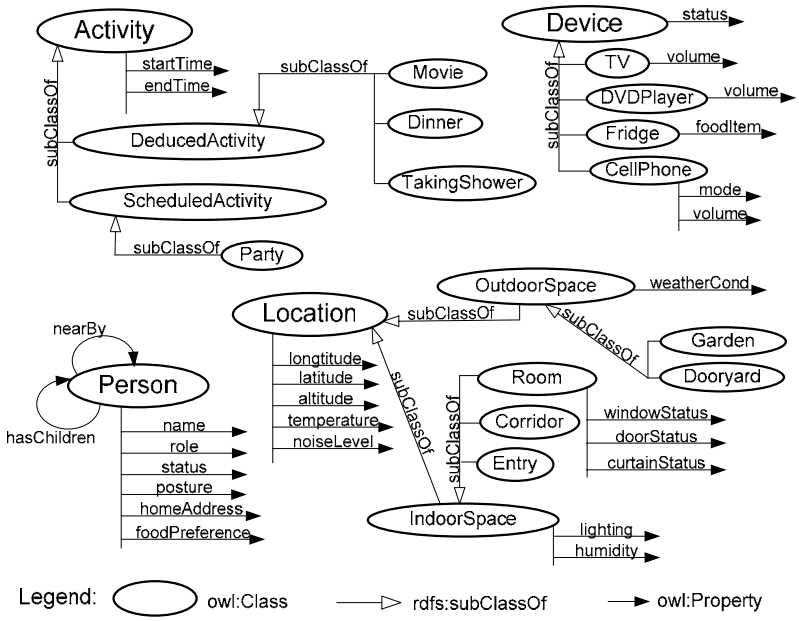



Figure 3.3: A partial definition of domain-specific ontologies

## 3.4 Context reasoning

In this section, we describe the details of context reasoning and present our earlier experiments on the feasibility study of applying logical reasoning in context-aware computing. We use logical reasoning which is based on first-order-logic to reason about context data. There are two kinds of logical reasoning in our system: ontology reasoning and user-defined rule-based reasoning. We discuss them in the following sections.

### 3.4.1 Ontology reasoning

Ontology reasoning is responsible for checking class consistency and implied relationship, asserting inter-ontology relations when integrating or switching domain-specific ontologies.

Ontology reasoning includes RDFS reasoning and OWL reasoning. RDFS reasoning supports all the RDFS entailments described by the RDF Core Working Group. OWL

reasoning supports OWL/lite [59], which includes constructs such as relations between classes (e.g., disjointness), cardinality (e.g., "exactly one"), equality, characteristics of properties (e.g., symmetry), and enumerated classes. A set of RDFS and OWL rules needs to be pre-specified; an example is shown in Table 3.1.

TABLE 3.1: A PARTIAL RDFS AND OWL RULE SET

| | |
|---|---|
| subClassOf | (?A rdfs:subClassOf ?B), (?B rdfs:subClassOf ?C) -> (?A rdfs:subClassOf ?C) |
| subPropertyOf | (?A rdfs:subPropertyOf ?B), (?B rdfs:subPropertyOf ?C) -> (?A rdfs:subPropertyOf ?C) |
| TransitiveProperty | (?P rdf:type owl:TransitiveProperty), (?A ?P ?B), (?B ?P ?C) -> (?A ?P ?C) |
| Disjointness | (?C owl:disjointWith ?D), (?X rdf:type ?C), (?Y rdf:type ?D) -> (?X owl:differentFrom ?Y) |
| InverseOf | (?P owl:inverseOf ?Q), (?X ?P ?Y) -> (?Y ?Q ?X) |

### 3.4.2 User-defined rule-based reasoning

User-defined rule-based reasoning provides forward chaining, backward chaining and a hybrid execution model. The forward-chaining rule engine is based on the standard RETE algorithm [63]. The backward-chaining rule engine uses a logic programming engine similar to Prolog engines. A hybrid execution mode performs reasoning by combining both forward-chaining and backward-chaining engines. Table 3.2 shows a partial rule set based on the forward-chaining rule engine.

TABLE 3.2: A PARTIAL USER-DEFINED RULE SET

| |
|---|
| (?user rdf:type socam:Person), (?user, socam:locatedIn, socam:Bedroom), (?user, socam:hasPosture, 'LIEDOWN'), (socam:Bedroom, socam:lightLevel, 'LOW'), (socam:Bedroom, socam:doorStatus, 'CLOSED') -> (?user socam:status 'SLEEPING') |
| (?user rdf:type socam:Person), (?user, socam:locatedIn, socam:BathRoom), socam:WaterHeater, socam:status, 'ON'), (socam:BathRoom, socam:doorStatus, 'CLOSED') -> (?user socam:status 'SHOWERING') |
| (?user rdf:type socam:Person), (?user, socam:locatedIn, ?room), (socam:TV, socam:locatedIn, ?room), (socam:TV, socam:status 'ON') -> (?user socam:status 'WATCHINGTV') |

### 3.4.3 Reasoning performance and discussion

To study the feasibility of applying logical reasoning in context-aware computing, we evaluate the performance of context reasoning by running the context reasoning engine on a Pentium II 600MHz PC. In our prototype, the context engine takes about 521 ms to load 96 context instances from various internal context producers, and it takes about 20 ms to merge these instances with the ontology. The context reasoning process takes about 1.9 seconds to derive high-level contexts. The context engine is able to answer queries for derived contexts at the average rate of a few milliseconds per query. This result shows that logical reasoning is a computationally intensive process and it may create a bottleneck when it is applied to pervasive computing domain. Many current pervasive devices may not be able to run the context engine.

To study reasoning performance over difference scales of context knowledge, we extend the home-domain ontology. In the experiment, we create five datasets with different sizes of classes and instances. The context engine validates and parses these OWL expressions into RDF triples and performs the reasoning task. For each dataset, we measure its average runtime. The result in Figure 3.4 shows that the runtime is appropriately linear in the scale of context knowledge. Based on this observation, we are able to improve overall performance by splitting the context reasoning process.

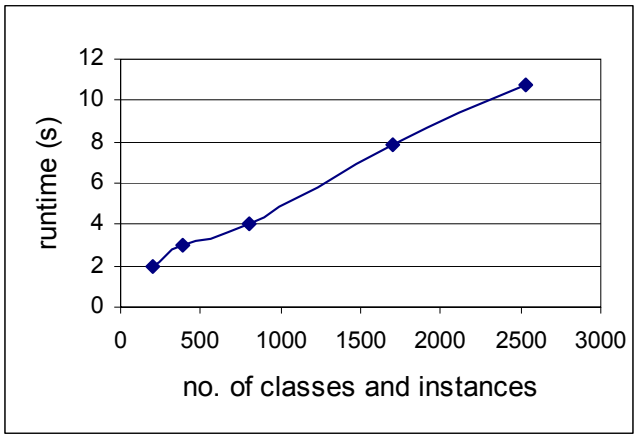



Figure 3.4: Performance of context reasoning

To have a better understanding of the reasoning performance, we study the reasoning process by comparing the two kinds of reasoning which are currently supported – ontology reasoning and user-defined rule-based reasoning. We measure the runtime of the two processes with respect to the different scale of context knowledge as shown in Figure 3.5. The experiment result shows the time taken for ontology reasoning is much more than that for user-defined rule-based reasoning. This is probably due to the large set of rules used in ontology reasoning whereas user-defined rule-based reasoning uses fewer rules. As the rule set for ontology reasoning remains unchanged

once defined, we are able to perform ontology reasoning in advance of a context query.

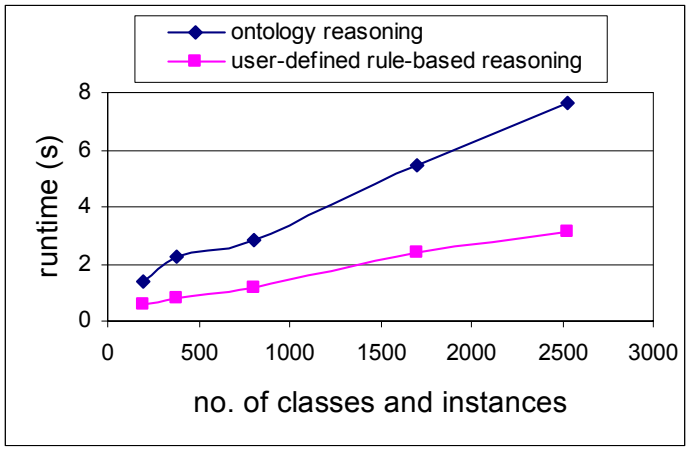



Figure 3.5: Reasoning comparison

Based on these experiments, we observe that logical reasoning is a computationally intensive process. Deploying a centralized reasoning engine to reason about context data in an application domain, which has a large number of context knowledge and user-defined rules, may result in poor reasoning performances. Distributed reasoning, where many reasoning engines are deployed in the system based on an application's requirements, can significantly ease the burden of processing and reduce memory consumption. The above experiments motivate us to deploy a distributed approach for context reasoning.

## 3.5 Summary

In this chapter, we have proposed an ontology-based context model to represent context data. As we have mentioned, the objective of our context model is to establish the foundation for our context-aware architecture. We limit our context model to providing the basic representation model and designing an appropriate context

ontologies. There remain numerous areas for future work. First, how to represent context classification, dependency, quality of context and uncertainty, and how to use these representations to reason about context data and solve context conflicts. Many reasoning techniques reported in the AI literature, such as probabilistic logic, temporal logic and Bayesian networks, can be applied to perform various context reasoning tasks. Second, privacy management in context-aware computing is an important issue. While many researchers such as Hong [17] and Chen [19] have focused on providing a management framework and solving interaction issues with users and applications, handling context privacy within the basic context model has not received enough attention yet. We believe by embedding privacy enforcing mechanisms into the basic model will allow context privacy issues be managed more efficiently. We will further discuss these issues in Chapter 7.

In this chapter, we have also evaluated the performance of logical and found it computational intensive. The experimental results have promoted us to propose a distributed approach to context reasoning. We will study and evaluate the distributed reasoning of our prototype implementation in Chapter 6. In the next chapter, we will describe the design of our P2P context lookup service in multiple context spaces.

# CHAPTER 4

# P2P CONTEXT LOOKUP[3]

In this chapter, we describe the P2P context lookup service in SCS. We first present an overview of SCS, followed by a detailed description of the various techniques we propose. Finally, we present the performance results of SCS obtained from a range of simulations. For ease of discussion, we use the terms node and peer interchangeable in the rest of the thesis.

## 4.1 Architecture overview

In SCS, a large number of nodes are arranged and self-organized into a semantic overlay network, in accordance with their semantics. A user or an application can act as a context producer, a context consumer, or both. Context producers provide various context data for sharing whereas context consumers submit their context queries and receive query results. Upon their creations, context producer peers are clustered according to their data semantics and mapped into a semantic cluster in SCS. Each peer is responsible for managing its own context data corresponding to a semantic cluster and publishing the data indices to peers in other semantic clusters. Each index serves as a node pointer to the physical location of the node where the context data is stored. The peers within the same cluster are interconnected and may be organized using any overlay structure. There is no restriction on the type of overlay used within a cluster. Upon receiving a context query, a peer first pre-processes the query and obtains the information about the semantic cluster associated with the query, and then

[3] The contents of this chapter have been presented in Papers 11, 12 and 14 in the author's publication list. The extension of this chapter has been submitted to a journal.

routes the query to an appropriate cluster in SCS. When the query reaches the designated cluster, it floods the query to all peers within the cluster. Peers that receive the query do a local search and report the index of the context data that the query is searching for. Each peer maintains a local context data repository which supports RDF-based semantic query using RDQL [64].

There are several critical issues to be addressed in the design of SCS. First, to facilitate semantic context search, we need to extract semantics from both context data and queries, and cluster peers in accordance with their data semantics. One solution is to use VSM from the Information Retrieval literature. The semantics of data objects can be abstracted and identified by *Term Vectors*. Each element of the vector represents a particular attribute or a term associated with the data object with weight reflecting the importance of that term in the given document. The semantic matching between context data and query is measured by computing the cosine of the angle between their vectors. However, VSM suffers from two problems: synonymy (a data object can be referred to in many ways) and polysemy (terms having more than one meaning). LSI aims to overcome these problems; however, the high computational and memory requirements of LSI and its inability to compute an effective dimensionality reduction in a supervised setting limit its applicability [66]. In this paper, we propose using ontologies as semantic metadata to extract data semantics, and arrange peers into various semantic clusters. The formal design of ontologies minimizes the problems of synonyms and polysemy. Based on ontologies, context data and queries can be mapped to appropriate semantic clusters directly without costly computation as in LSI while the same precision is retained. Then, in SCS, peers should be organized in such a way that those with semantically similar data are grouped together. To enable navigation and search across semantic clusters,

an intuitive solution is to construct *k*-dimensional semantic clusters by connecting each peer to all dimensions of the corresponding clusters. However, this approach incurs high maintenance cost for a high-dimensional semantic space due to the highly dynamic nature of peers in context-aware systems. To address this problem, we cluster peers and organize them into a ring space which maps a *k*-dimensional semantic space into a one-dimensional semantic space. Each peer in SCS keeps track of a node in each of its two adjacent clusters (i.e., neighbor clusters) so that all clusters can be interconnected in a ring fashion and navigation across different clusters becomes possible. To enable a query request to reach other semantic clusters quickly, each peer also keeps track of a certain number of *shortcuts*. Next, we need to retain all the good properties of a well-designed overlay network such as scalability, load balancing and fault tolerance. To address these issues, we propose a cluster encoding scheme which allows sub-clustering within a semantic cluster. Peers can be sub-clustered within a semantic cluster; the cluster splitting and merging mechanisms can be invoked so that the system can self-adapt to the number of peers. This scheme also enables us to search context data in parallel within a semantic cluster to improve search efficiency further.

We describe the ontology-based clustering technique in Section 4.2, and in Section 4.3, we present a simple solution to constructing a *k*-dimensional space, through which peers may connect to all dimensions of clusters. In Section 4.4, we describe the one-dimensional ring structure in detail, including peer placement, cluster naming, cluster splitting/merging, the routing algorithm and subscription.

## 4.2 Ontology-based semantic clustering

In this section, we describe how to use ontology to extract the semantics of both data and queries by using an example of context ontologies. As compared to attribute-value pairs, this context model has the advantage of enabling semantic representation and logic reasoning. The semantics of context data are represented by schema, i.e., context ontology. Various sets of context data are structured and classified according to these ontologies. This ontological structure is also exploited to extract query semantics and formulate context queries. We adopt a two-tier hierarchy in the ontology design as described in Section 3.3. The upper ontology which defines common concepts is shared by all peers. Each peer can define its own concepts in its lower layer ontologies. Different peers may store different sets of lower layer ontologies based on their application needs. An example of the ontological structure in context-aware systems is shown in Figure 3.4 in Section 3.3. The leaf nodes in the upper ontology are used as semantic clusters to cluster peers, and denoted as a set $E$ = {*Service*, *Application*, *Devices*, . . .}. Each of these pre-defined semantic clusters are assigned a unique ID upon their presence in the overlay network.

Mapping computation is done locally at each peer. For the mapping of RDF data, a peer needs to define a set of lower layer ontologies and store them locally. Upon joining SCS, a peer first obtains the upper ontology and merges it with its local lower layer ontologies. Then it creates instances (i.e., RDF data), and adds them into the merged ontology to form its local knowledge base. A peer can map its local data into one or more semantic clusters by extracting predicates of RDF triples. For example, as shown in the merged ontology in Figure 4.1(a), we can map predicate *locatedIn* into semantic cluster *IndoorSpace* by checking its *rdfs:range* if the predicate is of

type *ObjectProperty*. If the predicate is of type *DataTypeProperty*, for example, *lightLevel*, we will check its *rdfs:domain* to get the class – *Location*. As *Location* is not a leaf node in the upper ontology, we need to find out its subclasses/superclasses until the leaf nodes are reached. Finally, *lightLevel* is mapped into both the *IndoorSpace* and *OutdoorSpace* semantic clusters. To provide more precise mapping, we make use of the subject and object of an RDF data triple. Let $SCn_{sub}$ , $SCn_{pred}$ and $SCn_{obj}$ where $n$ = 1, 2, . . . denote the semantic clusters extracted from the subject, predicate and object of a data triple respectively. Unknown subjects/objects (which are not defined in the merged ontology) or variables are mapped to *E*. If the predicate of a data triple is of type *ObjectProperty*, we obtain the semantic clusters using $(SC1_{pred} \cup SC2_{pred} \cup ... SCn_{pred}) \cap (SC1_{obj} \cup SC2_{obj} \cup ... SCn_{obj})$. If the predicate of a data triple is of type *DatatypeProperty*, we obtain the semantic clusters using $(SC1_{sub} \cup SC2_{sub} \cup ... SCn_{sub}) \cap (SC1_{pred} \cup SC2_{pred} \cup ... SCn_{pred})$. Examples 1 and 2 in Figure 4.1(a) show the RDF data triples about the location and light level in a bedroom provided by a producer peer. In Example 2, we first obtain the semantic clusters from both the subject and predicate, and then intersect their results to get the final semantic cluster – *IndoorSpace*.

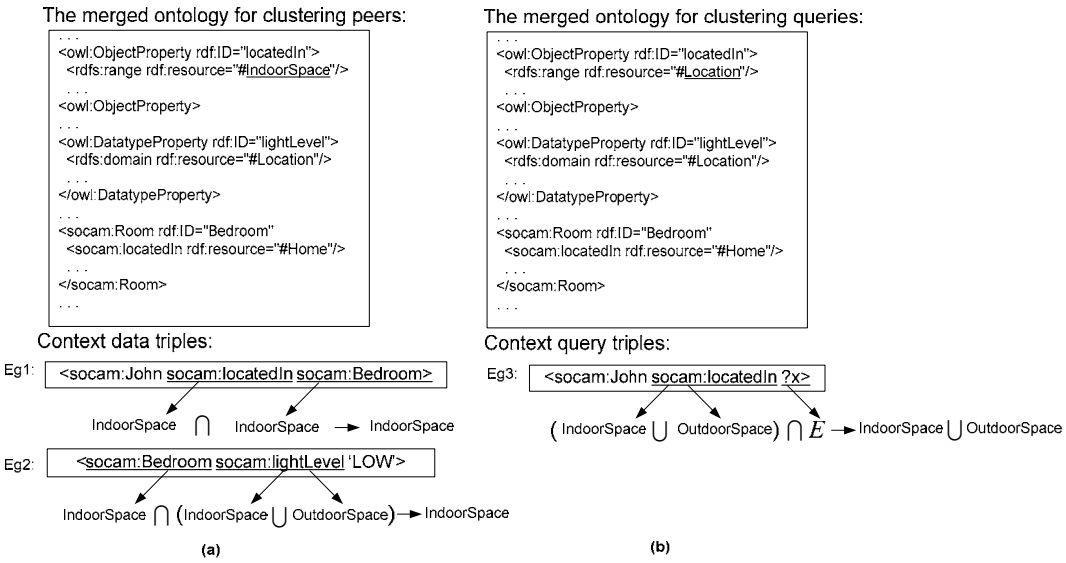



Figure 4.1: An example of semantic cluster mapping

A query follows the same procedure to obtain its semantic cluster(s), but it needs all the sets of lower layer ontologies. In real applications, users may create duplicate properties in their lower layer ontologies which conflict with the ones in the upper ontology. For example, the upper ontology defines the *rdfs:range* of predicate *locatedIn* as *Location* whereas the lower layer ontology defines its *rdfs:range* as *IndoorSpace*. To resolve this issue, we create two merged ontologies, one for clustering peers and the other for clustering queries. If such a conflict occurs, we select the affected properties defined in the lower layer ontology to generate the merged ontology for clustering peers' data, and select the affected properties defined in the upper ontology to generate the merged ontology for clustering queries. With this scheme, a peer can extract the semantics of its data triples more precisely, based on its lower layer ontology without losing generality for queries. For example, predicate *locatedIn* may have the *rdfs:range* of *IndoorSpace* (underlined in Figure 4.1(a)) in the merged ontology for clustering peers' data and have the *rdfs:range* of *Location* (underlined in Figure 4.1(b)) in the merged ontology for clustering queries.

Data triple <*socam:John socam:locatedIn socam:Bedroom*> will be mapped to *IndoorSpace*, and query <*socam:John socam:locatedIn ?x*> will be mapped to *IndoorSpace* and *OutdoorSpace* rather than only *IndoorSpace*. This is most likely the case in real life applications.

## 4.3 ContextBus

In this section, we present a simple approach – ContextBus [30] to construct a *k*-dimensional semantic space. ContextBus is an overly network where each ContextBus ties and manages context producer nodes with semantically similar data. Hence, one can view the address of a ContextBus as the index to the same category of context information. By visiting those nodes on one ContextBus or multiple ContextBuses in parallel or in some order, context queries can be resolved quickly. The basic idea of ContextBus is similar to SONs. ContextBus allows each peer to connect to all dimensions of the corresponding semantic clusters to facilitate navigation and search across these clusters. As shown in Figure 4.2, upon creation, each node may join a semantic cluster (i.e., ContextBus) by creating a physical connection to an existing peer in the cluster. A peer may participate in one or more semantic clusters, depending on the semantics of the context data it stores.

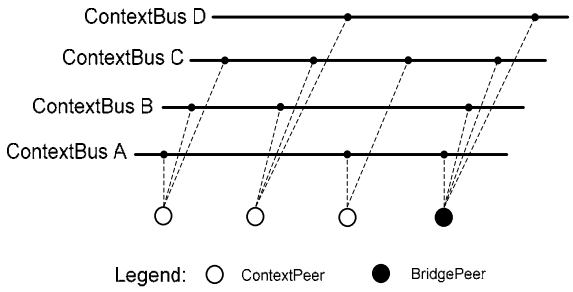



Figure 4.2: Overview of the ContextBus architecture

We recognize that peers have different capability constraints, such as maximum node degree (i.e., number of active connections per node). There are basically two classes of peers based on this constraint: high-degree and low-degree. Let $M$ be the maximum degree of a node and $C$ be the total number of semantic clusters in the system. A peer node is called a *high-degree* node if $M \geq C$ and a *low-degree* node if $M < C$. For a *high-degree* node, the bootstrap process ensures that it is connected to all semantic clusters. For a *low-degree* node, we first connect the node to those semantic clusters which are semantically similar to the context data stored in the node. This is to ensure that all the producer nodes within a semantic cluster are interconnected to each other, so that they can be searched by a query. We then assign at least one remaining connection of the node to a *high-degree* node. In this case, the *high-degree* node can act as a bridge (note: we call it *BridgePeer* as shown in Figure 4.2) for the *low-degree* node, so that a query can be routed to any other semantic clusters that the low-degree node is not able to reach by itself. Each peer keeps and maintains a list of direct (i.e., one-hop) neighbors. Upon receiving a query, a node extracts the semantic cluster(s) from the query and determines which semantic cluster to forward the query to. A *high-degree* node is able to forward a query to any semantic cluster, and a *low-degree* node, if it does not connect to the desirable semantic cluster directly, may forward the query to a *high-degree* node for the query to be forwarded to the appropriate semantic cluster. Once the query reaches the desirable semantic cluster, it will be flooded to all nodes within the cluster.

In the rest of this section, we describe the bootstrapping process of the ContextBus architecture and the routing process. We also discuss the drawbacks of this solution.

### 4.3.1 Bootstrapping

When a new node is created, it first goes through a bootstrapping process to join the network. A bootstrap server maintains information of available nodes for a certain region. A node's entry in the bootstrap server is a pair, <*nodeID*, *nodeClass*>, indicating the node's ID and its class (*high-degree* or *low-degree*). Entries are grouped according to the ContextBuses that nodes participate in. Multiple entries may exist across different ContextBuses as nodes may join multiple ContextBuses.

When a node, say *x*, joins a ContextBus, it first obtains one or more existing nodes in this ContextBus from the bootstrap server, and then connects to each of these nodes. These node's IDs are stored in node *x*'s routing table. A *high-degree* node will be able to join all the ContextBuses it wishes to join. A *low-degree* node will not be able to do so due to its limited number of available connections. In this case, we first satisfy those ContextBuses that provide the same type of context data as provided by the node, and then assign the remaining connections other ContextBuses. This ensures that a query for a particular type of context data reaches all nodes providing that type of context data. Here, we assume that the maximum degree of a *low-degree* node is more than the number of ContextBuses providing the same type of context data as the node does. For the assignment of the remaining connections, a *low-degree* node must connect to at least one *high-degree* node. This ensures that a *low-degree* node is able to route queries to any ContextBus, either by itself or through a *high-degree* node.

Recently, researchers in [67] realized the topology mismatching problem limits the performance of various search and routing techniques. To ensure that ContextBuses mirror the physical network as much as possible, we perceive that it is more efficient to perform topology optimization within each ContextBus upon a node joining or

leaving. This optimization requires the knowledge of link costs between every two nodes. In [68], a technique is proposed to determine these link costs, using the latency between each node to multiple servers. This technique may be employed to optimize ContextBus topologies.

### 4.3.2 Routing

Upon entry to the system, each node *x* creates a routing table containing a set of node IDs that are grouped according to ContextBus IDs. These nodes are the direct (or one-hop) neighbors of node *x*. As a *high-degree* node connects to at least one node in each ContextBus, it can forward any query to any ContextBus. If a query is generated at a *low-capacity* node, it forwards the query to a *high-degree* node if the query is destined for ContextBuses that it cannot connect to directly. In this case, the *high-degree* node acts as a bridge for the *low-degree* node, routing the query to the appropriate ContextBuses. The query is then flooded within a ContextBus using Cost-Aware Selective Flooding, which we will describe in Chapter 5.

### 4.3.3 Discussion

This approach works well (as we will demonstrate later in our simulation in Section 4.5.3) when the dimensionality of SCS (i.e., the number of semantic clusters in SCS) is reasonably low. However, the maintenance cost rises when the number of semantic cluster increases. In addition, as the ratio of low-degree nodes to high-degree nodes increases, a processing bottleneck may form at the high-degree nodes, and hence, search efficiency decreases.

## 4.4 Semantic Context Space

To deal with the efficiency concern of ContextBus as we discussed in Section 4.3.3, we present here a new approach to reduce maintenance cost, and facilitate efficient navigation and search in a high-dimensional semantic context space. In SCS, nodes are organized in such a way that those with semantically similar data are grouped into a semantic cluster. To enable navigation and search across semantic clusters, an intuitive solution is to construct *k*-dimensional semantic clusters by connecting each node to all dimensions of the corresponding semantic clusters such as in [51] and [30]. However, overlay maintenance cost rises when the number of semantic clusters increases because each node needs to maintain more nodes in its routing table. We seek to resolve the problem by using a one-dimensional ring structure to construct the overlay network, which enables mapping from a *k*-dimensional semantic context space into a one-dimensional semantic context space.

### 4.4.1 Peer placement

Upon joining SCS, a node needs to join an appropriate semantic cluster. We use the ontology-based semantic clustering technique to extract the semantics of its local RDF data and then map it to a semantic cluster(s). A node may have more than one data set each could be mapped into a different semantic cluster. We refer the semantic cluster of its biggest data set as the *major* semantic cluster, and the semantic clusters of its remaining data sets as the *minor* semantic clusters. We place a node into its major semantic cluster, and publish the indices of each minor data to its corresponding minor semantic clusters. This is to ensure a query reaches all the potential nodes which have the same semantics as that of the query. A node publishes the indices of its data to the minor semantic cluster(s) as follows: It selects a random

node in each of its minor semantic clusters, and places its indices (i.e., reference pointers) in these nodes. For example, as shown in Figure 4.3, *Peer 1* publishes its index to semantic cluster *SC1* by putting its index to a random node – *Peer 3* in *SC1*. As a result, a semantic cluster consists of a set of interconnected nodes which are grouped based on their semantics; and a collection of indices are also stored in these nodes.

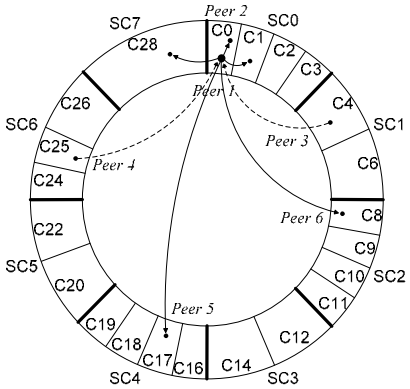



Figure 4.3: One-dimensional ring structure

The above scheme has several positive effects. For example, if a node has homogeneous data in its local repository, most of its data is categorized into one corresponding semantic cluster, therefore reducing the cost of publishing data indices. This is likely to be the case in real applications. Furthermore, many applications are designed in such a way that a node is likely to query for data available in its nearby nodes. By placing a node into one particular semantic cluster based on the category of its majority data, a query can be resolved very efficiently. Note that while we have only elaborated the joining of a single semantic cluster, the same principle can be applied by any node for joining multiple semantic clusters.

### 4.4.2 Cluster naming scheme

To interconnect different semantic clusters, we use one-dimensional ring structure. With this ring structure, a *k*-dimensional semantic space can be linearized. To place semantic clusters into a ring, we need to design an appropriate cluster naming scheme. In SCS, we distinguish the concepts of *cluster* and *semantic cluster*. A *cluster* refers to a partition that consists of a set of nodes grouped together, such as *C0* in Figure 4.3 (Note: a *cluster* is also referred to a *sub-cluster* in this thesis). A *semantic cluster* refers to a set of clusters corresponding to the same semantics. For example, cluster *C0*, *C1*, *C2*, and *C3* belong to semantic cluster *SC0*. We propose our cluster naming scheme as follows: A *Cluster ID* which is represented by a *k*-bit binary string (where $k = m + n$) is a unique ID that identifies a cluster in SCS. The first *m*-bit binary string (we call it *Semantic Cluster ID*) is used to identify a semantic cluster. Hence, an SCS can have a maximum of $2^k$ clusters and $2^m$ semantic clusters. An example of an SCS which assumes $k = 5$ and $m = 3$ is illustrated in Figure 4.3. The rationale behind this naming scheme is that, for a given query, we need to obtain the appropriate *Semantic Cluster ID* (rather than *Cluster ID*) to match the same semantics of the query. Semantic clusters can be viewed as an additional layer on top of actual clusters. Partitioning peers within a semantic cluster into a set of sub-clusters also provides better load balancing and enables parallel search within the same semantic cluster.

### 4.4.3 Ring construction

To construct SCS, each node in SCS creates and maintains a set of node entries in its routing table for message routing. A node, say *x*, first decides which semantic cluster to participate in. It then picks a cluster randomly within this semantic cluster to join by connecting to a number of nodes in this cluster. These node entries (called *x*'s

neighbors in its own cluster) will be maintained in *x*'s routing table. Node *x* also creates and maintains two node entries in each of its adjacent clusters. We call these two nodes *x*'s neighbors in its adjacent clusters. Each node joins the network by performing this operation, resulting in all the clusters being linked linearly in a ring fashion. Maintaining two neighbors in adjacent clusters for every node in SCS also ensures that a query generated at any node can reach any other cluster by navigating the ring space. However, queries have to be passed around the ring space linearly, either clockwise or anticlockwise, until the destination semantic cluster is reached. This approach may not be efficient when the number of semantic clusters is large. To accelerate search across semantic clusters, node *x* maintains a set of links to nodes in other semantic clusters except the two adjacent clusters. These nodes provide *shortcuts* (similar to long contacts in the small world network) for *x* to route a query to other semantic clusters quickly. For example, in Figure 4.3, *x* creates and keeps track of two *shortcuts*: one points to the opposite semantic cluster (i.e., *shortcut* to *Peer 5*) and the other points to the semantic cluster located in a quarter of the ring space (i.e., *shortcut* to *Peer 6*). In the process of cluster splitting and merging or when a new semantic cluster is inserted into the ring space, a node needs to update its neighboring nodes in both its own cluster and its adjacent clusters. However, a node only needs to update its *shortcuts* upon the insertion or deletion of a semantic cluster as a *shortcut* points to an appropriate semantic cluster rather than a cluster.

### 4.4.4 Cluster Splitting and Merging

The operations of cluster splitting and merging enable SCS to adapt and scale to a large number of nodes. Let *M* represent maximum cluster size. If the size of a cluster exceeds *M*, the splitting process is invoked to split the cluster into two. A simple way

of cluster splitting is to partition a cluster into two clusters of equal size without considering load distribution in the two clusters such as in Chord. To balance the load during splitting and merging, each node maintains a *CurrentLoad* which measures the node's current load in terms of the number of RDF triples and data indices the node stores. When node *x* joins the network, it sends a join request message to an existing node, say *y*. If *y* falls into the same semantic cluster that *x* wishes to join, *x* joins the cluster by connecting to *y* if its cluster size is below *M*; otherwise, *y* directs the request to a node, say *z*, in the semantic cluster that *x* wishes to join, and *x* connects to *z* if its cluster size does not exceed *M*. If the cluster size exceeds *M*, node *y* or *z* (called an initial node) initiates the splitting process. The initial node first obtains a list of all nodes in the cluster, which is sorted according to their *CurrentLoads*. Then it assigns these nodes in the list to the two sub-clusters alternately. After splitting, we obtain two clusters of relatively equal load. The initial node is also responsible for generating a new cluster *ID* for each of the two sub-clusters. To obtain a new cluster ID, each node maintains a *bit split pointer* which indicates the next bit to be split in the *n*-bit binary string (where $n = k - m$). For example, in Figure 4.3, we assume $m = 3$, $n = 2$, and there exists a cluster *C4* in the network. Initially the *bit split pointer* points to the most significant bit of the *n*-bit string. When cluster splitting occurs, the bit pointed by the *bit split pointer* is split into 0 and 1, and the pointer is moved forward to the next bit in the *n*-bit string. Therefore, we obtain cluster *IDs C4* and *C6*, which correspond to the same semantic cluster *SC1*. Cluster *C4* or *C6* can be further split into *C4* and *C5* or *C6* and *C7*, and finally the bit split pointer is set to null, indicating no cluster splitting is allowed. The same mechanism follows for insertion of a new semantic cluster in SCS. A semantic cluster can be split into a maximum number of $2^n$

clusters. After splitting, a node updates its *cluster ID*, the *bit split pointer*, and the neighbor lists in both its own cluster and its adjacent clusters.

When node $x$ leaves the network, it first checks whether its cluster size has fallen below a threshold $M_{min}$. If the current size is above $M_{min}$, $x$ simply leaves the network by transferring its indices to a randomly selected node in its cluster. Otherwise, this cluster needs to be merged into one of its neighboring clusters within the same semantic cluster. The leaving node triggers cluster merging, which is an inversed process of cluster splitting. To obtain the newly merged cluster *ID*, the *bit split pointer* moves backwards by 1 bit in the $n$-bit string, and the bit pointed to by the *bit split pointer* is set to 0. The nodes in the merged cluster need to perform the same updating as in the splitting process. For the selection of $M_{min}$, a simple method is to let $M_{min} = 1$ so that cluster merging is invoked when the last node in a cluster leaves. However, if there is only one node in a cluster, this node may become a hot spot as all the nodes in its two adjacent clusters have links to it. The actual value of $M_{min}$ should be determined by the statistics of nodes joining and leaving within this cluster. If the last node in a semantic cluster leaves, it initiates two messages to all the nodes in its two adjacent clusters, informing them to update their neighbor lists. Subsequently, the semantic cluster will be removed from SCS.

### 4.4.5 The routing algorithm

In this section, we describe the routing operation in SCS. As described above, each node in SCS maintains a routing table with a set of node entries (in the form of a pair <*NodeID*, *ClusterID*>) in its own cluster, two adjacent clusters and another two semantic clusters. It also keeps state information about its own cluster, consisting of a $k$-bit *ClusterID* (where $k = m + n$) which indicates the cluster it resides in, and

*ClusterSize* which specifies the current size of its cluster. Upon receiving a query, node *x* first obtains the *destination semantic cluster ID* (denoted as *D*), which is extracted from the query. Then node *x* checks whether *D* falls into its own semantic cluster by comparing *D* against the most significant *m*-bits of its *ClusterID*. If that is the case, *x* floods the query to all the nodes in its own cluster, and also forwards the query to the nodes in its adjacent clusters corresponding to *D*. The first node in a cluster that receives the query is always responsible for forwarding the query to its adjacent cluster(s) corresponding to *D*. This is achieved by turning on a special bit – the *first-node flag* that is appended to the query message (more details can be found in Section 6.5.2). In this way, search can be performed in parallel within a semantic cluster. The forwarding processes are recursively carried out until all the clusters corresponding to *D* have been covered and all nodes in each of the clusters have received the query. Every node, upon receiving a query, checks its local data repository and returns the matched data and indices. For example, as illustrated in the top of Figure 4.4(a), if a query is initiated at *Peer 1* with *D* = *SC0*, *Peer 1* first forwards the query to its neighboring node in its right adjacent cluster – *C1*, and then floods the query to all the nodes in its own cluster – *C0*. The same process is repeated in cluster *C1*, *C2* and *C3*. If *D* falls into node *x*'s adjacent semantic cluster, for example, in the case of a query generated at *Peer 2* with *D* = *SC3* as shown in the bottom of Figure 4.4(a), *Peer 2* will forward the query through its left neighboring node towards *SC3*. When the query reaches the destination semantic cluster, it will be flooded to all sub-clusters – *C14* and *C12* in *SC3*. If *D* neither falls into node *x*'s own cluster nor its adjacent semantic cluster, *x* relies on its *shortcuts* to route the query. A query can be routed to a semantic cluster which is closer to the destination semantic cluster quickly with the help of these *shortcuts*.

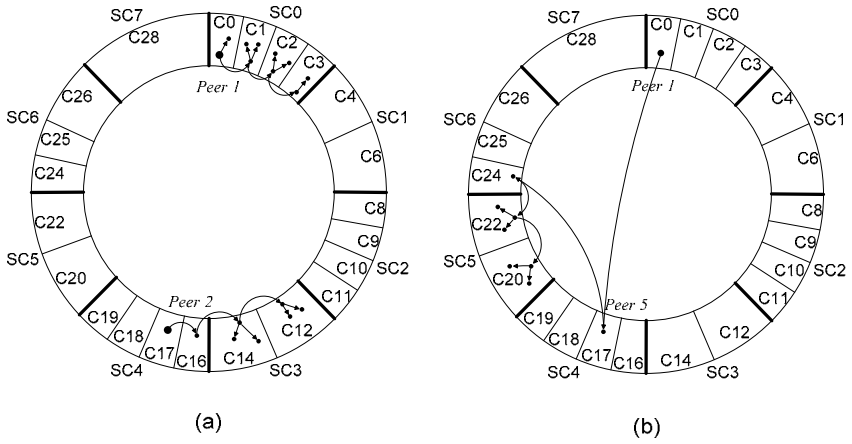



Figure 4.4: Query routing

In the design of these *shortcuts*, we have several options. We need to decide which semantic cluster a *shortcut* should point to and how many *shortcuts* each node should maintain. One strategy is to create a small number of random *shortcuts* (i.e., distant nodes) that is similar to the long contact in the small world network. Each node can have $s$ *shortcuts* ($s \geq 1$) with the tradeoff that the cost of creating and maintaining these *shortcuts* is proportional to $s$. Upon receiving a query, if the distance between $D$ and the semantic cluster that its *shortcuts* point to falls below a threshold – a preset minimum distance in terms of number of hops – the query is forwarded to the closest semantic cluster and hops towards the destination semantic cluster. If not, $x$ selects a *shortcut* randomly, and forwards the query to the *shortcut*. The same process is invoked until the distance to $D$ is below the threshold.

Our approach is based on the observation that a ring space can be equally divided into several partitions. Each node maintains two *shortcuts* ($s = 2$) that are used to partition the ring space. For example, we can partition a $2^m$ semantic space where $m = 3$ into four by creating two *shortcuts*: one pointing to the opposite semantic cluster and another pointing to the semantic cluster located in a quarter of the ring space. Given the maximum cluster size $M$, the system can have a total of $M \cdot 2^{m+n-1}$ nodes when $M_{min}$

= 1. Let *Cx* denote the cluster where *x* resides, and $SC_x$ denote the semantic cluster that *Cx* corresponds to. *SCx* can be obtained by truncating *Cx* to *m* bits from the most significant bit. The two semantic clusters $SC_{half}$ and $SC_{quarter}$ that *x*'s *shortcuts* point to are denoted as $(SC_x + 2^i) \bmod 2^m$, where $i = m - 1, m - 2$. To initial a search, *x* obtains *D* based on a query and checks which cluster range (partitioned by *x*'s *shortcuts*) *D* falls into. Then node *x* forwards the query to the closer semantic cluster through its *shortcut*. If *D* is closer to $SC_x$, node *x* forwards the query across its adjacent cluster towards *D*. A query takes a maximum of $2 + 2^{m-3}$ hops to reach the destination semantic cluster.

```
Assuming the longest shortcut point to 1/p (p=2^i,i=0,1,…) of the ring.
Obtain the destination semantic cluster D based on the query q.

if dist(SC_x, D) ≤ 2^m/4p then
  forward q to x's adjacent cluster towards D;
else if D falls into [SC_x, SC_quarter] or [SC_quarter, SC_half] or [SC_half, SC_x] then
  forward q to the semantic cluster that is closer to D
end if
```

Figure 4.5: Pseudocode of the search algorithm

The search algorithm is shown in Figure 4.5. To illustrate, consider Figure 4.4(b), where *Peer 1* generates a query and computes the destination semantic cluster as *SC5*. *Peer 1* first realizes that *SC5* falls into the interval [*SC4*, *SC0*] and *SC4* is close to *SC5*. Then *Peer 1* forwards the query to *Peer 5* at *C17*. As *SC5* falls into [*SC4*, *SC6*] and *C24* is closer to *SC5* as compared to *C17*, *Peer 5* forwards the query to *SC6* through its quarter *shortcuts*. Finally, the query reaches *SC5* and is then flooded in both *C22* and *C20*.

The more *shortcuts* created to partition the ring space, the finer the granularity we gain to locate the destination semantic cluster. As a result, we achieve better search

performance in terms of fewer routing hops. However, more *shortcuts* imply a higher cost of creating, updating and maintaining the *shortcuts*. In SCS, we set the number of *shortcuts* to two for the reasoning of keeping overlay maintenance cost low. To partition the ring space in a finer granularity when the number of semantic clusters *m* increases, we can place the longest *shortcut* into different points in SCS. The other *shortcut* always points to the middle semantic cluster between $SC_x$ and the semantic cluster that the longest *shortcut* points to. For example, if we place the longest *shortcut* to one-quarter of the ring, the ring space is divided by eight, and so on. We will evaluate our design decisions in various settings through simulation in Section 4.5.3.4. These *shortcuts* reduce network diameter and transform the network into a small world with a polylogarithmic search path length. More generally, the following theorem obtains the search path length for SCS.

**Theorem 1** Given a *m*-dimensional SCS of *N* nodes, with maximum cluster size *M,* number of bits to identify sub-cluster *n* and number of *shortcuts s*, the average path length for routing across semantic clusters is $O(\frac{1}{s} log^2(N/M \cdot 2^{n-2})^{1/m})$.

**Proof:** We follow a process similar to that in [56] to prove the theorem. In [56], Kleinberg proved that the optimal setting for shortcuts is $f_x = 1/x^m$, where m is the dimensionality. Thus, in SCS, a peer chooses another peer at distance *x* as one of its *shortcuts* using the pdf: $f_x = 1/x^m$ for $x \in [r, 1]$ where *r*, the minimum distance of a *shortcut*, is the average diameter of a semantic cluster (i.e., the maximum number of hops between two arbitrary to nodes in a semantic cluster). The average size of a semantic cluster is $\frac{M}{2} 2^{n-1}$, there are altogether $N/M \cdot 2^{n-2}$ semantic clusters in the system, and each semantic cluster takes charge of $M \cdot 2^{n-2}/N$ portion of the whole

semantic space on average. Therefore, the diameter of each partition $r$ is approximately $(M \cdot 2^{n-2} / N)^{1/m}$.

We extend the small world network model from 2-dimensional space to $m$-dimensional space. We use unit data space in SCS. Since each subspace has side length $r$ on average, there are $1/r$ subspaces along each side. The distance between two clusters along a dimension is the range of [1, 2, ... , $1/r$]. Thus, we separate the search process into phases 1, 2, ... , $log(1/r)$. Let $d$ be the distance from a query message's current node to the destination, and $d_i = 1/2_i$. Search is at phase $i$ if $d_{i+1} \leq d < d_i$. Phase $i$ ends when the message is forwarded to a peer less than $d_{i+1}$ distance away from the destination. The set of peers less than $d_{i+1}$ distance away from the destination is denoted as $D_{i+1}$, whose volume is $d_{i+1}^m$. The largest distance from a peer at phase $i$ to a peer in set $D_{i+1}$ is $d_i + d_{i+1}$. Since a peer has $s$ *shortcuts*, the probability that a peer at phase $i$ has contacts to set $D_{i+1}$ is at least $s \cdot d_{i+1}^m \cdot f_{d_i + d_{i+1}} = s / c \cdot \log(1/r)$ where $c$ is a constant that depends on $m$. Therefore, a query message requires $c \cdot \log(1/r) / s$ steps to reach the next phase on average. Since there are in total $log(1/r)$ phases, the total search path length is $O(\frac{1}{s} log^2(N / M \cdot 2^{n-2})^{1/m})$. □

### 4.4.6 Subscription

In addition to search requests which pull data from the network on a one-time basis, SCS enables consumers to issue subscription requests to the network and be notified when data changes over a period of time. When a subscription request is generated, it is first mapped to a semantic cluster ($D$) and then forwarded to all nodes in $D$. The mapping and routing processes of a subscription request are identical to a search

request. When a node in *D* receives a subscription request, it checks its local RDF data and decides whether it should accept the request. For example, an application can subscribe the event *John is in the bedroom* in the RDF triple form of <*socam:John socam:locatedIn socam:Bedroom*> to the network and trigger an action when this event occurs. As this RDF triple may not exist in the network (John may be in some other places) at the time of receiving a request, the subscription request may end up with no producers. To avoid losing potential producers or ending up with many irrelevant producers, we employ the subscription acceptance policy as shown in Figure 4.6, and illustrate how it works in context-aware computing and sensor network domains.

```
Given a subscription request in the form of a RDF triple pattern <Subs,
Preds, Objs>, a variable in the RDF triple represents any arbitrary
constant.
Let <Sub1, Pred1, Obj1> represents any RDF triple in a peer's local data
set called L.

accept = false; //initialization
for each RDF triple in L
  if Preds is of DatatypeProperty && ((Subs == Sub1) ∩ (Preds == Pred1))
     == true then
    accept = true;
    break;
  else if Preds is of ObjectProperty &&((Preds == Pred1) ∩ (Objs == Obj1))
          == true then
    accept = true;
    break;
  end if
end for
if accept == true then
   accept the subscription request;
else
   reject the subscription request;
end if
```

Figure 4.6: Subscription acceptance policy

Based on this policy, a producer peer attempts to match a subscription request against its local RDF data. This policy works for a subscription request in the form of any RDF triple pattern whose subject, predicate or object may take variables. Although predicates can be specified as variables, this situation seldom occurs since users or applications are always in favor of more specific events in real-life applications. We now consider the case that a predicate is specified in a subscription request. If a subscription request's predicate is of type *DatatypeProperty*, a producer peer determines if its local RDF data contains triple(s) with the same subject-predicate pair as in the request. For example, for a given subscription request <*socam:Bedroom socam:lightLevel 'LOW'*>, a producer peer will accept the request if there exists a RDF triple with subject "*socam:Bedroom*" and predicate "*socam:lightLevel*" in its local data. If a subscription request's predicate is of type *ObjectProperty*, a producer peer determines if its local RDF data contains triple(s) with the same predicate-object pair as the request. For example, for a given subscription request <*socam:John socam:locatedIn socam:Bedroom*>, a producer peer will accept the request if there exists a RDF triple with subject "*socam:locatedIn*" and predicate "*socam:Bedroom*" in its local data.

To understand the rationale behind these decisions, consider a subscription request in the form of the RDF triple <$Sub_s$, $Pred_s$, $Obj_s$>. Such a triple may be obtained from raw data generated by a sensor, which could be physical or virtual. In the domain of sensor networks, a predicate always corresponds to a sensor type. For example, "*socam:locatedIn*" corresponds to a physical location sensor and "*socam:participateIn*" corresponds to a virtual activity sensor. If $Pred_s$ is of *DatatypeProperty*, $Sub_s$ should correspond to the target this sensor is monitoring while $Obj_s$ should correspond to the sensor output. For example, the RDF triple of

<*socam:Bedroom socam:lightLevel 'LOW'*> can be interpreted as the output of a light level sensor monitoring the bedroom's light level. If a producer peer's local RDF data contains at least one triple with the $Sub_s$-$Pred_s$ pair, it can infer that this producer peer has the type of sensor specified by this pair. Hence, we can conclude that this producer peer can provide triples of this same subject-predicate pair. On the other hand, if $Pred_s$ is of *ObjectProperty*, $Obj_s$ should correspond to the target this sensor is monitoring while $Sub_s$ should correspond to the sensor output. In this case, the producer can provide triples with the same $Sub_s$-$Pred_s$ pair as in the subscription request.

Once a producer peer accepts a subscription request, it keeps monitoring the request. Whenever a change/event occurs (i.e., an RDF triple is added or removed), the producer peer notifies the subscribers when the RDF triple matches the subscription request. An RDF triple <$Sub_c$ $Pred_c$ $Obj_c$> is said to match the subscription request if $(Sub_c == Sub_s) \cap (Pred_c == Pred_s) \cap (Obj_c == Obj_s) == 1$. The routing of notification traces the exact path of the subscription request in the reverse direction. A subscriber can unsubscribe an event by sending an unsubscription request directly to the producers.

### 4.4.7 Peer dynamics and failure

In dynamic environments, a node may join and leave the system freely. In SCS, to keep track of its neighboring nodes, a node maintains a number of additional backup links for every link a node has. The approach has been used in many other P2P systems such as Pastry and CAN. However, in a highly dynamic system, detecting link failure during query routing can introduce additional overhead. Moreover, in the event of all its backup links fail, a node has to re-establish its neighboring links during

search, and this affects search performance. With this approach, a node needs to inform its neighboring nodes about its leaving and transfer its indices to a randomly selected node in its cluster before leaving. Another approach is that each node periodically sends a keep-alive message to each neighboring node such as the ping message in Gnutella-like overlay networks. If no response is received, the neighboring node is assumed dead, and a new link needs to be established. Failure detection is done in an off-line manner to avoid affecting search performance, but this may increase the overall traffic. In this approach, a node is not required to inform its neighboring nodes before its leaving. A node leaves the system by simply transferring its indices. In the above two approaches, when a node is involved in the back route of a subscription, it has to transfer its back route information to a node in its cluster or inform the subscriber about its leaving. Both the above two approaches have their pros and cons, which have to be evaluated carefully before applying to any real-life application. In the following evaluations, we rely only on backup links to study how well SCS performs in the presence of failure.

## 4.5 Performance evaluation and comparison

In this section, we use simulation to evaluate the effectiveness of SCS and compare SCS with SONs and Gnutella. We show the simulation results of setting various variables such as *m*, *n*, *M* and *shortcut* positions, and justify our choices. We first describe our simulation model and the performance metrics. Then we report the results obtained from a range of experiments.

### 4.5.1 Simulation model

To simulate SCS in a more realistic environment, we create two types of network topologies in our model: physical topology and P2P overlay topology. The physical topology represents a real-world Internet topology; the P2P overlay topology is built on top of the physical topology. All peer nodes are a subset of nodes in the physical topology. Previous studies have shown that both Internet physical topologies [69] and P2P overlay topologies [77] follow small world and power law properties. We use BRITE [78] to generate these topologies which are based on the AS model since it has both small world and power law properties.

We define the parameters used in our simulation as follows: $N$ is network size (i.e., the total number of nodes in the network); $M$ is maximum cluster size (i.e., maximum number of nodes in a cluster); $k$ ($k = m + n$) is the number of bits to represent a *Cluster ID*, where the first $m$-bit binary string is used to identify a *Semantic Cluster ID* , and the last $n$-bit binary string is used to identify a sub-cluster ID.

The simulation starts with having a pre-existing node in the network and then performing a series of join operations invoked by incoming nodes. A node joins a semantic cluster based on its local data and publishes its data indices. Various sets of RDF data are mapped into different semantic clusters and each semantic cluster is associated with a unique ID ranging from $0 \sim 2^m$. RDF data stored in each peer may be heterogeneous or homogeneous. To evaluate the capability of handling heterogeneous data in SCS, we introduce a parameter $\beta$, which is the ratio of the number of semantic clusters corresponding to all the local data stored in a node to the maximum number ($2^m$) of semantic clusters. $\beta$ falls into the range of $1/2^m$ to 1. When $\beta = 1/2^m$, it implies that a node has homogeneous RDF data in its local repository

which is exactly mapped to one particular semantic cluster in SCS. When $\beta = 1$, it implies that a node has heterogeneous RDF data which maps to all the semantic clusters; however, this case is unlikely to happen in real-life applications. In our experiments, we set $\beta$ to $1/2^m$, 0.25 and 0.5 respectively. The semantic cluster(s) are selected in random by each node according to $\beta$. A node also selects a random node in each of its minor semantic clusters to publish its indices if necessary. When a cluster exceeds the maximum size $M$, it is split into two. This operation may be performed recursively until the number of sub-clusters reaches $2^n$. When the network reaches a certain size, a mixture of node joining and leaving is invoked to simulate the dynamic characteristic of the overlay network. Each node is assigned with a query generation rate, which is the number of queries that it may generate per unit time. In our experiments, each node generates queries at a constant rate. If a node receives queries at a rate that exceeds its capacity to process them, the excess queries are queued in its buffer until the node is ready to read the queries from the buffer. Data are randomly replicated on nodes at a fraction $\alpha$. A query is selected randomly among different semantic clusters. When a node initiates a query, it is first mapped to a particular semantic cluster, then routed to the destination semantic cluster and flooded to all the sub-clusters in parallel. In our simulation study, we use a Gnutella overlay network to organize nodes within a sub-cluster. The average outgoing degree of a node in its sub-cluster is set to 4 by default, and *shortcuts* are set to half and quarter of the ring space unless otherwise specified. For the simplicity of generating RDF data in our simulation model, we use a set of keywords to represent RDF data triples; different sets of keywords correspond to different semantic clusters.

### 4.5.2 Performance metrics

In our simulation, we use the following performance metrics to measure the effectiveness of SCS:

*Fraction of nodes contacted per query* is the average fraction of nodes contacted for a query. It captures the efficiency of a lookup system. A smaller fraction of nodes implies less overhead in the network.

*Search path length* is the average number of hops traversed by a query to the destination.

*Search cost* is the average number of query messages incurred during a search operation in the network.

*Maintenance cost* is the average number of messages incurred as a result of a node joining or leaving the network. It includes the costs of node joining and leaving, cluster splitting and merging, and index publishing. We measure these costs in terms of number of messages.

*Routing load* is the average number of query messages that a node processes.

We present our simulation results in the following sections. For each experiment, we run the simulator 10 times. The average results of the 10 runs are presented.

### 4.5.3 Simulation results

#### *4.5.3.1 Search efficiency*

The efficiency of executing a search request is captured in the fraction of nodes contacted and the search path length in the search. In SCS, the nodes contacted per

query contain $N/2^m$ nodes and the nodes pointed to by a set of indices. Figure 4.7 plots the fraction of nodes contacted per query when $n$ is set to 0 (i.e., parallel search in a semantic cluster is disabled) and the number of semantic clusters is varied from $2^0$ to $2^8$. The values are obtained by taking the average over various network sizes.

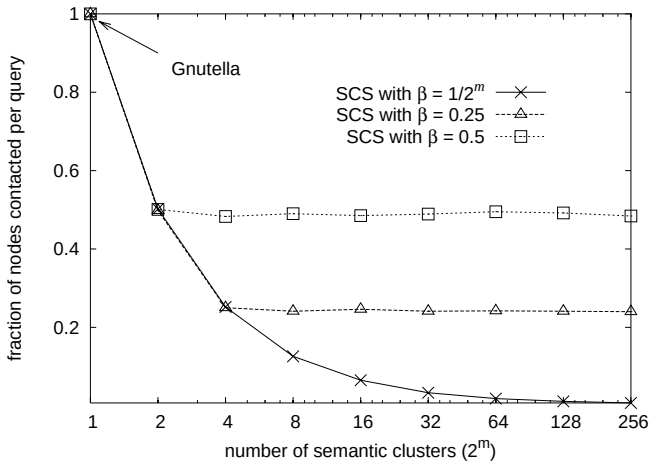



Figure 4.7: Fraction of nodes contacted per query

As expected, the fraction of nodes contacted per query decreases in proportion to $1/2^m$. For SCS with $\beta$ equals to 0.25 and 0.5, the number of nodes to be contacted is a quarter, and half of the nodes respectively. This is because besides contacting all the nodes in the destination semantic cluster, SCS has to contact the nodes in other semantic clusters pointed to by their indices. Due to the randomness of a peer's selection of semantic clusters and nodes to publish its indices, the fraction of nodes contacted is almost identical to $\beta$. Note that for a search request, Gnutella has to contact every node in the network. In the case of SONs, this fraction is equal to $\overline{c}/C_{max}$, where $\overline{c}$ is the average number of SONs each node participates in and $C_{max}$ is the maximum number of SONs in the system. SONs only contacts a fraction of the nodes depending on $\overline{c}$. The smaller the value of $\overline{c}$, the fewer the number of nodes

that are contacted for a query. With fewer nodes contacted by SCS and SONs, the network traffic load incurred by a query is also reduced.

Figure 4.8 compares the search path lengths of SCS, SONs and Gnutella when network size $N$ is varied from $2^8$ to $2^{13}$. We disable the clustering effect by setting $M$ to 1 for SCS since SONs and Gnutella do not have any clustering feature. We also disable parallel search within a semantic cluster by setting $n$ to 0. Hence, network size is $N = 2^{m-1}$. Since $M = 1$ and $n = 0$, there is no flooding within a semantic cluster. As shown in Figure 4.8, the search path lengths for both SCS and SONs increase slowly with network size when comparing to Gnutella, confirming that search path is bound. The search path length for SCS is almost identical to the one for SONs, showing SCS is of the same search effectiveness as SONs. In the case of a peer having heterogeneous local data (i.e., $\beta = 0.25$ or 0.5), the search path length is almost identical to the case of a peer having homogeneous local data (i.e., $\beta = 1/2^m$). This shows that homogeneous data in a peer does not have any negative effect on SCS in terms of search path length. This is because a peer can directly contact the node(s) in other semantic clusters using a set of indices that point to them.

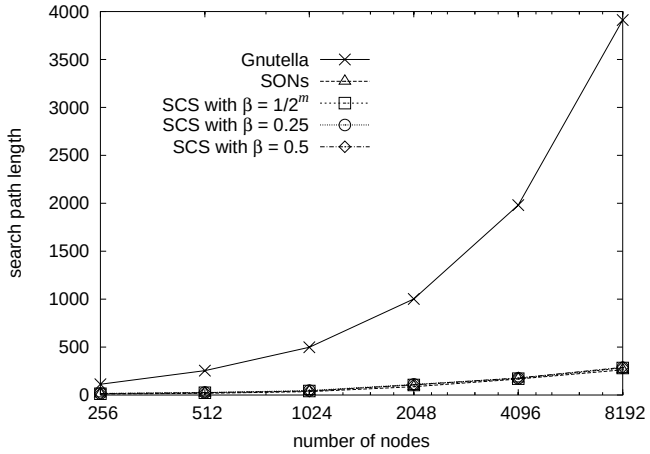



Figure 4.8: Search path length

In SCS, we explore the parallel search mechanism within a semantic cluster. We evaluate the parallel search effect by comparing SCS and SONs. We set up a network with $m$ = 4 and vary network size $N$ from $2^{10}$ to $2^{13}$. We set $n$ to 2 and 3 respectively for SCS; as a result, a semantic cluster is split into two when the size exceeds $N/2^5$ and $N/2^6$. Hence, a search can be performed in parallel among these sub-clusters. Figure 4.9 shows that the parallelism in SCS effectively reduces search path length in comparison with SONs. The result also shows that the parallel search effect increases (i.e., search path length decreases) with respect to $n$. The results in both Figure 4.8 and 4.9 show that search path length in SCS is sensitive to $2^m$ and $n$, but not to $\beta$.

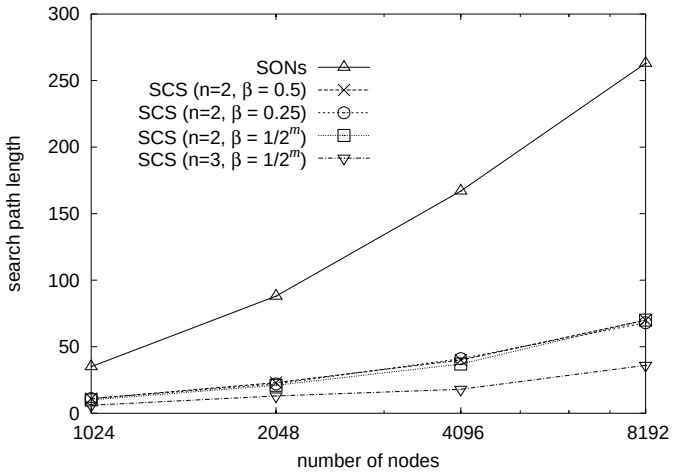



Figure 4.9: The effect of parallel search in SCS

*4.5.3.2 Overheads*

In this experiment, we evaluate search overhead by comparing search costs among SCS, SONs and Gnutella. We set $m$ to 5 (i.e., the number of semantic clusters is 32, as suggested by the result from Section 4.5.3.3), and $n$ to 0 (parallel search is disabled), and vary network size $N$ from $2^8$ to $2^{13}$. As shown in Figure 4.10, the search cost of Gnutella increases rapidly when network size grows. In contrast, SCS and SONs significantly reduce search cost with the setting of 32 semantic clusters. We repeat the

experiment by turning on the parallel search mechanism (i.e., $n$ = 2 and 3) while keeping other settings. We obtain results similar to those in the case where $n$ = 0. This confirms that the parallel search mechanism in SCS does not incur extra search overhead. When $\beta$ = 0.25 or 0.5, search cost increases because search requests have to reach nodes in other semantic clusters (other than $D$), which are pointed to by a set of indices.

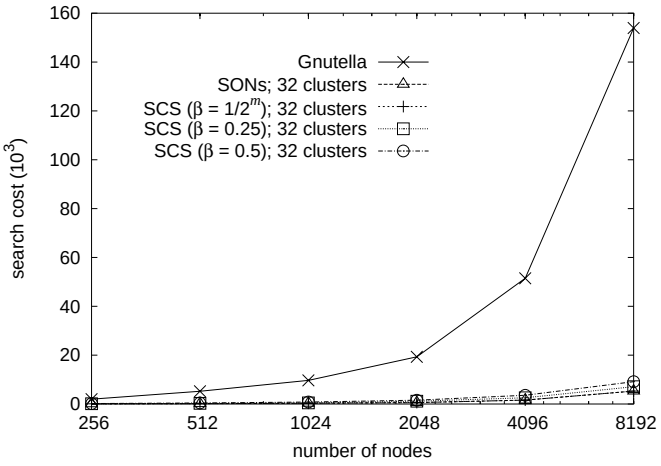



Figure 4.10: Search cost

We also evaluate and compare the maintenance cost of SCS and SONs in this experiment. The maintenance cost of SONs only contains the cost of nodes joining and leaving. As shown in Figure 4.11, the maintenance cost of SONs increases rapidly when the number of dimensions (i.e., semantic clusters) grows. This is because the required number of outgoing degrees for a node in SONs increases in proportion to dimension. In the case of SCS ($M$ = 32 and $n$ = 2 in this experiment), the maintenance cost of a node consists of the costs of node joining and leaving, cluster splitting and merging and index publishing. The maintenance cost in SCS also increases with respect to the dimension, but with a much slower rate. In the case of heterogeneous data stored in peers (i.e., $\beta$ = 0.25 or 0.5), maintenance cost increases with the rise in

index publishing cost; however, it is still much lower than that in SONs as shown in Figure 4.11. This confirms our design goal of reducing maintenance overhead incurred by using high-dimensional semantic overlay networks.

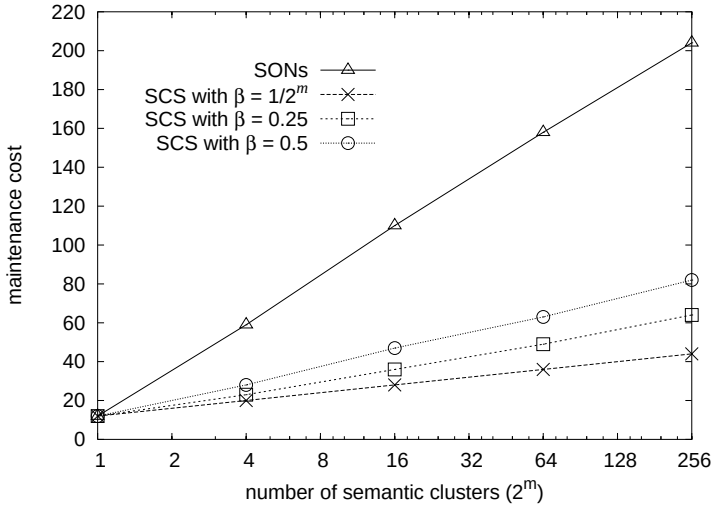



Figure 4.11: Maintenance cost

*4.5.3.3 Clustering effects*

In this section, we evaluate the effect of clustering in SCS by varying cluster size $M$ from $2^0$ to $2^{10}$. We first evaluate the effect of cluster size on search path length by setting a network of size $N$ to $2^{10}$. We turn off parallel search within a semantic cluster by setting $n$ to 0, and allow no data duplication in SCS. Hence all clusters are semantic clusters. We also set $\beta$ to $1/2^m$ as we focus on cluster operations in this section. Figure 4.12 plots the search path length in SCS when $M$ increases from $2^0$ to $2^{10}$. The search path length across clusters increases while the search path length within clusters decreases with larger cluster sizes (note that there are $2^{10}$ clusters in the network when $M = 1$ and only one cluster when $M = 2^{10}$). This is because with a fix network size, the total number of clusters in SCS decreases with larger cluster sizes. Figure 4.12 suggests that the search path length achieves its minimum when the

number of semantic cluster equals 32, 16 and 8, corresponding to $M$ = 32, 64 and 128 respectively.

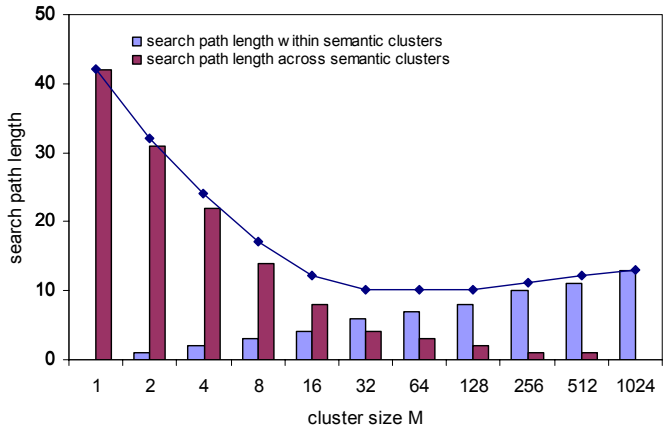



Figure 4.12: Search path length vs. cluster size $M$

With the same setting as in the previous experiment, we evaluate the search cost for within clusters and across clusters respectively. From Figure 4.13, we observe that search cost in SCS increases rapidly from a point where $M$ = 16. This is because of the effect of blind flooding within a cluster.

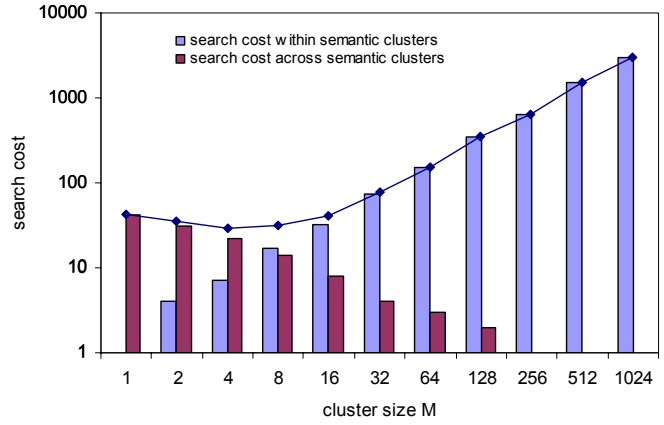



Figure 4.13: Search cost vs. cluster size $M$

We plot the cost of node joining/leaving and cluster splitting/merging over different cluster sizes in Figure 4.14. As there are fewer clusters in SCS with larger cluster sizes, a new node requires a smaller number of hops to join the network. Therefore, the cost of joining/leaving decreases with respect to *M*. Cluster splitting and merging also occur less frequently with larger cluster size, resulting in lower cluster splitting/merging cost.

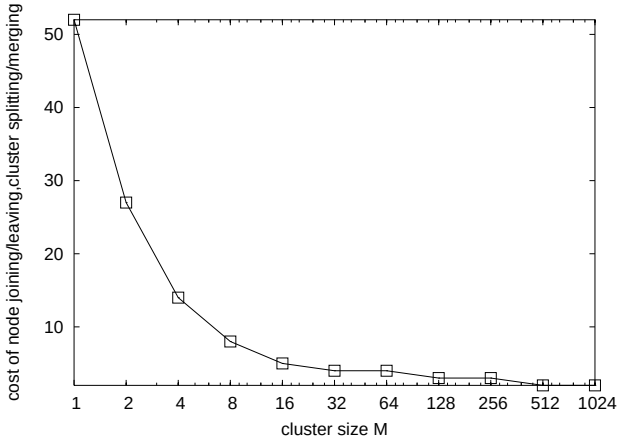



Figure 4.14: Costs of node joining/leaving and cluster splitting/merging vs. cluster size *M*

From the results in this section, we observe that the setting of 16 and 32 semantic clusters provides a good tradeoff between search efficiency and overhead. With larger cluster sizes, search path length and cost of node joining/leaving and cluster splitting/merging are not as sensitive to *M* as compared to search cost. Note that we set *n* to 0 in the experiments in this section. If the parallel search mechanism is turned on (i.e., $n > 0$), search path length can be further reduced as a query can be flooded in parallel in a semantic cluster. To further reduce the search cost incurred by blind flooding within a cluster in a Gnutella-like network, the Cost-Aware Selective Flooding technique (to be described in Chapter 5) can be used.

*4.5.3.4 Selection of shortcuts*

In this experiment, we evaluate the effect of different *shortcuts* in SCS and compare them to the random *shortcut* which is originally used in the small world network model. We started a network with the size of $2^{10}$ nodes. Each semantic cluster has only one node, with cluster size *M* set to 1 and *n* to 0. Hence, the search path length for intra-semantic cluster routing equals 0. We select two *shortcuts* –either fix-points or random-points in the network, and vary the location of the longest *shortcut*. The other *shortcut* always points to the middle semantic cluster between the semantic cluster where a node resides in and the semantic cluster that the longest *shortcut* points to. We plot the search path length for inter-semantic cluster routing with various semantic clusters in Figure 4.15. Compared to fix-point *shortcuts*, random *shortcuts* work well in low dimensional semantic context spaces, but perform worse in larger semantic context spaces. The location of fix-point *shortcuts* depends on the number of semantic context spaces. Among these *shortcuts*, the 1/8 *shortcut* seems to provide a balance for the size of semantic context spaces below 512.

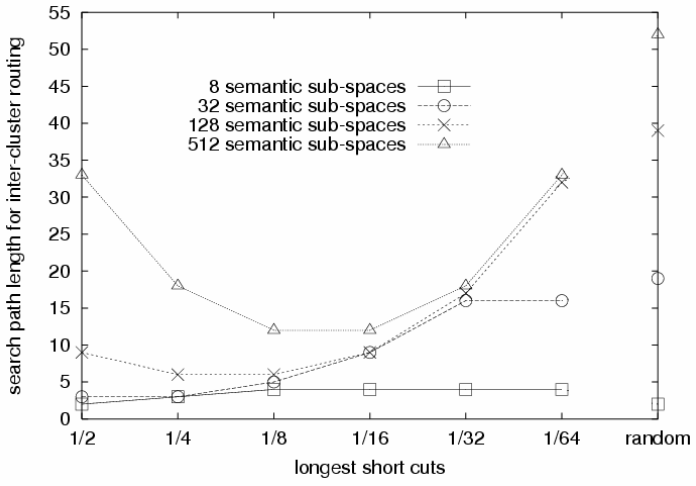



Figure 4.15: Shortcuts

*4.5.3.5 Load balancing*

We study load balance in SCS from the aspects of data load, index load and routing load. Since data load (i.e., in terms of number of context data triples) and index load (i.e., in terms of number of indices) are balanced under the uniform distribution of context data, we present only the result of routing load in this section. We evaluate the routing load per node in a network, setting $m$ to 3, $n$ to 2 and $M$ to 64. The average outgoing degree per node is set to 4 within a semantic cluster. A lookup query is drawn randomly among all the semantic clusters. Each node initials a lookup uniformly at random. Figure 4.16 shows that the routing load distribution across various nodes is relatively well balanced.

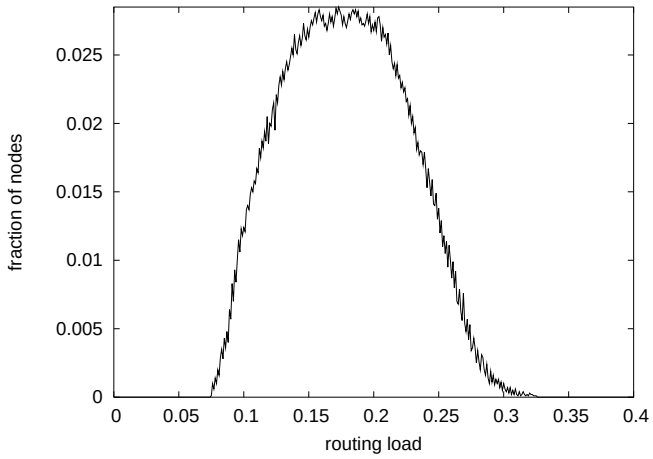



Figure 4.16: Routing load

## 4.6 Summary

We have presented our design of a semantic P2P context lookup system, with the design of various techniques such as an ontology-based semantic clustering scheme, a clustering naming scheme, parallel search, and push and pull operations, explained in detailed. Peers are self-organized into a one-dimensional ring space based on the

semantics of context data stored in each node. Cluster splitting and merging may be triggered when nodes join/leave the network or when data changes. We have also conducted extensive simulations to evaluate the performance of our proposed SCS system. The simulation results show that SCS offers good search performance and low overlay maintenance overhead as compared to SONs. Our system also exhibits good scalability and load balancing characteristics. In addition to our simulation, we have also developed the prototype system, and used the real measurement data to validate and calibrate our simulation model. We will describe the validation of our simulation model in Section 6.10.8.

In SCS, we group nodes with the same semantic together in a semantic cluster. A semantic cluster can be further split into multiple sub-clusters. In the design of a grouping technique, we should also consider to group nodes into a sub-cluster based on their proximity. If close-by nodes can be grouped together into a sub-cluster, query routing will be more efficient. To ensure the SCS overlay topology maps the underlying physical topology, further studies and evaluations need to be conducted and will be addressed in our future work.

In the current design of SCS, we assume using a Gnutella-like overlay to organize peers within a sub-cluster. In a Gnutella-like overlay network, a query is forwarded to all neighbors of a node. This blind flooding mechanism may generate a large amount of query messages in the network. In the next chapter, we will propose the Cost-Aware Selective Flooding algorithm to address this issue, and aim to provide a general solution to improving the scalability of *unstructured* P2P systems.

# CHAPTER 5

# COST-AWARE SELECTIVE FLOODING[4]

The blind flooding mechanism used in an *unstructured* P2P overlay network simply forwards a query to all the neighbors of a node. Such a mechanism may generate a large volume of unnecessary traffic in the network, and hence render the system not scalable. In recent years, many researchers have conducted studies on P2P traffic in the real world. For example, Ripeanu [70] analyzed the Gnutella network and showed that the blind flooding mechanism generates 330TB/month in a Gnutella network with 50,000 nodes and 36% of the total traffic is user-generated traffic (i.e., query messages). This is because the flooding-based routing mechanism generates a large amount of unnecessary traffic. It also incurs additional processing overhead at each node, and hence renders *unstructured* P2P systems far from scalable.

The causes for the problem are twofold. First, a query may be forwarded to multiple paths that are merged to the same node. As a result, a node may receive the same query multiple times. For example, as illustrated in Figure 5.1(a), D receives the same query three times as the query is forwarded along link $L_{AD}$, $L_{CD}$ and $L_{BD}$. In this case, only one of the paths is necessary, and messages generated along the other paths are redundant. Second, two neighboring nodes (nodes C and D in Figure 5.1(b)) may forward the same query message to each other if they have not received the query from each other before.

[4] The contents of this chapter have been presented in Paper 10 in the author's publication list. The extension of this chapter will be submitted to a journal.

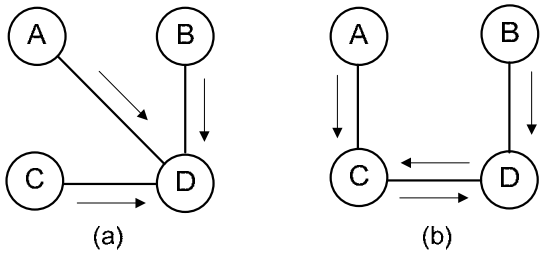


Figure 5.1: Unnecessary query messages in a Gnutella-like network

In the current design of SCS, we use a flooding-based routing mechanism to forward a query within a sub-cluster. As a consequence, excessive messages generated per query may render the system not scalable if we choose a larger cluster size. In this chapter, we propose the *Cost-Aware Selective Flooding* (CASF) technique that aims to reduce redundant query messages and improve system scalability. The basic principle of this technique is that, given a query, a node performs the CASF algorithm locally to compute and obtain a set of optimized paths using link cost information within its neighborhood to forward its query rather than forwarding the query to all its neighbors. CASF is performed by nodes in other relevant neighborhoods to determine a subset of forwarding paths in the whole overlay network. We define the neighborhood of a node, say $x$, as all its direct neighbors (1-hop neighbors of $x$) and the neighbors' neighbors (2-hop neighbors of $x$). CASF operates in three phases: neighborhood link cost measurement, basic routing algorithm, and routing decision mediation.

The rest of the chapter is organized as follows. We describe related work in Section 5.1. We then discuss the details of the CASF operations in Section 5.2, 5.3 and 5.4. We describe the main CASF algorithm in Section 5.5, and present a case study In Section 5.6. We present our simulation results in Section 5.7, and summarize the work in Section 5.8.

## 5.1 Related work

In this section, we survey and discuss previous work related to flooding in unstructured P2P overlay networks. There have been continuous efforts to improve flooding in Gnutella-like P2P systems.

In *Random Walks* [71], a node forwards its query message to *k* randomly chosen neighbors. The performance of this algorithm is highly variable. Success rates and hits vary greatly depending on network topology and the random choices made. In *Directed BFS* [72], a peer selects a subset of its neighbors to forward its query based on statistic information such as the neighbors that have returned the largest number of results received from previous queries. *Iterative Deepening* [72] or *Expanding Ring* [71] uses consecutive BFS searches at increasing depths, and works well when the search termination condition relates to a user-defined number of hits and it is possible that a small range of flooding will satisfy the query. In some cases, it may produce even bigger loads than the standard flooding mechanism. These flooding algorithms can reduce the total number of query messages compared to the standard flooding mechanism since queries are forwarded to a selected subset instead of all neighbors. However, the search may not be reliable as not all the nodes are covered by the query. Our CASF algorithm aims to reduce the message overhead of blind flooding directly while guaranteeing the search scope, especially in a dynamic network environment.

In super-peer systems such as Morpheus [73] and current Gnutella implementation, a super-peer acts as a centralized server to a subset of clients. It maintains the indices of its client peers and conducts searching and locating on behalf of its clients among super-peers. These super-peers connect to each other forming a Gnutella overlay network. With the expanding scale of the P2P systems, the inefficiency of flooding in

super-peer networks remains a grave concern. The cost of maintaining indices at a super-peer can be very high in a dynamic network. In *Local Indices* [74], each node indexes the files stored at all nodes within a certain radius *r* and can answer queries on behalf of all of them. The success rate is high since each node indexes many peers. However, message production in *Local Indices* is comparable to that of the flooding mechanism although the processing time is much smaller because not every node processes the query. The scheme also requires a flood in a radius *r* whenever a node joins/leaves the network or updates its local repository, and hence the overhead is potentially larger for dynamic environments. In *Routing Indices* (*RIs*) [75], a node forwards its query to a subset of its neighbors based on its local *RIs*. A *RI* is a data structure (and associated algorithms) that returns a list of neighbors for a given query, ranked according to their goodness for the query. The notion of goodness reflects the number of documents in nearby nodes. While *RIs* are bandwidth-efficient, they still require flooding in order to be created and updated; maintaining *RIs* could be very costly in highly dynamic networks. Moreover, stored indices can be inaccurate due to thematic correlations, over-counts or under-counts in context partitioning and network cycles.

Some researchers realized that topology mismatching is one of the key problems which cause excessive network traffic and limit search performance. For example, in LTM [67], each node detects and cuts most of the inefficient and redundant links, and creates new links to its closer neighbors. While this technique is efficient in reducing overall traffic and improving query performance, the cutting and creation of links for nodes in a global scale incurs a large amount of overhead. Such overhead may increase in a more dynamic network as LTM needs to be performed frequently to make the overlay network mirrors the physical network.

Multipoint Replaying [76] restricts the number of re-transmitters in wireless ad hoc networks as much as possible by efficiently selecting a small subset of neighbors which covers (in terms of 1-hop radio range) the same network region which the complete set of neighbors does. Only nodes chosed as forwarding neighbors (known as MPRs) rebroadcast the flooding message. The authors proposed a heuristic for the selection of MPRs. However, a node in a wireless ad hoc domain has a fix link pattern – a node always treats the nodes in its radio radius as its 1-hop neighbors and the link costs to all its 1-hop neighbors are identical, whereas a node in a wired network may have different link costs to its 1-hop neighbors. Hence, the algorithm for choosing MPRs does not apply directly to our case.

In summary, the CASF algorithm aims to reduce redundant query messages incurred by the blind flooding mechanism and improve system scalability for unstructured P2P overlay networks. CASF is a distributed algorithm; it only uses local information – link cost of 1-hop and 2-hop neighbors. While one-hop/two-hop neighbors and their link cost have been used in many other techniques such as in [67], as far as we know, no one has exploited topology patterns (i.e., *3-loop*, *4-loop* and *n-loop*) in a neighborhood, and use them to compute a set of optimized paths to forward queries. Many systems use flooding can easily adopt CASF to improve the scalability of their system.

## 5.2 Neighborhood link cost measurement

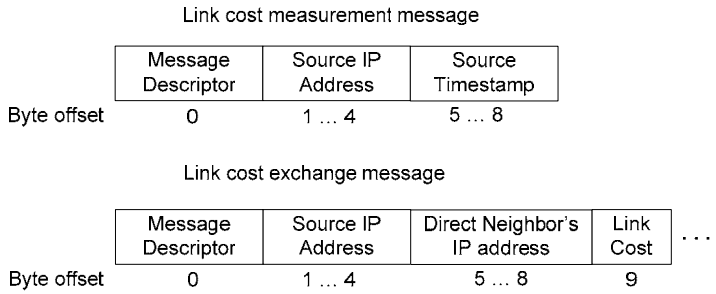



Figure 5.2: Link cost measurement and exchange messages

To measure and obtain the link costs of a node to all its direct neighbors, we define a *link cost measurement* message. The message format is shown in Figure 5.2. Each node in the system periodically sends this message to all its direct neighbors. This message contains the source node's IP address and the timestamp at the point this message is flooded. Upon receiving this message, a node can compute the link cost from the source node to itself by computing the difference between the source timestamp and the time it receives this message. This message is discarded by each node upon receipt.

To obtain the link costs of a node to all its neighbors' neighbors, we define a *link cost exchange* message. The message format is also shown in Figure 5.2. Each node periodically sends this message to all its direct neighbors. This message contains the source node's IP address, all its direct neighbors' IP addresses and the link costs of the source node to all its direct neighbors. When a node, say *x*, receives this message, it knows all the direct neighbors of the source node and their respective link cost information. Node *x* then stores the link costs to its direct neighbors, and also the link costs to its neighbors' neighbors in its routing table, as shown respectively in the first and the second column of Table 5.1.

TABLE 5.1: NODE *X*'S ROUTING TABLE

| **Direct Neighbor, *y*** | ***y*'s Neighbors** |
|---|---|
| <$nodeID_{y1}$, $cost_{y1}$> | <$nodeID_{y11}$, $cost_{y11}$>, <$nodeID_{y12}$, $cost_{y12}$>, … |
| <$nodeID_{y2}$, $cost_{y2}$> | <$nodeID_{y21}$, $cost_{y21}$>, <$nodeID_{y22}$, $cost_{y22}$>, … |
| … | … |

## 5.3 Basic routing algorithm

The P2P overlay topology within a node's neighborhood can vary in different ways. We recognize that there exist three fundamental cases: *3-loop*, *4-loop* and *n-loop*. We define a loop as a group of nodes linked together in a ring fashion. A loop may consist of three or more nodes, and may be closed or open. In particular, we call a closed loop with three or four nodes *3-loop* or *4-loop* respectively. We shall refer all other types of loops (containing five nodes or more, either closed or open) as *n-loop*. A node uses the CASF algorithm to compute and select a set of optimized forwarding paths with respect to these loops. CASF consists of the basic routing algorithm which computes the paths for one loop, and routing decision mediation which combines the decisions for all the loops. We now describe the basic routing algorithm for each type of loops.

The basic routing algorithm follows the least-cost principle, which means that it is desirable to forward a query along a set of least-cost paths. A source node, say *x*, is able to detect a *3-loop* if the two direct neighbors of *x* are direct neighbors of each other. For example, in Figure 5.3, node *x* detects a *3-loop* as B and C (both direct neighbors of *x*) are also direct neighbors of each other. For a *3-loop* with source node *x*, *x* computes and selects a set of optimized paths (shown as solid arrows for various cases in Figure 5.3) based on the link costs *d1*, *d2* and *d3*. The paths are selected

based on a minimum set of least-cost paths to ensure that a query reaches B and C quickly without redundant messages. For example, in Case 1 of Figure 5.3, a query is only forwarded along the links *xC* and *CB* as *d1* > *d2* + *d3*. This contrasts with the blind flooding mechanism where the same query message is flooded along the paths *xC*, *CB*, *Bx* and *xB* (shown as dotted arrows), resulting in the messages on *Bx* and *xB* being redundant. As for other cases in Figure 5.3, CASF avoids two redundant messages as well.

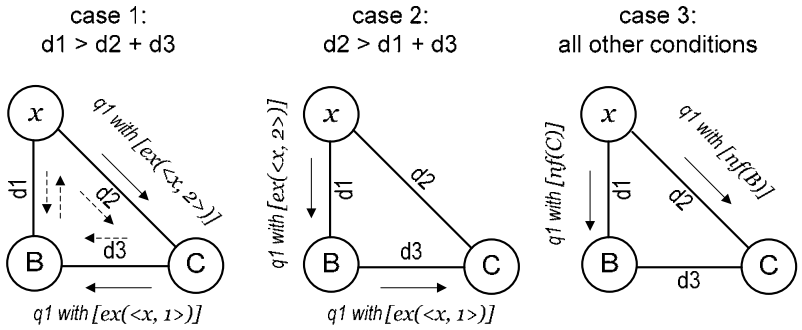



Figure 5.3: Optimized paths for a *3-loop* with source node *x*

To forward a query along the paths that are determined by CASF, we introduce two lists: *non-forwarding list* (*nf*) and *expected list* (*ex*). The *nf* list contains the node IDs of neighbors to which a query should not be forwarded (we call these neighbors *non-forwarding* nodes). The *ex* list contains the IDs of nodes (which may or may not be neighbors) that are expected to receive the query from another path (we call these neighbors expected nodes). Queries should also not be forwarded to nodes in the *ex* list. When receiving a query, each node executes the CASF algorithm and may add new entries into or delete existing entries from the *nf* and *ex* lists which are appended to the query message. The lists are subsequently retrieved by the receiver node and used for forwarding decisions. The lifetime of an entry in the *nf* list is always 1 hop, and hence this entry is purged upon use. Each entry in the *ex* list has a corresponding

TTL value which define the number of hops this entry can be used. An entry in the *ex* list will be purged either upon use or when its TTL equals zero.

For example, as shown in Case 3 of Figure 5.3, source node *x* computes and knows that both *xB* and *xC* are the optimized links to forward a query (i.e., *q1*) along. Node *x* forwards the list *nf(B)* to node C to inform C not to forward the query to B, and the list *nf(C)* to node B to inform B not to forward the query to C. For clarity, we only show the entries in the *nf* and *ex* lists which are added with respect to a loop with node *x*. The same reason applies in Figure 5.4 and Figure 5.5. In Case 1 of Figure 5.3, node *x* discovers that *xC* and *CB* are the optimized links and decides to forward a query to C only. Source node *x* adds itself to the *ex* list and set its TTL value to 2. When node C receives the query, it executes the algorithm and decreases the TTL value of *x* by 1. Then node C forwards the query with the list *ex(<x, 1>)* to node B. Node B does not forward the query back to *x* as *x* is the sender. When the query reaches node B, it decreases the TTL value of *x* to 0 and remove node *x* from the *ex* list. Hence, node B does not forward the query back to *x* along the path *Bx*.

A *4-loop* with source node *x* is detected if the two direct neighbors of *x* share a common direct neighbor. For example, in Figure 5.4, node *x* detects a *4-loop* as B and C (both direct neighbors of *x*) share a common direct neighbor D. The optimized paths for a *4-loop* with source node *x* and the appropriate *nf* and *ex* lists associated with the query (i.e., *q1*) for different cases are shown as solid arrows in Figure 5.4. In the case of equality for Cases 1 and 2, since both choices have the same effect, we arbitrarily fix the routing path to that in Case 1. The same reasoning applies in Cases 3 and 4. In all cases, two redundant messages are removed as compared to the blind

flooding mechanism. For example, in Case 1, the two redundant messages along the paths *BD* and *DB* (shown as dotted arrows) are removed.

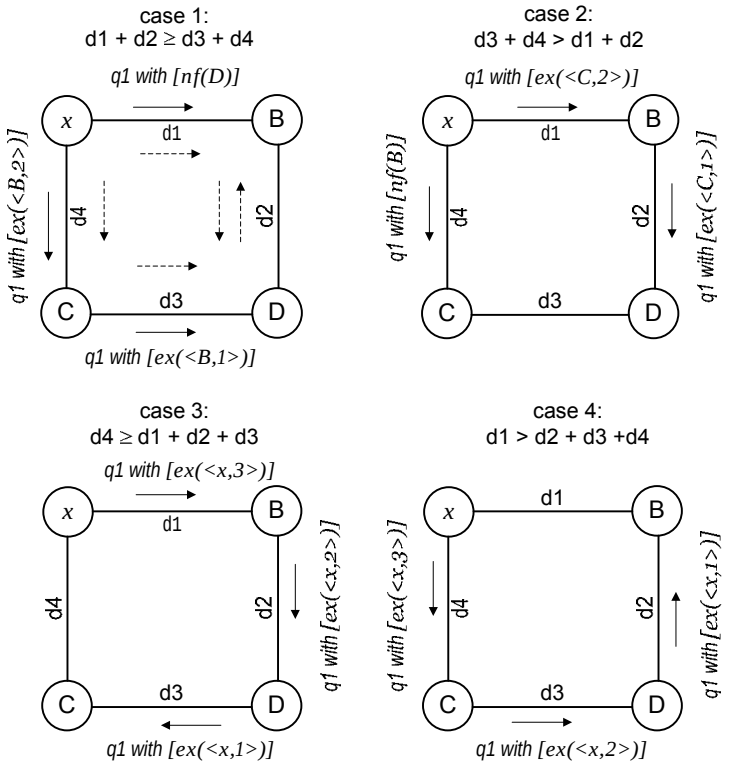



Figure 5.4: Optimized paths for a *4-loop* with source node *x*

The neighbors of node *x* for which no *3-loop* or *4-loop* is detected are considered to be part of an *n-loop*. Figure 5.5 shows an example of *n-loop* where $n = 5$. Node *x* detects an *n-loop* as B and C are neither direct neighbors of each other nor share a direct neighbor, and then forwards the query with the appropriate *ex* lists along both paths. As shown in Figure 5.5, two redundant messages along the paths *ED* and *DE* are removed. The same method applies in the case of $n = 6$. For an *n-loop* where $n \geq 7$ and there is no sub-loop within it, the algorithm is not able to remove redundant messages as we limit the scope of neighborhood information to two hops. However, this case seldom occurs in a P2P overlay network. As a result, the TTL value of the

entries in case of an *n-loop* is set to 4. We will give the justification of this choice in Section 5.4.

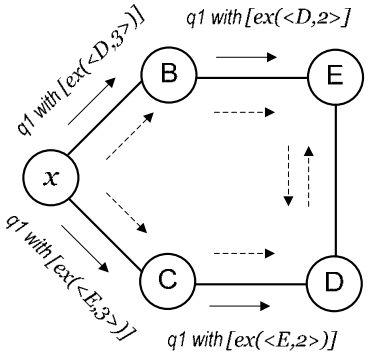



Figure 5.5: Optimized paths for an *n-loop* where $n = 5$

## 5.4 Routing decision mediation

In the Gnutella-like overlay topology, a node's neighborhood may contain a *3-loop*, a *4-loop*, an *n-loop* or any combination of these loops. A node, say *x*, will perform loop detection with respect to each of its neighbors. For each loop detected, the basic routing algorithm as we have described in the previous section is performed to obtain a sub-decision for that particular link. These sub-decisions is then mediated to a final forwarding decision, which would determine whether the query should be forwarded to a particular direct neighbor of node *x*.

In essence, the algorithm determines a subset of direct neighbors that node *x* should forward its query to. Routing decision mediation follows the two principles below:

- Node *x* forwards a query to a direct neighbor if and only if the computation for every loop detected with respect to that neighbor yields a positive forwarding sub-decision to that neighbor.

- If the computation for at least one loop concludes with a negative forwarding sub-decision to that neighbor, node *x* will not forward the query to that neighbor.

To understand the intent behind decision mediation, we note that a negative forwarding sub-decision signifies that there is a more favorable alternative path to the destination node than the current path being considered for a particular loop. On the other hand, a positive forwarding sub-decision indicates that the path being considered is the best possible path for a particular loop. Thus, a single negative sub-decision is sufficient to nullify the effect of all other sub-decisions (even if all of them are positive) as it is able to offer a better path than what the other loop computations suggest as the best. As the decision mediation is performed throughout the lifetime of a query flood, the query is thus flooded along a path that is almost optimized.

## 5.5 The main algorithm

The main CASF algorithm combines both the basic routing algorithm and routing decision mediation. We now present an overview of the main CASF algorithm which is shown in Figure 5.6. Let *N* represent the set of all direct neighbors of a node, say *x*. Upon receiving a query message *q*, node *x* starts to execute the CASF algorithm. First, it extracts the *nf* and *ex* lists appended to *q*. We denote these lists as the sender's *nf* and *ex* lists respectively. These lists will be used for making decisions. The *ex* list will also be forwarded to all neighbors later. Note that these lists are empty if *q* is initiated at node *x*. Node *x* also decreases the TTL values of all entries in the sender's *ex* list by 1. Node *x* than performs loop computation with respect to each of its direct neighbors such as *n*, provided *n* is not in the sender's *nf* or *ex* lists. If *n* is found to be in the sender's *nf* or *ex* lists, its entry is purged. The loop computation involves detecting a particular loop, updating the *nf* and *ex* lists for *n*, and setting the variable

*forwarding_decision* which represents the forwarding decision along the link from *x* to *n*. If *x*'s sender is a member of a particular loop and the link from the sender to *x* is part of that loop, no loop computation is necessary. This is because the result of the loop computation done previously by the sender has determined the forwarding decision for that loop. Note that *forwarding_decision* is a variable shared by all the loop computations with respect to *n*. It is set to *true* if all the loops desire to forward *q* along the link from *x* to *n*, or *false* when at least one loop does not desire to do so. This is where routing decision mediation takes place. Finally, the loop computation for each direct neighbor *n* of *x* concludes with a forwarding decision that determines whether *q* should be forwarded to *n*. All direct neighbors that node *x* decides to forward to are placed in a list known as the *forwarding_list* with their respective *nf* and *ex* lists.

After loop computation has been done for all direct neighbors of node *x*, node *x* begins post-processing which consists of the combination of *nf* and *ex* lists and the cleaning up of the *ex* list. For those direct neighbors of *x* in the *forwarding_list*, node *x* combines their *ex* and *nf* lists with those of the sender. Subsequently all the entries with zero TTL values in the *ex* list are purged. Those *n-loop* entries of node *n* which are not in the *forwarding_list* are also removed since they will not be used. When all the entries in the *forwarding_list* are finalized, the query message is forwarded to all direct neighbors of node *x* in the *forwarding_list* together with their corresponding *nf* and *ex* lists.

```
node x obtains sender's nf and ex from the query message q;
decrease TTL of all entries in sender's ex by 1;

/*node x performs computation for each direct neighbor n */
for each n ∈ N do
  if (n ∈ sender's nf) then purge n from sender's nf;
  else if (n ∈ sender's ex) then purge n from sender's ex;
  else
    forwarding_decision ← true; //the forwarding decision from x to n
    for each direct neighbor m of n do
      perform loop detection;
      update nf and ex lists for n based on link-cost;
      update forwarding_decision;
      if (forwarding_decision == false) then
        break;
    endfor
    if (forwarding_decision == true) then
      add n, nf, ex to forwarding_list; //keep nf and ex for each n
    endif
  endif
endfor

/* combination of nf and ex lists */
combine n's nf and ex with the sender's nf and ex;

/* cleanup of ex */
purge all entries in sender's ex with TTL = 0;
if n is not in forwarding_list then
  remove all n-loop entries of n from all ex in forwarding_list;
endif

/* forward */
forward q to all direct neighbors in forwarding_list;
End
```

Figure 5.6: Main CASF algorithm

CASF is a distributed algorithm; it does not require a global view of the entire network. Instead, each node has a limited view of the network within the scope of its neighborhood. This gives rise to the possibility of redundancy when *n-loop*s with 7 nodes or more (with no sub-loops within) exist, as a result of limited neighborhood information. In order to reduce this possibility, the scope of neighborhood information needs to be increased. However, this would incur greater computation complexity and storage overhead at each peer. Therefore, there is a tradeoff between the scope of neighborhood information and the degree of redundancy reduction. An overlay topology with a large number of *n-loop*s which have no smaller sub-loops contained within them can only exist when many nodes have only two neighbors each (i.e., node

degree = 2). However, the study in [77] showed that P2P overlay topologies follow the small world property of having large clustering coefficients (i.e., average of fraction of edges connecting neighbors of a node) and short average path lengths between two nodes. In fact, based on their observations in the study, the node degree of a real Gnutella overlay is approximately between 4 and 20. Hence, a large number of *n-loop*s without smaller sub-loops contained within them is unlikely to happen. Our simulation studies show that maintaining neighborhood information within a scope of two hops provides a good tradeoff between overhead cost and performance, owing to the fact that a large number of *n-loop*s in a Gnutella-like topology is rare.

## 5.6 A case study

In this section, we concretize the discussion of the CASF algorithm with an example topology consisting of 25 nodes, 45 links and their associated link costs as shown in Figure 5.7. This topology, which is adapted from [67], shows a typical Gnutella-like P2P overlay network. The link costs in the overlay are randomly assigned. We will demonstrate how a node executes the CASF algorithm and obtains a set of optimized links to forward a query with the example. We use *l(a,b)* to denote a link from node *a* to node *b*, and use *3-loop(a,b,c)* to denote a *3-loop* that involves node *a*, *b* and *c*. The query originates from node 1. We illustrate this example based on the execution timeline.

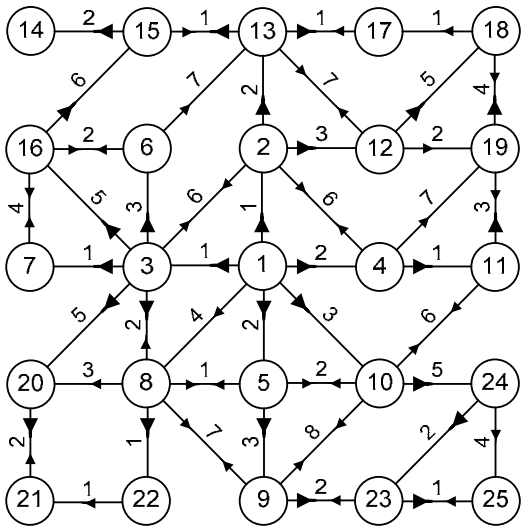



Figure 5.7: A case study

The overlay network is first constructed by a sequence of node joining operations. Upon joining the network, each node periodically sends link cost measurement and exchange messages to all its direct neighbors. Each node then creates and updates its routing table which contains its 2-hops neighbors and their associated link costs (shown aside each link in Figure 5.7). Node 1 starts to execute CASF by analyzing each of its direct neighbors. For example, first we consider link *l(1,2)* to node 2. Based on its routing table, node 1 detects *3-loop(1,2,4)*, *3-loop(1,3,2)* and *4-loop(1,8,3,2)* with respect to node 2. For each loop detected, the loop computation is performed to update its *nf* and *ex* lists and the sub-decision according to the basic routing algorithm described in Section 5.2. Note that the sender's *nf* and *ex* lists are empty as the query initiates at node 1. For example, if we consider *3-loop(1,2,4)*, node 4 is added to the *nf* list as *nf(4)*. In addition, node 1 detects several *n-loop*s, for example, *n-loop(1,3,6,13,2)*, and adds *<6,3>* to the ex list as *ex(<6,3>)*. For each of the *n-loop*s, the ex list is updated as necessary. As the computation for all the loops

concludes a positive forwarding decision to node 2, node 2 is added to the *forwarding_list* with the associated *nf(3,4)* and *ex(<6,3>,<19,3>...)* lists.

We now consider the link *l(1,8)* with respect to node 8. As the loop computation for *3-loop(1,8,3)* yields a negative forwarding sub-decision, node 1 does not forward the query to node 8. This is because one negative sub-decision is sufficient to negate the final forwarding decision based on the principle of routing decision mediation. The same process is repeated for all other direct neighbors of node 1. Finally, node 1 forwards the query to all the entries in the *forwarding_list*.

When node 2 receives the query, it first extracts the sender's *nf* and *ex* lists and then performs the loop computation for those direct neighbors which are not in the sender's *nf* and *ex* lists. In this case, node 2 computes for node 12 and 13 only as node 3 and 4 are in the sender's *nf* list. As a result, node 2 does not forward the query to node 3 and 4. Considering link *l(2,12)* with respect to node 12, node 2 detects *3-loop(2,13,12)* and *4-loop(2,12,19,4)*. As both loop computations conclude the positive forward decision to node 12, node 12 is added to the *forwarding_list* with *nf(13)* and combined *ex(<6,2>,<19,2>...)*. Note that entries 3 and 4 are removed from the *nf* list upon use and entries in the *ex* list are decreased by one. The computation for the rest of the nodes is similar to what we have discussed above. The final outcome following the execution of CASF by all the nodes is shown in Figure 5.7. Large arrows indicate forwarding decisions common to both the blind flooding mechanism and the CASF algorithm while small arrows indicate forwarding decisions pertaining to the blind flooding mechanism alone.

## 5.7 Performance evaluation

In this section, we use simulations to evaluate the effectiveness of CASF, and compare its performance to blind flooding in a Gnutella overlay network. We first describe our simulation model and the metrics. Then we report the simulation results from a range of experiments.

### 5.7.1 Simulation model and metrics

In our simulation, we assign degrees to nodes based on the power-law distribution as the study in [77] has shown that Gnutella networks follow the power-law property. We have two types of network topologies in our model: physical topology and P2P overlay topology. The physical topology represents the real-world Internet topology. The P2P overlay topology is built on top of the physical topology. The link cost between two nodes in the overlay is calculated based on the shortest physical path between these two nodes.

Each node $x$ is also assigned a query generation rate, which is the number of queries that node $x$ generates per unit time. In our experiments, each node generates queries at a constant rate. If a node receives queries at a rate that exceeds its capacity to process them, the excess queries are queued in its buffer until the node is ready to read the queries from the buffer. Queries are modeled as searches for different keywords stored randomly at each node. Keywords are randomly replicated on nodes at a fraction $\alpha$. Thus, querying for a keyword with fraction $\alpha$ implies that a query hit can be found at a fraction $\alpha$ of all the nodes in the system.

To measure the effectiveness of the CASF algorithm, we use the following performance metrics:

*Number of messages per query*: the number of query messages generated when executing a lookup request in the network. We aim to minimize the number of messages forwarded by each node while ensuring search completeness.

*Search completeness*: the ratio of the number of nodes contacted per query to the total number of nodes in the network. The value of this metric lies in the range 0 to 1.

*Bandwidth consumption*: the total bandwidth consumed in terms of bytes per second.

### 5.7.2 Simulation results

The goal of the CASF mechanism is to reduce redundant query messages as much as possible. In this experiment, we evaluate CASF by issuing a complete search request to a Gnutella overlay network. We compare the number of query messages incurred by the Gnutella protocol and by CASF and present the result based on the overlay network of 4000 nodes in Figure 5.8. The comparison shows that CASF reduces the average number of query messages significantly by about 60% as compared to blind flooding in the Gnutella protocol.

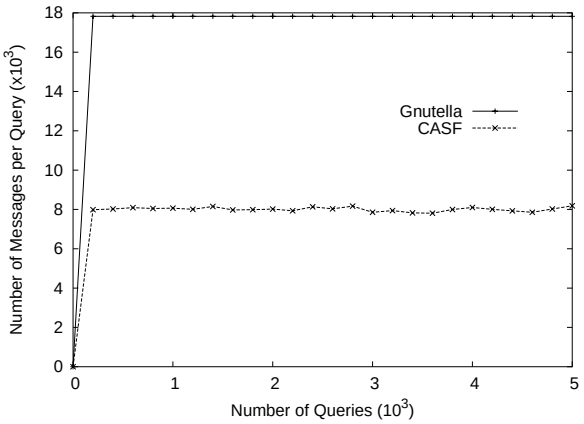



Figure 5.8: Effectiveness of the CASF algorithm

The search completeness when using CASF equals 1. This result verifies that CASF can guarantee every node in the overlay receives the query. The query response time when CASF is used is almost the same as the time taken when the standard Gnutella protocol is used. These results show that the CASF algorithm can significantly reduce redundant query messages without undermining search performance.

In this experiment, we evaluate all the possible overheads of the CASF algorithm. The overhead of CASF falls into two aspects: processing overhead and traffic overhead. The processing overhead is trivial as compared to the Gnutella protocol. Hence, we present the result of traffic overhead only. CASF uses two additional messages and adds *nf* and *ex* lists to the original Gnutella query message. In the experiment, we measure the network traffic incurred by Gnutella and CASF respectively in terms of number of bytes generated per second. We set the number of neighbors to 4. Figure 5.9 shows that CASF consumes less bandwidth compared to Gnutella. The two messages we created only generates about 3.5% of the total traffic. The bandwidth consumed by query messages is also reduced as expected. These results show that the additional overhead introduced by CASF only constitutes a small percentage of total network traffic.

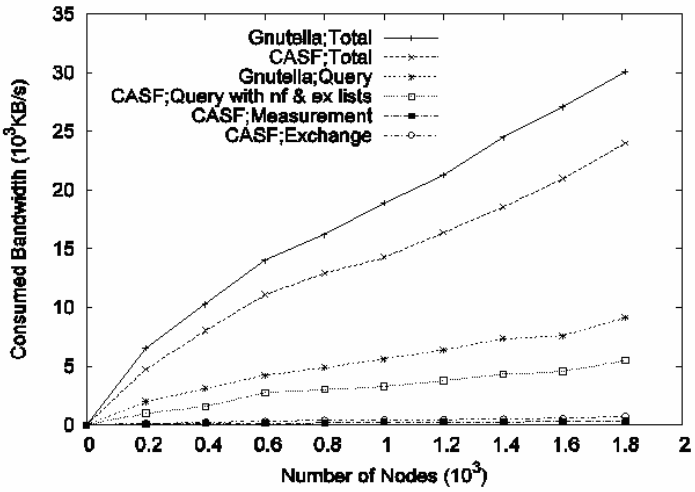



Figure 5.9: Bandwidth consumption

## 5.8 Summary

In this chapter, we have proposed the CASF algorithm to reduce redundant query messages incurred by the blind flooding mechanism in an *unstructured* P2P overlay network. The simulation results show that CASF significantly reduces redundant query messages while ensuring search completeness. The processing overhead is trivial and the traffic overhead is very low as compared to the Gnutella protocol. Note that while we intend to use CASF to forward a query in sub-clusters in the current design of SCS, CASF can be applied to any flooding-based P2P system. Hence, the CASF algorithm provides a potential solution to improving the scalability of *unstructured* P2P systems. In the next chapter, we will present our design of the SCS prototype system and the evaluation results based on our testbed.

# CHAPTER 6

# PROTOTYPE IMPLEMENTATION[5]

In this chapter, we present our design and implementation of various techniques proposed in SCS. The objective of this prototype is to demonstrate the working principle of SCS in a real-world setting and provide a basis for assessing practical issues. The prototype also enables us to conduct performance experiments aiming to validate and calibrate our simulation model. To demonstrate how application developers could benefit from the infrastructure services of SCS, we also present several typical context-aware applications based on a set of APIs implemented in our prototype system. The details of the development process are presented in this chapter as well.

The presentation of this chapter is outlined as follows. Section 6.1 provides an overview of the SCS prototype. Section 6.2 presents the bootstrapping implementation including semantic clustering, web cache, connection and reference registration. Section 6.3 describes the implementation of message receivers. We describe message forwarding and processing in Section 6.4, followed by the search and subscription services in Section 6.5. Next, we present the implementations for the LookupClient, the Context Producer and the Context Interpreter. In Section 6.9, we illustrate the application development process with several typical context-aware applications. We present the results of performance measurement of the prototype in Section 6.10.

[5] The contents of this chapter have been presented in Paper 13 in the author's publication list. The extension of Paper 13 has been submitted to a journal.

## 6.1 Overview

The prototype system implements the core system components realizing various techniques presented in Chapter 3 and Chapter 4. These include the context model, ontology-based semantic clustering, the construction of one-dimensional ring space, cluster splitting and merging, query routing, push and pull services, and context reasoning. A peer (also known as ContextPeer) can act as context producer, context consumer, or both in SCS. There are typically three types of ContextPeers in our system:

*Context LookupClient:* A LookupClient obtains context data from the network by issuing queries to a context producer. The context producer hence acts as a proxy to resolve queries for the LookupClient. The LookupClient is not required to participate in the SCS overlay network since it does not provide any context data.

*Context Producer*: Context producers usually contain context data, and hence they participate in the SCS overlay network. They usually provide low-level context data which is obtained from physical and software sensors. The functionalities of a context producer includes converting raw sensor data into RDF triples, managing the context data and the context ontologies it stores, managing subscription queries and subscribers, routing and responding to context queries from LookupClients. Queries can be generated at context producers as well.

*Context Interpreter*: A context interpreter is a special type of context producer. It has all the functionalities of a context producer. In addition, a context interpreter has the capability to derive high-level context data from low-level context data by using a

built-in logical reasoning engine and a set of user-defined rules. The reasoning engine is implemented using Jena2 – HP's Semantic Web Toolkit [62].

The SCS prototype system is implemented in Java using SDK 1.4.1. In the following sections, we present the implementation details of our prototype system.

## 6.2 Bootstrapping

When a ContextPeer starts, it first goes through the bootstrapping process which consists of a series of operations, i.e., semantic cluster mapping for local context data, obtaining an existing peer in the system by SWebCache operations and initiating connections to other peers in SCS. The class diagram of the bootstrapping process (shown in Figure 6.1) illustrates the various classes and their relationships.

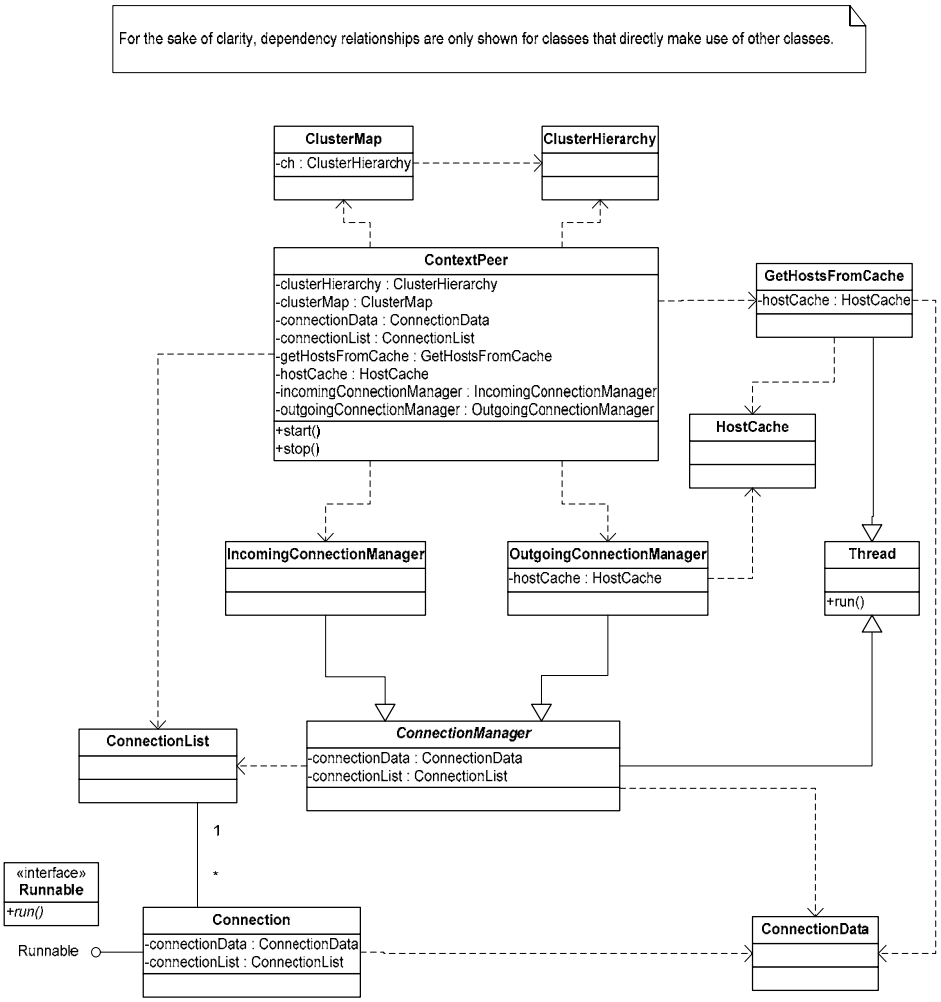



Figure 6.1: Classes responsible for connecting to the SCS network

### 6.2.1 Semantic cluster mapping

The semantic clusters in SCS are predefined in the upper context ontology. We have shown an example of the upper ontology in Figure 3.4 in Section 3.3. The leaf nodes in the upper ontology are used as semantic clusters to group ContextPeers. Each of these pre-defined semantic clusters is assigned a unique ID upon its presence in the SCS overlay network.

To determine which semantic cluster a ContextPeer should join, we create two structures, known as *ClusterHierarchy* and *ClusterMap*. *ClusterHierarchy* maps

OWL classes to their associated semantic clusters. It does this by tracing the hierarchy of classes as defined in the upper ontology. *ClusterMap* maps predicates defined in the upper ontology to their associated OWL classes. If a predicate is a *DatatypeProperty* (i.e., it describes the subject), it is mapped to the OWL class of its domain. On the other hand, if a predicate is an *ObjectProperty* (i.e., it describes the object), it is mapped to the OWL class of its range.

*ClusterHierarchy* and *ClusterMap* are created with respect to the context data provided by the ContextPeer. Subsequently, the predicates associated with the context data are mapped to OWL classes using the *ClusterMap*, and are further mapped to the appropriate semantic cluster(s) using *ClusterHierarchy*. For the semantic mapping of context data, the method *getPeerClusters* is invoked with an ontological model as input argument. This method creates and maintains a *HashMap* called *clusterTripleCount*, which maps a semantic cluster to the number of triples corresponding to the semantic cluster by iterating through all triples in the model. Upon successful execution, the method returns a vector containing all the semantic cluster IDs corresponding to all the context data in the input model. The first element in this vector indicates the ID of the semantic cluster corresponding to the majority of the context data, which we call the major semantic cluster; the rest of the semantic clusters are known as minor semantic clusters. A ContextPeer should join the major semantic cluster in SCS and create references to all its minor semantic clusters.

Context queries follow a similar mapping process to determine the corresponding semantic cluster by invoking the method *getTripleClusters*. This method returns all the semantic cluster(s) associated with the query triple. Context producers and interpreters require the semantic cluster mapping for both context data and queries

whereas LookupClients only require the mapping for context queries as they do not provide any context data.

### 6.2.2 SWebCache

When a ContextPeer joins its major semantic cluster, it first contacts a known bootstrap server which stores and maintains information about existing peers in SCS. We call this server an SCS Web Cache or SWebCache for short. In our prototype, we use one bootstrap server for simplicity reason, however, multiple bootstrap servers may exist; and each ContextPeer is required to know at least one of them to join the SCS. A ContextPeer obtains information of an existing ContextPeer by issuing a *hostfile* request to an SWebCache. The SWebCache responds to the request with the IP address of an existing ContextPeers and its related information. The ContextPeer then parses the response and stores it in its local *HostCache*. The *HostCache* is a data structure that holds information about currently connected ContextPeers. All SWebCache operations are handled by the *GetHostsFromCache* instance.

An SWebCache supports main operations such as *hostfile* request, *urlfile* request and SWebCache updating.

*Hostfile request:* This operation returns a ContextPeer randomly that is currently participating in the SCS overlay network with their respective information. Information for each ContextPeer is represented as a 3-tuple of the form (<IPAddress>, <Port>, <SID>). <IPAddress> represents the IP address of the ContextPeer, <Port> is the port at which the ContextPeer listens to for incoming connections, and <SID> is a string containing the semantic cluster ID the

ContextPeer is currently part of. A *hostfile* request is invoked with the following URL request string, where <swebcacheURL> is the URL of the SWebCache:

*<swebcacheURL>?hostfile=1*

An SWebCache responds to a *hostfile* request by returning a 3-tuple resembling *<IPAddress>:<Port>,<SID>*.

*Urlfile request:* This operation returns a list of other known SWebCache URLs. An SWebCache responds to a *urlfile* request by returning a list of known SWebCache URLs. A *urlfile* request is invoked with the following URL request string:

*<swebcacheURL>?urlfile=1*

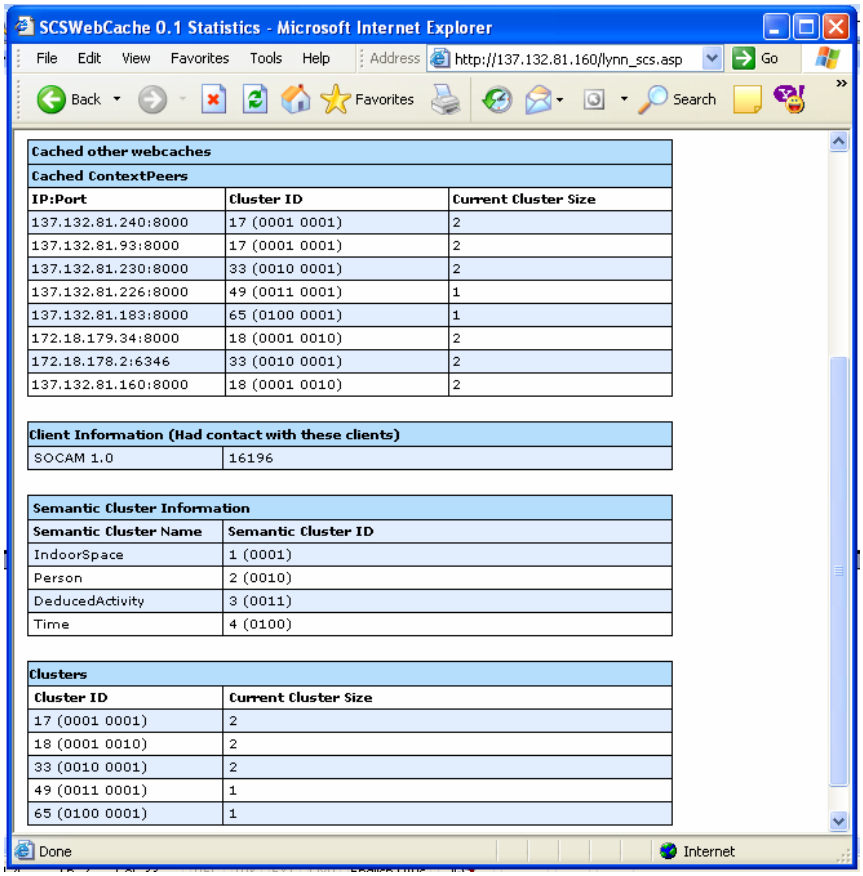
*SWebCache updating:* Upon successfully joining the overlay, each ContextPeer can perform an SWebCache update to add information about itself to the SWebCache. This is done by using the following URL request string:

*<swebcacheURL>?client=SCS&version=1.0&ip=<IPAddress>*
*:<Port>&sid=<SID>*

If the update is successful, the SWebCache responds with either "OK" or "Host updated", depending on whether the SWebCache has previously known this ContextPeer.

An SWebCache also collects and displays statistical information related to the current state of the SCS overlay such as the IP addresses and port numbers of existing ContextPeers, Cluster IDs, the semantic cluster(s) and associated IDs, and the number of peers in a semantic cluster. Each peer is required to update an SWebCache of its statistical information when it joins the SCS overlay network. This process, however,

is only used for our evaluations and experiments; it is not required in a real SCS system. The screen shot of an SWebCache is shown in Figure 6.2.




**Cached other webcaches**

**Cached ContextPeers**

| IP:Port | Cluster ID | Current Cluster Size |
|---|---|---|
| 137.132.81.240:8000 | 17 (0001 0001) | 2 |
| 137.132.81.93:8000 | 17 (0001 0001) | 2 |
| 137.132.81.230:8000 | 33 (0010 0001) | 2 |
| 137.132.81.226:8000 | 49 (0011 0001) | 1 |
| 137.132.81.183:8000 | 65 (0100 0001) | 1 |
| 172.18.179.34:8000 | 18 (0001 0010) | 2 |
| 172.18.178.2:6346 | 33 (0010 0001) | 2 |
| 137.132.81.160:8000 | 18 (0001 0010) | 2 |

**Client Information (Had contact with these clients)**

| | |
|---|---|
| SOCAM 1.0 | 16196 |

**Semantic Cluster Information**

| Semantic Cluster Name | Semantic Cluster ID |
|---|---|
| IndoorSpace | 1 (0001) |
| Person | 2 (0010) |
| DeducedActivity | 3 (0011) |
| Time | 4 (0100) |

**Clusters**

| Cluster ID | Current Cluster Size |
|---|---|
| 17 (0001 0001) | 2 |
| 18 (0001 0010) | 2 |
| 33 (0010 0001) | 2 |
| 49 (0011 0001) | 1 |
| 65 (0100 0001) | 1 |

Figure 6.2: Screen shot of an SWebCache

### 6.2.3 Connection

Upon obtaining an existing node in SCS, the ContextPeer that wishes to join the overlay network initiates a connection to the existing node and then send a *Join* message to this node. The format of the *Join* message is shown in Figure 6.3.

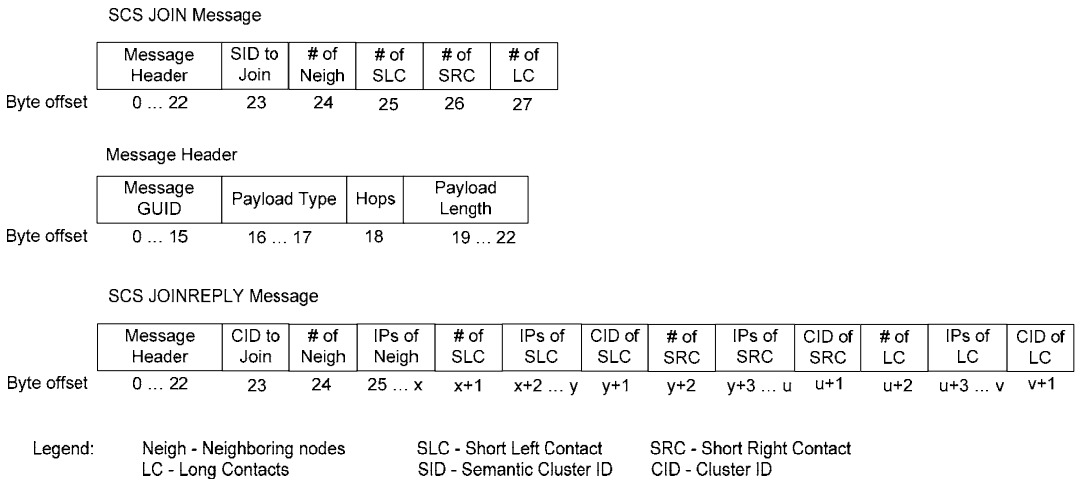



Figure 6.3: Structures of *Join* and *JoinReply* messages

The existing node is responsible for routing the *Join* message to a random node say *z* in the semantic cluster which the ContextPeer wishes to join. Then node *z* returns a *JoinReply* message (as illustrated in Figure 6.3) with information of a set of nodes: one or more neighboring node(s) in the cluster the ContextPeer joins, one neighboring node in the left adjacent cluster (short left contact), one neighboring node in the right adjacent cluster (short right contact) and one or more nodes in other semantic clusters (long contacts). These information are obtained through operations implemented as methods *getNeighbors*, *getShortLeftContact*, *getShortRightContact* and *getLongContacts*. Node *z* invokes each of these methods and sends the requests (*RandomNeighs*, *RandomShortLeftContact*, *RandomShortRightContact*, *RandomLongContact*) through its neighbor, the short left/right contact and the long contact. Upon successful execution, a set of nodes are obtained and returned to the joining ContextPeer.

Messages in SCS, e.g., *JoinMessage*, are implemented as instances of subclasses of the abstract *Message* superclass. The *Message* superclass encapsulates the fields that are common to all messages, namely the message GUID, payload type, hops and

payload length in the message header. It also stores the payload and a reference to the *Connection* from which the message originated. The classes implementing the different messages such as *JoinMessage*, *JoinReplyMessage, SearchMessage, SearchReplyMessage, RandomNeighsMessage*, *RandomShortLeftContactMessage*, *RandomShortRightContactMessage*, *RandomLongContactMessage and RandomReferenceHostsMessage* extend *Message* by defining different payload structures. They also provide methods for flattening a message to render it suitable for sending over the network, and for expanding a message after it is received from the network.

When a ContextPeer receives a *JoinReply* message, it obtains a set of nodes to connect to and its *OutgoingConnectionManager* starts to initiate outgoing connections to these nodes. For each connection attempt, the *OutgoingConnectionManager* sends the connect string:

```
SCS CONNECT\r\n
```

This connect string is followed by other headers indicating the local ContextPeer's IP address, port and SID.

When a remote ContextPeer receives a connection request via its *IncomingConnectionManager*, it verifies whether it falls into the maximum number of connections. If the remote ContextPeer accepts the connection request, it responds with:

```
SCS CONNECT OK\r\n
```

A connection is then established between the local and remote ContextPeers. Information for each such successful connection is stored in a *Connection* object. The

flags indicating various types of connections are listed in Table 6.1 (note that reference connection which is indicated with the flag `CONN_REFERENCE` need not to be created during the bootstrap). The structure known as the *ConnectionList* stores all the active *Connection*s.

TABLE 6.1: VARIOUS CONNECTION FLAGS

| **Flags** | **value** |
|---|---|
| CONN_NEIGHBOR | 0 |
| CONN_SHORT_LEFT_CONTACT | 1 |
| CONN_SHORT_RIGHT_CONTACT | 2 |
| CONN_LONG_CONTACT | 3 |
| CONN_REFERENCE | 4 |

There are also similar flags to indicate incoming connections or outgoing connections. An incoming connection and an outgoing connection may share the same *Connection* object if the two end hosts are the same. The number of incoming and outgoing connections for each ContextPeer may be specified in accordance with its capacity. Upon successfully joining the SCS overlay network, a ContextPeer keeps these connection objects in its routing table called *HostCache*. All ContextPeers in its *HostCache* are listed in the context producer's GUI with their IPs, Types, Modes and current Cluster IDs. The GUI also shows that the semantic cluster the ContextPeer joined in the "Semantic cluster joined" box and the cluster ID in the "ContextPeer Cluster ID" box. A screen shot is shown in Figure 6.4.

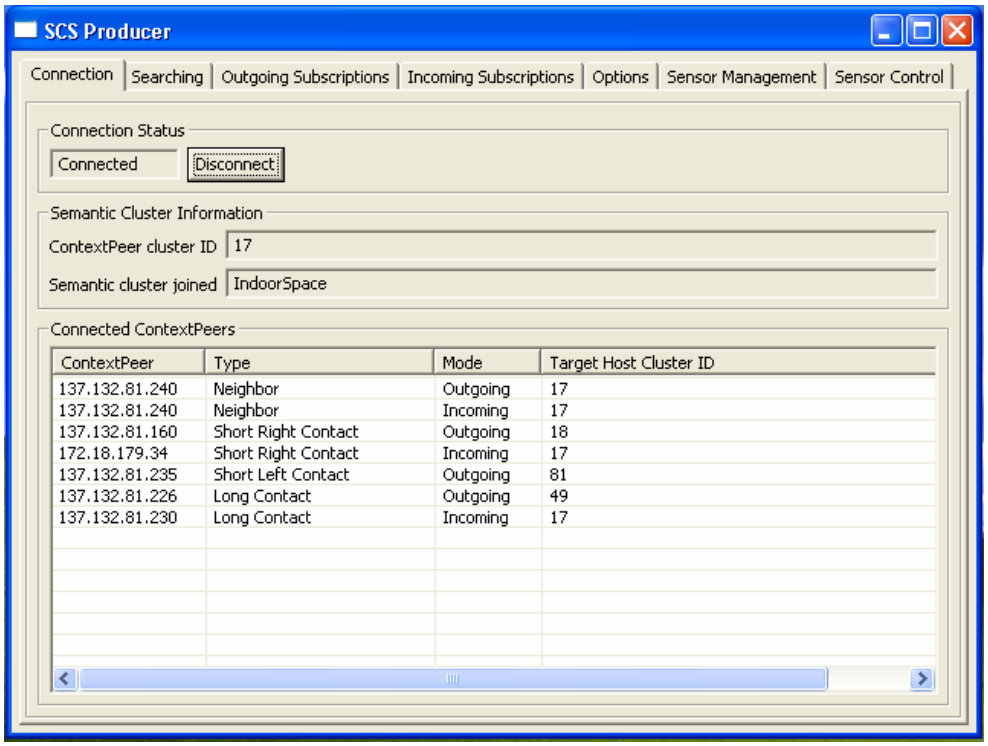



Figure 6.4: Screen shot of connections

### 6.2.4 Reference registration

After joining its major semantic cluster, a ContextPeer needs to register itself with all its minor semantic clusters. It invokes the method *getReferenceHosts* to randomly select reference hosting nodes of minor semantic clusters to store the index of the ContextPeer. This method in-turn sends the *RandomReferenceHostsMessage* to the network. Upon successful execution, a set of reference hosting nodes are obtained and returned to the ContextPeer. Then the ContextPeer initiates the outgoing reference connections to these nodes to register itself to all its minor semantic clusters. The reference connection string is shown below:

```
SCS REFERENCE_REGISTER\r\n
```

If the remote ContextPeer accepts the reference registration request, it responds with:

```
SCS REFERENCE_REGISTER OK\r\n
```

When a ContextPeer successfully registers itself with all its minor semantic clusters, its IP address and the associated semantic cluster are stored in the remote ContextPeer's *randomHostList* which basically is a *HashTable* structure containing the mapping between semantic clusters (i.e., minor semantic clusters) to nodes; then the connections are terminated. These reference connections are activated during a search request or a subscription request.

## 6.3 Message receivers

Different message receivers are defined to handle different types of messages. There are five types of message receivers, namely *JoinMessageReceivers, RandomContactMessageReceivers, SearchMessageReceivers, SearchResponseReceivers* and *InternalSearchResponseReceivers*. Message receivers are activated via callback methods. All message receivers implement the *MessageReceiver* interface, which defines the structure of these callback methods. The various receivers are outlined as follow:

*JoinMessageReceiver*: A *JoinMessageReceiver* implements *receiveJoin()*, which is defined as a callback method and is invoked when a *Join* message is received.

*RandomContactMessageReceiver*: A *RandomContactMessageReceiver* implements *receiveRandomContact()*, which is defined as a callback method and is invoked when a *RandomNeighsMessage*, *RandomShortLeftContactMessage, RandomShortRightContactMessage or RandomLongContactMessage* is received.

*SearchMessageReceiver*: A *SearchMessageReceiver* implements *receiveSearch()*, which is the primary callback method invoked when a *Query* message is received.

Both Context Producers and Context Interpreters use *SearchMessageReceiver* as they both provide context data and hence must be able to respond to queries. However, the implementation of *receiveSearch()* for Context Producers is different from that of Context Interpreters.

*SearchResponseReceiver*: In the same manner, a *SearchResponseReceiver* implements *receiveSearchReply()*, which is the primary callback method being invoked when receiving a *QueryHit* message.

*InternalSearchResponseReceiver*: An *InternalSearchResponseReceiver* also implements *receiveSearchReply()*, albeit in a different manner from that of a *SearchResponseReceiver*. An *InternalSearchResponseReceiver* is only utilized by Interpreters to receive responses for internal queries, which are in essence, premises required for context data reasoning.

## 6.4 Message forwarding and processing

Each ContextPeer implements a *Router* that is responsible for forwarding messages received from the network to other nodes. In addition, the *Router* also processes messages locally within the ContextPeer. Each ContextPeer keeps track of each *Query* message received by storing appropriate entries, such as the GUID and the associated semantic cluster ID of Query messages, in its *CacheTable*. The *Router* will not forward or process the message if its GUID has existed in the *CacheTable*. The class diagram of message forwarding and processing in Figure 6.5 illustrates various classes and their relationships.

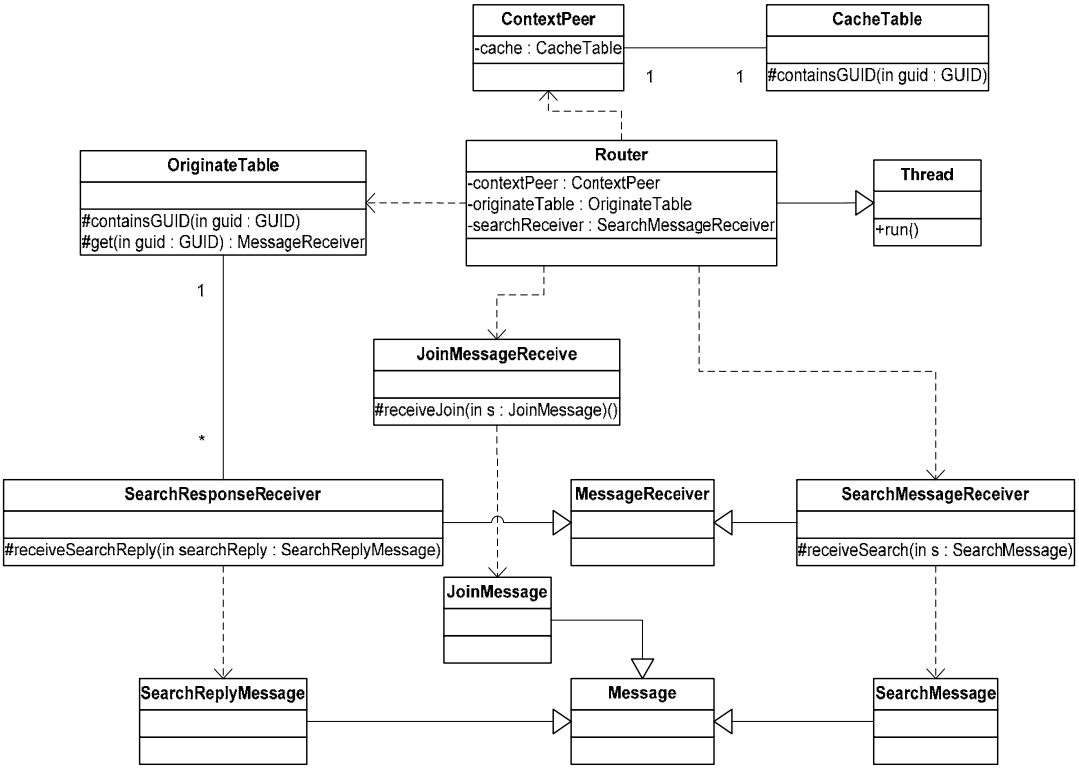



Figure 6.5: Classes responsible for message forwarding and processing

### 6.4.1 Message forwarding

Different types of messages are forwarded according to different forwarding rules.

A *Join* message is forwarded to a node in its *HostCache* whose CID (i.e., cluster ID) is closer to the CID in the message payload. When the *Join* message reaches a node whose CID matches the CID in the message payload, the node invokes the *RandomContactMessageReceiver* method to initiate a *RandomNeighsMessage*, a *RandomShortLeftContactMessage*, a *RandomShortRightContactMessage* and a *RandomLongContactMessage* to its neighbors, short left contact, short right contact and long contacts respectively. A node receiving any of these messages selects and returns a node randomly in its cluster.

A ContextPeer follows a similar rule to forward a *Query* message. When a *Query* message reaches the destination semantic cluster, it is forwarded to all the clusters in

that semantic cluster. This is done using a flag called *isFirstFlag* that indicates whether it is the first node in a cluster receiving the query. This flag is set to *True* by default. When the first node in the destination semantic cluster receives the *Query* message, it immediately forwards the message with the *isFirstFlag* remains as *True* to a node in its adjacent cluster, but it changes this flag to *False* and forward the *Query* message to all its neighbors in its *HostCache*. The first node in the adjacent cluster follows the same rule to forward the *Query* message until the next cluster until all clusters of the semantic cluster receive the query. Any node that receives the *Query* message with the *isFirstFlag* equal to *False* can only forward it to all neighbors in its *HostCache* (i.e., they are in the same cluster). If a ContextPeer receives a *Query* message whose CID matches the CID of a node entry in *randomHostList*, it initiates a reference connection with the flag `CONN_REFERENCE` to this node. Upon successfully creating the connection, the ContextPeer puts the connection object into the *referenceConnectionList*, which is an instance of the *ReferenceConnectionList*. The *Query* message is forwarded to this reference connection as well. The reference connection is terminated upon receiving the results from the remote ContextPeer. In the case of a subscription request, the reference connection is not terminated if the remote ContextPeer accepts the request until an unsubscription request is received.

A *QueryHit* message is forwarded only to the neighbor that has sent the corresponding *Query* message.

### 6.4.2 Message processing

In addition to message forwarding, the *Router* is also responsible for invoking callback methods of appropriate message receivers to enable ContextPeers to process *Query* and *QueryHit* messages received from the network.

*Query* messages are handled by the *SearchMessageReceiver* referenced to by the *Router*. For *QueryHit* messages, the *Router* maintains a structure called the *OriginateTable* that maps query GUIDs to their *SearchResponseReceiver*s. An entry is added to the *OriginateTable* each time a *Query* message is generated and sent to the network. When the *Router* receives a *QueryHit* message from the network, it looks up the *OriginateTable* to check whether the message is in response to a query sent by the local ContextPeer. If the *QueryHit*'s GUID exists in the *OriginateTable*, the corresponding *SearchResponseReceiver* is activated.

## 6.5 Search and subscription

In SCS, an RDQL string specifies a query's criteria. The RQDL string contains a triple pattern which enables statements that match the pattern to be returned as results. A triple pattern is structured like a statement, but its subject, predicate or object may take various values.

Before a query is sent to the network, it goes through a mapping process. For convenience, the subject, predicate and object of the query criteria's triple pattern will be referred to as the query's subject, predicate and object respectively in the remaining sections.

To initiate a search or subscription, a *SearchSession* is created. The *SearchSession* contains references to the query criteria, the semantic cluster that the query is mapped to, and the *SearchResponseReceiver* that is capable of handling replies from the network corresponding to the query. The *SearchSession* then sends the query to the SCS overlay network. This is handled by the *SearchSession*'s *SendThread* internal class, which sends the query to the relevant *Connection*s.

### 6.5.1 Query types

Queries in SCS can be broadly classified into two types: non-deduced and deduced. Non-deduced queries request for low-level context data, which is provided directly by sensors connected to Context Producers. Context Producers are only able to respond to non-deduced queries. On the other hand, deduced queries request for high-level data, which is derived from low-level context data. Such queries require reasoning, and thus, only Context Interpreters are capable of handling them.

### 6.5.2 Query messages

There are four different types of query messages in SCS, i.e., search requests, subscription requests, unsubscription requests and internal queries. A search request is issued only when a one-time response from the network is required. A subscription request enables a query to be subscribed to the network and a response be returned whenever there is a change in the relevant context data. When a Context Producer or a Context Interpreter receives a subscription request and decides to accept the request, it caches the query in the subscription request and responds to that query whenever there is a change in the context data. An unsubscription request does exactly as its name suggests; it instructs Context Producers and Context Interpreters to discard the corresponding subscription if they are currently maintaining it. An internal query is similar to a subscription request, but is issued exclusively by Context Interpreters to monitor context data for context reasoning. Internal queries are unsubscribed in exactly the same manner as unsubscription requests.

The structure of a *Query* message is illustrated in Figure 6.6. The query type field enables *Query* messages be differentiated into four types, namely search requests (00), subscription requests (01), unsubscription requests (10) and internal queries (11). The subscriber IP field contains the IP address of the ContextPeer that is subscribing to the

query. This field is only applicable for subscription requests and internal queries, and is not used for search requests. The first-node flag indicates whether the ContextPeer is the first node in a cluster. The reference flag indicates whether the connection is a normal connection or a reference connection. The destination CID field stores the semantic cluster ID that the query is mapped to.

SCS Query Message

| | Message Header | Query Type | Subscriber IP | First Node Flag | Reference Flag | Destination CID | Search Criteria |
|---|---|---|---|---|---|---|---|
| Byte offset | 0 … 22 | 23 … 24 | 25 … 28 | 29 | 30 | 31 … x | x+1 … |

Figure 6.6: Structure of *Query* message

### 6.5.3 QueryHit messages

For subscribed queries, the context data being monitored is usually subjected to changes. It is thus necessary to have two kinds of *QueryHit* messages, namely a *QueryHit* add (0) to indicate the presence of statements in *QueryHit*, and a *QueryHit* remove (1) to indicate the absence of statements in *QueryHit*. To differentiate these two kinds of *QueryHit* messages, a *QueryHit* type field is added to the *QueryHit* message. The structure of the *QueryHit* message is illustrated in Figure 6.7. The subscription flag field indicates *True* if this query hit is in response to a subscribed query. The hit count field indicates the number of hits returned. The IP address and port fields indicate the responding ContextPeer's IP address and port number. The last field contains the results.

SCS QueryHit Message

| | Message Header | QueryHit Type | Subscription Flag | Hit Count | IP Address | Port | Result Set |
|---|---|---|---|---|---|---|---|
| Byte offset | 0 … 22 | 23 | 24 | 25 | 26 … 27 | 28 … 31 | 32 … x |

Figure 6.7: Structure of *QueryHit* message

### 6.5.4 Subscriptions

*Subscription* and *Subscriber* are two data structures used in SCS to support query subscriptions. A *Subscription* holds information for an active query subscription while a *Subscriber* maintains information pertaining to a ContextPeer that is actively subscribing to a particular query. Each *Subscriber* instance keeps the subscription request GUID, the subscriber's IP address and a timer value. The latter is only used for non-subscribed deduced queries. Subscribers can be either normal subscribers or internal subscribers; the former refers to LookupClients that subscribe to non-internal queries while the latter refers to Context Interpreters that subscribe to internal queries. Three types of *Subscription*s exist, namely *OutgoingSubscriptions*, *IncomingSubscriptions* and *OutgoingInternalSubscriptions*. An *OutgoingSubscription* is created by a LookupClient for each outgoing subscription request that it issues to the network. It holds an outgoing subscribed query's GUID, criteria and a model that stores the responses received for the outgoing subscribed query. An *IncomingSubscription* stores an incoming subscribed query's criteria, its associated CID and a list of subscribers (either normal or internal) subscribing to that incoming subscribed query. When a subscription request is accepted by a Context Producer or a Context Interpreter, it checks whether an *IncomingSubscription* with similar criteria already exists. If such an *IncomingSubscription* exists, a new subscriber corresponding to the subscription request is added to that *IncomingSubscription*. Otherwise, a new *IncomingSubscription* is created, and the new subscriber added to it. Thus, only one *IncomingSubscription* is necessary for multiple subscription requests with the same criteria. An *OutgoingInternalSubscription* is maintained by a Context Interpreter for each internal query it sends to the network. Each *OutgoingInternalSubscription* maintains the internal query GUID and criteria, and

also a list containing the GUID and criteria of each deduced query that requires the internal query.

## 6.6 LookupClient

The functionality of a LookupClient includes initiating context queries according to the application's requirements, sending queries to a Context Producer, and returning the results to the application. The class diagram containing the major classes in the LookupClient is shown in Figure 6.8.

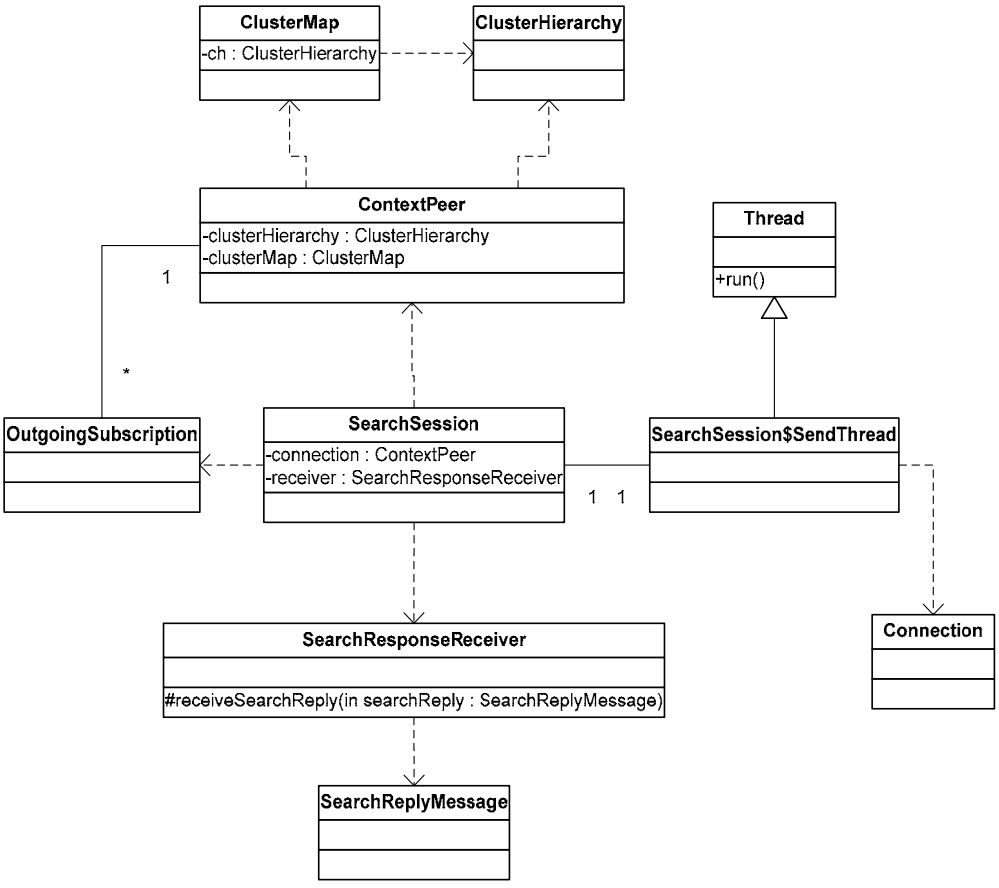



Figure 6.8: Class diagram of LookupClient

### 6.6.1 Initiating queries

A LookupClient needs to contact a Context Producer to obtain context data from the SCS overlay network. The LookupClient initiates a *SearchSession* to send a query to an existing Context Producer in SCS. The Context Producer serves as a proxy to resolve queries for the LookupClient. Before a query is sent, it is mapped to the appropriate semantic cluster(s) by means of the *ClusterMap* and *ClusterHierarchy*. If the query is a search request, the query is sent immediately to the network. However, if the query is to be subscribed, the LookupClient first creates an *OutgoingSubscription*. Subsequently, it issues a subscription request to the overlay network.

### 6.6.2 Receiving query responses

When receiving a *QueryHit* message, a LookupClient extracts the payload of the message by using *SearchResponseReceiver*'s *receiveSearchReply()* method. Usually, the method is defined either to display the contents of the *QueryHit* for debugging purposes or to pass it to an application for further handling, such as the activation of actuators.

## 6.7 Context Producer

A Context Producer includes all the functionalities provided by the LookupClient. In addition, it provides query routing, sensor management, context data management and query management. The class diagram related to these functionalities of a Context Producer is shown in Figure 6.9.

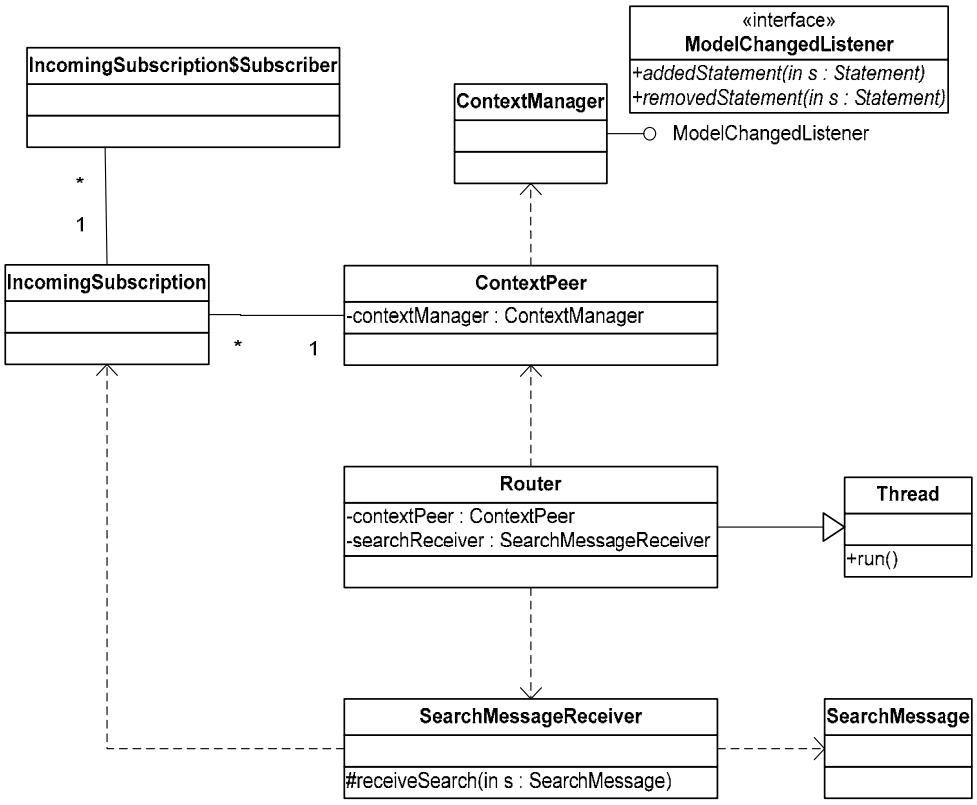



Figure 6.9: Classes responsible for context data and query management

### 6.7.1 Initiating queries

Similar to a LookupClient, a Context Producer can initiate both search queries and subscription queries. A screen shot is shown in Figure 6.10 to illustrate the GUI for initiating a search query or a subscription query. Users can choose to initiate a search request by typing an RDQL query in the "Search" box and clicking the "Search" button, or a subscription request by typing an RDQL query, selecting the "Subscribe" box and clicking the "Search" button. The results for a search request will be displayed in the "Responses" table whereas the results for a subscription request will be displayed in the "Outgoing Subscription" tab (described in Section 6.7.4.3).

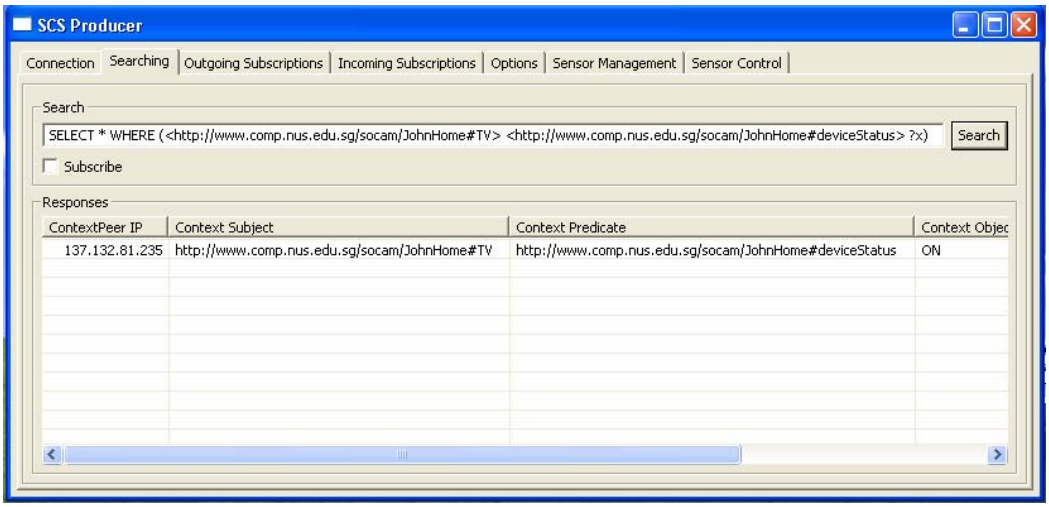



Figure 6.10: GUI of Context Producer for searching and subscribing context data

### 6.7.2 Sensor management

Various hardware or software sensors can be connected to a Context Producer. Any type of sensor may be used, as long as the appropriate wrapper is written to convert raw data from the sensor into RDF statements.

In this prototype implementation, we use software sensors to emulate hardware sensors such as location sensors, light level sensors and device status sensors. These software sensors can be added to or removed from a Context Producer easily. A screen shot of some of these software-emulated sensors is shown in Figure 6.11 and Figure 6.12. Figure 6.12 illustrates the GUI for attaching or detaching various sensors to a Context Producer. Figure 6.12 illustrates the GUI for selecting different values for the attached sensors. Each type of sensor is capable of generating a specific type of raw sensor data. The raw sensed data is then converted to an RDF statement by the Context Producer. For example, an RFID location software sensor can generate sensed data *<RoomID RFID_John>*, where *RoomID* represents the ID of a bedroom and *RFID_John* represents the RFID sensor attached to a person – John.

Upon receiving the sensed data, the Context Producer converts it to the RDF statement *<socam:John socam:locatedIn 'Bedroom'>*, representing the fact that John is currently located in the bedroom, and then store the RDF statement into its local context data repository. We have built in various software-emulated sensors in the prototype: location sensor, light sensor, noise sensor, device sensor, phone status sensor, door status sensor, temperature sensor, human count sensor, inventory sensor, calendar emulator and clock emulator.

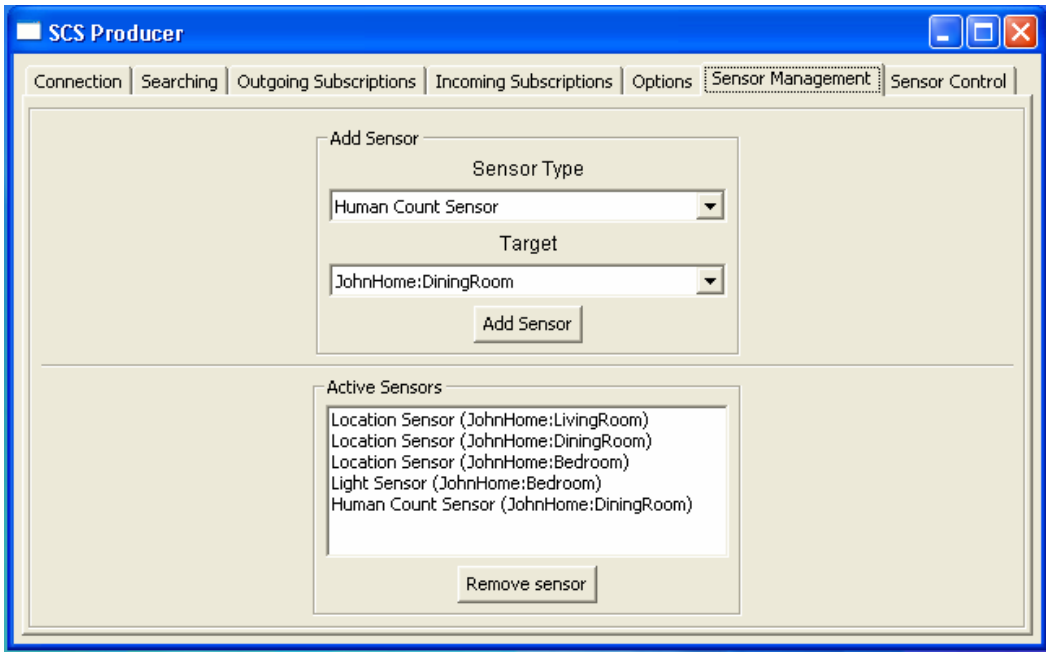



Figure 6.11: GUI of Context Producer for sensor management

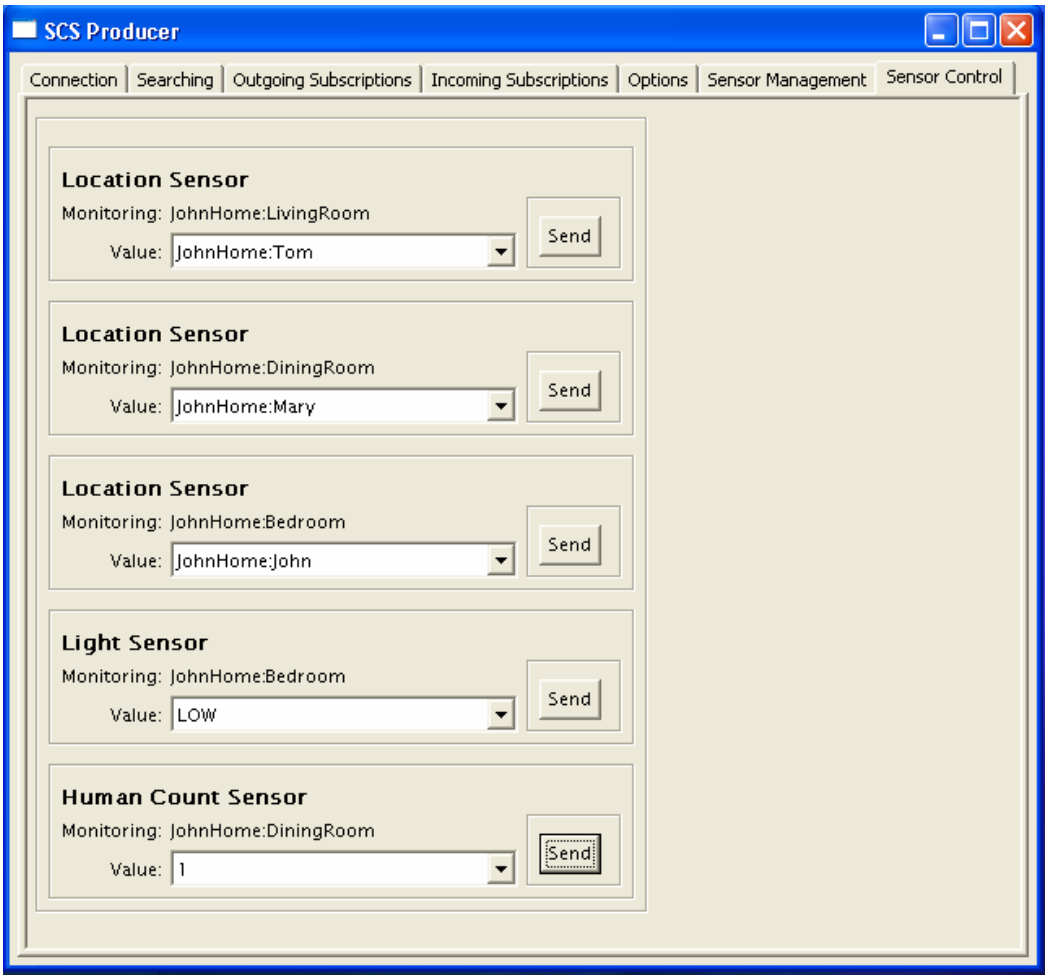



Figure 6.12: GUI of Context Producer for sensor value selection

### 6.7.3 Context data management

Each Context Producer maintains a local context data repository which supports RDF-based semantic querying using RDQL. This repository is realized by Jena *Model*s. Jena *Model*s are also used to store ontologies for context data. A Jena *Model* provides methods for the addition and removal of context data and for querying. It also provides a set of operations to combine itself with other Jena *Model*s. In addition, a *ModelChangedListener* may be attached to a *Model* to monitor changes in the *Model*, particularly with respect to context data addition and removal. *Model*-based context data management falls under the responsibility of the *ContextManager*.

The *ContextManager* maintains a few Jena *Model*s. These Jena *Model*s store context data ontologies, static context data and sensed context data. Static context data refers to context data that does not change frequently, such as the spatial information of buildings (e.g., John's bedroom is located in John's house). Sensed context data refers to data obtained from software sensors (e.g., John is located in John's bedroom). Such data is typically dynamic and changes frequently.

Besides providing accessor and mutator methods for the individual Jena *Model*s, the *ContextManager* also combines all the models into a base model which is used for local lookup. Ontology reasoning is performed on this base model. This is to be distinguished from the rule-based reasoning done by Context Interpreters. The former performs reasoning on context ontologies while the latter performs reasoning on context data based on a set of user-defined rules.

As sensed context data changes frequently, the *ContextManager* attaches a *ModelChangedListener* to the sensed context data *Model* to monitor these changes. This is especially important for responding to incoming subscribed queries and will be explained in the next section.

### 6.7.4 Query management

Query management involves three steps: local context lookup, subscription acceptance and subscription response. The first step applies to both unsubscribed and subscribed queries while the second and third steps apply only to subscribed queries.

#### *6.7.4.1 Local context lookup*

When a Context Producer receives a *Query* message, its payload is extracted by its *SearchMessageReceiver*. If the *Query* message is mapped to a non-deduced semantic

cluster, the *SearchMessageReceiver* then performs a local context lookup in the *ContextManager*'s base model. If the lookup returns non-empty results, a *QueryHit* message is then constructed using the *Query* message's GUID and the returned results, and subsequently sent to the SCS overlay network. Queries mapped to deduced semantic clusters are ignored by the Context Producer as it is not capable of performing context data reasoning.

*6.7.4.2 Subscription acceptance*

If the *Query* message happens to be a subscription request, the Context Producer determines whether it should accept the request based on the subscription acceptance policy described in Section 4.4.5.

Based on this policy, the Context Producer attempts to match the subscription request against the context data in its base model. If the request's predicate is a *DatatypeProperty*, the Context Producer determines if its base model contains statements with the same subject-predicate pair as the request. For example, for a given subscription request `<socam:Bedroom socam:lightLevel 'LOW'>`, the Context Producer accepts the request if there exists a statement with subject "`socam:Bedroom`" and predicate "`socam:lightLevel`" in its base model. Similarly, if the predicate is an *ObjectProperty*, the Context Producer determines if its base model contains statements with the same predicate-object pair as the request. For example, for a given subscription request `<socam:John socam:locatedIn socam:Bedroom>`, the Context Producer accepts the request if there exists a statement with subject "`socam:locatedIn`" and predicate "`socam:Bedroom`" in its base model.

If the Context Producer accepts the subscription request, an *IncomingSubscription* is constructed or updated as necessary.

*6.7.4.3 Subscription response*

Whenever a change occurs with respect to sensed context data, the *ModelChangedListener* informs the *ContextManager* of the RDF statement that has been added or removed. Subsequently, the *ContextManager* scans through all *IncomingSubscriptions* and identifies those that are affected by the change. This is done in the following manner: Let the added or removed RDF statement be $<subject_c, predicate_c, object_c>$. Let the RDF triple pattern of a particular *IncomingSubscription*'s criteria be $<subject_{sr}, predicate_{sr}, object_{sr}>$. Define the Boolean variable $isAffected_c$ as:

$$isAffected_c = (subject_c == subject_{sr}) \text{ ^ } (predicate_c == predicate_{sr}) \text{ ^ } (object_c == object_{sr})$$

where ^ denotes the logical AND operation. A variable can take the value of any arbitrary constant and is thus equal to any constant value. An *IncomingSubscription* is affected by a change *c* if $isAffected_c$ is true. For each affected *IncomingSubscription*, the *ContextManager* sends *QueryHit* messages to all its subscribers to supply them with the updated context data. The *QueryHit* messages sent are *QueryHit* adds if *c* is the addition of a statement or *QueryHit* removes if *c* is the removal of a statement. The screen shot of a subscription response is shown in Figure 6.13.

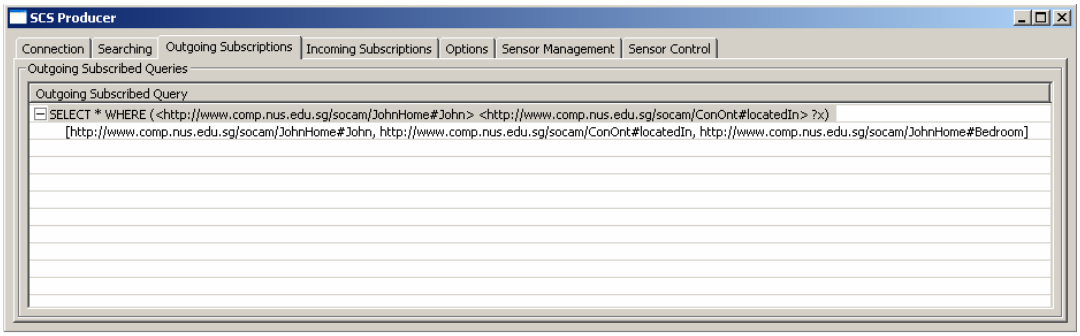



Figure 6.13: Screen shot of a subscription response

## 6.8 Context Interpreter

A Context Interpreter includes all the functionalities provided by a Context Producer. In addition, a Context Interpreter provides context reasoning service. This section describes the internal operation of a Context Interpreter including deduced query registration, rule management, deduced query management and context data reasoning. The class diagram related to the functionalities of context reasoning in a Context Interpreter is shown in Figure 6.14.

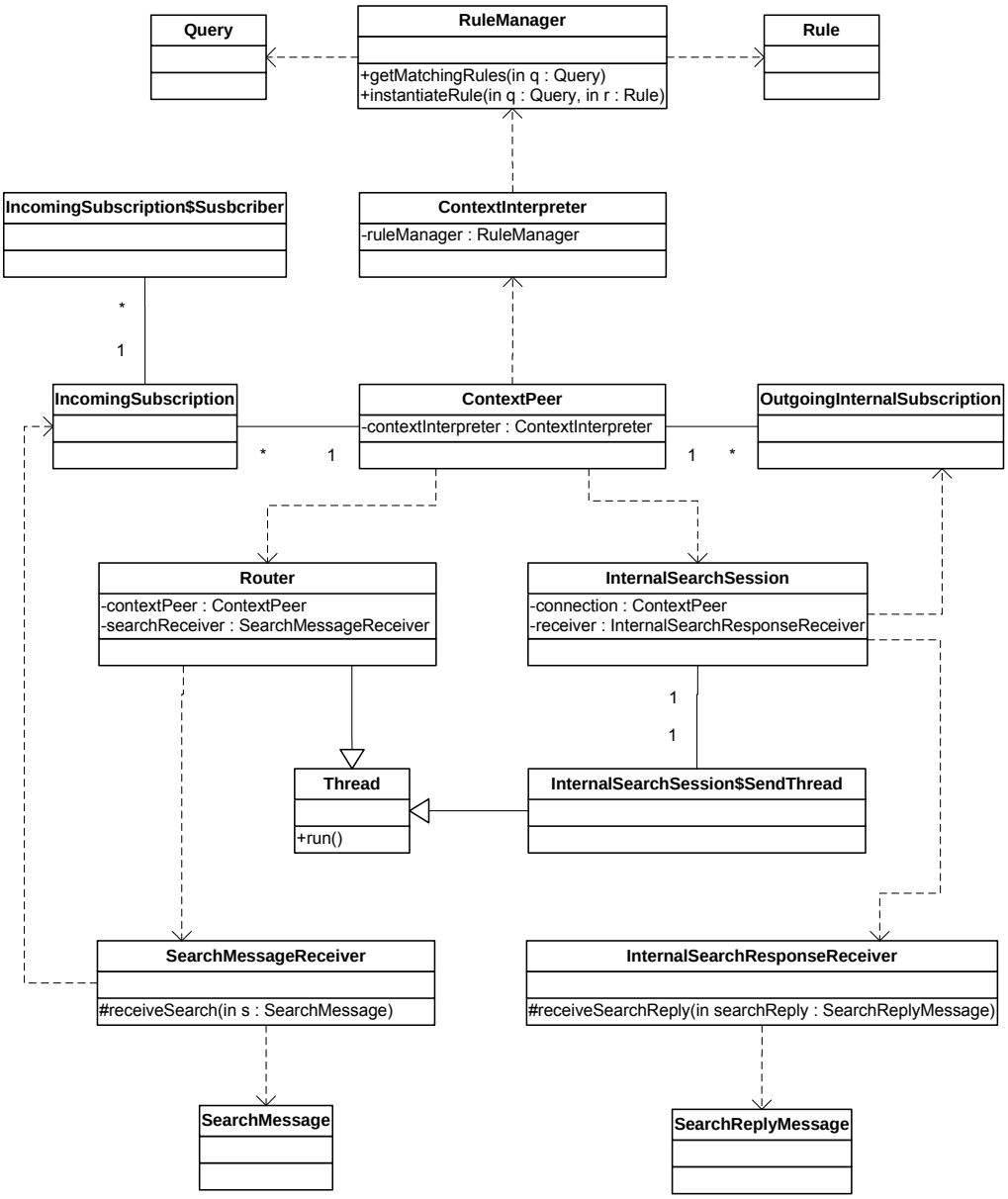



Figure 6.14: Classes responsible for context data reasoning

### 6.8.1 Deduced query registration

All deduced queries are required to register as incoming subscription requests in a Context Interpreter. In another words, they are treated as subscribed queries, regardless of whether they are subscribed or not. This is because the requested high-level context data is not usually available immediately as the necessary premises must

be obtained by the Context Interpreter before the high-level context data can be generated. The Context Interpreter constructs or updates an *IncomingSubscription* for each deduced query. The screen shot of incoming subscriptions for all the deduced queries in a Context Interpreter is shown in Figure 6.15. As shown in the figure, all deduced queries are listed in the "Incoming Subscribed Queries" table with the originator's IP address.

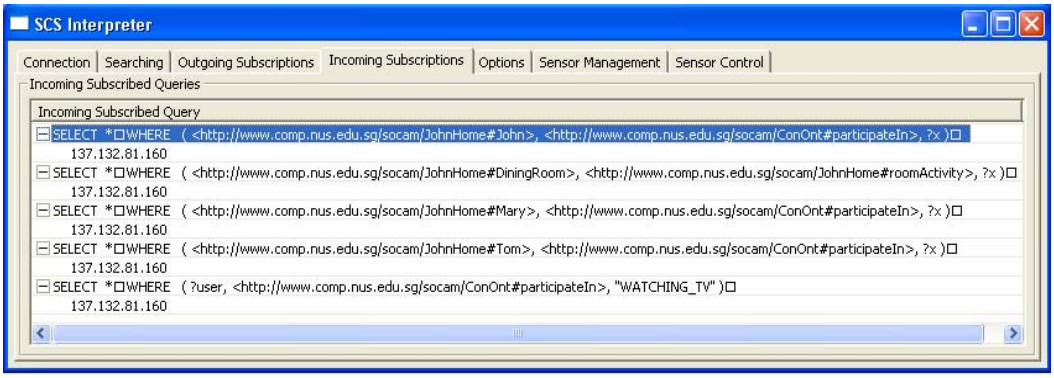



Figure 6.15: Screen shot of incoming subscriptions for all deduced queries in a Context Interpreter

### 6.8.2 Rule management

A Context Interpreter performs reasoning on low-level context data using a set of defined rules. These rules are encapsulated by *Rule* objects and are managed by a *RuleManager*. A Jena rule takes the form:

`[<RuleName>: <Premise`$_1$`> … <Premise`$_n$`> -> <Conclusion>]`

`<RuleName>` specifies the name of the rule. `<Premise`$_1$`> … <Premise`$_n$`>` are triple patterns representing the premises that make the conclusion true.

`<Conclusion>` is a triple pattern that specifies the high-level statement generated when all premises are satisfied.

Rules are specified in a rule file. The *RuleManager* reads this file, creates the corresponding *Rule* objects and stores them. Besides storing the rules, the *RuleManager* also provides methods for determining the matching rules for a query, as well as instantiating a rule with respect to a query.

*6.8.2.1 Determination of matching rules*

A query's subject is said to match the rule conclusion's subject if any of the following three conditions are true:

- the query's subject is a variable
- the rule conclusion's subject is a variable
- both the query's subject and the rule conclusion's subject are not variables and are equal

A query's object is said to match the rule conclusion's object in the same manner. If the query's predicate is the same as that of the rule conclusion, and both the subject and object of the query match those of the rule conclusion, the rule is said to match the query. Given a query, a *RuleManager* is able to return all matching rules for the query by performing the checks above on all the rules it manages.

*6.8.2.2 Rule instantiation*

The instantiation of a rule involves binding all variables in the rule with respect to a binding environment. This binding environment is constructed using the query and the rule conclusion. For each rule conclusion with a predicate matching that of the query, two bindings will potentially be added to the binding environment. If the query's subject is not a variable, a binding that binds the rule conclusion's subject to

the query's subject will be added to the binding environment. This applies similarly to the query's object.

### 6.8.3 Deduced query management

When a Context Interpreter receives a *Query* message, its payload is extracted by the Context Interpreter's *SearchMessageReceiver*. If the Context Interpreter identifies the *Query* as a deduced query, the *Query* is further processed. Deduced queries are treated differently from non-deduced queries. In particular, all deduced queries are treated as subscribed queries, regardless of whether they are subscribed or not. This means that the Context Interpreter constructs or updates an *IncomingSubscription* for each deduced query. This is done because the requested high-level context data is not usually available immediately as the necessary premises must be obtained by the Context Interpreter before the high-level context data can be generated. Thus, the Context Interpreter has to cache the deduced query while it attempts to obtain the necessary premises. However, since non-subscribed deduced queries are not true subscriptions, the Context Interpreter sets a timer value for the *Subscriber* instance corresponding to the ContextPeer that sent the query, allowing it to time out when the timer value expires. The Context Interpreter is unable to answer the deduced query if it is unable to generate the necessary high-level context data before the timer value expires. Subsequently, the *Subscriber* that timed out will be removed. On the other hand, if the Context Interpreter successfully answers the query before the timer value expires, the corresponding *Subscriber* object will be removed immediately. Subscribed deduced queries do not make use of the timer value.

### 6.8.4 Context data reasoning

A Context Interpreter performs context data reasoning using the *ContextInterpreter* class. Context data reasoning involves four phases. These phases are determination of matching rules, rule instantiation, internal query generation and high-level context data derivation. The first two phases are handled by the *RuleManager* (discussed in Section 6.8.2).

#### *6.8.4.1 Internal query generation*

After the *RuleManager* has completed all the necessary rule operations, a set of premises for all the rules that match the deduced query is obtained. The *ContextInterpreter* generates an internal query for each of these premises. Internal queries are analogous to subscribed queries. If an internal query has not been sent to the network before, the *ContextInterpreter* creates a new *OutgoingInternalSubscription* and adds the GUID and criteria of the deduced query to it. Otherwise, the details of the deduced query are simply added to the existing *OutgoingInternalSubscription* corresponding to the internal query. If a new *OutgoingInternalSubscription* is created, the internal query is subsequently sent to the network via *InternalSearchSession*.

#### *6.8.4.2 High-level context data derivation*

Before context data derivation can take place, a rule-based reasoner (subsequently referred to as the rule reasoner) and a *Model* containing the necessary premises (subsequently referred to as the premise model) are required. When a Context Interpreter starts up, its *ContextInterpreter* constructs the rule reasoner using all the rules managed by the *RuleManager* and creates a new empty premise model.

Premises received from the network in the form of responses for internal queries are placed in the premise model. The *ContextInterpreter* can create an inference model by attaching the rule reasoner to the premise model. The inference model contains high-level statements derived by the rule reasoner from the premises in the premise model. A lookup can be done against the inference model in the same manner as normal models.

Each time the Context Interpreter receives a premise from the network, the corresponding *InternalSearchResponseReceiver* is activated. The *InternalSearchResponseReceiver* instructs the *ContextInterpreter* to add the new premise to the premise model and rebuild the inference model. Subsequently, the *ContextInterpreter* does a lookup against the inference model with respect to each deduced query that requires the new premise. The relevant deduced queries can be determined by using the information stored in the *OutgoingInternalSubscription* for the internal query corresponding to the received premise. A *QueryHit* is sent to the appropriate subscribers for each deduced query that can be answered.

## 6.9 Development of context-aware applications

In the previous sections of this chapter, we have presented the details of our prototype implementation. In this section, we describe how application developers can make use of the SCS prototype to build various context-aware applications. First, we describe a set of APIs provided by our SCS prototype system. Then we present several typical context-aware applications we have developed to illustrate the development process and the use of APIs. Finally, we point out some other application scenarios and highlight our ongoing work in application development.

### 6.9.1 SCS APIs

The SCS prototype system defines a set of APIs for application developers to make use of the functionalities of SCS and build various context-aware applications. These APIs provide basic functionalities such as joining the SCS network, initiating queries and receiving results. These methods are defined in the *SCSAPI* class and include the following methods:

- *connect()* and *disconnect()*

  These methods allow an application to connect to the SCS overlay network via a ContextPeer.

- *search(String query)*

  This method enables an application to search for context data in the SCS overlay network.

- *subscribe(String query)* and *unsubscribe(String query)*

  These methods allow an application to subscribe a query to SCS as well as cancel a subscription.

- *reply(String response)*

  This is a callback method that is invoked by the ContextPeer when a response for a query is received. It in turn invokes the *reply(String response)* method defined in an application (this method is defined in the *ScsApp* interface which the application must implement).

### 6.9.2 Sample context-aware applications

Various context-aware applications can be built by using the SCS APIs. In this section, we describe two typical context-aware applications we have developed to demonstrate the different functionalities of SCS and how the development process can be made easier with the SCS APIs.

*6.9.2.1 SmartHome application*

The SmartHome application enables users to monitor home appliances, activities and provide intelligent services in smart home environments. These are typically context-aware applications that people envision to realize context-aware computing. We describe some of the application scenarios implemented in the SmartHome application below:

> We follow through a day in John's house where John, his wife Mary and their son Tom live. When any of these residents arrives home, the SmartHome application detects his/her presence and plays a voice greeting such as "Good Evening, John!" in accordance with the current time of the day. If John proceeds to his bedroom, turns the light off and takes a short nap, the SmartHome application deduces that John is currently sleeping and proceeds to switch his mobile phone to silent mode and turn on the "Do Not Disturb" indicator on his bedroom door. Meanwhile, Tom goes to the living room and switches on the television. Shortly after, the fixed line phone rings. As the SmartHome application notes that the phone is ringing while Tom is watching television, it proceeds to lower the television volume so that Tom can answer the call without distraction. In the evening, Mary prepares dinner and the family makes their way to the dining room. The SmartHome application then deduces that they are having dinner and starts playing the dinner music for the day, filling the dining room with enjoyable, ambient music. If anyone's birthday falls on that day, the SmartHome application then chooses to play the "Happy Birthday" song . . .

The GUI of the SmartHome application is shown in Figure 6.16. The application is able to connect to the SCS overlay network and disconnect from it, send search requests and subscription requests to the network and receive responses for them as well as execute action plans when certain context events occur. Specifically, in the example above, the application monitors and subscribes different types of context data such as a person's location (i.e., John's location), room activities (i.e., the dining room activity), device statuses (i.e., television status), and physical environment (i.e., bedroom light level). The application fires actions based on a set of context data subscribed, i.e., playing music when the family is having dinner in the dining room, decreasing the television volume when someone is watching television and the phone is ringing, etc. The action criteria are shown in the "Conditions" box and the name of the action is shown in the "Actions" box. Apart from the built-in subscription requests, users can also initiate a one-time search request to the network from the "Search" tab which has a similar GUI as the Context Producer.

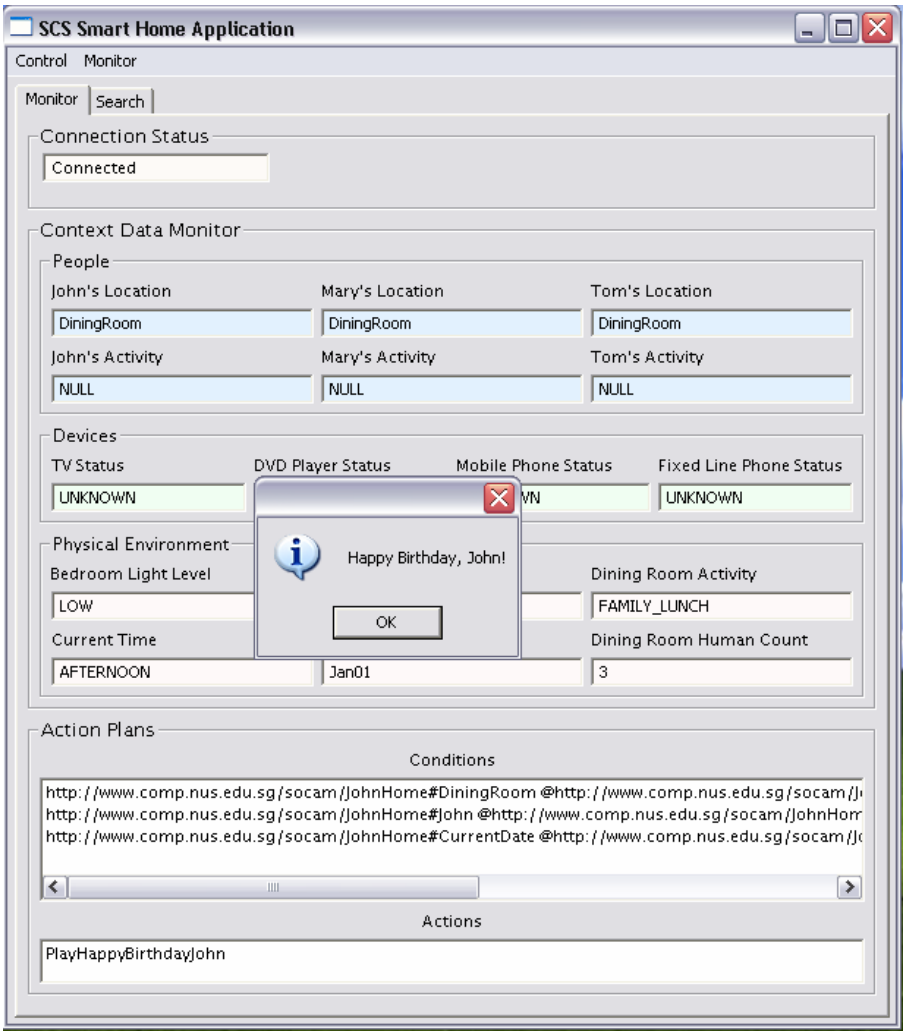



Figure 6.16: Screen shot of the SmartHome application

To make use of the SCS APIs, each application must implement the *ScsApp* interface, which specifies two methods:

- *reply(String response)*

  This callback method is called by the SCS API when a response is received from the SCS network.

- *updateConnectionStatus(boolean connected)*

  This callback method is called by the SCS API when a response is received from the SCS network.

In order to start monitoring the dining room activity, the SmartHome application subscribes to the query:

```
"SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom>
 <http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity>
 ?x)"
```

by calling the *subscribe()* method in the SCS API. Monitoring is stopped by calling the *unsubscribe()* method. This is a type of deduced query that is then routed to a context interpreter responsible for the same type of deduced query. The context interpreter derives the high-level contexts based on a set of user-defined rules. One of the rules is shown in Figure 6.17 (the rule format has been re-arranged for user readability) whereas the rest of the rules are listed in Appendix B. Based on this rule, if John, Mary and Tom are located in the dinning room, the dining room has three persons and the current time is evening, the context interpreter is able to derive that the dining room is having the room activity – FAMILY_DINNER.

```
[roomActivity_FAMILY_DINNER:

(http://www.comp.nus.edu.sg/socam/JohnHome#John
 http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
 http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),

(http://www.comp.nus.edu.sg/socam/JohnHome#Mary
 http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
 http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),

(http://www.comp.nus.edu.sg/socam/JohnHome#Tom
 http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
 http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),

(http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom
 http://www.comp.nus.edu.sg/socam/
JohnHome#numberOfPersons '3'),

(http://www.comp.nus.edu.sg/socam/JohnHome#CurrentTime
 http://www.comp.nus.edu.sg/socam/JohnHome#hasTime
'EVENING')

 ->

(http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom
 http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity
 'FAMILY_DINNER')]
```

Figure 6.17: A sample rule for context reasoning in the SmartHome application

In this application, the *reply()* method is implemented to invoke the *fireActionPlan()* method, which starts an action plan according to the response received from the network. In this case, *fireActionPlan()* invokes *playDinnerMusic()* when the statement

```
"<http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom>
<http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity>
'FAMILY_DINNER'"
```

is received. *playDinnerMusic()* then executes the appropriate code to play the dinner music for the day. The skeleton of the code in the SmartHome application is shown in Figure 6.18. As we can see for the code, application developers only need to specify their context queries that are of interest and define the required callback methods. Hence, they are able to put more development efforts into defining application-

specified actions and the application's GUIs. We have run the SmartHome application in various network setups including the testbed which is described in Section 6.10.

```
import com.socam.ScsAPI;

class MyApp implements ScsApp {
  ScsAPI scsAPI;

  public MyApp() {
    scsAPI = new ScsAPI(this);
  }
  public static void main(String[] args) {
    ...
  }
  public void connect() {
    scsAPI.connect();
  }
  public void disconnect() {
    scsAPI.disconnect();
  }
  public void search(String query) {
    scsAPI.search(query);
  }
  public void reply(String response) {
    fireActionPlan(response);   // application specific
  }
  public void startMonitor() {
    scsAPI.subscribe("SELECT * WHERE (socam:DiningRoom
      socam:roomActivity 'FAMILY_DINNER')");
  }
  public void stopMonitor() {
    scsAPI.unsubscribe("SELECT * WHERE (socam:DiningRoom
      socam:roomActivity 'FAMILY_DINNER')");

  // application specific method
  public void fireActionPlan(String response) {
    if (response.split(",")[0].equals("DiningRoom") &&
      response.split(",")[1].equals("roomActivity") &&
      response.split(",")[2].equals("FAMILY_DINNER")) {
        playDinnerMusic();
    }
  }
  // application specific method
  private void playDinnerMusic() {
    ...
  }
}
```

Figure 6.18: Skeleton of the code in the SmartHome application

*6.9.2.2 ShoppingAssistant application*

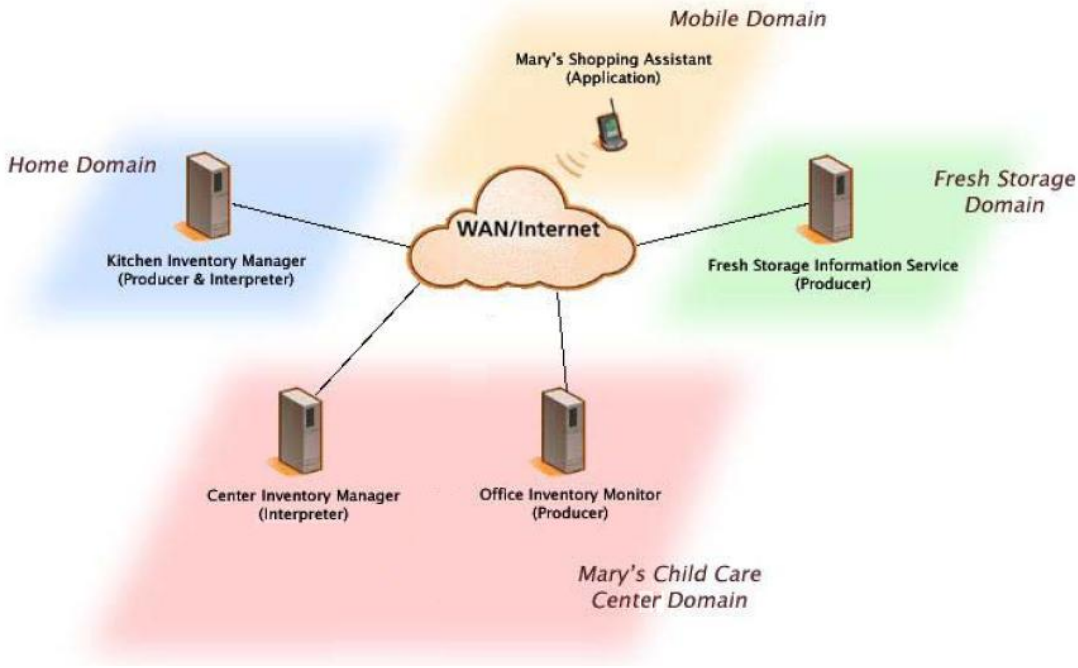



Figure 6.19: Scenario of the ShoppingAssistant application

Shopping assistant is another typical context-aware application which involves cross-domain context information as shown in Figure 6.19. It provides helpful shopping suggestions for users. In this example, we implement the following application scenario:

> While John is away at work, Mary decides to go shopping. She brings along her portable ShoppingAssistant and drives to town. Shortly after, Mary arrives at her favorite grocer, Fresh Storage. Her ShoppingAssistant recognizes that Mary is near one of her favorite shops. It then queries the network for items that have to be purchased. At home, the Kitchen Inventory Manager monitors the refrigerator inventory and deduces items that require purchasing as well as the number to purchase for each item, by checking the current quantities of items against their preset required quantities. The Kitchen Inventory responds

to the ShoppingAssistant, suggesting that a dozen eggs and two cartons of fresh milk should be purchased. Meanwhile, the Center Inventory Manager (which manages the inventory of items in Mary's child care center) responds to the ShoppingAssistant, indicating that ten boxes of cereals and three reams of printing paper should be purchased. The ShoppingAssistant then queries the Fresh Storage Information Service for the pricing and availability of the items suggested for purchase. After determining that the eggs, milk and cereals are available, the ShoppingAssistant alerts Mary and displays a table showing the items suggested for purchase at Fresh Storage (eggs, milk and cereals but not paper), with their respective quantities and prices. After purchasing the items, Mary drops by the child care center and then makes her way back home.

The development process is quite similar to the one in the SmartHome application, and hence, we will not go into details. However, we highlight here that cross-domain context data is used in this application, i.e., inventory data from the Kitchen Inventory Manager at home as well as from the Center Inventory Manager at Mary's child care center, merchandise data from the grocery store and book store, location data which changes from one store to another store, etc. Different namespaces are allowed in SCS and can be used to differentiate the domain. Application developers can define their own context ontology and data by specifying their namespaces. For example, one may use "`http://www.comp.nus.edu.sg/socam/JohnHome`" as the namespace for John's home, and others may use "`http://www.comp.nus.edu.sg/socam/ChildCare`" as the namespace for Mary's child care center. An example of definition of domain-specific context ontologies used in this application such as grocery store, book store and child care

center are listed in Appendix C. Through this application, we demonstrate that SCS works effectively in cross-domain context-aware applications.

### 6.9.3 Other scenarios and ongoing work

Many other context-aware applications can make use of the SCS APIs. For example, we describe a MeetingAssistant scenario as follows:

> At 9.00 a.m., John leaves home for work and drives to his office. In the office, the MeetingAssistant (which keeps track of predefined meeting schedules, prepares meetings and provides reminder services) notes that there is a meeting scheduled at 10.00 a.m. to be held between John and his colleagues at the Singapore office and their counterparts at the Sydney office, hosted by the Singapore office. The MeetingAssistant determines all registered participants of this meeting, which include John and his colleague Steve, and queries the network for their respective locations. John's PersonalAssistant (which acts as his personal information manager) responds that John is in his car. Thus, the MeetingAssistant sends a Short Message Service (SMS) to him, reminding him of the upcoming meeting at 10.00 a.m. The EmployeeMonitor (which keeps track of the locations of employees in the office) also responds, indicating that Steve is at his desk in the office. The MeetingAssistant proceeds to send a pop-up meeting reminder to Steve's desktop screen instead of sending an SMS to him. By 9.55 a.m., all participants in Singapore and Sydney have taken their seats in the respective offices. The MeetingRoomMonitors (which monitor meeting room activity) in both offices deduce that a meeting has started in their respective locations and inform the MeetingAssistant and its counterpart in Sydney. The MeetingAssistant

proceeds to dim the lights, switch on the projector and communications equipment and attempts to establish a connection with the office in Sydney. Meanwhile, its counterpart prepares the meeting room in Sydney. When the meeting concludes and all participants have left the meeting venues, the MeetingRoomMonitors deduce that the meeting has ended and inform both MeetingAssistants. Both MeetingAssistants then proceed to shut down all equipment and turn off all lights.

In continuing efforts to demonstrate the useful features of the SCS prototype system, I am currently supervising two university students to develop various context-aware applications. Particularly, we will focus more on building cross-domain context-aware applications and exploring all the useful features provided by the SCS prototype.

## 6.10 Prototype evaluation

We conduct a series of experiments to evaluate the SCS prototype system. The purpose of this evaluation is to test the prototype performance in close-to-real scenarios and to validate and calibrate our simulation model which has been evaluated previously. In this section, we present our evaluation results. First, we describe the setup of our prototype testbed, followed by the results obtained from a series of experiments such as bootstrapping, dynamic characteristic, query response time, query processing capability, deduced query processing time and memory consumption. Finally, we use the evaluation results to validate and calibrate our simulation model.

### 6.10.1 The prototype testbed

We set up the prototype testbed which consists of eight ContextPeers (seven Context Producer peers and one Context Interpreter peer) in the NUS campus network. Most

of the ContextPeers run on Pentium 800MHz desktop PCs with 256MB memory except one runs on a Pentium 1GHz desktop PC with 256MB memory, and another which runs on a 2.4GHz laptop with 512MB memory. SWebCache runs in a separate desktop PC. The physical layout of our prototype is shown in Figure 6.20. All the ContextPeers are connected to the NUS campus network. Each ContextPeer will be assigned a public IP address upon plugging in to the NUS campus network. The laptop uses a WLAN connection whereas the rest of the PCs use LAN connections.

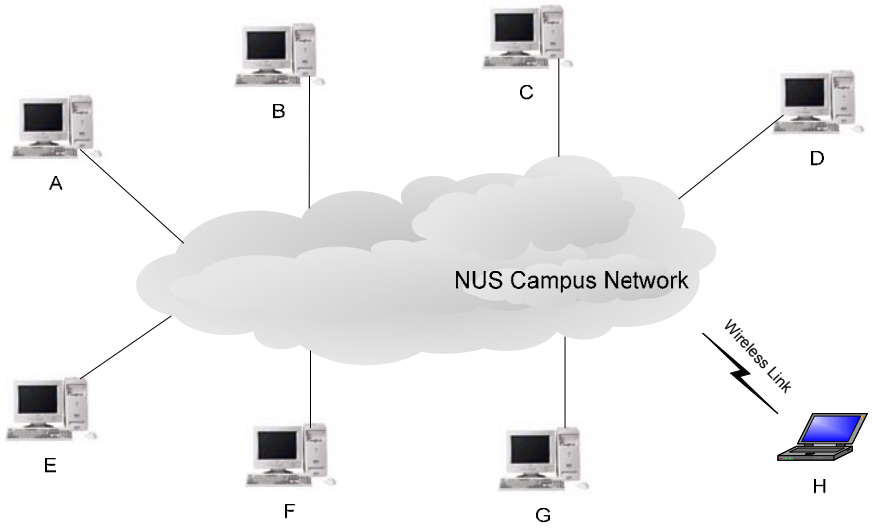



Figure 6.20: The physical layout of our prototype testbed

We create a set of context ontologies and context data whose semantics correspond to semantic clusters: *IndoorSpace*, *Person*, *DeducedActivity*, *OutdoorSpace*, *Merchandise*, *Time* and *Device*. These context ontologies and data have actually been used in the SmartHome and ShoppingAssistant applications. Each ContextPeer stores the upper context ontology and one or more domain-specific context ontologies. Context data stored in each ContextPeer may correspond to one or more semantic clusters. Before the evaluation starts, we need to place context ontologies and context data at each ContextPeer. We use two data placement schemes: homogeneous data

and heterogeneous data. In the earlier scheme, each ContextPeer stores its local data which corresponds to one particular semantic cluster. In the later scheme, each ContextPeer stores its local data which corresponds to multiple semantic clusters. The evaluation starts by connecting each ContextPeer to the SCS network. The statistics of the SCS overlay network will be displayed on SWebCache in the bootstrapping server PC. We have shown a sample screen shot of SWebCache in Figure 6.2 in Section 6.2.2. The SCS overlay network is constructed when ContextPeers randomly join the network. A ContextPeer obtains the IP of an existing ContextPeer from the bootstrap server. We test the bootstrap process by connecting the eight ContextPeers to the network in different joining orders, hence the structure of the ring space obtained may differ from one to another. Figure 6.21 shows one example of the ring spaces captured during the evaluations. Most of the results presented in this section are based on this ring space.

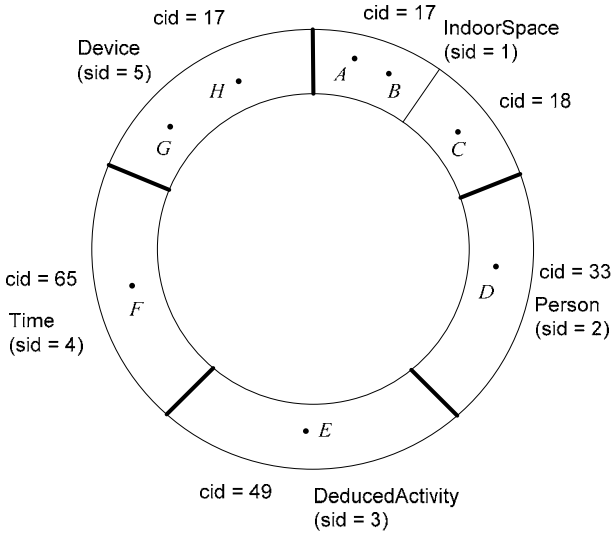



Figure 6.21: An example of the ring space constructed during the evaluations

For the homogeneous data scheme, the parameter $\beta$ equals 0.2 since the total number of semantic clusters is 5. In the case of the heterogeneous data placement, we specify

that the context data which is stored at each ContextPeer corresponds to 2 or 3 semantic clusters on average, and hence $\beta$ equals 0.5. In the evaluation, we set $m$ to 4, $n$ to 4 and $M$ to 2 ($m$ is the number of bits representing semantic clusters, $n$ is the number of bits representing sub-clusters and, $M$ is the cluster size). Context queries are randomly generated at each ContextPeer. Context queries can be non-deduced queries, for example:

```
SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn>
?x)
```

or deduced queries, for example:

```
SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/ConOnt#participateIn>
?x)
```

A full set of context queries used in the evaluation is listed in Appendix D. Different context queries are routed in the SCS overlay network and results are returned to the requestors if context data is available in the network. In the following sections, we present our evaluation results.

### 6.10.2 Bootstrapping

When a ContextPeer starts, it first goes through the semantic clustering mapping process to identify which semantic cluster to join. The mapping process is done by iterating each of the RDF data triples and identifying its corresponding semantic cluster. Then the ContextPeer chooses the major semantic cluster to join. On average,

the program initialization process takes about 4.26 seconds, and the mapping process for each RDF data triple takes about 0.251 ms. The initialization process involves reading and merging the ontology files stored locally and generating internal data structures for mapping. It is done only once when a peer starts and is only repeated if there is a change in these ontologies. The computation cost of the semantic cluster mapping process in SCS is much lower than the computation cost of LSI (the results can be found in [53]). Upon joining the network, each node creates and maintains the connections to one neighboring node in its own cluster (if available), two short contacts in its two adjacent (left and right) clusters and one long contact (if available) in any other semantic cluster. We also evaluate the ring construction and the cluster splitting/merging operations in our prototype by forcing a ContextPeer to join and leave different semantic clusters. ContextPeers join and leave the system smoothly. The joining process involves initiating the *Join* message, connecting to those nodes in the *JoinReply* message received and registering its reference if needed. The results for different steps in the bootstrap process are summarized in Table 6.2. Note that the reference registration process is only required for the experiment setup of $\beta = 0.5$.

TABLE 6.2: THE RESULTS FOR THE BOOTSTRAP PROCESS

| **Processes** | **Average Time Taken** |
|---|---|
| Program Initialization | 4.26 s |
| Semantic Clustering Mapping | 0.251 ms/RDF triple |
| Joining Process (for $\beta = 0.2$) | 2.56 s |
| Joining Process (for $\beta = 0.5$) | 2.98 s |

### 6.10.3 Dynamic characteristic

We also evaluate the dynamic characteristic of the ring space in our prototype by forcing ContextPeers to join and leave different semantic clusters randomly. Since we set $M$ to 2 in this evaluation, cluster splitting occurs when the cluster size is greater than 2 as more ContextPeers join the cluster. When the last node leaves, the cluster is merged with its neighboring sub-cluster (within the same semantic cluster). The ContextPeer joining/leaving and cluster splitting/merging processes run smoothly in our prototype. For testing the dynamic characteristic of the ring space, we introduce a parameter: time-to-stability. We define the steady state of ContextPeer as the state in which a ContextPeer maintains live connections to at least one neighboring node, one left contact, one right contact, at least one long contact as well as reference hosting nodes (the nodes storing its index). The steady state of a ContextPeer may collapse if one of the following events occurs:

- Its neighboring node(s) or short left/right contacts or long contact(s) leave the network or some of these nodes change their major semantic clusters (due to changes in their local context data).

- Reference hosting node(s) leave the network or change their major semantic clusters.

Queries routing may be affected when ContextPeers are not in the steady state. The time-to-stability parameter is measured from the time when the steady state of a ContextPeer collapses until it reaches the steady state again. We measure the time-to-stability of the affected ContextPeers for different test cases (for $\beta = 0.5$ only) and the results are summarized in Table 6.3 (note that no backup links are used in these cases). In the experiments, the number of neighboring nodes and the number of long contacts

are set to 1 respectively. In Case 1, the affected ContextPeer is required to initiate a *RandomNeighsMessage* or *RandomShortLeftContactMessage* or *RandomShortRightContactMessage* or *RandomLongContactMessage* to the network and start to connect to the nodes in the reply messages. In Case 2, the affected ContextPeer needs to initiate a *RandomReferenceHostsMessage* and re-register itself to the nodes in the reply message.

TABLE 6.3: RESULTS ON TIME-TO-STABILITY (WITHOUT BACKUP LINKS)

| **Test Cases (for $\beta$ = 0.5, without backup links)** | **Average Time-To-Stability** |
|---|---|
| Case 1: The neighboring node or short left/right contact or long contact leaves the network or changes its major cluster or cluster splitting/merging occurs | 271 ms per connection |
| Case 2: Reference hosting nodes leave/change | 87 ms per reference |

In a highly dynamic SCS network, peers leave and join frequently; this may result in relapse rate very high. A high relapse rate may affect query routing in SCS. To prevent this, we use a backup link for each type of connections. Once the steady state collapses, a ContextPeer can switch to the backup link immediately for the affected connection. With this backup scheme, we can minimize the disruption to query routing in the highly dynamic SCS network where peers frequently leave and join.

### 6.10.4 Query response time

In this experiment, we measure the query response time of the prototype system. The purpose of this experiment is to analyze the important factors which affect the query response. We also compare the SCS prototype system with the ContextBus architecture. In the experiment, we randomly select non-deduced queries and deduced

queries from the query pool listed in Appendix D, and measure the average response time for both types of queries. The query response time can be broken down into three portions: query mapping, query processing and communication. Query mapping is the time taken by a ContextPeer to map a query to the appropriate semantic cluster(s). Query processing is the time taken by a Context Producer to process a non-deduced query or the time taken by a Context Interpreter to process a deduced query. For a Context Producer, query processing involves performing a local lookup against the base model. In the case of a Context Interpreter, query processing involves rule processing, internal query generation and high-level context data derivation and lookup. Communication represents the time taken for queries and their responses to travel over the network. For non-deduced queries, it is the sum of the time taken to send a query from the LookupClient to the Context Producer and the time taken to send the query's response from the Context Producer back to the LookupClient. For deduced queries, besides communication between the LookupClient and the Context Interpreter, also included is the additional time taken for the Context Interpreter to send internal queries to Context Producers as well as for the Context Producers to send appropriate internal query responses (i.e., premises) back to the Context Interpreter. In this experiment, we also measure the query response time for the ContextBus architecture. The testbed setup is the same as for the SCS prototype. The context ontologies and data placement for ContextBus are the same as the one for the case of $\beta = 0.5$ in the SCS prototype. The results are shown in Figures 6.22 and 6.23.

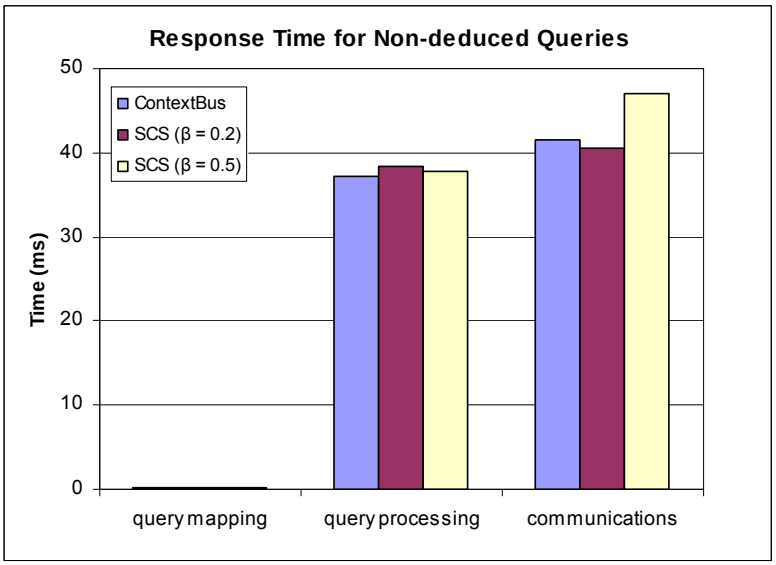



Figure 6.22: Response time for non-deduced queries

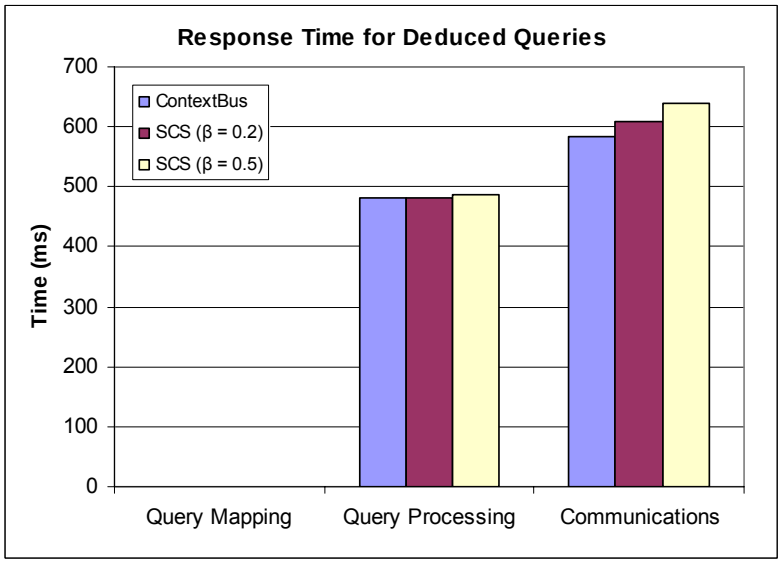



Figure 6.23: Response time for deduced queries

On average, the total query response time of SCS from the perspective of the LookupClient is 83.5ms for a non-deduced query and 1108 ms for a deduced query. The average query response time of ContextBus is 78.9ms for a non-deduced query and 1066 ms for a deduced query. The results for these two architectures are similar in

term of query response time in the current testbed setup. As we can see from the above results, the processing time for query mapping can be ignored; the costs of query processing and communication are the major factors. The response time for a deduced query is longer than that for a non-deduced query because the worst case scenario is assumed, i.e., the Context Interpreter's premise model is empty and all premises have to be obtained from the network by issuing internal queries. If the Context Interpreter is already maintaining certain incoming subscribed queries, the response time for further queries may be much lower as some of the premises required by the new queries may already be present in the premise model. We will investigate and analyze this issue further in Section 6.10.6. For the SCS prototype, the communication cost of non-deduced queries in the case of $\beta = 0.5$ is slightly higher than that in the case of $\beta = 0.2$ could be due to the extra communication costs incurred by reference connections. Communication time for a deduced query is higher than that of a non-deduced query as this time includes the time taken for internal queries and their respective responses to travel over the network. In a real application scenario, communication costs are highly network dependent especially in a wide-area network.

### 6.10.5 Query processing capability

This section evaluates the capability of the Context Producer and Context Interpreter to process simultaneous queries. We conduct two experiments. In the first experiment, the LookupClient continuously sends a varying number of queries to the Context Producer by randomly picking them from a large query pool which is generated based on the list in Appendix D. The second experiment is similar to the first, but the LookupClient continuously sends deduced queries to the Context Interpreter instead. We measure the average processing time for both experiments. Figure 6.24 plots

average query processing time against number of simultaneous queries. When a logarithmic scale is used for both axes, both graphs display a linear relationship. This shows that the capabilities of both Context Producer and Context Interpreter scale well to number of queries.

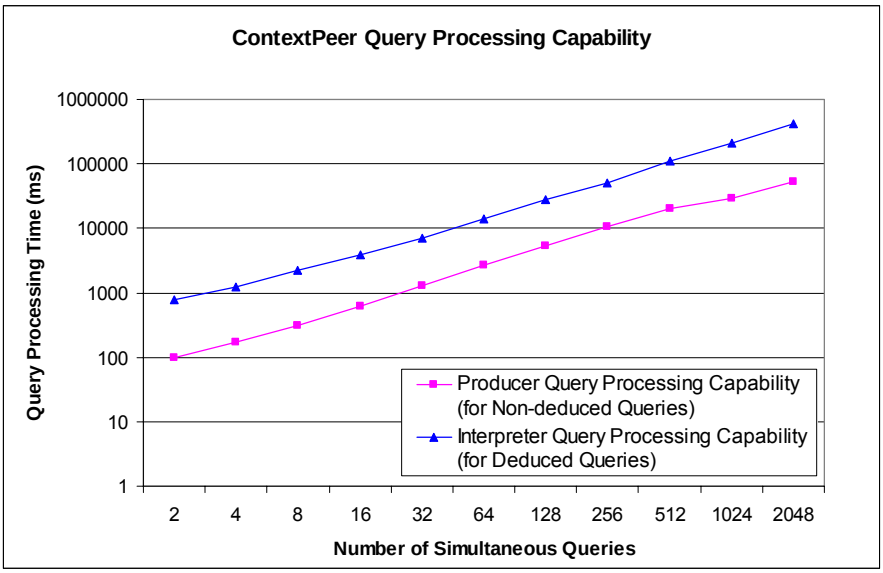



Figure 6.24: ContextPeer query processing capability

### 6.10.6 Improving deduced query processing

In Section 6.10.4, we have analyzed and identified that query processing and communication are the two main factors that affect query performance in SCS. In this section, we propose and evaluate different methods aiming to improve performance in processing deduced queries.

We propose four possible methods to handle deduced queries in a Context Interpreter: (Table 6.4 summarizes these four methods)

- **Method A – Pre-subscribe all premises:** The Context Interpreter analyzes all the rules it maintains and sends internal queries to the network for all possible

premises upon startup. It also derives all possible high-level context data corresponding to all the rules it stores.

- **Method B – Internal queries are not shared:** For each deduced query received, the Context Interpreter sends internal queries for all relevant premises and unsubscribes these internal queries when the deduced query is answered. Internal queries are not shared between rules.
- **Method C – Internal queries are shared:** This method is similar to Method B, but internal queries are shared between rules. The Context Interpreter only sends an internal query if it has not already been sent and only unsubscribes an internal query if there are no deduced queries pending to be answered that require that internal query.
- **Method D – Pre-subscribe certain premises /Internal queries are shared:** This method is a combination of Methods A and C. The Context Interpreter pre-subscribes the premises of the rules corresponding to frequent deduced queries and uses Method C for the rules corresponding to infrequent deduced queries.

TABLE 6.4: DIFFERENT METHODS FOR DEDUCED QUERY PROCESSING

| Method | Pre-subscription | Internal Subscription Sharing |
|---|---|---|
| A (Pre-subscribe all premises) | ✓ | × |
| B (Internal queries are not shared) | × | × |
| C (Internal queries are shared) | × | ✓ |
| D (Pre-subscribe certain premises/Internal queries are shared) | ✓ | ✓ |

We evaluate the effectiveness of each method in this experiment (the setup is based on $\beta = 0.2$). Method A is performed by manually placing the necessary high-level

statements in the Context Interpreter's inference model beforehand. Thus, the Context Interpreter responds to deduced queries in the same way as a Context Producer does with non-deduced queries. Method C is performed by checking for internal queries that have already been sent and only initiating internal queries if they have not been sent before. Method B is performed by removing the check for internal queries that have already been sent. Hence, duplicated internal queries may be sent over the network. Method D is performed by manually placing a portion of all possible high-level statements in the Context Interpreter's inference model beforehand. Note that we use one-third of the high-level statements in this experiment, the ratio should be computed based on the query statistics obtained from real scenarios)

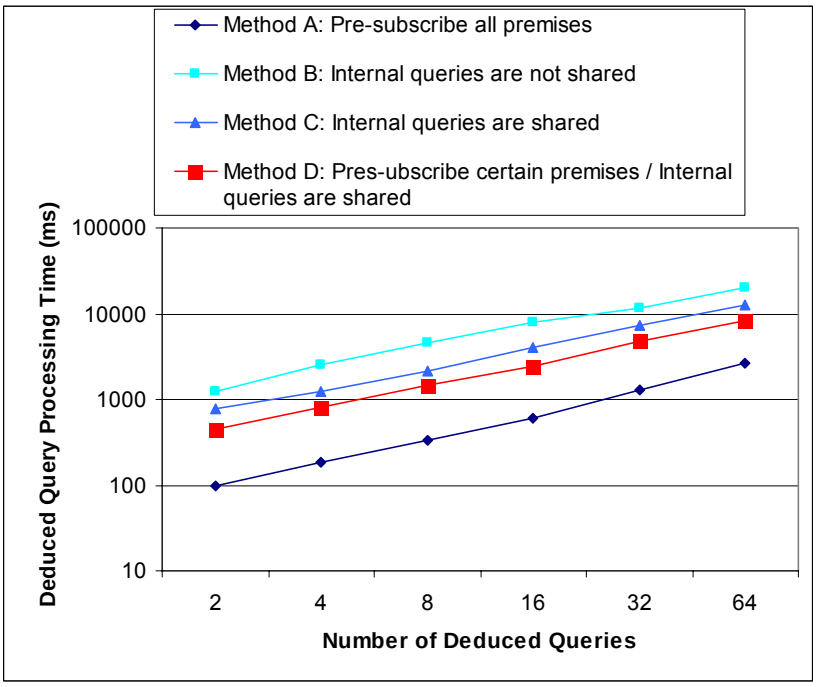



Figure 6.25: Deduced query processing time

Figure 6.25 plots the processing time of deduced queries for different methods. It can be seen that Method A gives the shortest response time. This is because the Context Interpreter process reasoning rules, internal query pre-subscription and high-level

context data derivation beforehand. The result for Method A is similar to the non-deduced query processing time for the Context Producer in Figure 6.24. Method B performs the worst as the worst-case scenario is assumed (i.e., the Context Interpreter has to issue an independent internal query for each of the premises).

Although the response time for Method A is very low, it may not scale if the Context Interpreter maintains too many rules. This is because the Context Interpreter would have to maintain a very large number of internal subscriptions at all times. In addition, many irrelevant internal queries may be sent to the network, thus increasing the network load unnecessarily. Also, the Context Interpreter may have to periodically resend all internal queries to ensure that it is able to obtain premises from Context Producers that have just joined the network. Method B removes the need for the Context Interpreter to keep track of the internal queries it sends to the network and ensures that the premises obtained are fresh as internal queries are only sent when deduced queries are received. However, this method is inefficient as it generates many redundant internal queries, which increase the network load and the response time for deduced queries. Method C provides a good compromise between Methods A and B. Although its response time is greater than that of Method A, it generates a significantly smaller number of internal queries compared to both Methods A and B and thus reduces the network load. In addition, the premises obtained are also fresh as the internal queries are subscribed on demand as it is with Method B. Method D improves the response time of Method B by pre-subscribing the premises for deduced queries that are popular. It requires the Context Interpreter to keep track of and maintain the statistics of deduced queries received and deploy an algorithm to decide which queries should be pre-subscribed beforehand. This method needs to be further

studied in our future work. In the current SCS prototype, Method C is selected to handle the deduced query processing.

### 6.10.7 Memory consumption for deduced query processing

In the previous section, we have evaluated four methods for handling deduced queries and analyzed their respective communication costs. In this section, we evaluate the memory consumption of the different methods. We assume the same experimental setup as in Section 6.10.6. Figure 6.26 plots the memory consumption in term of MB (megabytes) for the above four methods.

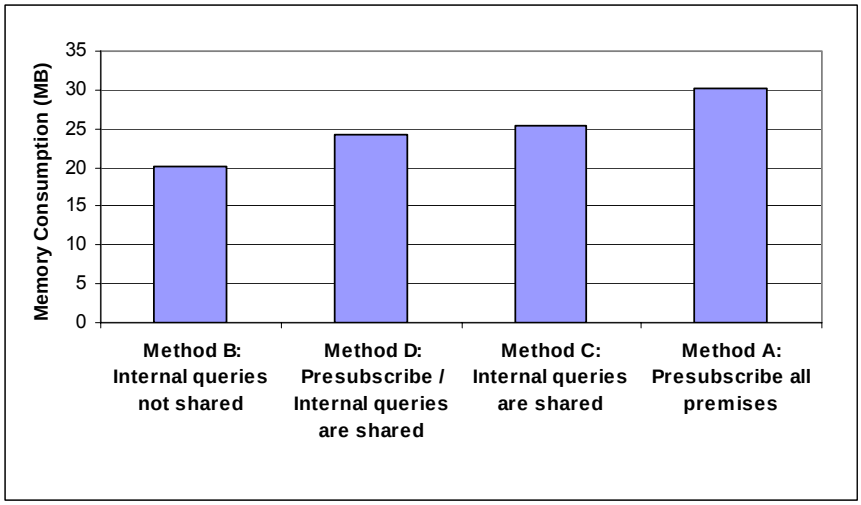



Figure 6.26: Memory consumption for the different methods

Among them, Method A consumes the most memory. This is because in Method A, the Context Interpreter has to maintain internal subscriptions for all internal queries it has pre-subscribed to. Method B consumes the least memory because the Context Interpreter does not need to maintain pre-subscribed internal queries as all internal queries are subscribed on demand. The memory consumption of Method C and Method D fall between that of Methods A and B. Clearly, there is a tradeoff between query response time and memory consumption. This evaluation also reveals that the

computing device that runs the Context Interpreter does require certain hardware capabilities (i.e., processing power and memory) apart from certain software platform capabilities. Some embedded computing device may not be capable of running the Context Interpreter, for example, mobile phone, etc. Further studies are needed on deploying our SCS prototype into embedded devices.

### 6.10.8 Validation of our simulation model

In the last few sections, we have presented the evaluation results obtained from our prototype system. In this section, we will use some of these results to validate our simulation models.

First, we set up the simulation based on the SCS prototype testbed (shown in Figure 6.21). In the simulation, we follow the same setup as the prototype testbed. We create eight ContextPeers in the overlay network. The propagation delay between every two ContextPeers is configured based on the measurements from the prototype testbed. We also use two data placement schemes: $\beta = 0.2$ and $\beta = 0.5$; however, context data triples are replaced as keywords in the simulation. Each set of keywords corresponds to different semantic clusters. Context queries are modeled as searches for specific keywords. We test this simulation model by running the same procedure as for the prototype system, such as starting the bootstrap process, constructing the ring space, initiating queries and receiving results. We obtain the same ring space as in the prototype as well as similar results in terms of query routes, search path length and search cost as in the prototype. This confirms that our simulation model can predict the behavior of the prototype system.

Next, we use some of the evaluation results such as the time taken for mapping and processing a context query in the Context Producer, which are obtained from the

prototype system, to re-run the experiments in Section 4.5.3.1 and 4.5.3.2. We obtain similar results, as described in the above two sections. We do not test for deduced queries because of the complexity of developing a reasoning engine in our simulator.

## 6.11 Summary

In this chapter, we have described the implementation of our SCS prototype in detail. We have also measured the performance of our prototype and reported the results. Our experimental results indicate that the SCS prototype works practically and achieves fair good performance in real scenarios. In addition, our experiences on developing sample context-aware applications in an application domain or cross multiple domains show that the application development can be greatly simplified based on the SCS APIs. We believe the SCS system can have a significant practical impact on building a large variety of context-aware applications in multiple context spaces.

# CHAPTER 7

# CONCLUSIONS AND FUTURE WORK

This chapter concludes the thesis with a summary of our research contributions and outlines several directions for future work.

## 7.1 Summary

In this thesis, we address the problem of the provision of context-aware infrastructure support for collaborative context-aware applications over multiple context spaces. We aim to provide infrastructure support for designing scalable and self-organized context-aware systems in multiple context spaces, facilitating context-aware application development, and conducting a thorough and rigorous evaluation of our system. We provide a set of core infrastructure services – wide-area context lookup and distributed context reasoning, coupled with a context representation model.

For context modeling, we have proposed an ontology-based context model and two-tier context ontologies.

For context lookup, we have proposed a semantic P2P overlay network named SCS to provide users and applications with an efficient lookup service. We have designed various techniques in SCS including:

- an ontology-based semantic mapping scheme for fast semantic abstraction,
- a one-dimensional ring space for reducing overlay maintenance cost and enabling efficient routing,

- cluster splitting/merging for self-scaling to number of nodes,
- cost-aware selective flooding for minimizing redundant query messages, and
- a context push service for notifying context consumers of context changes quickly.

For context processing, we have proposed a distributed logical reasoning approach to interpret various contexts. Through logical reasoning, we are able to raise the level of context abstraction based on users' or applications' needs. Context reasoning is done in a distributed fashion so that reasoning engines can be embedded into different application domains.

We have implemented and evaluated the SCS system using both simulation and prototype. The evaluation results show that SCS works effectively in both simulation and practice. The SCS system is self-organized and scalable to the growth and changes of nodes. It also has better search efficiency and low overlay maintenance overhead. Our experiences of developing context-aware applications using the SCS prototype show that the development process can be greatly simplified. Application developers need only focus on application-level tasks without wasting time and effort on low-level details of acquiring context data from multiple context spaces.

## 7.2 Future work

In this thesis, we have mainly focused on context lookup, which is a fundamental service supporting context-aware systems and applications. Further investigations and studies are needed to explore and utilize the features in our context model and deploy

distributed context reasoning in reality. We present here some directions for our future research[6]:

*Performance evaluation over the global internet involving multiple domains*

To further study our prototype system in the Internet setting, we are working to set up and run ContextPeers in other organizations outside the NUS campus network, or even in different countries. Furthermore, we are currently designing and developing various context-aware applications over multiple context spaces.

*Deployment of physical sensors*

Due to the limitations of our research facilities and available resources, the current SCS prototype system is lack of the deployment of physical sensors. Instead, we create software sensors to emulate the behaviors of physical sensors. We will seek external funding and supports to deploy real sensors into our SCS prototype and further evaluate our system.

*Interoperability of context ontology models*

Although our two-tier context ontology approach allows a certain degree of interoperability between different context-aware systems over multiple context spaces, there are often cases where there are multiple ways to model the same information. Different ontologies may model the same concepts in different ways. This may be due to differences in the perspectives of different systems, different developers, different professions, etc. In order for different context-aware systems to share context information that commits to heterogeneous ontologies, ontology interoperability

[6] Based on this thesis work, we applied for and successfully got a 3-year research grant from Agency for Science, Technology and Research (A-star) under the Ultra Wide Band – Sentient Computing (UWB-SC) program. Many of these research directions will be addressed in this project.

mechanisms are needed to map terms in one ontology to their equivalents in other ontologies. We believe such mechanisms can great improve the interoperability of the SCS system, and hence our system can be potentially deployed in a wide range of applications and domains.

*Uncertainty management*

Uncertainty management is important for ensuring the robustness of context-aware applications. In our earlier attempt [25], we have extended our basic context model by incorporating probabilistic information, and used the Bayesian network to reason about uncertain contexts. More extensive studies are needed, such as how to acquire probabilities, and how to use other techniques to reason about these probabilities.

*Privacy in context model*

The privacy issue has been recognized as an important research area in pervasive computing. Existing work [17][19] has focused on providing a management framework and solving interaction issues with users and applications. We observe that context privacy can be embedded into the basic context model. It should be more efficient to manage and handle context privacy issues from the ground up.

*Deploying ContextPeers to mobile and embedded devices*

As an increasing effort to embed computing into mobile devices and users, context-aware computing is emerging as part of our lives in the future. To enable users and applications to benefit from our system, it is desirable to deploy and integrate ContextPeers to mobile and embedded devices such as PDAs, cell phones, etc. Many practical issues need to be investigated and studied further. For example, different mobile and embedded devices may require different software platforms for their

applications. Since our system is based on Java, it is easy to port and integrate the functionalities of a ContextPeer into different software platforms. Another critical requirement is computing resources such as processing power and memory. Again, more investigation and studies are needed to address these practical issues.

*Embedded context reasoning*

We envision that it will be more useful to embed context reasoning into mobile and embedded devices. As we have studied in this thesis, logical context reasoning is really a computationally intensive process and it is difficult to embed into mobile devices. One of the solutions is that we can design and customize a logical reasoning engine specifically for mobile and embedded devices. This customized reasoner enables logical reasoning to be performed at resource-constraint devices with the tradeoff of limited reasoning functionalities. The customized reasoner can be designed in such a way that it contains a minimum set of core components and some reconfigurable components. Users and applications can choose optional components based on their own requirements. Other solutions such as reasoning agents for mobile and embedded devices can also be considered.

# Bibliography

[1] Henricksen K, Indulska J, Rakotonirainy. An Infrastructure for Pervasive computing: Challenges. In Proceedings of the Workshop on Pervasive computing (INFORMATIK 2001), Viena, September 2001.

[2] Roy Want, Andy Hopper, Veronica Falcao, Jonathon Gibbons. The Active Badge Location System. Journal of ACM Transactions on Information Systems, Vol. 10, No. 1, pp 91-102, January 1992.

[3] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom 1996), November 1996.

[4] Jason I. Hong and James A. Landay. An Infrastructure Approach to Context-Aware Computing. Journal of Human-Computer Interaction , Vol. 16, 2001.

[5] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dartmouth College, November 2000.

[6] R.Hull, P.Neaves and J. Bedford-Roberts. Towards Situated Computing. In Proceedings of the 1st International Symposium on Wearable Computers, Cambridge, October 1997.

[7] P.J. Brown. The Stick-e Document: A Framework for Creating Context-aware Applications. Electronic Publishing, Palo Alto, 1996.

[8] B. Schilit, N. Adams and R. Want. Context-aware Computing Applications. In Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, December 1994.

[9] A. K. Dey. Understanding and Using Context. Journal of Personal and Ubiquitous Computing, 5(1):4-7, February 2001.

[10] Henricksen K, Indulska J, Rakotonirainy A. Modeling Context Information in Pervasive Computing Systems. In Proceedings of the 1st International Conference on Pervasive Computing (Pervasive 2002), Zurich, August 2002.

[11] Wolfgang Kellerer, Anthony Tarlano. Mobile P2P Overlay for Context-Aware Computing. In Dagstuhl Seminar Peer-to-Peer-systems and Applications, March 2004.

[12] http://aire.csail.mit.edu/.

[13] T. Kindberg and J. Barton. A Web-based Nomadic Computing System. Journal of Computer Networks, 35(4):443–456, 2001.

[14] Shilit, B.N. A Context-Aware System Architecture for Mobile Distributed Computing. Ph.D. thesis, Department of Computer Science, Columbia University, 1995.

[15] Dey, A.K., Salber, D. Abowd, G.D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. Anchor article of a special issue on Context-Aware Computing, Journal of Human-Computer Interaction, Vol. 16(2-4), pp. 97-166, 2001.

[16] Guanling Chen and David Kotz. Solar: An Open Platform for Context-Aware Mobile Applications. In Proceedings of the 1st International Conference on Pervasive Computing (Pervasive 2002), Switzerland, June 2002.

[17] Jason Hong. An Architecture for Privacy-Sensitive Ubiquitous Computing. PhD thesis, University of California at Berkeley, Computer Science Division, Berkeley, 2005.

[18] Anand Ranganathan and Roy H. Campbell. A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. In Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil, June 2003.

[19] Harry Chen, Tim Finin, Anupam Joshi. Semantic Web in the Context Broker Architecture. In Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), Orlando FL., March 2004.

[20] T. Gu, H. K. Pung, D. Q. Zhang. A Service-Oriented Middleware for Building Context-Aware Services. Elsevier Journal of Network and Computer Applications (JNCA), Vol. 28, Issue 1, pp. 1-18, January 2005.

[21] Dey, A., Manko., J., Abowd, G. Distributed mediation of imperfectly sensed context in aware environments. Technical Report GIT-GVU-00-14, Georgia Institute of Technology, 2000.

[22] G. Judd, P. Steenkiste. Providing Contextual Information to Pervasive Computing Applications. In Proceedings of the 1st IEEE Conference on Pervasive Computing and Communications (PerCom 2003), Fort Worth, Texas, March 2003.

[23] Hui Lei, Daby M. Sow, John S. Davis, II, Guruduth Banavar and Maria R. Ebling. The design and applications of a context service. Journal of ACM SIGMOBILE Mobile Computing and Communications Review, vol 6, no. 4, pp 44-55, 2002.

[24] Anand Ranganathan, Jalal Al-Muhtadi, Roy H. Campbell. Reasoning about Uncertain Contexts in Pervasive Computing Environments. IEEE Pervasive Computing, pp 62-70, Apr-June 2004 (Vol.3, No 2).

[25] T. Gu, H. K. Pung, D. Q. Zhang. A Bayesian Approach for Dealing with Uncertain Contexts. In Proceedings of the Second International Conference on Pervasive Computing (Pervasive 2004), in the book "Advances in Pervasive Computing" published by the Austrian Computer Society, vol. 176, ISBN 3-85403-176-9, Vienna, Austria, April 2004.

[26] Karen Henricksen, Jadwiga Indulska. A Framework for Context-Aware Pervasive Computing Applications. PhD Thesis, the University of Queensland, September 2003.

[27] Heer, J., A. Newberger, C. Beckmann, and J.I. Hong. Liquid: Context-Aware Distributed Queries. In Proceedings of the 5th International Conference on Ubiquitous Computing (Ubicomp 2003). Springer-Verlag. pp. 140-148, Seattle, Washington, October 2003.

[28] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang. An Ontology-based Context Model in Intelligent Environments. In Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004), San Diego, California, January 2004.

[29] T. Gu, H. K. Pung, and D. Zhang. A Peer-to-Peer Overlay for Context Information Search. In Proceedings of the 14th IEEE International Conference on Computer Communications and Networks (ICCCN 2005), San Diego, California, October 2005.

[30] T. Gu, E. Tan, H. K. Pung, and D. Zhang. A Peer-to-Peer Architecture for Context Lookup. In Proceedings of the International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous 2005), San Diego, California, July 2005.

[31] T. Gu, E. Tan, H. K. Pung, and D. Zhang. ContextPeers: Scalable Peer-to-Peer Search for Context Information. In Proceedings of the International Workshop on Innovations in Web Infrastructure (IWI 2005), in conjunction with the 14th World Wide Web Conference (WWW 2005), Japan, May 2005.

[32] Harry Chen, et al. SOUPA – standard Ontology for Ubiquitous and Pervasive Applications, http://pervasive.semanticweb.org, 2004.

[33] OMG. The Common Object Request Broker: Architecture and Specification Version 3.0.3. OMG Technical Document Number formal/2004-03-12, 2004.

[34] R.V.Guha. rdfDB: An RDF Database. http://guha.com/rdfdb.

[35] RDFStore. http://rdfstore.sourceforge.net.

[36] B. McBride. Jena: Implementing the RDF Model and Syntax specification. In Proceedings of the 2nd International Workshop on the Semantic Web, May 2001.

[37] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Proceedings of the 1st International Semantic Web Conference, Sardinia, Italia, June, 2002.

[38] Napster website. http://www.napster.com.

[39] Gnutella website. http://www.gnutella.com.

[40] Freenet website. http://freenet.sourceforge.net.

[41] KaZaA website. http://www.kazaa.com.

[42] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM 2001), 2001.

[43] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM 2001), 2001.

[44] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. IEEE Journal on Selected Areas in Communications, 22(1):41–53, January 2004.

[45] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. Lecture Notes in Computer Science, 2218:161–172, November 2001.

[46] Min Cai, Martin Frank. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In Proceedings of the 13th International World Wide Web Conference (WWW 2004), New York, May 2004.

[47] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-attribute Addressable Network for Grid Information Services. In Proceedings of 4th International Workshop on Grid Computing, 2003.

[48] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In Proceedings of the 12th International World Wide Web Conference (WWW 2003), Budapest, Hungary, May 2003.

[49] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, and M. Palm. Edutella: A P2P Networking Infrastructure based on RDF. In Proceedings of the 11th International World Wide Web Conference (WWW 2002), Hawaii, USA, May 2002.

[50] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Alexander L¨oser, Ingo Bruckhorst, Mario Schlosser, and Christoph Schmitz. Super-Peer-Based Routing and Clustering Strategies for RDF-based Peer-to-Peer Networks. In Proceedings of the 12th International World Wide Web Conference (WWW 2003), Budapest, Hungary, May 2003.

[51] A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. Technical report. Stanford University.

[52] E. Cohen, A. Fiat, and H. Kaplan. A Case for Associative Peer to Peer Overlays. Journal of ACM SIGCOMM Computer Communication Review, 33(1):95–100, January 2003.

[53] C. Q. Tang, Z. C. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM 2003), Karlsruhe, Germany, August 2003.

[54] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by Latent Semantic Analysis. Journal of the American Society of Information Science, 41(6):391–407, 1990.

[55] M. Li, W. C. Lee, Anand Sivasubramaniam, and D. L. Lee. A Small World Overlay Network for Semantic Based Search in P2P. In Proceedings of the Second Workshop on Semantics in Peer-to-Peer and Grid Computing, in conjunction with the World Wide Web Conference, May, 2004.

[56] J. Kleinberg. The Small-World Phenomenon: an Algorithm Perspective. In Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000.

[57] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems, pp 127-140, Mar 2003.

[58] Tim Berners-Lee, James Hendler, Ora Lassila. The Semantic Web. Scientific American, May 2001.

[59] M.Smith, C. Welty, and D. McGuinness. Web Ontology Language (OWL) Guide. August 2003.

[60] Dan Brickley, R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium, January 2003.

[61] Ian Horrocks. DAML+OIL: a Reasonable Web Ontology Language. In Proceedings of the 8th International Conference on Extending Database Technology (EDBT), Prague, March 2002.

[62] Jena2. http://www.hpl.hp.com/semweb/jena2.htm.

[63] C.L Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence, 1982.

[64] RDQL, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[65] M. Berry, Z. Drmac, and E. Jessup. Matrices, Vector Spaces, and Information Retrieval. Journal of SIAM Review, 41(2):335–362, 1999.

[66] G. Karypis and E. Han. Fast Supervised Dimensionality Reduction Algorithm with Applications to Document Categorization and Retrieval. In Proceedings of the 9th ACM International Conference on Information and Knowledge Management, pp 12-19, New York, US, 2000.

[67] Y. Liu, X. Liu, L. Xiao, L. M. Ni, and X. Zhang. Location-Aware Topology Matching in P2P Systems. In Proceedings of the IEEE Conference on

Computer Communications (INFOCOM 2004), Hong Kong, China, March 2004.

[68] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays using Global Soft-State. In Proceedings of International Conference on Distributed Computing Systems (ICDCS 2003), 2003.

[69] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network Topology Generators: Degree-Based vs. Structural. In Proceedings of the Conference of the Special Interest Group on Data Communication (SIGCOMM 2002), 2002.

[70] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. IEEE Internet Computing, 2002.

[71] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In Proceedings of ACM International Conference on Supercomputing, pages 84–95, June 2002.

[72] B. Yang and H. Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In Proceedings of International Conference on Distributed Computing Systems (ICDCS'02), July 2002.

[73] Morpheus website. http://morpheus.com/.

[74] B. Yang and H. Garcia-Molina. Improving Search in Peer-to-Peer Networks. In Proceedings of International Conference on Distributed Computing Systems (ICDCS'02), July 2002.

[75] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. In Proceedings of International Conference on Distributed Computing Systems (ICDCS'02), July 2002.

[76] Amir Qayyum, Laurent Viennot, and Anis Laouiti. Multipoint Relaying: An Efficient Technique for Flooding in Mobile Wireless Networks. Technical Report RR-3898, INRIA, February 2000.

[77] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In Proceedings of Multimedia Computing and Networking (MMCN 2002), 2002.

[78] BRITE, http://www.cs.bu.edu/brite/.

[79] M. Li, W. C. Lee, and Anand Sivasubramaniam. Semantic Small World: An Overlay Network for Peer-to-Peer Search. In Proceedings of the International Conference on Network Protocols (ICNP), 228-238, October, 2004.

[80] Wang, X., Dong, J.S., Zhang, D., Chin, C.Y., Hettiarachchi, S.R. Semantic Space: An Infrastructure for Smart Spaces. IEEE Pervasive Computing Magazine (2004) 32–39.

## Appendix A: The upper context ontology

```
<rdf:RDF
    xmlns="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xml:base="http://www.comp.nus.edu.sg/socam/ConOnt#">

  <owl:Class rdf:ID="ContextEntity"/>

  <owl:Class rdf:ID="Activity">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>

    <owl:Class rdf:ID="DeducedActivity">
      <rdfs:subClassOf rdf:resource="#Activity"/>
    </owl:Class>

    <owl:Class rdf:ID="ScheduledActivity">
      <rdfs:subClassOf rdf:resource="#Activity"/>
    </owl:Class>

  <owl:Class rdf:ID="CompEntity">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>

    <owl:Class rdf:ID="Application">
      <rdfs:subClassOf rdf:resource="#CompEntity"/>
    </owl:Class>

    <owl:Class rdf:ID="Device">
      <rdfs:subClassOf rdf:resource="#CompEntity"/>
    </owl:Class>

    <owl:Class rdf:ID="Service">
      <rdfs:subClassOf rdf:resource="#CompEntity"/>
    </owl:Class>

  <owl:Class rdf:ID="Location">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>

  <owl:Class rdf:ID="IndoorSpace">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Location"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#OutdoorSpace"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#lightLevel"/>
        </owl:onProperty>
        <owl:allValuesFrom
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
      </owl:Restriction>
```

```
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#noiseLevel"/>
        </owl:onProperty>
        <owl:allValuesFrom
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID="OutdoorSpace">
    <rdfs:subClassOf rdf:resource="#Location"/>
    <owl:disjointWith rdf:resource="#IndoorSpace"/>
  </owl:Class>

  <owl:Class rdf:ID="Merchandise">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>

  <owl:Class rdf:ID="Person">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>

  <owl:Class rdf:ID="Time">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>


  <owl:DatatypeProperty rdf:ID="startTime">
    <rdfs:domain rdf:resource="#Activity"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="endTime">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Activity"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasParticipant">
    <rdfs:domain rdf:resource="#Activity"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="participateIn">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#DeducedActivity"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="locatedIn"
     rdf:type="http://www.w3.org/2002/07/owl#TransitiveProperty">
    <rdfs:domain rdf:resource="#ContextEntity"/>
<!--
    <rdfs:domain rdf:resource="#Activity"/>
    <rdfs:domain rdf:resource="#CompEntity"/>
    <rdfs:domain rdf:resource="#Merchandise"/>
    <rdfs:domain rdf:resource="#Person"/>
```

```
-->
    <rdfs:range rdf:resource="#Location"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="lastLocatedIn"
     rdf:type="http://www.w3.org/2002/07/owl#TransitiveProperty">
    <rdfs:domain rdf:resource="#ContextEntity"/>
<!--
    <rdfs:domain rdf:resource="#Activity"/>
    <rdfs:domain rdf:resource="#CompEntity"/>
    <rdfs:domain rdf:resource="#Merchandise"/>
    <rdfs:domain rdf:resource="#Person"/>
-->
    <rdfs:range rdf:resource="#Location"/>
  </owl:ObjectProperty>

  <owl:DatatypeProperty rdf:ID="spatialContains"
   rdf:type="http://www.w3.org/2002/07/owl#TransitiveProperty">
    <rdfs:domain rdf:resource="#Location"/>
    <rdfs:range rdf:resource="#ContextEntitiy"/>
    <owl:inverseOf rdf:resource="#locatedIn"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="use">
    <rdfs:domain rdf:resource="#Activity"/>
    <rdfs:range rdf:resource="#CompEntity"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="own">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#CompEntity"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

## A set of domain-specified ontologies

```
<!DOCTYPE rdf:RDF [
    <!ENTITY socam "http://www.comp.nus.edu.sg/socam/ConOnt#">
]>

<rdf:RDF
    xmlns="http://www.comp.nus.edu.sg/socam/JohnHome#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xml:base="http://www.comp.nus.edu.sg/socam/JohnHome#">

  <owl:Class rdf:ID="SystemTime">
    <rdfs:subClassOf rdf:resource="&socam;Time"/>
  </owl:Class>

  <owl:Class rdf:ID="SystemDate">
    <rdfs:subClassOf rdf:resource="&socam;Time"/>
  </owl:Class>

  <owl:Class rdf:ID="DeducedQuantity">
    <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
```

```
</owl:Class>

<owl:Class rdf:ID="Showering">
  <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
</owl:Class>

<owl:Class rdf:ID="DeducedMeeting">
  <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
</owl:Class>

<owl:Class rdf:ID="DeducedParty">
  <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
</owl:Class>

<owl:Class rdf:ID="Sleeping">
  <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
</owl:Class>

<owl:Class rdf:ID="WatchingTV">
  <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
</owl:Class>

<owl:Class rdf:ID="ScheduledDinner">
  <rdfs:subClassOf rdf:resource="&socam;ScheduledActivity"/>
</owl:Class>

<owl:Class rdf:ID="ScheduledMeeting">
  <rdfs:subClassOf rdf:resource="&socam;ScheduledActivity"/>
</owl:Class>

<owl:Class rdf:ID="ScheduledParty">
  <rdfs:subClassOf rdf:resource="&socam;ScheduledActivity"/>
</owl:Class>

<owl:Class rdf:ID="JBuilder">
  <rdfs:subClassOf rdf:resource="&socam;Application"/>
</owl:Class>

<owl:Class rdf:ID="PowerPoint">
  <rdfs:subClassOf rdf:resource="&socam;Application"/>
</owl:Class>

<owl:Class rdf:ID="RealPlayer">
  <rdfs:subClassOf rdf:resource="&socam;Application"/>
</owl:Class>

<owl:Class rdf:ID="AlarmClock">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="DVDPlayer">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="Fridge">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="Light">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Phone">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="Projector">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="TV">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="WaterHeater">
  <rdfs:subClassOf rdf:resource="&socam;Device"/>
</owl:Class>

<owl:Class rdf:ID="Building">
  <rdfs:subClassOf rdf:resource="&socam;IndoorSpace"/>
</owl:Class>

<owl:Class rdf:ID="Door">
  <rdfs:subClassOf rdf:resource="&socam;IndoorSpace"/>
</owl:Class>

<owl:Class rdf:ID="Room">
  <rdfs:subClassOf rdf:resource="&socam;IndoorSpace"/>
</owl:Class>

<owl:Class rdf:ID="Yard">
  <rdfs:subClassOf rdf:resource="&socam;OutdoorSpace"/>
</owl:Class>

<owl:Class rdf:ID="Adult">
  <rdfs:subClassOf rdf:resource="&socam;Person"/>
  <owl:disjointWith rdf:resource="&socam;Child"/>
  <owl:disjointWith rdf:resource="&socam;Elderly"/>
</owl:Class>

<owl:Class rdf:ID="Child">
  <rdfs:subClassOf rdf:resource="&socam;Person"/>
  <owl:disjointWith rdf:resource="&socam;Elderly"/>
  <owl:disjointWith rdf:resource="&socam;Adult"/>
</owl:Class>

<owl:Class rdf:ID="Elderly">
  <rdfs:subClassOf rdf:resource="&socam;Person"/>
  <owl:disjointWith rdf:resource="&socam;Child"/>
  <owl:disjointWith rdf:resource="&socam;Adult"/>
</owl:Class>

<owl:Class rdf:ID="Milk">
  <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
</owl:Class>

<owl:Class rdf:ID="Eggs">
  <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
</owl:Class>

<owl:Class rdf:ID="Cereals">
  <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
```

```
  </owl:Class>

  <owl:Class rdf:ID="Paper">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:DatatypeProperty rdf:ID="altitude">
    <rdfs:domain rdf:resource="&socam;Location"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="humidity">
    <rdfs:domain rdf:resource="&socam;Location"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="latitude"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:domain rdf:resource="&socam;Location"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="longitude">
    <rdfs:domain rdf:resource="&socam;Location"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="temperature"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
    <rdfs:domain rdf:resource="&socam;Location"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="weather">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;OutdoorSpace"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="windSpeed">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;OutdoorSpace"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="lightLevel">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;IndoorSpace"/>
    <socam:rangevalue>OFF</socam:rangevalue>
    <socam:rangevalue>LOW</socam:rangevalue>
    <socam:rangevalue>MEDIUM</socam:rangevalue>
    <socam:rangevalue>HIGH</socam:rangevalue>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="noiseLevel">
```

```
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;IndoorSpace"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="curtainStatus">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Room"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="doorStatus">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <socam:rangevalue>CLOSED</socam:rangevalue>
    <socam:rangevalue>OPEN</socam:rangevalue>
    <rdfs:domain rdf:resource="#Room"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="numberOfPersons">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Room"/>
    <socam:rangevalue>0</socam:rangevalue>
    <socam:rangevalue>1</socam:rangevalue>
    <socam:rangevalue>2</socam:rangevalue>
    <socam:rangevalue>3</socam:rangevalue>
    <socam:rangevalue>4</socam:rangevalue>
    <socam:rangevalue>5</socam:rangevalue>
    <socam:rangevalue>6</socam:rangevalue>
    <socam:rangevalue>7</socam:rangevalue>
    <socam:rangevalue>8</socam:rangevalue>
    <socam:rangevalue>9</socam:rangevalue>
    <socam:rangevalue>10</socam:rangevalue>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="roomActivity">
      <rdfs:domain rdf:resource="#Room"/>
    <rdfs:range rdf:resource="&socam;DeducedActivity"/>
  </owl:ObjectProperty>

  <owl:DatatypeProperty rdf:ID="windowStatus">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Room"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="deviceStatus">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <socam:rangevalue>ON</socam:rangevalue>
    <socam:rangevalue>OFF</socam:rangevalue>
    <rdfs:domain rdf:resource="&socam;Device"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasInterval">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#AlarmClock"/>
  </owl:DatatypeProperty>
```

```
  <owl:DatatypeProperty rdf:ID="phoneStatus">
      <rdfs:subPropertyOf rdf:resource="#deviceStatus"/>
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <socam:rangevalue>IDLE</socam:rangevalue>
    <socam:rangevalue>RINGING</socam:rangevalue>
    <rdfs:domain rdf:resource="#Phone"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="age"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:domain rdf:resource="&socam;Person"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="birthday">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="firstName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="fullName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="gender">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="homeAddress">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:ID="instantMessageID">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="lastName"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="middleName">
```

```
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="mobilePhoneNum">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="nickName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="personStatus">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="RFID"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="role">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="title">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Person"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="deviceID"
     rdf:type="http://www.w3.org/2002/07/owl#FunctionalProperty">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Device"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="deviceName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Device"/>
  </owl:DatatypeProperty>

   <owl:DatatypeProperty rdf:ID="hasTime">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#SystemTime"/>
    <socam:rangevalue>MORNING</socam:rangevalue>
    <socam:rangevalue>AFTERNOON</socam:rangevalue>
```

```
    <socam:rangevalue>EVENING</socam:rangevalue>
    <socam:rangevalue>NIGHT</socam:rangevalue>
  </owl:DatatypeProperty>

   <owl:DatatypeProperty rdf:ID="hasDate">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#SystemDate"/>
    <socam:rangevalue>Jan01</socam:rangevalue>
    <socam:rangevalue>Jan02</socam:rangevalue>
    <socam:rangevalue>Feb01</socam:rangevalue>
    <socam:rangevalue>Mar01</socam:rangevalue>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandiseID">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandiseName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandisePrice">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasInventoryQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
    <socam:rangevalue>0</socam:rangevalue>
    <socam:rangevalue>1</socam:rangevalue>
    <socam:rangevalue>2</socam:rangevalue>
    <socam:rangevalue>3</socam:rangevalue>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasRequiredQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="hasQuantityToBuy">
    <rdfs:range rdf:resource="#DeducedQuantity"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

## Appendix B: User-defined rules in the SmartHome application

```
[userStatus_SLEEP:(?user  rdf:type
http://www.comp.nus.edu.sg/socam/ConOnt#Person),(?user,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn,
http://www.comp.nus.edu.sg/socam/JohnHome#Bedroom),(http://www.comp.n
us.edu.sg/socam/JohnHome#Bedroom, rdf:type
http://www.comp.nus.edu.sg/socam/JohnHome#Room),(http://www.comp.nus.
edu.sg/socam/JohnHome#Bedroom,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn,http://www.comp.nus
.edu.sg/socam/JohnHome#Home),
(http://www.comp.nus.edu.sg/socam/JohnHome#Bedroom,
http://www.comp.nus.edu.sg/socam/JohnHome#lightLevel, 'LOW') ->
(?user  http://www.comp.nus.edu.sg/socam/ConOnt#participateIn
'SLEEPING')]

[userStatus_WATCHING_TV:(?user  rdf:type
http://www.comp.nus.edu.sg/socam/ConOnt#Person), (?user,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#DVDPlayer,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#TV,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#TV,
http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus
'ON'),(http://www.comp.nus.edu.sg/socam/JohnHome#DVDPlayer,
http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus 'OFF')  ->
(?user  http://www.comp.nus.edu.sg/socam/ConOnt#participateIn
'WATCHING_TV')]

[userStatus_WATCHING_MOVIE:(?user  rdf:type
http://www.comp.nus.edu.sg/socam/ConOnt#Person), (?user,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#DVDPlayer,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#TV,
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn, ?room),(http://www
.comp.nus.edu.sg/socam/JohnHome#TV,
http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus
'ON'),(http://www.comp.nus.edu.sg/socam/JohnHome#DVDPlayer,
http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus 'ON')  ->
(?user  http://www.comp.nus.edu.sg/socam/ConOnt#participateIn
'WATCHING_MOVIE')]

[roomActivity_FAMILY_BREAKFAST:(http://www.comp.nus.edu.sg/socam/John
Home#John http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Mary
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Tom
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#DiningRoom
http://www.comp.nus.edu.sg/socam/JohnHome#numberOfPersons
'3'),(http://www.comp.nus.edu.sg/socam/JohnHome#CurrentTime
http://www.comp.nus.edu.sg/socam/JohnHome#hasTime 'MORNING') ->
(http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom
```

```
http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity
'FAMILY_BREAKFAST')]

[roomActivity_FAMILY_LUNCH:(http://www.comp.nus.edu.sg/socam/JohnHome
#John http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Mary
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Tom
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#DiningRoom
http://www.comp.nus.edu.sg/socam/JohnHome#numberOfPersons
'3'),(http://www.comp.nus.edu.sg/socam/JohnHome#CurrentTime
http://www.comp.nus.edu.sg/socam/JohnHome#hasTime 'AFTERNOON') ->
(http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom
http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity
'FAMILY_LUNCH')]

[roomActivity_FAMILY_DINNER:(http://www.comp.nus.edu.sg/socam/JohnHom
e#John http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Mary
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#Tom
http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn
http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom),(http://www.com
p.nus.edu.sg/socam/JohnHome#DiningRoom
http://www.comp.nus.edu.sg/socam/JohnHome#numberOfPersons
'3'),(http://www.comp.nus.edu.sg/socam/JohnHome#CurrentTime
http://www.comp.nus.edu.sg/socam/JohnHome#hasTime 'EVENING') ->
(http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom
http://www.comp.nus.edu.sg/socam/JohnHome#roomActivity
'FAMILY_DINNER')]
```

## Appendix C: An example of domain-specific context ontologies such as grocery store, book store and child care center used in the ShoppingAssistant application

### The ontology for grocery store

```
<!DOCTYPE rdf:RDF [
    <!ENTITY socam "http://www.comp.nus.edu.sg/socam/ConOnt#">
]>

<rdf:RDF
    xmlns="http://www.comp.nus.edu.sg/socam/GroceryStore#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xml:base="http://www.comp.nus.edu.sg/socam/GroceryStore#">

  <owl:Class rdf:ID="Adult">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
  </owl:Class>

  <owl:Class rdf:ID="Child">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Elderly">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Eggs">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="Milk">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="Cereals">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="DeducedQuantity">
    <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
  </owl:Class>

  <owl:DatatypeProperty rdf:ID="merchandiseID">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
```

```
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandiseName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandisePrice">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasInventoryQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <socam:rangevalue>0</socam:rangevalue>
    <socam:rangevalue>1</socam:rangevalue>
    <socam:rangevalue>2</socam:rangevalue>
    <socam:rangevalue>3</socam:rangevalue>
    <socam:rangevalue>4</socam:rangevalue>
    <socam:rangevalue>5</socam:rangevalue>
    <socam:rangevalue>6</socam:rangevalue>
    <socam:rangevalue>7</socam:rangevalue>
    <socam:rangevalue>8</socam:rangevalue>
    <socam:rangevalue>9</socam:rangevalue>
    <socam:rangevalue>10</socam:rangevalue>
    <socam:rangevalue>11</socam:rangevalue>
    <socam:rangevalue>12</socam:rangevalue>
    <socam:rangevalue>24</socam:rangevalue>
    <socam:rangevalue>36</socam:rangevalue>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasRequiredQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="hasQuantityToBuy">
    <rdfs:range rdf:resource="#DeducedQuantity"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

### The ontology for book store

```
<!DOCTYPE rdf:RDF [
    <!ENTITY socam "http://www.comp.nus.edu.sg/socam/ConOnt#">
]>

<rdf:RDF
    xmlns="http://www.comp.nus.edu.sg/socam/BookStore#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xml:base="http://www.comp.nus.edu.sg/socam/BookStore#">
```

```
  <owl:Class rdf:ID="Adult">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
  </owl:Class>

  <owl:Class rdf:ID="Child">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Elderly">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Paper">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="DeducedQuantity">
    <rdfs:subClassOf rdf:resource="&socam;DeducedActivity"/>
  </owl:Class>

  <owl:DatatypeProperty rdf:ID="merchandiseID">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandiseName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandisePrice">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasInventoryQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <socam:rangevalue>0</socam:rangevalue>
    <socam:rangevalue>1</socam:rangevalue>
    <socam:rangevalue>2</socam:rangevalue>
    <socam:rangevalue>3</socam:rangevalue>
    <socam:rangevalue>4</socam:rangevalue>
    <socam:rangevalue>5</socam:rangevalue>
    <socam:rangevalue>6</socam:rangevalue>
    <socam:rangevalue>7</socam:rangevalue>
    <socam:rangevalue>8</socam:rangevalue>
    <socam:rangevalue>9</socam:rangevalue>
    <socam:rangevalue>10</socam:rangevalue>
    <socam:rangevalue>11</socam:rangevalue>
    <socam:rangevalue>12</socam:rangevalue>
    <socam:rangevalue>24</socam:rangevalue>
    <socam:rangevalue>36</socam:rangevalue>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
```

```
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasRequiredQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="hasQuantityToBuy">
    <rdfs:range rdf:resource="#DeducedQuantity"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

**The ontology for child care center**

```
<!DOCTYPE rdf:RDF [
    <!ENTITY socam "http://www.comp.nus.edu.sg/socam/ConOnt#">
]>

<rdf:RDF
    xmlns="http://www.comp.nus.edu.sg/socam/ChildCare#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:socam="http://www.comp.nus.edu.sg/socam/ConOnt#"
    xml:base="http://www.comp.nus.edu.sg/socam/ChildCare#">

  <owl:Class rdf:ID="Adult">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
  </owl:Class>

  <owl:Class rdf:ID="Child">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Elderly"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Elderly">
    <rdfs:subClassOf rdf:resource="&socam;Person"/>
    <owl:disjointWith rdf:resource="&socam;Child"/>
    <owl:disjointWith rdf:resource="&socam;Adult"/>
  </owl:Class>

  <owl:Class rdf:ID="Milk">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="Eggs">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="Paper">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
  </owl:Class>

  <owl:Class rdf:ID="Cereals">
    <rdfs:subClassOf rdf:resource="&socam;Merchandise"/>
```

```
  </owl:Class>

  <owl:DatatypeProperty rdf:ID="merchandiseID">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandiseName">
    <rdfs:range
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="merchandisePrice">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasInventoryQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
    <socam:rangevalue>0</socam:rangevalue>
    <socam:rangevalue>1</socam:rangevalue>
    <socam:rangevalue>2</socam:rangevalue>
    <socam:rangevalue>3</socam:rangevalue>
    <socam:rangevalue>4</socam:rangevalue>
    <socam:rangevalue>5</socam:rangevalue>
    <socam:rangevalue>6</socam:rangevalue>
    <socam:rangevalue>7</socam:rangevalue>
    <socam:rangevalue>8</socam:rangevalue>
    <socam:rangevalue>9</socam:rangevalue>
    <socam:rangevalue>10</socam:rangevalue>
    <socam:rangevalue>11</socam:rangevalue>
    <socam:rangevalue>12</socam:rangevalue>
    <socam:rangevalue>24</socam:rangevalue>
    <socam:rangevalue>36</socam:rangevalue>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:ID="hasRequiredQuantity">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:DatatypeProperty>

  <owl:ObjectProperty rdf:ID="hasQuantityToBuy">
    <rdfs:range rdf:resource="#DeducedQuantity"/>
    <rdfs:domain rdf:resource="&socam;Merchandise"/>
  </owl:ObjectProperty>

</rdf:RDF>
```

## Appendix D: Sample queries used in prototype evaluations

### Non-deduced queries:

```
SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/JohnHome#mobilePhoneNum> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/JohnHome#role> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/JohnHome#birthday> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/JohnHome#age> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/JohnHome#mobilePhoneNum> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/JohnHome#role> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/JohnHome#birthday> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/JohnHome#age> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/ConOnt#locatedIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/JohnHome#mobilePhoneNum> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/JohnHome#role> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/JohnHome#birthday> ?x)
```

```
SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/JohnHome#age> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom>
<http://www.comp.nus.edu.sg/socam/JohnHome#lightLevel> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Bedroom>
<http://www.comp.nus.edu.sg/socam/JohnHome#lightLevel> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#LivingRoom>
<http://www.comp.nus.edu.sg/socam/JohnHome#lightLevel> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Bathroom>
<http://www.comp.nus.edu.sg/socam/JohnHome#lightLevel> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom>
<http://www.comp.nus.edu.sg/socam/JohnHome#numberOfPersons> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#MainDoor>
<http://www.comp.nus.edu.sg/socam/JohnHome#doorStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#CurrentTime>
<http://www.comp.nus.edu.sg/socam/JohnHome#hasTime> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#CurrentDate>
<http://www.comp.nus.edu.sg/socam/JohnHome#hasDate> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#MobilePhone>
<http://www.comp.nus.edu.sg/socam/JohnHome#phoneStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#FixedLinePhone>
<http://www.comp.nus.edu.sg/socam/JohnHome#phoneStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#WaterHeater>
<http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#ElectricOven>
<http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#TV>
<http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#DVDPlayer>
<http://www.comp.nus.edu.sg/socam/JohnHome#deviceStatus> ?x)
```

**Deduced queries:**

```
SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#John>
<http://www.comp.nus.edu.sg/socam/ConOnt#participateIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Mary>
<http://www.comp.nus.edu.sg/socam/ConOnt#participateIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#Tom>
<http://www.comp.nus.edu.sg/socam/ConOnt#participateIn> ?x)

SELECT ?x WHERE
(<http://www.comp.nus.edu.sg/socam/JohnHome#DiningRoom>
<http://www.comp.nus.edu.sg/socam/ConOnt#roomActivity> ?x)
```