

A CONTEXT MANAGEMENT FRAMEWORK
TO SUPPORT CONTEXT-AWARE PERVASIVE COMPUTING

By

CYRUS BOADWAY

A thesis submitted to the School of Computing
in conformity with the requirements for the degree of Master of Science

Queen's University
Kingston, Ontario, Canada

September, 2012

Copyright © Cyrus Boadway, 2012

Abstract

As computing devices become more pervasive and integrated with our lives, it becomes useful for these devices to understand the environment which surrounds them and the activities of their users, so that they may adapt themselves to suit the circumstances and best assist their users. To accomplish this, it is necessary that these devices have a model for expressing and sharing contextual data which characterizes the state of the surroundings, the devices and people in an environment, and the relationships between each.

This thesis describes a framework for managing the exchange of contextual data between devices and sensors with information about their shared environment, with other devices whose goals are to adapt their behaviour to best suit the circumstances. This function supports the greater goal of implementing a pervasive computing environment. We identify three roles in context sharing: Context Providers which are sources of contextual data such as sensors; Context Consumers which are applications which use contextual data to adapt themselves; and Context Hosts which act as intermediaries facilitating the exchange of data and provide a proxy host for contextual data on behalf of Context Providers. This framework includes protocols to facilitate data exchange between each with considerations for the security and data privacy.

This thesis also outlines an extensible model for defining and expressing contextual data and related meta data. The context model can be passed among the context sharing actors to communicate state and relationships between entities in the working environment.

Acknowledgements

I would like to express my sincere thanks to my advisor, Dr. Patrick Martin, for giving me guidance, for his patience, and for giving me the opportunity to be a part of the Database Systems Laboratory. Thanks also to Wendy Powley for her support and fierce encouragement, through to the end of my degree. Thank you both for helping me through difficult times and for helping me to find direction.

Thanks also to my family for their unending support and to my friends for their encouragement, blind faith, and threats of various kinds, all of which contributed to pushing me to the finish line.

Table of Contents

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Figures.....	vii
Chapter 1 Introduction	1
1.1 Motivation	1
1.1.1 Objective	2
1.2 Thesis Contributions.....	3
1.3 Thesis Organization	4
Chapter 2 Previous Work	6
2.1 Existing Frameworks	6
2.1.1 The Context Toolkit (1999).....	6
2.1.2 Jini (2002)	7
2.1.3 Reconfigurable Context-Sensitive Middleware (RCSM) (2002).....	8
2.1.4 Semantic Task Execution Editor (STEER)(2003).....	9
2.1.5 BASE (2003)	10
2.1.6 Context Broker Architecture (CoBrA)(2003)	12
2.1.7 Tuples On The Air (TOTA)(2004)	13

2.1.8	SOCAM (2004).....	14
2.1.9	PersonisAD (2007).....	15
2.1.10	ECSTRA – Distributed Context Reasoning Framework (2011).....	16
2.1.11	Agent-Based Context Aware Information Sharing for Ambient Intelligence (2011).....	17
2.1.12	Ambitalk.....	18
2.2	Shortcomings of Existing Frameworks	19
Chapter 3	Context Sharing Framework	21
3.1	Context Framework Roles	22
3.1.1	Role Overlap and Privacy Issues.....	25
3.2	Context Model	26
3.2.1	Context expression requirements.....	28
3.2.2	Ontology in XML.....	29
3.3	Security Features	39
3.3.1	Public Key Encryption.....	39
3.3.2	Letter of Mark	40
3.3.3	Man in the Middle Attacks.....	43
3.4	Message Passing Scheme	45
Chapter 4	Implementation and Testing.....	47
4.1	Implementation	47
4.1.1	Network Layer.....	48
4.1.2	Service Discovery Layer.....	49
4.1.3	Service Layer	51
4.1.4	Actor Internal Architecture	52
4.2	Use Cases	57

4.2.1	Use Case 1: An Adaptive Home Theatre	57
4.2.2	Use Case 2: Cellphone Desktop Notifications	62
4.3	Testing	66
4.3.1	Query Performance	66
4.3.2	Client Requests via TCP/IP Network Sockets	68
4.3.3	Bluetooth Network Query Performance	73
4.3.4	Context Service Discovery, IP Network and Bluetooth	74
4.3.5	Leaving the Situation.....	75
4.4	Experimental Conclusions	77
Chapter 5	Conclusion and Future Work.....	78
5.1	Thesis Summary	78
5.2	Future Work.....	80
References.....		83
Appendix A	Context Schemas	89
Appendix B	The Situation Document	104
Appendix C	Implementation Source and Design.....	108

List of Figures

Figure 3-1 Illustration of context sharing roles.....	23
Figure 3-2 Context Actors and their interactions	24
Figure 3-3 Private key sharing decryption process.....	41
Figure 3-4 Encrypted data sharing decryption process	42
Figure 3-5 “Letter of Mark” decryption process.....	43
Figure 3-6 An illustration of the message interactions between actors in a typical Situation.	46
Figure 4-1 Host Architecture	53
Figure 4-2: The relationships between the Entities and Hosts of the scenario.	58
Figure 4-3: State for the context elements of three devices.	61
Figure 4-4 Throughput for a local scope query, with bars showing 1 standard deviation	69
Figure 4-5 Query response time, locally scoped query, with bars showing 1 standard deviation	70
Figure 4-6 Host throughput for multi-Host Situations, with bars showing 1 standard deviations.....	71
Figure 4-7 Query Speed for multi-Host Situations, with bars showing 1 standard deviation.....	72

Chapter 1 Introduction

The computing age has been a time of constant innovation, with computers becoming increasingly embedded in our daily lives and interactions. Increasingly cheap and efficient manufacturing processes have made computers affordable and available. These devices are inter-connected through a myriad of technologies and are increasingly equipped with sensors that allow them to better understand the environment in which they are operating. These heterogeneous sources of information, properly leveraged, allow devices to better understand their environment and interpret the goals of their users, giving them the opportunity to modify themselves to more readily achieve those goals.

The idea of applications and devices sharing information in a way that allows them to dynamically and autonomously reconfigure themselves to changing contextual circumstances, remains a vision. Some implementations of frameworks and middleware exist, but each is insufficient in some fashion. This thesis describes the design and implementation of a framework that provides applications with a mechanism by which to share context across devices.

1.1 Motivation

In 1991, the head of Xerox PARC, Mark Weiser, defined his vision of the future of computing [1]. In it, he foresaw devices receding into the environment and becoming indistinguishable from it. These devices would interact with each other and the environment and change themselves to best assist the users in their goals. Distinct from the “intimate computer”, which a user would direct with explicit commands to accomplish tasks, these “ubiquitous devices” would infer the correct action and perform it in such a way that the user felt they were responsible for the action [2].

Schilit et al. [3] refined this vision by asserting that essential components of the paradigm were the concepts of “context” and “context-awareness”. For applications to adapt themselves effectively and usefully, they must have an understanding of the circumstances in which they are operating; circumstances that are dynamic and composed from heterogeneous sources. This understanding comes from a device’s interpretation of data from sensors to which it is attached, acquired from other devices, or inferred from other data. Informally, this data can be thought of as context, information that characterizes the circumstances within the operating environment. A context aware device would be able to interpret and act upon context it has acquired to deliver its services in a fashion tailored to the users and the users’ goals.

The visions of Weiser and Schilit were of the future, predicated on technology and theory not then realised. Their envisioned technology is becoming available today. Network-attached and adaptive home theatre systems[4], thermostats [5] and lights[6], and Smart Phones are common features of a modern home [7], each technologically capable of communicating with each other over various network types.

A few wireless networking technologies are widely included in common consumer electronics. In particular, Bluetooth and Wi-Fi networking each offer the necessary means for inter-device communication. What still remains to be ratified to achieve the vision of context-aware pervasive computing includes a context language that is expressive and open to extension and a protocol and infrastructure to distribute and share that context in a secure and time-sensitive fashion.

1.1.1 Objective

The objective of our research is to develop a platform for context management supporting pervasive computing through the secure sharing of context data in a decentralised fashion, in a shared environment. Via this framework, applications would be able to adapt themselves to dynamic context

provided from heterogeneous sources occupying the same environment. The context expression must take a form that assures the context producer's intended meaning is not mistaken by the consumer.

1.2 Thesis Contributions

This thesis leverages well known and standardized technologies to implement a context sharing framework which allows applications to adapt themselves to a contextually dynamic environment. Specifically, the contributions of this thesis can be broken down into two categories. First, this thesis defines a flexible language by which context can be expressed. Second, it defines a framework that allows this context to be shared between applications that wish to consume context and sources of context.

The proposed context language uses XML to express context in a way that is communicable, flexible, and extensible, and expressive. It is communicable in that its use of XML namespaces ensures that the producer of a contextual datum is assured that its consumer will not misclassify the datum as having another meaning. It is flexible in its ability to express concepts in multiple ways, allowing varied granularity in the expression of context, or even different ways of expressing the same idea, such as a location being expressed as both a street address and by the longitude-latitude geographic coordinate system. The granularity can even be restricted on a per-consumer basis, providing privacy controls. The language is extensible, meaning that basic contextual data can be augmented with specific refinements not prescribed by the base ontology. For example, a general "printer", which provides context related to the location and printer queue length, may also include an extension to describe further features such as the list of jobs that have been completed and those still in queue. The context language does not simply express state values, but is ontological, allowing for the expression of relationships between various context data, for example relating a device to its owner or the set of users currently logged into a system.

The proposed framework allows various sources of context to make this data available to interested consumers. Our framework provides decentralized and secure transmission from sources of context to context-aware applications. Bluetooth capable devices and devices attached to IP network infrastructure can participate in context sharing. By separating the tasks of context sharing into three components, Context Producers, Context Consumers and Hosts, devices with limited resources can rely on those with more resources to share their context as a form of proxy. The use of RSA public key cryptography as a central feature of the framework ensures that privileged information is accessible only to authorized consumers. Decentralization is provided through service discovery technologies Domain Name Service – Service Discovery (DNS-SD)[8], for IP networks and Device and Service Discovery Protocols [9] for Bluetooth.

While techniques for interpreting the context data through XQuery are described, the policies that relate a situation’s contextual state to intended action are outside the scope of this thesis. Example implementations of context aware applications which monitor and react to context are made herein, but their policies for adaptation are simply demonstrative.

The developed technology described in this thesis constitutes a complete solution for context sharing in pervasive computing environments. Implementations of this technology allow applications to become context aware, which properly leveraged, could provide an improved user experience.

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter two provides a general overview of context aware pervasive computing, and contains observations regarding the deficiencies of work related to this topic. A survey of the currently available technologies makes clear the need for a flexible common language to describe context, and an accompanying set of technologies to discover and share that context. Chapter three describes our context sharing framework. The design of the context

description language is described in detail with several examples illustrating its versatility. The actors involved in the sharing context are defined and protocols for context exchanges between them outlined, including provisions for security and query and context proxying. Chapter four outlines the implementation of the framework described in Chapter three, putting the described technologies into action. Several implemented use case scenarios which illustrate the features of the context description language and the various features of the framework are described. Finally, this chapter includes the results of a battery of tests which have been used to profile the performance characteristics of the implementation, paying particular attention to network responsiveness and probing the upper limitations of the Host context services. Chapter five summarizes the thesis and presents a discussion of the design's limitations, modifications that might be made to address these limitations, and future work.

Chapter 2 Previous Work

In this section, several implementations of pervasive computing frameworks are evaluated. Each is shown to lack provisions for essential component of a pervasive computing platform.

2.1 Existing Frameworks

2.1.1 The Context Toolkit (1999)

The designers of the *Context Toolkit* [10] recognized the importance of context and the difficulty of generalizing the presentation of context. The Context Toolkit provides a consistent interface to context consuming applications. The framework was designed to address several difficulties in handling context:

1. Context comes from heterogeneous sources, which results in a wide range of raw data types.
2. For applications to make use of context, it must be abstracted, so that information can be understood in many different ways. For example location may be GPS coordinates or a room location.
3. Devices must be able to acquire data from a variety of sources, not just locally attached sensors.
4. Context is mutable, and thus the architecture must support dynamic context.

Inspired by the manner in which “GUI toolkits insulate the application from interaction details handled by widgets, the context toolkit insulates the application from context sensing mechanics

through widgets.” These context widgets encapsulate the functionality of sensors and present a standard interface with which applications can interact.

A widget maintains a set of attributes which are remotely accessible by the interface. The format of the message passing is ASCII text in XML formatting. Messages are passed over a pre-existing TCP/IP network. The architecture does not provide a discovery mechanism, but it is referred to as further work. As well, a subscription/notification service is provided, to address the requirement for dynamic context.

2.1.2 Jini (2002)

Gupta et al [11] presented a pervasive computing middleware called Jini that relies on a Java remote code execution (code portability) system to provide pervasiveness. Their motivation was that in their view, to make the system truly flexible, the service consumers would need a way to access a service provider without any prior knowledge of the service’s requisite communication methods or protocols. Service providers therefore provide a section of code that a consumer downloads and executes. The downloaded code, a serialized object, contains all the relevant instructions to access the service provider correctly, and makes the resource available to the consumer via standard access methods.

In the Jini architecture, a service provider (or context provider) that wishes to make available a service registers itself with the Jini lookup service. It does so by providing a “proxy object”, which is a Java based interface object that contains the functionality to query and access the service. A device wanting to use a service queries the Jini lookup server for the availability of services. The lookup service returns a relevant proxy object. The device then uses the object to satisfy its context query.

Java is a resource intensive environment. To accommodate resource poor devices, Jini allows for surrogate hosting, in which a simple device communicates with a surrogate host via a private

protocol. The host in turn manages the communication and device registration with the rest of the federated pervasive devices. As well, particular attention is paid to moderately powerful devices by providing a middleware for wireless devices supporting JavaME, the Java Virtual Machine (Micro Edition)[12] which is widely implemented on PDAs and other limited devices.

2.1.3 Reconfigurable Context-Sensitive Middleware (RCSM) (2002)

A pervasive computing platform Reconfigurable Context-Sensitive Middleware (RCSM) was proposed by Yau et al. [13] to satisfy what they suggest are two fundamental characteristics of pervasive computing: context sensitivity and ad-hoc communication. The authors first propose a Context Aware Interface Description Language (CA-IDL) in which context applications may express what contextual information is required. An Adaptive Object Container (ADC) is compiled to be an executable interface produced from instructions arranged in CA-IDL. CA-IDL can be compiled to create objects for many major programming languages, including Java, C++, and C#.

CA-IDL can include a combination of state variables and context sensitive methods whose output is dependent on a combination of the objects' states. For example, state variables may include a room's lights as On and Off. When the state transitions from On to Off, a projector is instructed to begin showing content.

The design also includes a Reconfigurable Context-Sensitive Middleware (RCSM) Object Request Broker (R-ORB) which is responsible for hiding the networking and service/device discovery features from the application and delivering relevant context from a device's sensors or from other devices to all of the local applications. When an R-ORB is presented with new raw contextual information from a sensor or another device, it looks for pervasive applications whose ADCs match the context and updates their contextual model.

The design also features Ephemeral Groups, through which users can dynamically form collections where information can be more easily shared. The devices in such a group provide context to each other not accessible to other groups. Members can dynamically join or leave the groups. The utility of the design is illustrated with an example in which students are broken into working groups during a class. The teacher can walk around the room and join the groups in turn, contributing to the work of the group.

2.1.4 Semantic Task Execution Editor (STEER)(2003)

Masuoka et al. [14] describe their solution to their pervasive computing paradigm. It is a design based on Web Services as the resource distributing mechanism with a separate technology providing the decentralized networking.

The proposed design is described as a Task Computing Environment (TCE), which is an environment in which the functionality of an environment (the device functionality or third party functionality) is delivered as Web Services. Any device can access any device's features and make use of them in a consistent fashion.

A Task Computing Environment is composed of several components, the first of which is the Semantic Task Execution Editor (STEER). STEER is the main user interface of the environment, allowing a user to interact with the various available services. The user is presented with the available services, categorised by the inputs and outputs which the services expose.

The second component of the architecture is called White Hole. If a user is interested in having a file interact with a service, such as having a presentation displayed on an overhead projector, a user can have a Web Service composed based on that file. From within STEER a user can compose complex Web Services by matching a set of inputs of one service with the outputs of another. In the example

above, a user would match the presentation file's Web Service outputs with the inputs of the overhead projector's Web Service.

The third component is the Pervasive Instance Provision Environment (PIPE). This is a Web Service managing the user environment, where the device's available services are published as local or pervasive. As well, this environment includes the ability to "save" pervasive Web Services for future reference, if they are of particular interest.

This architecture includes a device and service discovery mechanism based on Universal Plug and Play (UPnP). This technology allows for peer-to-peer networking and dynamic discovery.

The Web Ontology Language for Semantic Web Services (OWL-S, formerly DAML-S)[15] is an important part of the architecture. The semantic web is an initiative that is looking to add hints to online content that will allow autonomous access and understanding. OWL-S provides the ontological requirements for allowing the purpose of a web service to be autonomously understood.

OWL-S extends the Resource Description Language to allow for the "natural description of many subject domains." This feature is particularly useful in making associations between the various web services for complex service composition, but the authors do not describe a complete ontology for pervasive devices.

2.1.5 BASE (2003)

Becker et al.[16] propose a micro-broker based middleware as a solution to pervasive computing, which they call BASE. The authors were interested in creating a middleware that is able to adapt to dynamically changing communicating media and accommodate devices of various abilities. In particular, they provide a modular approach in which device capabilities are encapsulated in plug-ins, with which applications indirectly interact.

In the BASE architecture, each device has a set of applications that require access to context. Many devices also have context sensors, i.e. data sources that can be made available to applications on this device as well as others. Applications request contextual information via a micro-broker. The micro-broker either accesses the local sensor via a sensor-specific plug-in, or if the information is contained on another device, requests the information via a transport protocol plug-in. The transport plug-in communicates with the other device's plug-in, which in turn makes a context request to that device's micro-broker which queries the local sensor and makes a reply to the originating micro-broker. Requests and replies need not be made via the same transport plug-in. The application is then presented with the result by the micro-broker.

The devices communicate with what are referred to as invocations. These invocations maintain unique message ids, sender and receiver ids, and service ids, as well as a data payload. As such, the micro-brokers are able to queue incoming requests and process them sequentially.

The authors account for the dynamic nature of the network by providing a lookup service. Each device maintains a list of nearby devices and a list of locally available services. If a request of a service cannot be satisfied locally, the nearby devices are queried. Services are queried either by name or by functional properties, which is to say a service object's interface.

The authors cite several advantages of the design. Foremost, their middleware allows for spontaneous context exchange over any set of media. This is an important feature when the devices communicating are in a dynamically changing environment in which some protocols may fail while others do not. Secondly, by providing each device with a micro-broker, each context consuming application is presented with a uniform programming interface. Sensor-specific communication protocols and variable networking protocols are concealed from the application completely. The

authors also claim that the modular nature of the design allows for device-specific design which accommodates that device's resource limitations.

2.1.6 Context Broker Architecture (CoBrA)(2003)

Chen et al. [17][18] propose a pervasive computing middleware called Context Broker Architecture (CoBrA). Its central features are an ontological approach to context modelling and an intelligent agent context management design.

The authors motivate their work by stating that, "previous work lacked a foundation of common ontology with explicit semantic representation." As well, in their estimation, the informal manner in which data was represented reduced interoperability. They claim that a common context ontology is an essential element of pervasive computing.

As such, the authors proposed a context ontology, COBRA-ONT, expressed in the Web Ontology Language (OWL). COBRA-ONT provides models for people, places, agents, and presentation events for meeting room environments, as well as the properties and relationships between these entities.

The context broker architecture is as follows: in a pervasive computing environment, a centralized context broker maintains a model of the contextual data in the environment for the community of devices occupying it. Devices equipped with context agents communicate with the central server updating and interacting with the data model. Context brokers in adjacent environments periodically synchronize their context models to ensure that in the event that one broker is unavailable, a device may still retrieve context via another.

Security and privacy are enforced centrally by restricting access to elements of the context model on a device-by-device basis. Devices providing context to the model provide associated access rules with that context.

2.1.7 Tuples On The Air (TOTA)(2004)

The design of a pervasive computing middleware described by Mamei and Zambonelli [19] is a distinctly different approach, which they called Tuples on the Air (TOTA). These researchers were concerned with providing a middleware that accommodates a collection of devices communicating using a tuple space model.

Their interpretation of a pervasive environment is one in which devices are constantly moving through their environment, passing other devices. All the devices work cooperatively to disseminate context through a variant of a Mesh Mobile Ad-Hoc Network (Mesh MANET).

Context providing devices are regularly broadcasting Context Tuples, which are representations of state or changes to their contextual state derived from each device's sensor set. If any device receives a tuple, they in turn relay the tuple to their neighbours. To ensure that the tuples are not broadcast indefinitely, each tuple is assigned a propagation rule which can be a combination of a number of relay hops after 7 which relays will not re-broadcast the tuple and a maximum propagation distance (along with the originating location) after which devices should disregard the tuple.

TOTA is not simply a distributed replication scheme as the tuples are not necessarily propagated verbatim: the tuples can interact with the context of other devices changing the tuple's data payload before being rebroadcasted. Further, a tuple may be rebroadcasted several times as events change. A given example is that a new device arrives and therefore all the relevant tuples need to be delivered.

With devices modifying context as it passes, and with devices updating their context status as they move about, the tuple distribution topology is constantly changing.

When a device receives a tuple, it compares it to its own requirements for context consumption. If the context is useful, it acts on and, applies any relevant changes, redistributes, and caches the tuple.

The tuples are represented as serialized Java objects, which means that remote code execution is an important component of architecture.

The authors state that the strength of their design lies in its uncoupled design. In a real-world MANET, devices are not always able to directly query each other. Their multi-hop broadcasting scheme allows devices to extend their range.

2.1.8 SOCAM (2004)

Tau Gu, Hung Keng Pung, and Da Qing Zhang have produced a context sharing framework called a Service Oriented Context Aware Middleware (SOCAM)[20]. This framework has several relevant and interesting features, including a service discovery layer; the separating of applications which produce context, store context, and share context, interpret context, and act on context; a fully defined extensible domain-based context ontology, and considerations for access control and security.

To provide an ontology for expressing context, they use web semantics and OWL to define a series of ontology schemas for specific domains, such as a 'vehicle' domain a 'home' domain etc. Each datum is expressed using the semantics of a domain ontology to express events or states, such as "A person lies down on the bed" or "a user is elderly". Context interpreters can reason about this logic and even form higher level context based from this information, and context aware applications can then act on this reasoned about information.

Software filling the roles of producers, applications, interpreters, etc., are registered with a central service repository for the Local Area Network (LAN). An Open Service Gateway Initiative compliant service provides this in a platform independent way, as well as providing a centralized access control authority which leverages the Java 2 security model. Further, the gateway can facilitate access to sources of context on the Wide Area Network (WAN) on behalf of the context aware application in

the LAN. The framework supports context querying and event subscription, maintained by the OSGi endpoint, which keeps the relevant parties notified of relevant events and changes in context.

The context model outlined for SOCAM is effective and contributed greatly to the model in this thesis. The centralization of a service repository and access control manager is, however, important limitations which prohibit ad-hoc communication.

2.1.9 PersonisAD (2007)

Of the pervasive computing middlewares considered in this overview, the most recently published was presented by Assad et al. [21]. Their focus was on two components: the provision of scrutable context, which is to say the source of the context upon which a device is acting and the inferences made using that context are transparent to the user; and the modeling of context in four primary classes, people, sensors, devices, and places.

The architecture outlined was as follows. Each device, person, sensor, and place is represented by a context model, stored on various distributed context model servers. As devices pass through the environment they acquire contextual information via their sensors, which they call evidence. The device then adds this evidence to the context model. For example, if a device passes an RFID sensor informing it that it is now in a certain room, the device would then send this as evidence to the context model server, to be stored as a part of its own context model.

If an application is interested in acting on context, it resolves the information from the server and acts upon it. For example, a device interested in the occupants of a given room could, using context querying rules, determine the people who currently have that room set as their location in the context model.

Rules are attached to the components of a context model, such that changes in context trigger changes in other context components. For example, if the location of a device changes, the location of the owner of that device is also changed.

The middleware relies on the Apple implementation of the Zeroconf standard Bonjour for dynamic device discovery and communication with the context servers. This allows seamless transition between places.

The authors claim that the most important contribution of their work is that the context is scrutable. The source of evidence stored in the context model is stored as well, allowing users interested in the source of a piece of context are able to determine that.

2.1.10 ECSTRA – Distributed Context Reasoning Framework (2011)

ECSTRA is a framework for distributed context reasoning designed by Andrey Boytsov and Arkandy Zaslavsky [22]. It is an agent based framework which uses the theory of Context Spaces to represent context. A Context Space is a multidimensional space in which each dimension represents one measure that might characterize the environmental state. A vector in this space represents the current situation's particular state. A sub-dimensional space of only the attributes which are relevant to a particular application is called the context space or application space.

Sensors provide raw data that characterizes one aspect of an environment. This data is sent to a centralized gateway service. The gateway provides a publish/subscribe service for each of these attributes, emitting messages to the interested parties when a change of state has occurred. Reasoning engines manage subscriptions of the attribute events which defined a particular application space and maintaining a state vector which can be queried or accessed by an application at any time. The reasoning engines themselves can be sources of data, creating sources of higher-level context inferred from the low-level data. Reasoning engines must also update and maintain their subscriptions from

determinations from the contextual data. For instance, a change of location of a user might cause a reasoning engine to change its context subscription for 'light level' from one location to another.

This framework supports several interesting features, like a separation of context roles. However, the framework's design doesn't allow for a flexible ontology, as each measure must be expressed as a dimensional value, and cannot characterize relationships or implicit meanings. As well, the framework does not address access-control or fine-grained security concerns.

2.1.11 Agent-Based Context Aware Information Sharing for Ambient Intelligence (2011)

Researchers Andrei Olaru and Cristian Gratie have constructed a framework for sharing of context in an interesting fashion named AmIcyTy[23]. Their framework uses agents, typically situated on individual sensors distributed in an environment, which are responsible for interpreting data from their sensors into "Facts", which each then shares with their neighbours. The guiding use case is to be able to disseminate contextual data through a self-organized peer-to-peer network such as a sensor network.

The nodes typically have limited storage space and power. The way in which the information is distributed is designed to accommodate these constraints. An agent sends the data of a certain topic preferential to its neighbours who have established domain interests to which the new datum is relevant. It determines neighbour domain specialties by observing which data the target has contributed in the past. As well, other factors like 'timeliness', how recently the data was generated and 'pressure', the relative importance of this particular datum rated on a scale of 0 to 1. This selective dissemination protocol accommodates the power constraints by limiting the number of transmissions.

The agents maintain a local database of facts which have been passed to them or generated locally. It regularly prunes this database based on its own domain interests, how recent the data is, and

how important the data is. Thus, their locally stored information is generally up to date and of personal interest, while accommodating the resource constraints.

The agent based approach effectively addresses the need for a decentralized distribution network. The framework is not, however, extensible or flexible. Each device must have apriori understanding of what a particular expression of context means. As well, there are no provisions for security, assuming that all actors in the network have full access to all data.

2.1.12 Ambitalk

For his Master's thesis, Eric Karmouch designed a framework called Ambitalk that allowed for communications channels between users to adapt themselves to the changing circumstances[24]. The framework is centered around using Session Initiation Protocol (SIP) for communications between two parties, and using contextual knowledge about the surroundings to adapt the form of the communication to changes in that data. This is done through a communication and negotiation framework which allows the devices to agree upon the parameters and changes of in the parameters of the conversation, such as changes in location and their accompanying changes in privacy policy of each participant in the conversation. The framework also allows the devices to adapt to changes in the network domain, carrying active communication sessions to the new domain and re-negotiating their parameters.

The implementation accompanying the framework has many different communication media, such as audio, video, and an interactive whiteboard. Each makes use of Ambitalk framework to negotiate aspects of their communication.

While the underlying technologies supporting the interactions are open protocols, the negotiation interactions are proprietary, and inflexible. Every devices has to be pre-programmed with a deep

understanding of the capability of the other devices in order to negotiate the terms of the communication.

2.2 Shortcomings of Existing Frameworks

The referenced works each provide features that satisfy some aspect of a full pervasive computing middleware, but each fails to provide a complete solution. The works can be categorized as incomplete for one or more of the following reasons: they rely on centralized architecture for the distribution of context or their context model is inadequate.

Decentralization of the mechanisms for sharing context is important. The increasingly mobile nature of electronic devices means that context sharing might happen in any environment with no guarantee about the infrastructure available. A reliance on central context repositories limits where context sharing might happen. Given the presence of some communication fabric, the framework should be designed to support communication in a random collection of devices. JINI requires the presence of a central lookup service while CoBrA and PersonisAD each use centralized context model servers [11] [17][18] [21].

While Weiser's original vision did not specifically entail context [1], it was not long before context was identified as a key component for providing truly pervasive knowledge sharing[3]. For devices to adapt themselves to the environment and intended tasks of the user in a seamless way, as Weiser had envisioned, it is important that information be passed from one device to the other in a manner that can be understood by the other devices. Context Toolkit, Jini, STEER, and TOTA [10][11][14] [19] provide pervasive networking schemes that allow the delivery of services, but fail to provide explicit context models that could constitute a standard .

It is important that context be modelled in a standard fashion. Many of the pervasive middleware proposed promote a new protocol for context modelling and sharing. None has been

adopted globally. The contextual data should take a standard format and be transmitted via standard protocol to facilitate adoption and extensibility. Use of non-standard components limits the scope of application to the speculation of the protocol's designer. Non-standard data models for context sharing are used in the works of Becker, Mamei, and Yau[16][19][13].

When sharing information, it is important to take into consideration privacy and its security implications. It is essential that access to context data can be restricted on a per-access basis. Few of the frameworks have identified this critical component. While their publications do not propose this, STEER's use of web services as their delivery technology implies that the use of WS-Security might be leveraged to provide the necessary security to ensure data privacy. RCSM provides provisions for security and privacy through their "ephemeral groups" design[25]. Each grouping of devices uses security certificates validated through a centralized security dispatcher and limits access to information based on group-specific rules.

Chapter 3 Context Sharing Framework

This chapter discusses the design of a framework for representing and sharing contextual knowledge in a pervasive computing environment. The framework allows pervasive computing applications to adapt themselves to the current environmental circumstances, which in turn allows them to more effectively perform their tasks and assist their users.

For a complete solution, three aspects must be addressed. First, a design that describes the various actors in the production, storage and consumption of context must be considered. These three roles can be labelled Context Producers, Hosts, and Context Consumers. Context Producers take sources of context, such as environmental sensors or user status, and format that data according to a schema. Hosts collect the Context Producers' data and amalgamate it into a world view, or Situation. This Situation Document includes all contextual information collected by a Host. They then make the Situation document available to interested applications, assuring that the sensitive components of the available information are accessed only by those that have the correct authorization. Context Consumers are applications which query a Host's knowledge base and adapt their behaviour accordingly. These three components are outlined in detail Section 3.1.

Second, a model for contextual data must be designed, providing a vocabulary with which Context Producers can format their knowledge and in which it can be correctly interpreted by context consuming applications. It must be presented in such a way that it can be queried. The model must be flexible enough to describe the contextual state of various sources of context. The model must also

allow for the expression of relationships between various elements of contextual knowledge. The proposed context model is detailed in Section 3.2.

Finally, having defined those who consume, produce, store and query contextual data and how that context should be represented, the various ways that this context is communicated between the actors must be defined. Context Producers must be able to find and register their data with a Host. Similarly, Consumers must be able to find and query a Host. Lastly, Hosts must be able to query each other to share data in a decentralized fashion. Communication between actors must be secure and information that is restricted must only be accessible to those who have the correct authorization. This process becomes complicated when one actor is asking about information on another actor's behalf, such as when a Host is querying another Host on behalf of a Context Consumer. These issues are addressed in Section 3.3.

3.1 Context Framework Roles

At the highest level of abstraction, context sharing can be divided into three distinct roles: Context Providers which provide contextual data, Context Consumers which are responsible for acting on that data, and Hosts which are responsible for facilitating the exchange of context. By defining the responsibilities of each and the relationships between them, we can better encapsulate their individual functionality. This section describes the responsibilities of each.

Consider the situation illustrated in Figure 3-1: a living room, in which there is a home theatre system and a user with a cellphone. In this environment, the home theatre would like to adapt to the changes in the cellphone's ring state, pausing the movie on an incoming call. The cellphone has contextual information which the home theatre can adapt itself to. As well, because the cellphone has limited battery life and responding to network queries would consume power, it has another device in the living room which does not have the same resource constraints, share its context on its behalf. The

home theatre can then access the cellphone's context data through this third party. In this example, the three distinct roles of context sharing are illustrated: a source of contextual information, a consumer of contextual information and a third party hosting context information for the cellphone. These are the Context Producer, Context Consumer, and Context Host respectively.

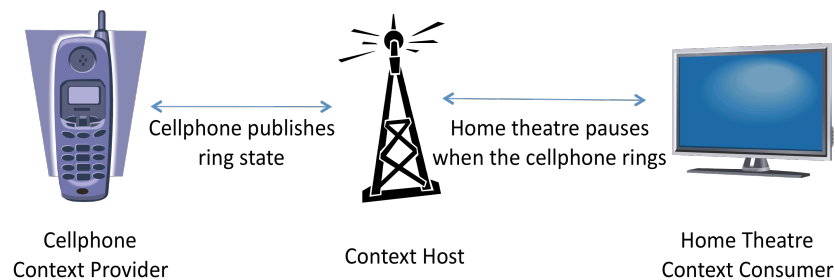


Figure 3-1 Illustration of context sharing roles

Context Providers have direct access to sources of data, including environmental sensors, ethereal data sources like calendars and phonebooks, and relationships between the various pieces of information. The Context Providers format this data in accordance with a schema, and give it to a Context Host to make it available. The Context Providers monitor their respective data sources and update the Host when the data changes.

Context Consumers are applications that adapt themselves based on contextual knowledge. They formulate context queries, which they pose to context Hosts, the results of which they can use to alter their behaviour. When Context Consumers make a request of a host, they can define the scope of their query to be either 'local' or 'global'. Local queries are requests for the information stored on that particular host alone. Global queries require the host to collect the context from other hosts, and apply the query to all available accessible contextual knowledge. Formalized policy definitions that govern the relationship between contextual states and application actions are outside the scope of this thesis. This thesis only provides the tools by which Context Consumers access contextual data in an ordered

and prescribed fashion. Some rudimentary policies are implemented as proof of concept and testing in Section 4.1.

Context Hosts play a central role to the actual communication of the data sources. Typically, they are resourceful devices (plenty of RAM and available CPU cycles) with unlimited power supply. First, they act as a repository for Context Providers. The host provides an API through which Context Providers can register their context for public consumption. These data are validated against public schemas and stored in a database. Second, they provide a query API for use by Context Consumers.

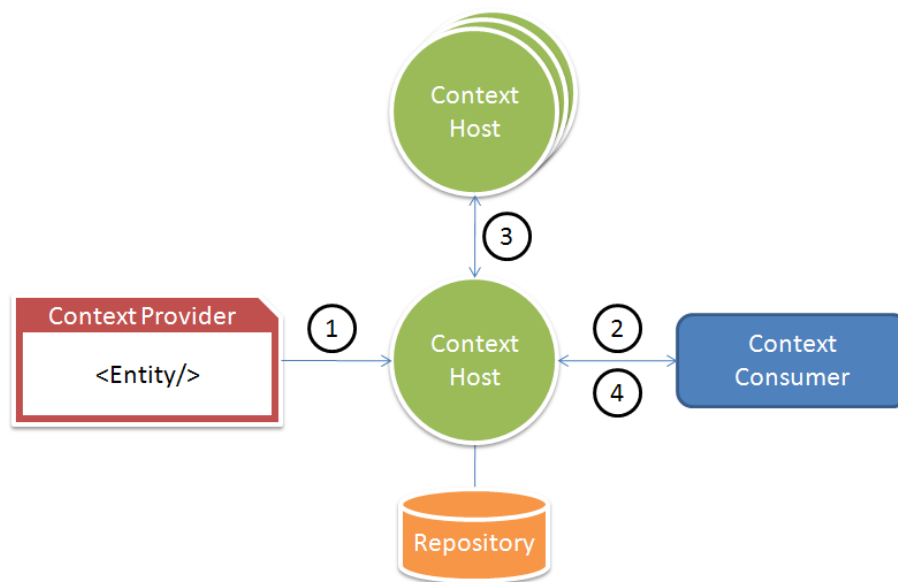


Figure 3-2 Context Actors and their interactions

The relationships between the Actors defined in this section are illustrated in Figure 3-2, with the actor interactions numerically labelled. For the “interaction 1”, the Context Provider produces context and registers it with a Context Host. In “interaction 2” a Context Consumer queries a host. In “interaction 3”, if the scope of the query calls for it, the queried Host contacts other nearby Hosts to

collect their context on the Consumers behalf. In “interaction 4”, the Host returns the result of the processed query to the Consumer.

3.1.1 Role Overlap and Privacy Issues

The separation of roles allows Consumers and Providers to rely on devices with more resources in the event that they themselves do not have sufficient resources. The central role of Host can require significant communication with other actors, a costly role when communication is over wireless networking. If a Context Provider is frequently queried regarding its current state, it might quickly exhaust a limited power supply. Likewise, a Context Consumer can request that a Host perform a query on its behalf, allowing the costly act of querying each available Host in the environment to be handled by a more resourceful device. This ‘proxying’ of request handling and context hosting suits circumstances in which mobile devices wish to conserve their energy. Zebedee et. al proposed a proxy-like context sharing of which this is an extension [26].

The separation of logical roles does not imply that each must be implemented separately or run on separate devices. In fact, there are specific circumstances under which it might be advantageous to implement two roles together. For example, if a Context Provider has particularly sensitive information, it is necessary that the Host distributing that context be held at an ultimate level of trust. Highly mobile devices might have to maintain a minimal Host in the event that there are no other Hosts to which they can offload, in order to keep the local Context Providers and Context Consumers functioning. It is also possible that merging the role of Context Consumers and Context Providers would be useful. An application might be to take low level contextual knowledge and process it to produce and publish higher level context. For example, a collection of telephone capable devices might announce their presence in a room, which could be abstracted into a new high level contextual datum relating the number of telephones in a room.

3.2 Context Model

Representing contextual information is a critical component of a pervasive computing framework. The Context Providers must present their data to the applications in such a fashion that they can be understood. Equally, the Context Consumers must have an understanding of the context schemas and a way in which Context Providers' knowledge relates to these schemas. Further, the context should not be limited to simply representing state. The model must not only express the contextual data but also the relationships among contextual data. Such an expression of contextual data and the relationships between these data constitutes an ontology.

Several approaches to designing an ontology could have been adopted. The ontology could contain a minimal set of ontological elements and relationships, allowing all the Context Providers and Context Consumers a complete and agreed upon language with which to articulate their data. The restrictiveness of a small ontology, however, would insufficiently address the incredible variety of devices currently available, and the sources of information they contain. Conversely, we could build an exhaustive database of current devices, their functionality, and all possible relevant contextual data; however, this would poorly address the reality of technology: constant evolution and advancement. The form and function of the devices available is constantly changing, making a consistent, universally available ontology impractical.

The adopted solution is intermediate to the extreme positions above. As it is impossible to predefine a schema representing all context, but insufficient to rely simply on primitives for expressing all context, an intermediate solution has been adopted in which a base schema is universally agreed upon, and extended to express more specific domains. Those designing software to share a device's context are able to build a specialized schema that references the foundation elements and extend them in unique directions. For example, the base schema defines "devices". Devices are extended in a

schema extension document to define “telephones”, used by all context-sharing telephones. A specific model of telephone would then further extend “telephone” to their model in another schema document. If a Context Consumer application were to make an inquiry about this device, and was familiar with the specific model of telephone, they would fully understand the context presented. However, if the Consumer were only familiar with telephones, but had not been programmed with an understanding of that particular model’s schema extension, they would still understand the context expressed using the general ‘telephone’.

The base schema, from which all other schema are derived includes basic contextual data types and the rules for their expression, such as strings and integers, as well as three higher level elements to which these data can be related: Entities, Hosts, and the Situation. Entities are concepts like a person, place, or device, to which contextual data can be attributed. For example, a telephone would be a type of “Device” entity, which could have context expressing its current ringing state, ringer volume, and an ownership relationship with a “Person” entity. A Host is the schematic representation of a Context Host which is the representative of many Entities to querying devices. The Situation can be considered a view of the currently available contextual data, expressed in terms of the available Entities, represented by Hosts. As well, the base schema includes rules by which to express relationships between entities and their data.

Basic definition documents may outline common Entity types, such as printers, cell phones, email devices, or locations. These types can be used as a base, from which more specific Entity definitions may be derived. For example, while a general printer may be defined with such features as a printer queue, status, and Internet Printing Protocol endpoint URI, a printer with further features, such as a pages-remaining-in-tray feature, can extend the basic type.

A software developer, wishing to share a device's data with this framework would find a set of schemas that best represent the data the device provides. Ideally, the device's data are completely represented by the schema documents, but in the event that they are not, the developer may write their own schema document to extend the base schemas to express the missing data. For example, a developer might find that the commonly used "telephone" schema does not include expression for a cellphone's ambient light sensor, and choose to extend the "telephone" definition to include a light sensor contextual data point.

This design has two important features which make it effective. First, the extensibility features mean that while a Context Consumer may not be familiar with a particular Entity type it may be familiar with the Entity definitions from which the Entity derives. Its familiarity with the ancestor gives the Context Consumer a general understanding of that particular Entity. Since Context Consumers with specific areas of interest would be programmed with a more comprehensive understanding of one branch of Entity types, it is not necessary for them to have an understanding of the complete ontology. Second, if a Context Consumer is not familiar with a particular extension of an Entity, the definition document is expressed using universally accepted primitives. Thus it can analyse the definition document to make an educated guess about that feature set's intended meaning. While the technology for such inference is outside the scope of the thesis, a form of applied schema translation and inference [27] may be used to address this problem.

3.2.1 Context expression requirements

The choice of technology to implement the schema language outlined in Section 3.2 was selected to support the following essential features:

- **Communicable Ontology:** the data must be represented in a consistent way, whose form is pre-defined to ensure that it is understood and by which relationships between contextual knowledge can be defined.
- **Flexible:** the schema must be extensible so that new ideas and data sources can be added without affecting previous implementations.
- **Aware of Time:** in a dynamic and mobile environment, information can become stale; it is important that Context Consumers are aware of the data's validity with respect to time.
- **Maintain Privacy & Security:** privacy and security concerns must be addressed as essential components to the data model, rather than as an afterthought.

The manner in which these features were implemented follows.

3.2.2 Ontology in XML

Extensible Mark-up Language (XML) is a semi-structured data format[28]. Its features are ideally suited to the requirements of expressing contextual information for many reasons. This section outlines the core technologies of XML and how they relate to and support a context model. A complete set of example schema are provided in 5.2Appendix A.

3.2.2.1 Tree-like Hierarchy

XML can express a tree-like hierarchy of data, which gives us the ability to express relationships between various ontological individuals. For a given Situation there are many Hosts. A Host is responsible for representing a set of Entities provided by the Context Providers. The Entities each contain a hierarchy of contextual knowledge. The following XML document represents the relationship between the Situation, the Hosts within it, and the Entities hosted by each:

```

<Situation>
  <Host id="host1">
    <Entity id="entity1" />
    <Entity id="entity2" />
  </Host>
  <Host id="host2">
    <Entity id="entity3" />
  </Host>
</Situation>

```

Similarly, the following example illustrates how XML's hierarchy could be used to express an Entity with telephone features and email features.

```

<Entity id="smartphone" type="deviceType">
  <FeatureSet type="telephoneDeviceType">
    <PhoneNumber mutable="false">
      <AreaCode>613</AreaCode>
      <Number>533-6000</Number>
    </PhoneNumber>
  </FeatureSet>
  <FeatureSet type="emailDeviceType">
    <Address public="false">cyrus@cs.queensu.ca</Address>
  </FeatureSet>
</Entity>

```

Attributes are used to qualify the type of generic elements. In the above example the Entity is defined as a device type and the feature sets are defined as the telephone and email feature sets. By setting the types of the FeatureSet element, the form and type of the subsequent nested XML elements are prescribed, meaning that Context Consumers can consistently find information in the same place.

Context can often have multiple representations. For example, location can be expressed in different ways; two examples being longitude and latitude in degrees, minutes, and seconds, or a colloquial term like "Goodwin Hall". With XML, it is possible to represent multiple expressions of the same information simultaneously.

```

<?xml version="1.0" encoding="UTF-8"?>
...
<Entity>
  <Location>
    <GeographicCoordinates>
      <Longitude type="DegreesMinutesSecondsType">
        <Degrees>44</Degrees>
        <Minutes>13</Minutes>
        <Seconds>39.48</Seconds>
        <CardinalDirection>N</CardinalDirection>
      </Longitude>
      <Latitude>
        <Degrees>76</Degrees>
        <Minutes>29</Minutes>
        <Seconds>32.65</Seconds>
        <CardinalDirection>W</CardinalDirection>
      </Latitude>
    </GeographicCoordinates>
    <Colloquial>Goodwin Hall</Colloquial>
  </Location>
</Entity>...

```

Here, geographic coordinates and the colloquial expression for the location are displayed side by side. A Context Consumer reasoning about this information can use either of the forms to understand the location. The complete schema for Location is available in 5.2Appendix A.

3.2.2.2 Graph-like Referencing

While the hierarchical dimension can be used to express the relationships between ontological individuals from different classes (e.g. a host can be related to its entities), it does not allow for the expression of the relationship between individuals of different classes. For example, strictly through hierarchy, the relationship between two Entities, or one Entity and another Entity's context cannot be expressed.

XML does, however, support graph-like features which can be used to express such relationships. An XML node with an ID attribute can be referenced by a node of IDREF type, a native XML type. In the following example, a device Entity expresses that it is owned by a person Entity. The Subject element is an IDREF.

```

<Entity id="person1" type="personType" />

<Entity id="device1" type="deviceType">
  <Relationship verb="ownership" >
    <Subject>person1</Subject>
  </Relationship>
</Entity>

```

A relationship is defined with three elements: subject, verb, and object. One or more subjects must be defined. These are defined by a reference to that element's id. In this case, the subject element is an Entity. The verb describes the relationship between the object and the subject as an attribute of the Relationship term. The object element is not expressly defined, but implied; the Entity in which the relationship is nested is the object. As such, only an Entity can express how other Entities are related to it, and prohibits one Entity from expressing how two other entities are related.

3.2.2.3 XML Schema

XML Schema is a metadata language that defines the grammar by which an XML document may be formed[29]. It is itself expressed in XML. Using schemas, the context documents can be restricted to a predictable layout; an essential feature of a context model if it is to be commonly understood. The following example is an excerpt of the schema "ContextBase.xsd", which defines many of the primitive elements and types used in this context model. The complete document is available in 5.2Appendix A. Here you see a definition for an attribute group "ContextAttributes", and a complexType "ContextString". The ContextAttributes group defines a set of optional attributes which can be used to characterize an expression of contextual data. Any schema that wishes to prescribe a contextual datum of 'string' type, can use a ContextString, which is defined as an element of 'string' type with the ContextAttributes set. The ContextAttributes are explained in depth in 3.2.2.6.

```

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

...

<attributeGroup name="ContextAttributes">
  <attribute name="contextID" type="ID"/>
  <attribute name="lifetime" type="integer" use="optional"/>
  <attribute name="expiry" type="long" use="optional"/>
  <attribute name="mutability" type="boolean" use="optional"/>
  <attribute name="private" type="boolean" use="optional"/>
</attributeGroup>

<complexType name="ContextString">
  <simpleContent>
    <extension base="string">
      <attributeGroup ref="base:ContextAttributes"/>
    </extension>
  </simpleContent>
</complexType>

...

</xs:schema>

```

XML Schema has provisions for extending type definitions. This is an important feature, allowing for the representation of a hierarchy relating general Entity types and their context to more specific Entity types. Through this method of extension, primitive types can be extended with new features and contextual qualities without altering their backwards compatibility, as the original contextual forms are maintained. Below is an example which demonstrates schema type extension. A generic abstract XML element “Entity” is declared. The abstract declaration implies that the element “Entity” may not itself appear in an XML document, but other elements which extend it may[30]. This schema element defines that all Entities have an ID attribute, and a location defined by the LocationType (defined elsewhere in the schema). A “PersonType” element is defined next, extending its abstract parent, declaring that PersonTypes have names. Thus, an XML document using this schema and declaring a PersonType entity would be allowed to give that element an id and location as defined in the

Entity type, and a name, as defined in the PersonType. This example is an excerpt and would be preceded and followed by other XML content.

```
...<complexType abstract="true" name="EntityType">
  <sequence>
    <element name="Location" type=" LocationType" />
  </sequence>
  <attribute name="id" type="ID" use="required"/>
</complexType>

<complexType name="PersonType">
  <complexContent>
    <extension base="EntityType">
      <element name="Location" type=" LocationType" />
      <element name="Name">
        <sequence>
          <element name="firstName" type="string"/>
          <element name="lastName" type="string"/>
        </sequence>
      </element>
    </extension>
  </complexContent>
</complexType>...
```

XML schema can be broken into separate documents. This allows the schematic layout to be partitioned by theme (e.g. definitions for devices can be separate from persons). Definitions for more general forms of context can be separate from more specific definitions. This allows for a Context Consumer to use some of an Entity's data, without having to understand its entire schema. As long as the Context Consumer understands one of the Context Provider's ancestors, they can have limited interaction with this Entity. Devices can have a coarse-grained understanding of general Entities, and a finer-grained understanding of Entities and context of interest to that consumer's scope of application. From an adoption perspective, this would also allow device and service implementers to develop their own schemas, picking and choosing a selection of base documents as the starting point from which they would extend to define their Entity's capabilities.

3.2.2.4 Namespaces

To ensure that Context Consumers and Context Providers are using the same vocabulary, the schema documents are defined within namespaces. This means that when referring to a type definition within a context document, the namespace from which it comes is explicitly stated. The context type 'name' can be defined separately in the device schema document and the person schema document, and any document referring to either definition states the namespace to which they are referring. In this fashion there is no confusion about which meaning of 'name' is being discussed.

```
<entity type="person:PersonType"
  xmlns:person="http://cs.queensu.ca/PersonBase"
  xsi:schemaLocation="http://cs.queensu.ca/PersonBase
    http://cs.queensu.ca/Pervasive/PersonBase.xsd">

  <person:Name>
    <person:FirstName>Alice</person:FirstName>
    <person:LastName>Anderson</person:LastName>
  </person:Name>

</entity>
```

The above example makes use of schema location references, an important feature of namespace reference. While a Context Consumer might not have encountered the schema to which a Context Provider is referring, since the schema document's URI is explicitly stated, the consumer can access it and attempt to understand it. An application implementing such schema translation features is outside the scope of this thesis, but their availability further supports this design as a viable context model.

3.2.2.5 Querying XML

There are many standard languages for querying XML documents and thus there is no need to implement a new context reasoning language. This particular implementation of the framework supports XPath[31] and XQuery[32] statements, but it could easily be extended to support XSLT[33].

The following example illustrates how the question “For the given situation, what email addresses are accessed by email devices owned by a person whose last name is Anderson” can be posed in XQuery.

```

declare namespace base="http://cs.queensu.ca/ContextBase";
declare namespace person="http://cs.queensu.ca/PersonBase";
declare namespace device="http://cs.queensu.ca/DeviceBase";

for $userid in
//base:Entity[person:Name/person:Last/string()='Anderson']/@id
return
for $email in //base:Entity[base:Relationship[@verb="owns"]
[base:SubjectElement=$userid]//device:EmailAddress
return $email/string()

```

The namespaces are explicitly stated in the query ensuring that the query and the context document share a vocabulary. The namespace labels (e.g. base:, person:, and device: used in the above query example) are only references to the namespace URI, so the labels used by the Context Provider in their context document are not required to match the labels used in the Context Consumer's query. Schema sensitive implementations of xQuery are able to fetch and process the schema documents, even validating them. This is an important feature as it allows the queries to be xml schema type-extension sensitive. For instance, in the XML schema, a "Device" is an extension of the "Entity" type. In an XML document, there would be no reference of a device being an Entity. Since xQuery is schema aware references to devices in a query will be recognized as Entities.

3.2.2.6 Context Attributes

The contextual knowledge is only part of the relevant information that must be available in a context document. Associated with the context data is a set of metadata, which includes that datum's mutability (i.e. can the value change over time), the datum's longevity (i.e. the expected period over which a context consumer can assume the datum is fresh), and a unique identifier with which relationships may be defined. The expression of this metadata is expressed in context documents as a set of element attributes.

Access control is an important design requirement. Context providers must also be able to state which consumers have access to which context data. This qualifies as a form of metadata which can be represented within a context document's XML. To protect a certain branch of XML from certain users, a Context Provider defines an access group within the relevant Entity's XML. An access group is a list of the RSA public keys of the Context Consumers that are to be given access to a branch of context, and a reference to the context node which is to be protected. The context node also has a Boolean valued attribute named "public" that is set to false. At query time, it is then the responsibility of the host to prune the XML tree allowing the Context Consumer access only to the branches of XML to which they are entitled.

3.3 Security Features

Building security into the design of this pervasive computing context management framework has been a primary goal of our work from its inception. Layering security on top of this system would have been a half measure at best, and would not have addressed some of the peculiarities of this architecture, in particular the hosts acting as query proxies to the Context Consumers.

Secure communication is essential to assuring the privacy of privileged information. There are many parties involved in the collection and consumption of context. These parties pass messages between one another regularly, often containing contextual information that should only be accessible by eligible parties. It is therefore essential that, when two parties are sharing information over an open wireless network, the information is secured.

3.3.1 Public Key Encryption

Public key cryptography has been essential to encrypted communication through the electronic age [34]. In general terms, for two parties to communicate, each produces a public and a secret key. Each party publishes the public key for the world to use while keeping their secret key to themselves. To send a message that only one party can decode, the sender encrypts the message with the receiver's public key. Even if the message was intercepted in transit by a third party, it is computationally impractical for them to decode the message. Such encryption schemes rely on trap-door functions, mathematical one-way transformations whose inputs are difficult to determine from their outputs without extra information which creates a shortcut.

With a user's public key, a sender may be assured that the receiver is the only person who will be able to decode a message, but since the receiver's public key is available to the world, the receiver cannot confirm who sent the message. This issue is addressed by public key cryptography's ability to

digitally sign content. The sender computes a hash of the content they are sending and using their private key produces a signature. This signature can only be produced by the sender's private key, but can be verified by anyone with the sender's public key.

RSA is a widely used public key encryption algorithm, whose strength and reliability have been thoroughly investigated [35]. For this reason RSA is employed to support private communication between Context Consumers, Context Providers, and Hosts. If each uses its public key as its unique identifier within the network, there can be no confusion over the relationship between a public key and its owner. More importantly, a Context Provider can make rules about permissions based on the public key of the individual making the query, and ensure that the result is passed back to them and them alone.

3.3.2 Letter of Mark

Communication between various endpoints is now done in a secure fashion, but a problem persists: Context Consumers are not interacting directly with Context Providers. Hosts are acting as intermediaries, querying other hosts on behalf of the Context Consumer.

The problem is best illustrated with an example. A Context Consumer has permission to access a particular piece of otherwise restricted information. This information has been registered by a Context Provider to a nearby Host, $Host_R$ (i.e. the Repository Host). The Context Consumer makes a query with global scope to a different Host, $Host_Q$ (i.e. the Queried Host). On behalf of the Context Consumer, $Host_Q$ then queries each nearby host for all knowledge relevant to the querying Context Consumer. The problem arises here: how can $Host_R$ release this information in a secured way such that it can be consumed by $Host_Q$ on behalf of the Consumer, with no prior knowledge of the relationship between $Host_Q$ and the Consumer?

Before considering possible solutions, a condition must be stated. The Context Consumer should not have to process the data themselves; that is the responsibility of a Host. A Context Consumer is only concerned with the solution to their query. This is done to provide an opportunity to minimize the processing done by the Consumer by offloading it to a Host. Given that the responsibility of processing the Consumer's query is assigned to the originally queried Host (i.e. $Host_Q$) rather than the Context Consumer themselves, three solutions exist. First, if $Host_R$ has a copy of the Consumer's private key, $Host_R$ can encrypt the data using the Consumer's key, and $Host_Q$ can simply decrypt the data, process it, and pass the result back to the Consumer, as is illustrated in Figure 3-3. This violates an essential rule of public key cryptography, which is "never share your private key". At this point, the Consumer can never revoke the Host's right to query on its behalf.

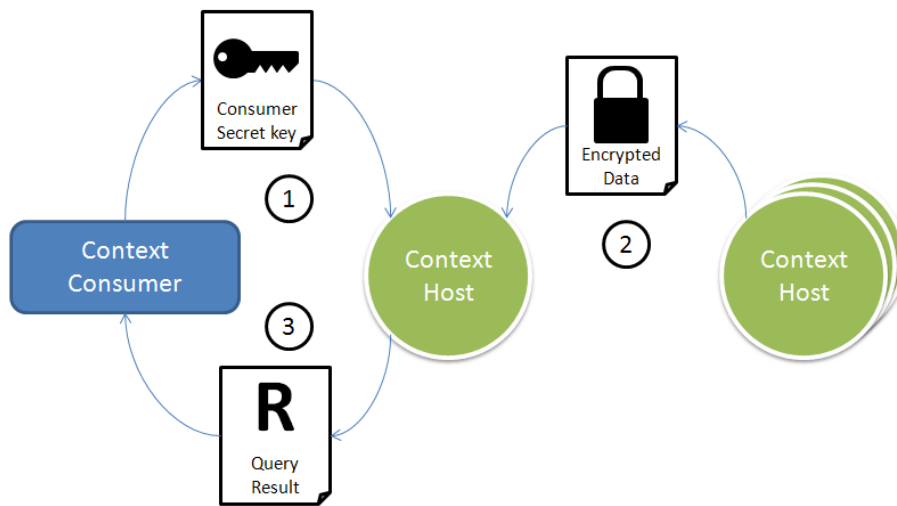


Figure 3-3 Private key sharing decryption process

Second, as in the first case, $Host_R$ can encrypt the data with the Consumer's key and pass it back to $Host_Q$. $Host_Q$ can then pass the encrypted package back to the Consumer, unprocessed, have the Consumer decrypt it and pass it back to $Host_Q$, at which point $Host_Q$ can process the query on the unencrypted data. This procedure is illustrated in Figure 3-4. While this arrangement would maintain

key privacy, this method requires a lot of message passing which consumes power and takes time. Having the Host act as an intermediary in this fashion provides little if any advantage.

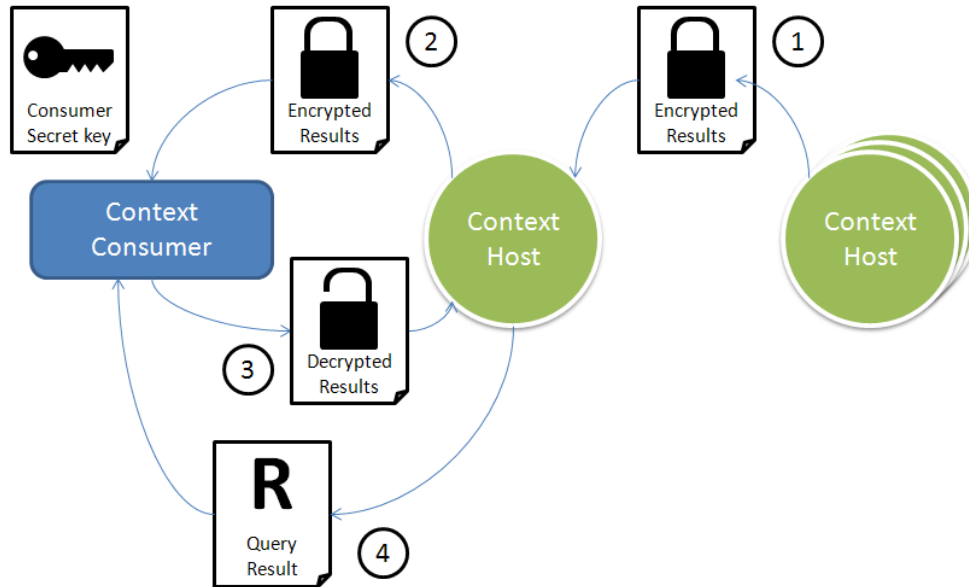


Figure 3-4 Encrypted data sharing decryption process

The third solution is the one implemented for our framework and is illustrated in Figure 3-5. When the Context Consumer makes a globally scoped query through a Host, they provide the Host with a Letter of Mark, which is a data package authorizing the Host to operate on their behalf for a limited amount of time. Specifically, this Letter of Mark identifies the public key of the querying Context Consumer, the public key of the Host making the query, and the UNIX timestamp of the expiry of the Letter of Mark. It also includes an RSA signature of content of the letter of mark, which assures the receiver of the validity of the authorization. When $Host_Q$ queries $Host_R$, $Host_Q$ provides a copy of the Letter of Mark. Once the Letter of Mark has been verified by checking the signature against the Consumer's public key, $Host_R$ can then encrypt the contents directly to $Host_Q$, using $Host_Q$'s public key. $Host_Q$ can process the data directly and pass only the result back to the Consumer.

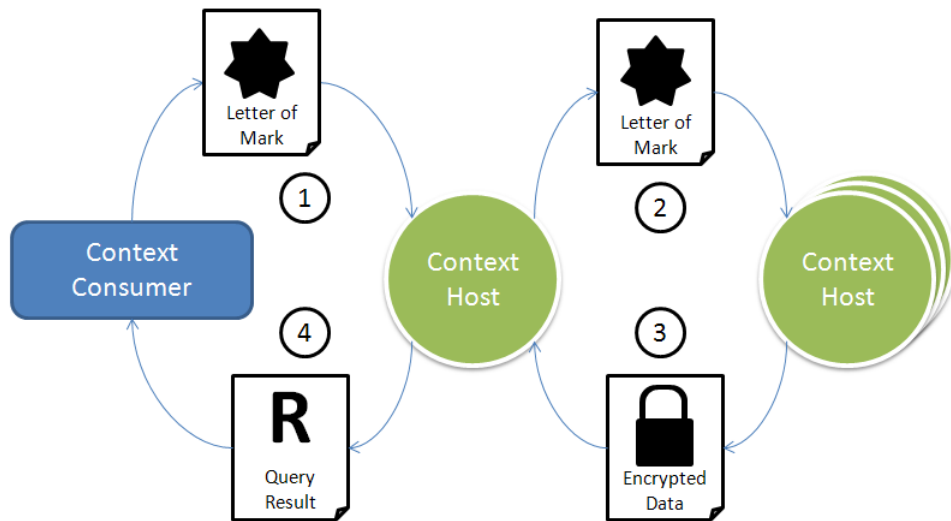


Figure 3-5 “Letter of Mark” decryption process

Context Consumers and Hosts are merely logical separations of actors, which can allow, but does not require, the actors to be on separate physical devices. As such, if an application which implements a Context Consumer is unwilling to authorize third parties, they can also implement a Host and maintain full control of their querying of other hosts in the environment. In this case, the Consumer does not need to offload the query processing to another device.

3.3.3 Man in the Middle Attacks

In this implementation, there is no mechanism to establish the relationship between a Host’s identifier and its public key; the Host is directly identified by its public key. Several problems exist with this: first, the actors cannot change their key without having to re-establish its relationship with each entity, breaking or discarding all context data previously established about its relationships; second this introduces the possibility of one host representing another host’s context as its own, which permits a Man in the Middle attack.

A Man in the Middle attack is a potential attack against communication systems. In a scenario in which two parties, Alice and Bob, would like to communicate using public key encryption over a network but have not yet exchanged public keys, it is possible for a third party, Eve, to represent herself to Alice as Bob, and to Bob as Alice. Any messages passed from Alice to Bob, first go through Eve. Eve decrypts the message intended for Bob, reads it, and re-encrypts it with Bob's public key before passing it along. The essential problem is that there is no way to verify the relationship between a particular entity and their public key.

There are several solutions to this problem. The first is a centralized repository of relationships between keys and identities. Everyone starts off with the public key of the central repository. Those involved in communication have their own keys signed by the centralized authority, which is to say the relationship between the key and the user is certified by the central authority. Everyone can then use the central authority's public key to verify that the key is related to an identity. This model requires a centralized authority, which is not a reasonable solution given this design's decentralized model.

Another solution would be to use what is known as a "web of trust" model. Each member of the community signs the public keys of the people they trust. If Alice encounters Bob for the first time, she cannot immediately trust the key she received belongs to him, as it may have been swapped in transmission. If, however, Charlie had signed Bob's public key, and Alice already trusts Charlie's key, she can rely on Charlie's assertion that the key-user relationship is valid. This method has a clear advantage of being decentralized. This model does require a critical mass of the keys to have been signed by other members of the community before the coverage is sufficient that if two members have not yet interacted, they are at least likely to have had their keys each signed by a member with which the other had already interacted.

3.4 Message Passing Scheme

As well as the content being passed around between actors in the context sharing environment, the messages themselves must be defined. There are three pairs of messages required for the three actors to effectively communicate their knowledge:

- 1) Context Host Request & Reply: A Context Provider sends a request for hosting to an available Host. The message contains several things, including the context XML payload and an expiry UNIX timestamp. The Host responds with an acknowledge message, either granting permission to host the content until the UNIX timestamp is met, or rejects the request altogether.
- 2) Context Request & Reply: This message is sent to a Host from either a Context Consumer or another Context Host. Context Consumers use this message to request that a Host process a query, and Hosts use it to request Context on behalf of a Consumer to build a Situation view of the available context to which a query may be applied. The message also contains a Letter of Mark on whose behalf the query is being made, so that a Host knows to whom to encrypt the message, and by whose authority the request is being made. The response contains the query response or if an error was encountered, an error flag and message.
- 3) Host Key Request & Reply: This message is used by Context Providers to obtain the public keys of the available Hosts through which to publish their context, Context Consumers when looking for a Host to query, and by other Hosts when looking to determine the current Situation. Hosts respond with their public key.

Figure 3-6 illustrates a typical message passing procedure, as a Context Provider registers its context with a Host, a Context Consumer makes a globally scoped query to a Host, and the Host queries other hosts to determine the answer to the Consumer's question.

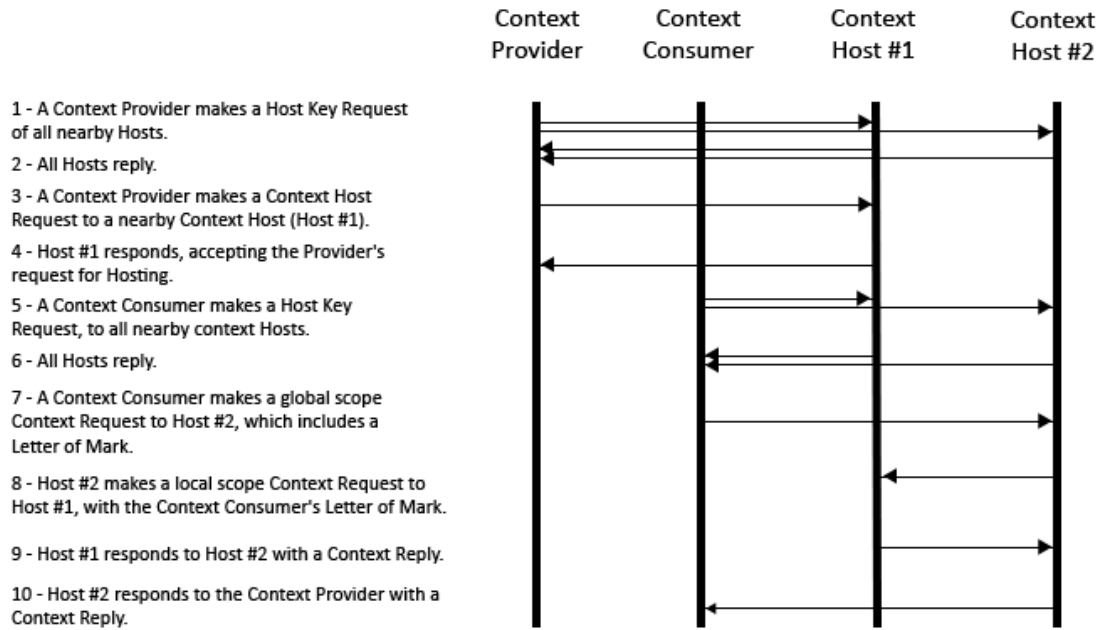


Figure 3-6 An illustration of the message interactions between actors in a typical Situation.

Chapter 4 Implementation and Testing

This chapter outlines the implementation of our framework based on the designs of Chapter 3. The various technologies used in the implementation are described and the mapping between their features and the requirements of the Context Aware Pervasive Computing Platform are enumerated. Implemented case studies are described to illustrate the usefulness of the technology and to identify its weaknesses. Tests which assess the implementation's usability are discussed.

4.1 Implementation

The use of software and technologies that are in widespread use was an important goal of the project. Using common components and technologies meant that such a framework could easily be used in existing platforms with minimal or no alterations to an existing configuration. The use of open source software and Java code provided portability across platforms.

Our implementation was done in a Linux environment using a suite of 32 bit and 64 bit desktop and laptop computers. Stock 802.11g wireless cards and Bluetooth dongles provided the networking hardware interfaces.

Our software was written in Sun Open Java SE 1.6[36]. The code base consisted of 5709 lines of code in 49 classes (3 abstract), and additional packages from 3rd parties such as the Apache Saxon project, JmDNS, and Bluecove, each described in detail in this chapter. The class hierarchy expressed in a UML diagram, and references for downloading the source code are outlined in Appendix C.

The framework implementation is comprised of three layers: a network layer, a discovery layer, and a service layer. Each is responsible for a different aspect of the connection and exchange of context. The network layer provides the fabric by which data is exchanged. The discovery layer relies on the network layer and provides a service by which pervasive devices can discover other devices that are sharing context. The service layer is comprised of client and server applications which constitute the Actors in context exchange. The following subsections describe the layers and their related technologies as well as other implementation considerations.

4.1.1 Network Layer

Essential to context sharing is a fabric by which devices can communicate. Several different technologies exist for connecting devices, allowing them to share data wirelessly. Our framework makes use of two different technologies, wireless IP network infrastructure and Bluetooth, each with their own advantages. Internet Protocol (IP) Network infrastructure is widely available and provides an excellent conduit by which these devices can communicate. There are many configurations on which this pervasive computing platform may operate, of which those supporting the wireless data link layer 802.11 commonly called Wi-Fi [37], are the most useful due to their widespread use. Our framework has been designed so that it operates regardless of the IP Network being used.

Commonly, wired and wireless devices are connected through a pre-existing Local Area Network (LAN) infrastructure. Many home and office environments have Wi-Fi access points. These access points often provide a Dynamic Host Configuration Protocol (DHCP) service, assigning unique IP addresses to each device. If a DHCP service is not available, devices can assign themselves addresses in a decentralized way using link-local addressing [38]. Using wireless access points has the added advantage of grouping devices which are proximate in the same LAN since only devices in range of the router can join it. As well, devices are grouped by their decision to use that particular network over

others. The disadvantage is the reliance on a centralized networking infrastructure, but since access points are so widely available, this issue is negligible.

Our implementation allows Context Providers and Context Consumers to interact with Hosts over Bluetooth connections, if each are equipped with the required technology. Bluetooth operates in a decentralized fashion, each device identified uniquely by a Media Access Control (MAC) address issued by the manufacturer. While routing protocols do exist, Bluetooth is generally used for direct device to device communication, as a replacement for physical cables. The core Bluetooth specification is designed for a 10 meter range[39].

Bluetooth supports multiple protocols, one of which is the Radio Frequency Communication (RFCOMM) protocol, which acts as an analogue to the RS-232 serial port connection [9]. Support of this protocol allows for the creation of socket connections between the two devices which can be used for message passing in an identical fashion to the IP network. Each device requires an implementation of the Bluetooth Stack, which support the layers and protocols of Bluetooth. Bluecove is an open source JAVA API to Bluetooth stacks, including BlueZ, which has been used in this implementation[40].

Our implementation of a Host has some limitations with respect to Bluetooth. While a Bluetooth endpoint can be used for context queries and context registration, Bluetooth notifications were not implemented.

4.1.2 Service Discovery Layer

In a pervasive computing environment devices must not only be able to communicate with each other, but must also be capable of discovering each other and know how to interact once they make contact. Specifically, all the Actors in the exchange of context must be able to discover Hosts with which they can interact. Hosts must have a way to advertise their presence and all actors must have a way to seek out Host services. This is accomplished in different ways for Bluetooth and IP network fabrics.

4.1.2.1 IP Network Discovery

Domain Name Service - Service Discovery (DNS-SD)[8] is a technique for using specially crafted DNS records to advertise the availability of services over local area IP networks (LANs). A service registered using DNS-SD is advertised as a DNS record with four parts: the name of the service; the service type, such as “_printer” or “_pervasive” which we have defined (but not registered with the official type registry[41]) for devices which are available as Context Hosts; the packet transmission protocol by which communication with the service happens, such as “_tcp”, and the “.local” top level domain, indicating that the service is local to the LAN. In the case of pervasive Hosts, the service name is a SHA1 hash value of its public key. Thus a service might be announced with the following: DuBlzz8P0w8o7Wv2inZnnkD7JFy._pervasive._tcp.local.

Combined with Multicast DNS (mDNS), which provides DNS services in a network environment without a central DNS service, DNS-SD allows devices to advertise their services to the LAN without a centralized registry. An interesting advantage of mDNS is that every time a DNS record (including service records) is requested on the network, the DNS cache of all the devices is updated with the response. This means that devices will be more up to date on the available Host services than with a polling scheme. This deliberate misuse of the DNS protocol has been leveraged to advertise printing services, iTunes libraries, bittorrent, and a long list of other network services. This technique was originally developed by Apple as a part of their solution to Zero-configuration Networking (Zeroconf), originally implemented as Rendezvous and now known as Bonjour. The implementation outlined in this thesis makes use of the open source JmDNS project [42], a Java DNS-SD and mDNS API which is compatible with Bonjour.

4.1.2.2 Bluetooth Discovery

We use the Bluetooth service discovery protocol to provide service discovery over Bluetooth. The system is not as elegant as the DNS-SD and mDNS solution for IP networks but it satisfies the requirements of allowing devices to discover published services, in particular Context Host services, on nearby devices. Once a device has been discovered, it may be queried using Bluetooth Service Discovery Protocol (SDP) to see which services are available [9]. A context Host would advertise itself with a Bluetooth services URI in the following manner:

```
btspp://localhost:0000110100001000800000805f9b34f  
b; name=ContextHost-9wx_mM4wmTYA0fv-j5imoKuwD41;
```

Bluetooth can be used to transmit several different protocols. It is necessary, therefore, to specify in the service URI which protocol is used. The btspp component of the above URI refers to the Serial Port Profile (SPP), which emulates the RS-232 serial port connection allowing socket connections to be made in the same fashion as IP network socket connections. While the 32bit UUID that follows the protocol statement uniquely identifies the service, the set of parameters passed with the statement have been made to include a name parameter which is the string "ContextHost-" followed by the Context Host's public key in a Base 64 encoded SHA1 hash. This allows for the service's differentiation using the same identifier used within the context sharing framework.

4.1.3 Service Layer

Once a Host's endpoint presence have been shared using the techniques described in Section 4.1.2, the Context Consumers and Context Hosts need a protocol by which they may interact. The service layer is the set of protocols for communication between the logical Actors, independent of the communication technology used. A basic client-server socket architecture provided by Bluetooth and

Wi-Fi combined with a messaging protocol allows for Actors to make requests of Context Hosts to add, update or query context.

As was described in Section 3.4, a set of predefined messages are passed between the Actors. For the implementation of this framework, each message is defined and implemented as a set of serialized Java objects. For example, one Host's local scope context request to another contains the request Host's public key, the Letter of Mark issued by the Context Consumer for whom this query has been made (which in turn contains an expiry, Consumer's public key, Host's public key, and the Consumer's signature of these data). This message object is serialized and sent via a socket connection from one Host to another.

The reliance on Java features is for convenience in this demonstrative implementation. In further refinements of the framework, the messages could be expressed in a technology-agnostic format, such as XML, or Web Service interactions. As well, our implementation encapsulates the message receipt and parsing in such a fashion that it could be exchanged with such protocols.

4.1.4 Actor Internal Architecture

The implementation details of each of the three Actors are outlined in this section, with particular attention to the specific challenges of each.

4.1.4.1 Context Hosts: Request Handling & Context Repository

Our implementation of a Host consists of three main components: the network endpoints (servers), the worker threads, and the repository. Each is responsible for a specific aspect of handling requests. The implementations of these components and their relationships, illustrated in Figure 4-1, are explained in this section.

Both Bluetooth and Wi-Fi endpoints have socket listeners, to which other actors may make connections and send requests. The Host server manages incoming connections from both endpoints by issuing each request a worker thread from a shared thread pool. The thread pool consists of 100 pre-instantiated worker threads; this removes some of the overhead associated with handling a request. If the number of active requests exceeds the number threads, the requests are queued.

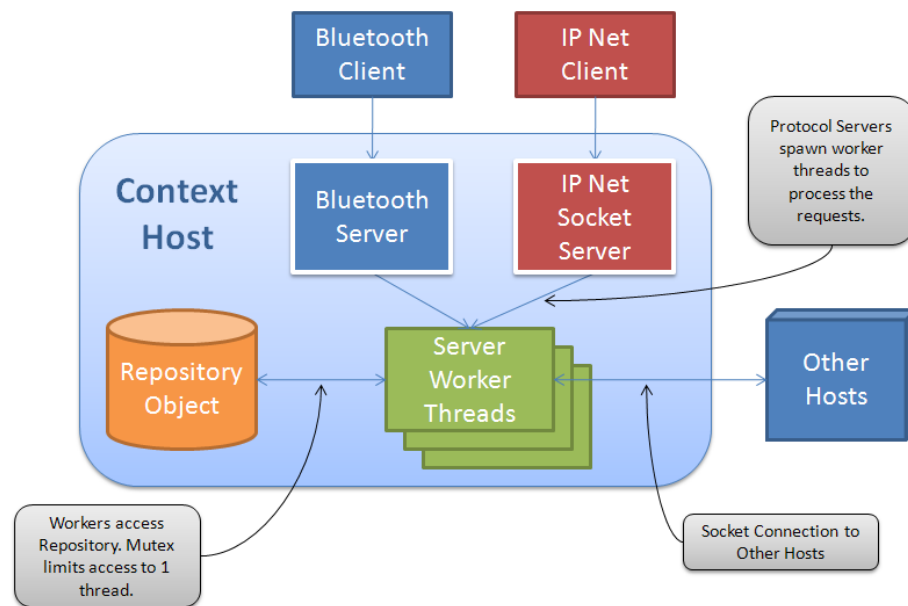


Figure 4-1 Host Architecture

The Host's repository maintains a store of the context it has agreed to make available to context consumers. Our implementation of a Context Host maintains its Repository as a concurrent hash map[43], which is essentially a key-value pair mechanism in which the hash keys are the RSA public keys of the Context Providers the Host is making available, and the respective hash values are the context representations which those Providers contribute. A worker thread can compile a local view of this Host's context by iterating through the hash map and concatenating the Host's context Entities into a single Host-level View XML document. Concurrent hash maps allow for updates to a key value pair to take place at the same time as the hash map is being read. Though they do not guarantee that items added after iteration has begun will all be retrieved, completely locking the table during iteration to

ensure that the snapshot is for an exact time window would only delay those modifications, slowing the table access and providing little advantage in improving the consistency of the view. Unlike a database, since the context from each element is entirely separate, there is no particular need for the response to View to be perfectly consistent. Writes to the Repository hash map are limited by a mutex, a mechanism which ensures only a single worker can modify the map at a time. This was included to prevent the situation in which one worker would read the Repository with the intent of making an update dependent on that read, then another worker would modify the data before the first would make their change. The repository hash map has the default concurrency level of 16, meaning that 16 workers may interact with the repository at the same time. Combined with the write mutex, that means that 16 threads may interact with the repository at the same time, one of which may make modifications.

If the request comes from a Context Provider updating the Host's context model, or the request is a simple locally scoped query from a Context Consumer, the worker thread interacts with the Host's Repository, updating or querying the context model depending on the form of the request.

In the event of a globally scoped query request, a Host's worker queries nearby Hosts with a locally scoped query to collect their context. These Hosts' context responses are a view of that Host's context XML, limited to the data accessible to the requesting consumer. These XML documents are combined with the Host's local context into a Situation view, an amalgam of all available context XML documents. The Host applies the consumer's query statement to that view, and finally returns the result to the querying Consumer.

When a Context related request is made of a Host, a worker thread builds a single XML document from the Repository's contents to generate a view of the available context. The worker

applies the query to the document and provides a response message to the requestor containing the result.

4.1.4.2 Context Consumers: Keeping Current

The Context Consumers, at their core, rely on making periodic requests, polling, to a Host in order to keep an up to date view of their environment. Our framework implements both Bluetooth and Wi-Fi clients. Each uses service discovery techniques described in Section 4.1.2 to find context Host endpoints. They then make a request to the Host, issuing a Letter of Mark and a context query, sent via a socket connection. The Host replies with the result of the query as the response of the request, signed appropriately.

In this implementation, the Context Consumers preferentially select Hosts to which they have already made requests. In a more complete implementation, it would become necessary for the Context Consumer to have a ‘whitelist’ of Host ids that it considered trustworthy, and would be willing to have operate on their behalf.

Maintaining a current view of the Situation is important for the Context Consumers to ensure timely adaptation to changes in context. There are two ways to achieve this, notification or polling. In the first solution, if the context a host is managing changes, the relevant Consumers, that is, Consumers that have requested that they be notified upon any change, are notified immediately. Our implementation has an important limitation: Consumers are only notified of changes to Context held on the Host to which they have subscribed.

An alternative to the subscription model is a polling system in which Consumers poll their Host to ensure that they stay up to date. This can be a costly network activity. To minimize the amount of querying, the Consumers must determine an appropriate polling time that keeps them up to date without wasting energy. Since the context elements are attributed with lifetime values, a simple query

that finds the minimum lifetime of all the query's relevant context values provides the Application with a useful polling frequency.

A final feature assists the Context Consumers in keeping up to date. When Hosts join an IP network, they advertise their arrival through DNS-SD[8]. A Context Consumer can passively monitor the network for such events and proactively request an update from the Host to which they subscribe for new context which might have arrived with the new Host. While not implemented, a Consumer might also actively monitor for changes in the presence of Bluetooth devices and react in a similar fashion to Context changes on the IP Network, by requesting updates to changes in the environment.

4.1.4.3 Context Providers: Maintaining a Host

Context Providers need to make decisions about which Host to use in a dynamic environment. The Context Providers and Context Consumers that use the IP Network to communicate with Hosts each maintains a list of preferred Hosts, identified by their public key. This list may consist of trusted Hosts (i.e. Hosts they are familiar with and comfortable relying on to honour their privacy) or simply preferred hosts. The context client selects the host they wish to use, uploads their context, makes queries, or requests notification subscriptions. From then on, they continue to interact with the same host. If the Host becomes unresponsive, or disappears from the network (their DNS-SD entry is removed), the clients immediately seek out another Host. There is no explicit notification system for services leaving a network, but if a DNS request is made, the other Hosts reply, thus updating the list of available Hosts.

4.2 Use Cases

In this section, the features of our pervasive platform implementation are demonstrated. A set of applications which were implemented as a part of this thesis are described that illustrate use cases and the breadth of the framework's abilities. In particular, these examples make use of the entire suite of functionality described in previous sections, including both Wi-Fi and Bluetooth network protocols, multiple hosts communicating, and applications which modify themselves by querying and adapting to the contextual Situation. The first use case is concerned with the framework's implementation; specifically with demonstrating the message passing between the various Actors, the use of the various network protocols, and basic context querying. The second use case demonstrates the relationships between Entities, private and public context, and the use of the schema's extensibility to provide specialized context without disrupting the delivery of basic context.

4.2.1 Use Case 1: An Adaptive Home Theatre

To demonstrate how devices might dynamically adapt themselves to a changing contextual environment, a set of Actors simulating different aspects of a living room with a home theatre were implemented.

There are three Entities in the example: a telephone device, a home theatre device, and the ambient lighting of a room. Each Entity has a Context Provider, which maintains the Entities' contextual states. The telephone provides context related to the ownership of the telephone, the telephone's state (i.e. ringing, connected, or waiting), the ring volume level (i.e. silent, vibrate, normal, or loud), and the phone's current location. The home theatre publishes context related to its DVD player feature set including the current state (i.e. stopped, playing, paused), and the television feature set including the

current input method. The ambient lighting Context Provider publishes context related to the current light level in the room (i.e. off, low, or high).

The home theatre, telephone and lights act also as Context Consumers. The Consumers each make dynamic and autonomous modifications to the devices' behaviour, and consequently to their context, as a function of changes observed in the Situation. The home theatre changes its own behaviour (and context) as a function of changes in the state of the telephone in the same room. The query used by the home theatre can be expressed in English as: "if there is a telephone located in the same room and its current state is 'ringing', and the ring volume is not set to 'vibrate' or 'silent', and the DVD player state is 'playing', then set the state to 'paused'." The query used by the lights Consumer can be expressed in English as: "if the state of the DVD player is currently 'paused' and was previously 'playing', then change the light level to 'high'; otherwise, if the state of the DVD player was previously 'stopped' or 'paused' and is now 'playing', set the light level to 'low'."

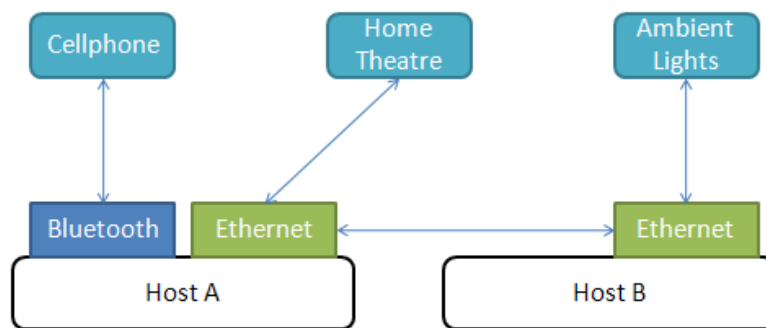


Figure 4-2: The relationships between the Entities and Hosts of the scenario.

The contexts of the three Entities are stored between two Context Hosts, Host A and Host B. *Host A* is enabled with both an IP network endpoint and a Bluetooth endpoint, while *Host B* has only an IP network endpoint. The telephone uses Bluetooth to register its context with *Host A*. The home theatre uses IP networking to register its context with *Host A*. The lights use IP networking to register their context with *Host B*. The various Entities and their relationships with the Hosts are illustrated in

Figure 4-1. Since all the Hosts have a mode of communication, the context for each Context Provider is available to any Context Consumer that makes a query via any network fabric.

A compiled Situation document in which the cellphone is ‘waiting’, the DVD player is ‘playing’ and the lights are ‘low’ is available in 5.2Appendix B.

4.2.1.1 *Determining the acceptable polling frequency*

To determine an appropriate polling period for updating the context, the home theatre system uses the following query to determine the minimum lifetime of the statuses of all telephones in the Situation. In English, this query would be, “for all the telephones, what is the minimum status lifetime no less than 5 seconds and no more than 30?” The 30 second limit ensures an upper bound on the polling frequency, to ensure that the home theatre is never too far out of date. The 5 second lower limit is applied to ensure that even if the lifetime of a telephone’s status is short, the available Hosts are not overwhelmed with queries. As an xQuery statement, that can be expressed as:

```
declare namespace device="http://cs.queensu.ca/DeviceBase";

let $lifetime:= //device:featureSet[@type="device:telephoneFeatureSet"]/
    device:status/xs:integer(@device:lifetime)

return
    if(count($lifetime)>0) then
        max((min(($lifetime),30)),5))
    else
        30
```

The lighting system uses a slightly more complicated query, searching specifically for the playing state of all DVD players that occupy the Location with colloquial label (a base Context element of Locations Context) “Living Room”, and determining the minimum lifetime between 5 and 30 seconds.

```
declare namespace base="http://cs.queensu.ca/ContextBase";
declare namespace device="http://cs.queensu.ca/DeviceBase";
declare namespace location="http://cs.queensu.ca/LocationBase";
declare namespace homeTheatre="http://cs.queensu.ca/HomeTheatreBase";
```

```

let $lifetime :=
//base:Entity[location:Location/location:Colloquial/string()="Living
Room"]/device:featureSet[@type="homeTheatre:dvdPlayerFeatureSet"]/homeTheatre:
dvdPlayerStatus/xs:integer(@base:lifetime)

return
  if(count($lifetime)>0) then
    min((max($lifetime)),30))
  else
    30

```

4.2.1.2 Applications Adapting to Context Use Case

The following example illustrates the context sharing capabilities of our platform. In particular, this example illustrates how the environment can become a dynamically changing system of elements which interact automatically without any previous exposure to the various devices. All of the discovery and registration steps outlined in Section 4.1.2 are assumed complete at this point.

The initial contextual state of the system is as follows: the home theatre's DVD player is 'paused', the light level is 'high', and the cellphone is in 'waiting' state. The DVD player begins playing, as a result of a user action. Detecting the DVD player's context change, the lights dim from 'high' to 'low'. After a while, the cellphone starts ringing, changing its context state to 'ringing'. The home theatre recognizes this and sets the DVD player to 'paused'. The change in state of the DVD player prompts the lights to be set to 'high'. The phone changes back to the 'waiting' state as a result of a user action (i.e. phone call terminated or sent to answering service). The DVD player is notified of the context change and changes its play state to "playing" again. The lights detect the DVD player's new context state and dim the lights accordingly. Figure 4-3 illustrates the contextual state changes of specific relevant elements of the various devices in this scenario, and the causal relationships between state changes. The curly braces indicate a change in state and arrows indicate a causal relationship between context changes.

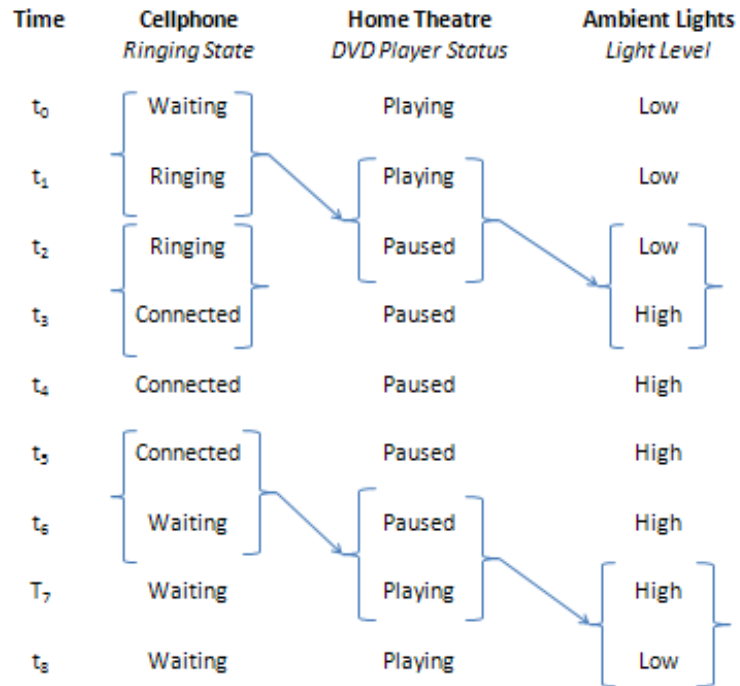


Figure 4-3: State for the context elements of three devices.

The DVD player uses the following xQuery statement to monitor the ringing contextual state of the telephones in the “living room” Location.

```

declare namespace base="http://cs.queensu.ca/ContextBase";
declare namespace person="http://cs.queensu.ca/PersonBase";
declare namespace device="http://cs.queensu.ca/DeviceBase";

for $status in
//base:Entity/device:featureSet[@type="device:telephoneFeatureSet"][not (device
:ringVolume="silent")] /device:status
where not ($status="waiting")
return count($status)

```

The Ambient Lighting system uses the following xQuery to monitor the room for the state of DVD players currently in the same room.

```

declare namespace base="http://cs.queensu.ca/ContextBase";
declare namespace person="http://cs.queensu.ca/PersonBase";
declare namespace device="http://cs.queensu.ca/DeviceBase";
declare namespace location="http://cs.queensu.ca/LocationBase";
declare namespace homeTheatre="http://cs.queensu.ca/HomeTheatreBase";

for $status in

```

```
//base:Entity[location:Location/location:Colloquial/string()="Living
Room"]/device:featureSet[@type="homeTheatre:dvdPlayerFeatureSet"]/homeTheatre:
dvdPlayerStatus/string()
return $status
```

4.2.2 Use Case 2: Cellphone Desktop Notifications

The second use case is designed to illustrate the ontological features of the context model. In this scenario, there is one Context Provider, one Context Consumer, and one Host. All use Ethernet for communication. The Context Provider is a cellphone. The Context Consumer is a desktop application that monitors the state of telephones belonging to the user currently logged into that machine. When the phone rings, the desktop application takes the caller-ID phone number and checks it against the user's Google Contacts address book, using a RESTful web service[44]. It then displays a notification using the Gnome system notification system [45] including the caller's name and contact information.

The cellphone Context Provider uses the type definitions of several schema documents to define its context. In the context document below, XML elements with the prefix "base" have been defined in the base context schema. The telephone device featureSet, is defined by the schema with namespace <http://cs.queensu.ca/Telephone> and referenced by the prefix "phone". That featureSet includes the basic features of a phone, including its phone number. It is extended in the schema with the namespace <http://cs.queensu.ca/Blackberry>, in which a device feature set that extends the basic telephoneFeatureSet with an incoming phone number caller ID feature is defined, reusing the generic 'telephoneNumber' type defined in the "phone" namespace. These namespaces are each included in

5.2Appendix A. In the ringing state, the context that device might provide would be as follows:

```

<?xml version="1.0" encoding="UTF-8"?>

<entity:Entity
  xmlns:situation="http://cs.queensu.ca/Situation"
  xmlns:host="http://cs.queensu.ca/Host"
  xmlns:base="http://cs.queensu.ca/ContextBase"

  xmlns:device="http://cs.queensu.ca/DeviceBase"
  xmlns:phone="http://cs.queensu.ca/Telephone"
  xmlns:bb="http://cs.queensu.ca/Blackberry"
  xmlns:person="http://cs.queensu.ca/PersonBase"
  xmlns:location="http://cs.queensu.ca/LocationBase"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:entity="http://cs.queensu.ca/Entity"
  xsi:schemaLocation="http://cs.queensu.ca/ContextBase ContextBase.xsd
    http://cs.queensu.ca/LocationBase LocationBase.xsd
    http://cs.queensu.ca/DeviceBase DeviceBase.xsd
    http://cs.queensu.ca/Telephone Telephone.xsd
    http://cs.queensu.ca/Blackberry Blackberry.xsd
    http://cs.queensu.ca/Entity Entity.xsd "

  id="AxngzBaGgI9kDSnruEjpTxfMbN2"
  xsi:type="device:DeviceType"
>

  <base:Relationship verb="ownership">
    <base:Subject>agOi32NLpLlzY40fC9auZ0kxqLa</base:Subject>
  </base:Relationship>

  <base:Location>
    <location:Colloquial>Office</location:Colloquial>
  </base:Location>

  <base:PermissionGroups>
    <base:PermissionGroup>
      <base:ContextID>AxngzBaGgI9kDSnruEjpTxfMbN2-callerid</base:ContextID>
      <base:EntityID>3gOi32NLpLlzY40fC9auZ0kxqLa</base:EntityID>
    </base:PermissionGroup>
  </base:PermissionGroups>

  <device:FeatureSet xsi:type="bb:BlackberryFeatureSet">

    <phone:TelephoneNumber>
      <phone:areacode>613</phone:areacode>
      <phone:number>533-6000</phone:number>
    </phone:TelephoneNumber>

    <phone:ringVolume>Quiet</phone:ringVolume>

    <phone:status>Ringing</phone:status>

    <bb:callerID contextID="AxngzBaGgI9kDSnruEjpTxfMbN2-callerid" private="true">
      <phone:areacode>613</phone:areacode>
      <phone:number>533-6001</phone:number>
    </bb:callerID>

  </device:FeatureSet>

</entity:Entity>

```

In the above example, there are other attributes expressed, whose definitions are inherited from ancestor context primitives. The caller ID value is identified as ‘private’, and the user with public key ‘3gOi32NLpLlZy40fC9auZOxqLa’ can access that value via the PermissionGroup. The Context Consumer can use the following xQuery to list the incoming phone numbers to ringing phones owned by the desktop user identified by the id “3gOi32NLpLlZy40fC9auZOxqLa”:

```
declare namespace base="http://cs.queensu.ca/ContextBase";
declare namespace device="http://cs.queensu.ca/DeviceBase";
declare namespace blackberry="http://cs.queensu.ca/Blackberry";

let $x:=
//base:Entity[base:Relationship[@verb="ownership"]/base:ObjectElement="3gOi32N
LpLlZy40fC9auZOxqLa"]/device:featureSet[@type="phone:telephoneFeatureSet"] [ph
one:status="ringing"]//blackberry:callerID

return $x
```

This use case illustrates the schema extensibility features. This cellphone has features that are not defined for a basic telephone. Beyond the simple ringing state and declaration of its phone number, it also contains the caller ID features defined outside of the basic cellphone schema. This extensibility enables Context Consumers with an understanding of Blackberry’s caller ID context to make use of it, without disturbing the ability of Context Consumers that lack this understanding to continue making use of the basic context features.

This example also makes use of the Relationships concept, stating that the cellphone is owned by a user identified by a certain public key. The desktop application can identify the phone as belonging to the user currently logged in and perform user-specific actions such as collecting relevant information about the caller and displaying on the user’s desktop.

There are privacy issues related to the caller ID information, which are illustrated in this use case as well. The incoming caller information should not be public knowledge. The Context Provider has marked the relevant branch of XML as private information by setting the xml public attribute to ‘false’

and has defined an `PermissionGroup` of permitted user keys. The Context Consumer, identified by the desktop user's key can then run queries that access this otherwise obscured section of context XML.

4.3 Testing

To assess the implementation's capacity to deliver timely responses to queries, reasonably supporting context sharing, aspects of query response performance were measured. Many factors can influence the performance of query response. For example, if a Context Consumer is communicating wirelessly with a Host, factors such as the network fabric and protocol, device hardware, the proximity, network interference, and network congestion unrelated to context sharing can all result in reduced performance. For these tests, a dedicated wireless network was used for the wireless IP network testing, unconnected with any other devices, with few other wireless network nearby. As well, for the Bluetooth testing, an environment outside the range of any other wireless devices was used.

4.3.1 Query Performance

Performing queries with XQuery is a central component of this framework. To better understand what fraction of the overall processing of a query is attributed to the evaluation of the query, a set of tests were performed on two different XQuery engines. These queries were performed within one machine, removing network effects from the experiment, and focusing on the query engine's performance alone.

To examine how Situation size affects the performance, four XML documents were composed each describing a Situation of one Host and 1, 20, 100, and 1000 Entities respectively; one can imagine an environment such as a living room containing dozens of Context Providers sharing their Entity documents. For the purpose of this experiment, the Entities are simple, each having only a telephone feature set with a telephone number. Two queries were run against each document: a simple query which returns the entire Situation with the query statement `"/`, returning all available XML; and the

more complicated query return a count of the number of ringing phones owned by a person whose first name is “Cyrus” expressed as follows:

```
...namespace declarations...
for $id in //base:Entity[person:Name/person:FirstName="Cyrus"]/@id

let $ringingPhones :=
//base:Entity[base:Relationship[@verb="ownership"]/base:ObjectElement=$id]/
  device:featureSet[@type="device:telephoneFeatureSet"][not(
    device:ringVolume="silent")] /device:status

where not($ringingPhones="waiting")

return count($ringingPhones)
```

These queries are not an effective measure of the XML querying engine’s performance generally, but do provide insight into the effectiveness of the library within the scope of its use in this platform, quantifying the query engine’s performance in calculating the result or in the overhead of preparing the document and query for execution. These queries are typical of the types of queries expected for a Context query. Since processing XML queries is at the core of a Host’s ability, the goal is to keep the responsive time very low, in the order of a few millisecond. This implementation relies on the Apache Saxon HE 9.2 [46] XML processor processing the XQueries. The experiment was run on a Linux machine with a 2.4GHz Intel processor and 2GB of RAM. The results are shown in Table 1.

	1 Entity Document		20 Entity Document		100 Entity Document		1000 Entity Document	
	Avg. Time (ms)	σ (ms)	Avg. Time (ms)	σ (ms)	Avg. Time (ms)	σ (ms)	Avg. Time (ms)	σ (ms)
Simple Query	0.22	0.99	0.60	1.10	1.33	2.11	2.91	3.07
Complex Query	0.63	1.25	0.97	1.07	1.83	2.15	7.59	20.76

Table 1 Comparison of XQuery engines.

A few conclusions and observations can be made from these results. First, for each of the query types and document sizes, the engine is able to handle the queries sufficiently quickly – within a few

milliseconds. While query completion time does grow as the document size grows, the time per-entity in the document decreases. Likely, the query overhead is being spread among more entities, lowering the per-entity cost of the query. There is a large variance to the response times, around double the mean. It is unclear why the engine was so varied in its response times. It is possible that the sample set of 1000 queries was too small to be statistically significant. In any case, even a standard deviation from the mean, then query engine performs well.

4.3.2 Client Requests via TCP/IP Network Sockets

To measure the robustness of our Host's architecture and its ability to handle a large number of incoming concurrent queries, several tests were performed, to show that the framework can support hundreds of incoming requests, each with response times measured at a few seconds or less. Two desktop computers, running Linux, with 2.4Ghz processors and 2GB of RAM were connected via an 802.11g network. The first provided a Host managing a single device's Entity. The second queried the first with multiple querying threads to simulate concurrent requests. Each thread was made to query the host for a complete listing of the publicly available context on that Host (i.e. a local scope query) 100 times. To ensure that the measurements were being made for the prescribed number of concurrent tabulation was halted when the first client completed its run of queries. Host query throughput, and average query completion time were each examined under these conditions.

For small numbers of concurrent requests, the framework's throughput increases with the number of requests. Between 20 and 25 concurrent clients, the host begins to reach its maximum throughput of approximately 275 requests per second. Beyond 25 concurrent clients, the system throughput remains flat. The results are plotted in Figure 4-4. From this plateau, and the previous experiment's conclusion that showed the query handler was able to handle a much larger number of

concurrent requests with millisecond-order response times, we can conclude that the framework has some network and request management overhead causing this limitation, rather than the querying engine itself.

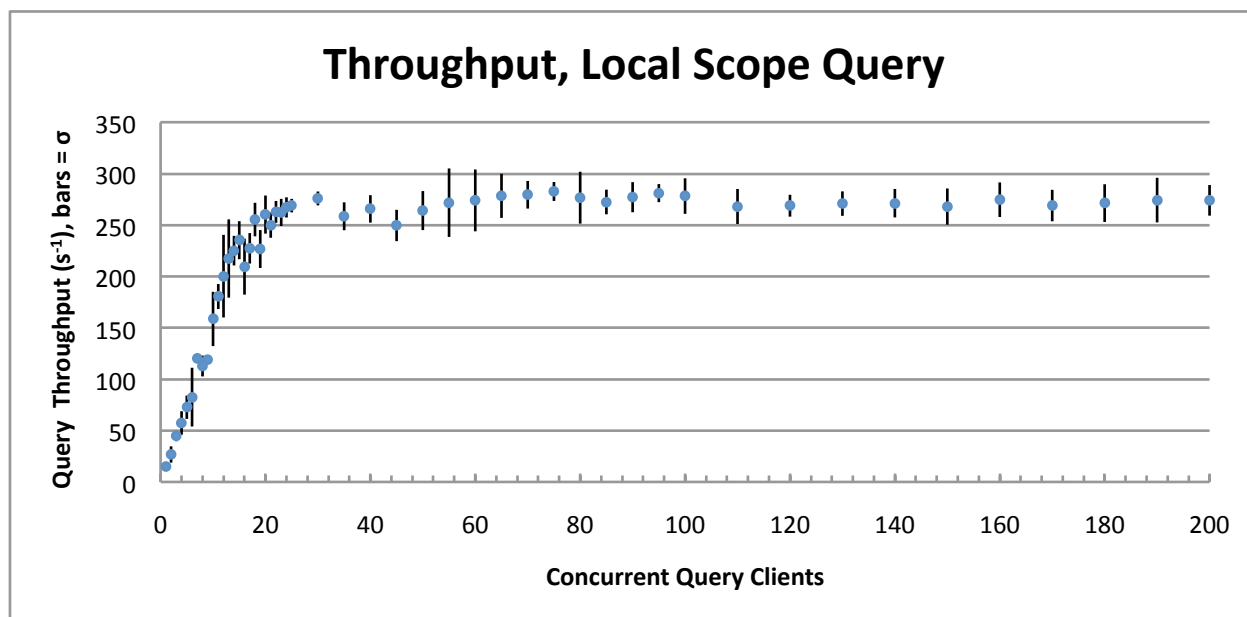


Figure 4-4 Throughput for a local scope query, with bars showing 1 standard deviation

The query completion times for the same experiment are plotted in Figure 4-5. For the first 20 or so concurrent clients, the average response time remains between 60 and 70 ms, with a standard deviation of about 10 milliseconds. As the request overhead begins to limit throughput and the Host becomes the limiting factor, the response times become slower and much more variable. In the range between 20 and 200 concurrent requests, the average response time increases linearly ($R^2 = 0.999$).

The standard deviation of the response times was as much as half of the query time. This is likely a result of a network limitation, such as network saturation, or the network interface being unable to process so many simultaneous connections, or may be the framework implementation's own request handling overhead.

These experiments shows that, over an wireless IP network, the framework is able to handle hundreds of incoming requests with sub-second response times.

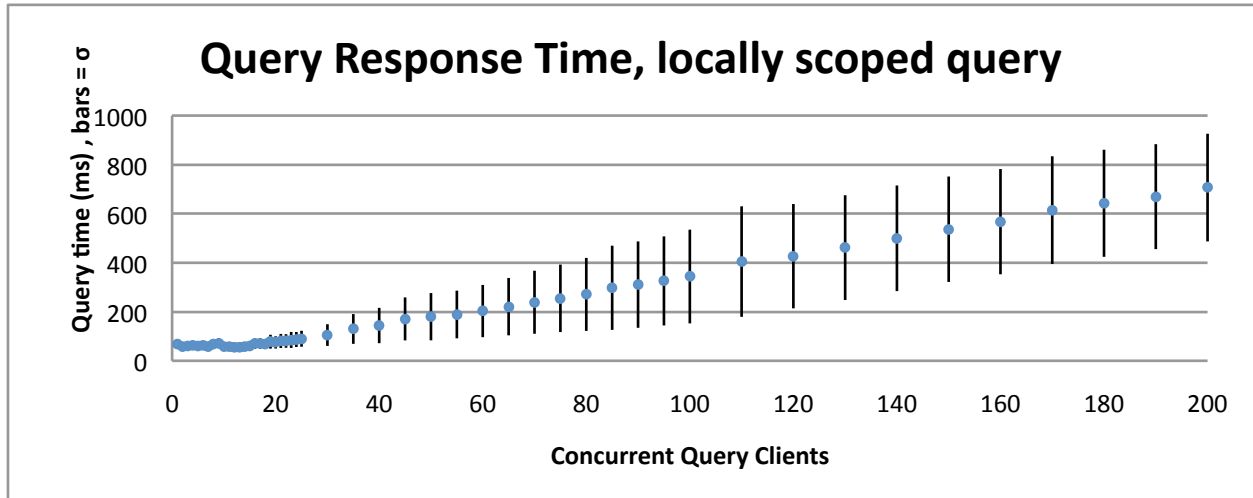


Figure 4-5 Query response time, locally scoped query, with bars showing 1 standard deviation

A final query response test was performed to evaluate the framework's performance with multiple Hosts in a Situation. This experiment best models what a real environment might look like: devices interacting with one Host which must consult with other Hosts to satisfy the query. Since this implementation of the framework does not include software that would allow one Host caching context from other Hosts, every request must be relayed to all Hosts. This experiment was performed with a set of ten and a set of fifty Consumers querying a Host with a globally scoped query for Situations with between one and ten other Hosts on the network. Each of these consumers queries their Host constantly, effectively giving ten and fifty concurrent requests at all times. The Hosts were distributed between two linux desktops each with 2.4GHz processors and 2GB of RAM, connected via a wired 100Mbit Ethernet network. The goal was to be able to respond to any request within a few seconds. The request throughputs for each are plotted in Figure 4-6.

Several observations about the network's behaviour can be made. With a single Host, the response times differ slightly depending on the number of Consumers, though the mean of each is within one standard deviation of the mean of the other. Once multiple hosts are involved, any difference erodes, leaving a difference of only a few requests per second. This comparison implies that the limiting factor is the cost of collecting the Context from each of the other Hosts in the Situation. The query can only be processed when each of the other Hosts has responded, which means the slowest Context collecting response is the fastest possible response the Host can make to the Client. Comparing the results for these globally scoped query tests with the locally-scoped query tests before them show that accessing the list of Host services in the Situation, a list maintained by JmDNS, is a bottleneck. The results for a single Host, but a globally scoped query result in an overall throughput of approximately 27 and 36 requests processed per second, while the locally scoped queries (also to a single Host) processed between 200 and 300 requests per second.

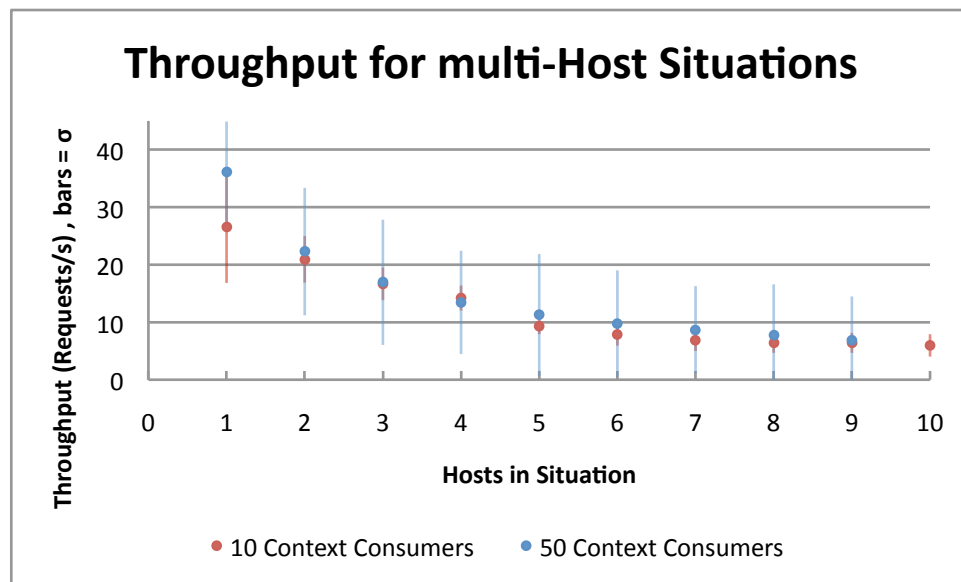


Figure 4-6 Host throughput for multi-Host Situations, with bars showing 1 standard deviations

Evidently, getting a fresh topology of the Hosts in the area using the DNS-SD service is expensive, and further implementations would likely need a better implementation of the DNS-SD library, which is supposed to maintain an fast lookup of the available devices.

The response times from the Consumer's perspective is plotted in Figure 4-7, with 10 and 50 concurrent Consumer queries for various numbers of Hosts in the Situation. The results show that the response times are very sensitive to the number of Consumers and the number of Hosts in the Situation, linearly increasing with the number of Hosts ($R^2 = 0.999$ and $R^2 = 0.990$ for 10 and 50 concurrent Consumer queries respectively).

While with 10 concurrent requests the Host infrastructure performs quite consistently, with 50 concurrently requesting clients, the query time becomes much more variable, with a standard deviation up to 1/3 of the mean. The source of the variance is not certain, but it is likely network related, as the number of requests being passed on the network increases in proportion with the product of clients and Hosts.

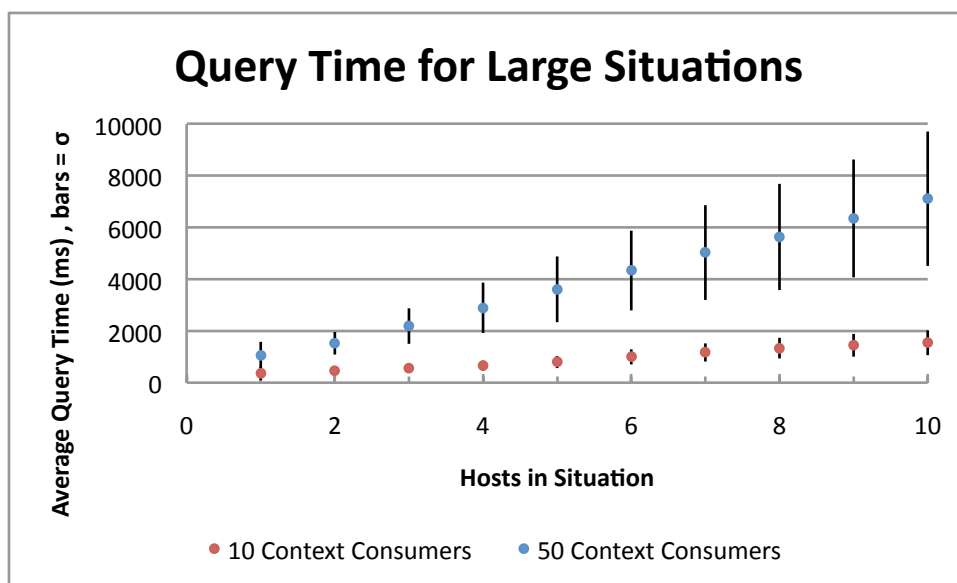


Figure 4-7 Query Speed for multi-Host Situations, with bars showing 1 standard deviation

This experiment demonstrates that the framework is able to support many clients, all operating simultaneously in the same environment. With ten Consumers making requests constantly in an environment with ten Hosts, the requests are all handled within a few seconds. With a Host network with 50 concurrent active requests, the limits of the infrastructure start to appear with a network of 4 or 5 Hosts. Features such as Host-based context caching and a context-change notification infrastructure might reduce the need for active polling and improve the framework's performance.

4.3.3 Bluetooth Network Query Performance

The tests designed to measure the performance of the Bluetooth Host endpoint are not comprehensive due to limitations with the Bluecove Bluetooth Java library: this library limits the number of outbound connections to one, making it impossible to simulate multiple Bluetooth devices through one adapter.

In the first test, a single Bluetooth enabled device, a linux laptop with 1GB of RAM and a 2.3GHz AMD processor, queried a Bluetooth Host, a desktop computer with 2GB of RAM and a 2.4GHz Intel processor. In the second, two Context Consumers (an 2.3GHz AMD laptop and a 1.8GHz AMD laptop, both with 2GB RAM), simultaneously connected to a Bluetooth Host, a desktop computer (a 2.4GHz Intel processor with 2GB of RAM). All three machines in both experiments were using the Bluecove implementations of the Bluetooth networking stack.

The Consumers, as in the IP Network connection tests, each requested a complete view of the Host's local Context. Physically, the machines were in close proximity, spaced approximately 1 metre apart, well within the 10 meter limit of the devices[9]. In each test, the clients repeated their queries 100 times each, effectively giving two concurrent requests throughout the experiment.

With a single Context Consumer, the Host was able to process requests in 112 milliseconds on average with a standard deviation of 30 milliseconds. Two Consumers resulted in response times of on

average 218 milliseconds with a standard deviation of 60 milliseconds. In effect, it took nearly twice as long to process two concurrent queries as it did to process one, suggesting that the Bluetooth stack might be limited to one concurrent request at a time. The concurrent connection bottleneck is certainly a problem, but for a sufficiently small number of devices, the query response times are adequate with a sufficiently small variance. While Bluetooth may not be a network fabric that supports much more than a handful of simultaneous device requests, it would be functional in small environments.

4.3.4 Context Service Discovery, IP Network and Bluetooth

Device discovery is an essential component of this framework. When a Context Provider or Context Consumer enters a Situation, they need to be able to discover Hosts with which they can interact. As well, a Host must be aware of other Hosts that might be hosting Context relevant to the query. Tests were performed to determine how quickly devices could be discovered under ideal network conditions, using both Ethernet and Bluetooth networking protocols.

DNS-SD provides the device discovery technology for IP network communication. The average time for Hosts to discover new entries on the network was tested, from the moment a new Host publicises its presence to the time when other hosts are aware of the device. Two networked computers alternately issued new service records to each other, each publishing the moment of receipt of service record from the other. The time between when a device published a service and when it received the service publication message from the other is twice the time required to publish a new service.

The service publication time was measured for 1000 service publishing events, and on average were 4218 milliseconds with a standard deviation of 27 milliseconds (very consistent). While these results show that device discovery is not to be considered instantaneous, it is a short enough time to be considered viable for this framework.

Unlike Host discovery using the IP network, the Bluetooth discovery process is active rather than passive: devices must be actively searching for devices and services. The time it takes to discover a new device is dependent on the time it takes for devices to respond to discovery requests as well as the frequency at which the searching device queries the environment.

Discovery of Hosts through Bluetooth is a two-part process. The devices supporting Bluetooth must first be discovered using the Bluetooth discovery protocol. Once a list of nearby devices has been discovered, each device must be searched for relevant services. The device search must be performed regularly, while the service search need only be performed once when the device is first discovered.

Device discovery is performed by issuing a request for all nearby devices to respond. Requests have a default timeout of 10 seconds. While service searches could theoretically begin immediately upon a device's discovery, the Bluecove Bluetooth API limits interaction with the Bluetooth stack to one connection at a time. The testing environment was free of all other Bluetooth devices, and the devices were approximately 1 meter apart, well within the specified Bluetooth range. Over 100 attempts, a service search took on average 234 milliseconds with a standard deviation of 65 milliseconds, demonstrating that for a small set of devices, devices can get an update of their environment in a timely fashion.

4.3.5 Leaving the Situation

Equally important to the speed with which devices can be discovered is the manner in which devices disappear and the speed with which this information is delivered to a Host. To maintain an accurate representation of the Situation, Hosts must be aware of the arrival and disappearance of Context Providers. This problem has several parts: the manner in which devices leave the situation, the manner in which Context Providers leave the Situation, and the speed with which each of these events are detected by other devices in the Situation.

Context Providers, Context Consumers and Hosts need to know when Hosts disappear. Context Providers and Context Consumers need to know because they form a relationship with an individual Host for querying and registering Context, and if that host disappears they need to be able to transfer their Context to a new Host as quickly as possible. Hosts need to know because they query other Hosts to collect context for queries. If this is done “gracefully”, the Host will unregister its service record from the LAN actively. 100 service de-registrations were performed, and on average, a graceful Host deregistration event was discovered in 103 milliseconds with a standard deviation of 2 milliseconds. Actors are therefore consistently and promptly notified of a device’s disappearance if they disappear gracefully. Since many devices are mobile, it is possible for devices to leave “gracelessly”, moving outside the range of a particular wireless network. Under these circumstances, the Host may not properly remove its own service record from the DNS caches of the other devices, in which case the Host’s disappearance is not discovered as quickly. Dynamic DNS Update Leases (DNS-UL) [47] makes provisions for this, adding a lease expiry time to the service’s DNS record. The default record expiry is 1800 seconds, but for mobile devices, a shorter lease time can be applied. The Host is required then to re-issue the record at least as frequently as the lease (the DNS-UL) draft recommends at 75% of the lease. As well, provisions are made in this implementation that if a query reaches a timeout of 10 seconds, Context Consumers and Context Clients will seek out another, hopefully more reliable, Host to maintain their context.

To maintain awareness of the disappearance of Bluetooth Hosts, Bluetooth Context Consumers and Providers must regularly search the environment for the presence of Hosts using the Bluetooth discover protocol, a query which has a timeout of 10 seconds. As well, a failure to connect to a Host causes a client to seek out other more reliable Hosts.

If a Context Provider using either Bluetooth or LAN leaves a Situation, it can exit gracefully by unregistering its context document from its Host. This is done by submitting a context document with a lapsed expiry timestamp. The Host purges all expired context at the next query. If a Context Provider leaves the Situation gracelessly the Host responsible for its context will continue to distribute the Provider's context until its expiry. It is therefore the responsibility of the Provider to choose appropriate expiries for its context based on its mobility and expected departure from the Situation.

4.4 Experimental Conclusions

The implementation of the described framework was tested with a focus on its ability to handle requests in different configurations and over IP and Bluetooth network fabrics, demonstrating its viability as a context sharing framework. The context querying engine can process requests in milliseconds, even with hundreds of concurrent requests. Over an IP network, the software processes requests in a few seconds, even when handling many simultaneous multi-Host context requests. These experiments demonstrate the implementation successfully supports context sharing in a timely fashion for IP networks. Performance improvements could be made by allowing Host-based context caching, so that multi-Host requests are not necessary, and in particular, by addressing the performance bottleneck in the service discovery mechanism, DNS-SD/JmDNS. For Bluetooth interactions, context requests can be performed in a similar the same time-frame as IP requests, so long as the number of concurrent requests is small. Making use of a Bluetooth stack that can handle multiple concurrent requests would address the network fabric's greatest bottleneck.

Chapter 5 Conclusion and Future Work

Presented in this final chapter is a summary of the thesis, followed by a discussion of the limitations of the design and further work that would address these limitations.

5.1 Thesis Summary

Computers are increasingly a pervasive fixture of our day-to-day lives, permeating everything we do. New generations of hardware are increasingly aware of their surroundings through the addition of more and better sensor systems, networking technologies, and further integration with data services. These sources of information allow devices to adapt and reconfigure themselves according to the circumstances in order to better interact and assist the user in their intended task.

These data which characterize the circumstances and surroundings of devices and their users constitute context. Devices are currently limited to the data which they are able to collect via their own sensors or in a limited number of circumstances through proprietary protocols. Those computing devices which are trying to assist their users could benefit greatly from a protocol for exchanging contextual data in a fashion that is secure and in a language that is both expressive and extensible.

This thesis describes a framework that allows devices to share their contextual data, giving them a better representation of their surroundings, and includes the overview and test results of an implementation of this framework. The framework offers three specific contributions. First, a modeling language for expressing context is outlined with specific provisions for its interpretation (querying),

security and privacy. Second, roles for producing, sharing and consuming these data are defined. Third, a set of protocols are outlined to facilitate the communication of context between the various devices while maintaining the privacy and security features of the context model.

Using XML and XML schema to express context has several valuable consequences. The context data are ensured to be expressed in the same way for all devices, which is critical for interoperability. XML Schema has provisions for type extension, which means that while primitive and common context elements are predefined, the context model can evolve and expand to represent new data, and to refine primitive definitions with new expressions all while maintaining backward compatibility. As well, XML and XML Schema are well known technologies, with many powerful tools for producing and interacting with them. In particular, XQuery is an XML querying tool used in this framework's implementation for interpreting context documents.

This thesis identifies three roles of context usage: Context Producers, devices that have sensors and other context data they wish to share; Context Consumers, those that are able to adapt themselves to changes in a situation's context; and Context Hosts, those that facilitate the exchange between the other actors, often acting as a form of context proxy. By constructing the infrastructure around these roles, with communication protocols between each, devices are able to implement only the components that they need. Thus Context Producers with limited power or wireless range are able to offload most of the network communication to other devices with more resources.

The framework's messaging protocol facilitates the sharing of context, in a novel fashion. A Context Consumer is able to issue a request to a Context Host to fetch the contextual data of all nearby devices, process a query, and return the results. To authorize a Host to access private data on behalf of the Consumer, a cryptographically signed limited-time query request is passed from the consumer to the Host. Thus, the Host is able to act on behalf of the Consumer, offloading the workload entirely.

A prototype implementation of the framework was produced to demonstrate each of the capabilities. Support for Bluetooth, IP Network, and mixed network topologies were implemented and tested in several configurations, examining the design's query throughput characteristics using different XML engines. The framework's Bluetooth implementation was functional, but limited in its capacity to serve concurrent requests, whereas the IP network solution showed robust support. A Host could process over 200 concurrent queries with response times of less than a second. A Situation with 10 Hosts representing multiple Entities was able to process 10 concurrent multi-Host queries with response times of less than 2 seconds, and at the extreme, 50 concurrent requests with response times of less than 6.5 seconds. These results demonstrate that this implementation provides adequate performance.

5.2 Future Work

While the framework design and demonstrative implementation offer a functional system through which contextual data may be passed, there are several avenues of research that could improve the performance and usefulness in real-world applications.

The current framework requires that a Host represent the full context document of any Entity in the network. A Context Provider must either choose to trust one of the available Context Hosts on the network, or provide its own instance of a Host. The Provider is forced to trust that their sensitive data will not be compromised to receive the energy savings of allowing a powerful Host to represent its context. Two immediate solutions come to mind that would lessen the trade-off, but introduce new complexities. First, a Context Provider couldn't encrypt the sensitive data with their own key before giving the data to the Host. This however implies that Hosts acting on behalf of the Context Consumer will not be able to process these data as a part of a query. Second, perhaps the representation of the data could be shared between two Hosts, one for the commonly available data which would satisfy most requests, and a second Host representing the sensitive context data. The cost for this solution would be

that queries would have to be broken down so that the Provider resolved the components of a query result that were sensitive while the general Host answered the rest.

To speed up implementation, the messaging protocols were implemented as socket connections passing serialized Java objects. Since Bluetooth and TCP/IP networks both support socket connections, the same infrastructure could be used for both networks. An open infrastructure such as this should not rely on Java serialized objects for formatting the request, but rather an open and perhaps network-dependent protocol. Many Web Service protocols would be well suited to this task.

Public key communication suffers from a common vulnerability called a Man in the Middle attack, which stems from the inability to verify that a public key is tied to a particular identity without centralization or an extensive trust network that might not be available. In this implementation, individuals were identified directly by their public key, meaning there is no distinction to be made. Unfortunately, this has some important limitations. If an individual needed to change their public key, they would be unable to, since their identity is tied to the public key. Some method of identity verification should be explored.

Discussed in this thesis, but stated as outside the scope of implementation was the topic of context caching. One of the slowest aspects of querying context is in the multi-host context queries. A clear improvement would be to add caching that relied on the timeliness attributes already tied to the context data to answer the Context Consumer's query without the expensive gathering task. The caching scheme outlined in ECSTRA[22] is particularly interesting, as it also assigns confidence values to the results, based on how recently the data were collected.

An important feature of the infrastructure designed in this thesis is the reliance on schema documents referenced by namespaces. For this implementation, the documents were processed with locally available schema documents. However, since the schemas can be extended, it is not always clear

what the inheritance relationship is without a local copy of the schema. An XML document may indicate a web URL at which the schema document for a referenced namespace resides. These documents in some circumstances would be necessary to properly evaluate a query of a context document.

The testing of the framework suggested that JmDNS was a performance bottleneck for determining the available Hosts in the local area IP network. For a high-traffic environment, faster implementations of DNS-SD or some form of caching might be in order.

While this framework provides a method by which context can be accessed and queried effectively, it does not consider a formalization of the layer above the simple querying. Having access to the context is only aspect of the problem, while policies which govern when and how devices are permitted to interact with a user are another. It may be that this is an entirely separate architecture, but it would be interesting to consider how user profiles expressed as a part of the context might provide rules or hints about the way in which these interactions might take place.

References

1. Marc Weiser. "The Computer for the 21st Century". In: *Pervasive Computing*, IEEE 1.1 (1992), pp. 19–25.
2. Marc Weiser. "Ubiquitous Computing". In: *Computing* 26.10 (1993), pp. 71–72.
3. B. Schilit, N. Adams, and R. Want. "Context-aware computing applications". In: *IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'94)*. 1994, pp. 89–101.
4. Official XMBC Website. [Online; accessed 1-5-2010]. May 2010. url: <http://xbmc.org>.
5. Michael C. Mozer. "The Neural Network House: An environment that adapts to its inhabitants." In: *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*. Ed. by M. Coen. Menlo Park, CA, 1998, pp. 110–114.
6. Home Controls, Inc. *Home Automation, Programmable Lighting, and Home Security Products from the Leader in Home Automation*. [Online; accessed 1-2-2010]. url: <http://www.homecontrols.com>.
7. Gartner Inc. *Worldwide Mobile Device Sales Grew 13.8 Percent in Second Quarter of 2010, But Competition Drove Prices Down*. [Online; accessed 2-9-2010]. url: <http://www.gartner.com/it/page.jsp?id=1421013>.
8. Krochmal Marc Cheshire Stuart and Apple Inc. "DNS-Based Service Discovery IETF Draft". In: *Internet RFC 4844*, ISSN 2070-1721 (2010). url: <http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt>.

9. Bluetooth SIG, Inc. *The Official Bluetooth Technology Info Site*. [Online; accessed 2-9-2010]. 2003. url: <http://www.bluetooth.com/SiteCollectionDocuments/rfcomm1.pdf>.
10. Dey Anind K. Salber Daniel and Gregory D. Abowd. "The Context Toolkit: Aiding the Development of Context-Enabled Applications." In: *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*. 1999, pp. 434 –441.
11. Rahul Gupta, Sumeet Talwar, and Dharma P. Agrawal. "Jini Home Networking: A Step toward Pervasive Computing". In: *Computer* 35.8 (2002), pp. 34–40. issn: 0018-9162.
12. Oracle Corporation. *Java ME and Java Card Technology*. [Online; accessed 17-2-2012]. url: <http://www.oracle.com/technetwork/java/javame/index.html>.
13. Stephen S. Yau et al. "Reconfigurable Context-Sensitive Middleware for Pervasive Computing". In: *IEEE Pervasive Computing* 1.3 (July 2002), pp. 33–40. issn: 1536-1268. doi: 10.1109/MPRV.2002.1037720.
14. R Masuoka et al. "Ontology-enabled pervasive computing applications". In: *IEEE Intelligent Systems* 18.5 (2003), pp. 68–72.
15. World Wide Web Consortium (W3C). "OWL-S: Semantic Markup for Web Services." In: *W3C Member Submission* (Nov. 2004). Ed. by Mike Dean et al. [Online; accessed: 27-11-2011.] url: <http://www.w3.org/Submission/OWL-S/>.
16. Christian Becker et al. "BASE - A Micro-Broker-Based Middleware for Pervasive Computing". In: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications. PERCOM '03*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 443–. isbn: 0-7695-1893-1.

17. Harry Chen, Tim Finin, and Anupam Joshi. "An ontology for context-aware pervasive computing environments". In: *The Knowledge Engineering Review* 18.3 (2003), pp. 197–207.
18. Harry Chen, Tim Finin, and Anupam Joshi. "Using OWL in a Pervasive Computing Broker". In: *Workshop on Ontologies in Agent Systems, AAMAS-2003* (2003).
19. Marco Mamei and Franco Zambonelli. "Programming pervasive and mobile computing applications with the tota middleware". In: *PerCom 2004. Proceedings of the Second IEEE Annual Conference on Pervasive Computing*. 2004, pp. 263–273.
20. Tao Gu, Hung Keng Pung, and Da Qing Zhang. "Toward an OSGi-Based Infrastructure for Context-Aware Applications". In: *IEEE Pervasive Computing* 3.4 (Oct. 2004), pp. 66–74. issn: 1536-1268. doi: 10.1109/MPRV.2004.19.
21. Mark Assad et al. "PersonisAD: Distributed, Active, Scrutable Model Framework for Context-Aware Services Pervasive Computing". In: *Pervasive Computing* (2007). Ed. by Anthony LaMarca, Marc Langheinrich, and Khai N. Truong. Vol. 4480. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2007. Chap. 4, pp. 55–72. isbn: 978-3-540-72036-2. doi: 10.1007/978-3-540-72037-9_4.
22. Andrey Boytsov and Arkady Zaslavsky. "ECSTRA: distributed context reasoning framework for pervasive computing systems". In: *Proceedings of the 11th international conference and 4th international conference on Smart spaces and next generation wired/wireless networking. NEW2AN'11/ruSMART'11*. St. Petersburg, Russia: Springer-Verlag, 2011, pp. 1–13. isbn: 978-3-642-22874-2.

23. Andrei Olaru and Cristian Cratie. "Agent-Base, Context-Aware Information Sharing for Ambient Intelligence". In: *International Journal on Artificial Intelligence Tools (IJAIT)* 20 (2011), pp. 985–1000. doi: 10.1142/S0218213011000498.
24. Eric Karmouch. *Design and Implementation of Ambitalk, A Ubiquitous System for the Automatic Adaptation of Mobile Communication*. Queen's University MSc Thesis. 2006.
25. Stephen S. Yau, Yu Wang, and Dazhi Huang. "Middleware Support for Embedded Software with Multiple QoS Properties for Ubiquitous Computing Environments". In: *Proc. 8th Intl. Workshop Object-Oriented Real-Time Dependable Systems*. 2003, pp. 250–256.
26. Jared Zebedee. "An Adaptable Content Management Framework for Pervasive Computing". MA thesis. Kingston, Ontario: Queen's University, 2009.
27. Dieter Fensel et al. "OIL: An Ontology Infrastructure for the Semantic Web". In: *IEEE Intelligent Systems* 16.2 (Mar. 2001), pp. 38–45. issn: 1541-1672. doi: 10.1109/5254.920598.
28. World Wide Web Consortium (W3C). "Extensible Markup Language (XML) 1.0 (Fifth Edition)". In: *W3C Recommendation* (Nov. 2008). Ed. by C. M. Sperberg-McQueen Eve Maler Francois Yergeau Tim Bray Jean Paoli. [Online; accessed 24-01-2012]. url: <http://www.w3.org/TR/2008/REC-xml-20081126/>.
29. World Wide Web Consortium (W3C). "XML Schema". In: *W3C Recommendation* (Oct. 2004). Ed. by Priscilla Walmsley David C. Fallside. [Online; accessed 1-6-2010]. url: <http://www.w3.org/TR/xmlschema-0/>.
30. World Wide Web Consortium (W3C). "Element Declarations". In: *XML Schema Part 1: Structures Second Edition* (Oct. 2004). Ed. by Thompson Henry S. et al. [Online; 24-10-2010]. url: http://www.w3.org/TR/xmlschema-1/#cElement_Declarations.

31. World Wide Web Consortium (W3C). "XML Path Language (XPath)". In: *W3C Recommendation* (Nov. 1999). Ed. by Steve DeRose James Clark. [Online; accessed 17-2-2012]. url:
<http://www.w3.org/TR/xpath/>.
32. World Wide Web Consortium (W3C). "XQuery 1.0: An XML Query Language (Second Edition)". In: *W3C Proposed Edited Recommendation* (Apr. 2009). Ed. by Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simon, Scott Boag, Don Chamberlin. [Online; accessed 17-2-2012]. url:
<http://www.w3.org/TR/2009/PER-xquery-20090421/>.
33. World Wide Web Consortium (W3C). "The official XSL Transformations (XSLT) specification". In: *W3C Recommendation* (Nov. 1999). Ed. by James Clark. [Online; accessed: 17-2-2012]. url:
<http://www.w3.org/TR/xslt/>.
34. Karlton Philip Freier Alan O. and Paul C Kocher. "The SSL Protocol". In: *The Internet Engineering Task Force (IETF), Internet Draft version 3* (Nov. 1996). url: <http://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00>.
35. D. Boneh. "Twenty years of attacks on the RSA cryptosystem". In: *Notices of the American Mathematical Society (AMS)* 46.2 (1999), pp. 203–213.
36. Oracle Corporation. *Java Standard Edition (SE)*. [Online; accessed 17-2-2012]. url:
<http://www.oracle.com/technetwork/java/javase/documentation/index.html>.
37. S Matthew Gast. *802.11 Wireless Networks: The Definitive Guide, Second Edition*. Sebastopol, CA: O'Reilly Media Inc., 2005. isbn: 978-0-596-10052-0.
38. Daniel H. Steinberg and Stuart Cheshire. *Zero Configuration Networking, The Definitive Guide*. Sebastopol, CA: O'Reilly Media Inc., 2006.

39. Bluetooth SIG, Inc. *A Look at the Basics of Bluetooth Wireless Technology*. [Online; accessed 17-2-2012]. url: <http://www.bluetooth.com/Pages/Basics.aspx>.
40. BlueCove Team. *Bluecove, Java Library for Bluetooth*. [Online; accessed 1-6-2009]. url: <http://code.google.com/p/bluecove/>.
41. Stuart Cheshire. *DNS SRV (RFC 2782) Service Types*. [Online; accessed 14-2-2009]. 2006. url: <http://www.dns-sd.org/ServiceTypes.html>.
42. Arthur van Hoff. *JmDNS*. [Online; accessed 11-6-2010]. url: <http://jmdns.sourceforge.net/>.
43. Oracle Corporation. "Class ConcurrentHashMap<K,V>". In: *Java SE 6 Documentation*. [Online; accessed 17-11-2011]. url: <http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentHashMap.html>.
44. Google Inc. "Portable Contacts API". In: *Developer's Guide* (2010). [Online; accessed 27-03-2010]. url: http://code.google.com/apis/contacts/docs/poco/1.0/developers_guide.html.
45. Canonical Ltd. *NotifyOSD, Onscreen Display Notifications*. [Online; accessed 27-03-2010]. url: <https://wiki.ubuntu.com/NotifyOSD>
46. Michael H. Kay. *Saxon: The XSLT and XQuery Processor*. [Online; accessed 01-09-2009]. url: <http://saxon.sourceforge.net/#F9.3HE>.
47. Cheshire Stuart et al. "Dynamic DNS Update Leases, DNS Service Discovery (DNS-SD)". In: *Internet Draft* (Sept. 2006). [Online; accessed 10-09-2007]. url: <http://files.dns-sd.org/draft-sekar-dns-ul.txt>.

Appendix A Context Schemas

There are several schema documents included in this Appendix. These documents provide the basis for expression of all context documents presented in this thesis, and provide a basis from which schemas to model all sorts of other forms of Context can be built.

The “ContextBase” usually referenced by the prefix “base:” provides definitions for many primitive types. From these primitives, the general form of context’s representation is codified.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:base="http://cs.queensu.ca/ContextBase"
  xmlns:location="http://cs.queensu.ca/LocationBase"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  elementFormDefault="qualified"
  targetNamespace="http://cs.queensu.ca/ContextBase"
  xsi:schemaLocation="http://cs.queensu.ca/LocationBase LocationBase.xsd ">

  <import
    namespace="http://cs.queensu.ca/LocationBase"
    schemaLocation="LocationBase.xsd" />

  <annotation>
    <documentation>
      This is the base schema, defining the essential components
      for context's representation. Extensions may further extend
      the entity types to represent data in specific ways.
    </documentation>
  </annotation>

  <complexType abstract="true" name="EntityType">

    <annotation>
      <documentation>
        The EntityType is an abstract complexType. Entities must
        use an extending type with this as a base to represent
        an entity in a Situation.
      </documentation>
    </annotation>

    <sequence>
      <element maxOccurs="unbounded" minOccurs="0"
        name="Relationship" type="base:RelationshipType"/>
      <element maxOccurs="1" minOccurs="0"
        name="Location" type="location:LocationType"/>
      <element maxOccurs="1" minOccurs="0"
        name="PermissionGroups">

        <annotation>
          <documentation>
            Entity access is controlled through this group.
            While Context elements marked with attribute
            public set to false will not be broadcast,
            exceptions can be made by creating a reference
          </documentation>
        </annotation>
      </element>
    </sequence>
  </complexType>

```



```

        to both the element and the permitted entity.
    </documentation>
</annotation>
<complexType>
    <sequence>
        <element maxOccurs="unbounded" minOccurs="0" name="PermissionGroup"
type="base:PermissionGroupType" />
    </sequence>
</complexType>
</element>
</sequence>

<attribute name="id" type="ID" use="required"/>

</complexType>

<complexType name="RelationshipType">
    <sequence>
        <element name="Subject" type="IDREF" maxOccurs="unbounded" minOccurs="1"/></element>
    </sequence>
    <attribute name="verb" type="string"/></attribute>
</complexType>

<complexType name="PermissionGroupType">
    <annotation>
        <documentation>
            This type lists the references to entities with common
            group permissions.
        </documentation>
    </annotation>
    <sequence>
        <element name="ContextID" type="IDREF" maxOccurs="unbounded" minOccurs="1"/></element>
        <element name="EntityID" type="IDREF" maxOccurs="unbounded" minOccurs="1"/></element>
    </sequence>
</complexType>

<simpleType name="keyType">
    <annotation>
        <documentation>
            This element defines a base 64 encoded RSA public key.
            Allowed characters are A-Z, a-z, . (period), and /
            (forward slash).
        </documentation>
    </annotation>
    <restriction base="ID">
        <pattern value="[a-zA-Z0-9./] +"/>
    </restriction>
</simpleType>

<complexType abstract="true" final="restriction" name="ContextType">
    <attributeGroup ref="base:ContextAttributes"/>
</complexType>

<attributeGroup name="ContextAttributes">
    <attribute name="contextID" type="ID"/>
    <attribute name="lifetime" type="integer" use="optional"/>
    <attribute name="expiry" type="long" use="optional"/>
    <attribute name="mutability" type="boolean" use="optional"/>
    <attribute name="private" type="boolean" use="optional"/>
</attributeGroup>

<complexType name="ContextString">
    <simpleContent>
        <extension base="string">
            <attributeGroup ref="base:ContextAttributes"/>
        </extension>
    </simpleContent>
</complexType>

<complexType name="ContextInteger">

```

```

    <simpleContent>
      <extension base="integer">
        <attributeGroup ref="base:ContextAttributes"/>
      </extension>
    </simpleContent>
  </complexType>

  <complexType name="ContextFloat">
    <simpleContent>
      <extension base="integer">
        <attributeGroup ref="base:ContextAttributes"/>
      </extension>
    </simpleContent>
  </complexType>

  <complexType name="ContextBoolean">
    <simpleContent>
      <extension base="boolean">
        <attributeGroup ref="base:ContextAttributes"/>
      </extension>
    </simpleContent>
  </complexType>
</schema>

```

The Host and Situation schemas define how a collection of Entities and a collection of Hosts will be represented to Context Consumers.

Situation Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema

  targetNamespace="http://cs.queensu.ca/Situation"
  xmlns:tns="http://cs.queensu.ca/Situation"

  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"

  xmlns:host="http://cs.queensu.ca/Host"
  >

  <import schemaLocation="ContextBase.xsd"
    namespace="http://cs.queensu.ca/ContextBase"></import>
  <import schemaLocation="Host.xsd" namespace="http://cs.queensu.ca/Host"></import>

  <element name="Situation" type="tns:SituationType"></element>

  <complexType name="SituationType">
    <sequence>
      <element ref="host:Host" minOccurs="0" maxOccurs="unbounded"></element>
    </sequence>
  </complexType>

</schema>
```

Host Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema

  targetNamespace="http://cs.queensu.ca/Host"

  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:host="http://cs.queensu.ca/Host" elementFormDefault="qualified"

  xmlns:base="http://cs.queensu.ca/ContextBase"
  xmlns:entity="http://cs.queensu.ca/Entity"
  >

  <import schemaLocation="ContextBase.xsd" namespace="http://cs.queensu.ca/ContextBase" />
  <import schemaLocation="Entity.xsd" namespace="http://cs.queensu.ca/Entity" />

  <element name="Host" type="host:HostType"></element>

  <complexType name="HostType">
    <sequence>
      <element name="key" type="base:keyType" maxOccurs="1" minOccurs="0"></element>
      <element ref="entity:Entity" minOccurs="0" maxOccurs="unbounded">
      </element>
    </sequence>
    <attribute name="id" type="ID"></attribute>
  </complexType>

</schema>
```

The “Person Base” schema defines fundamental context attributes for a person. Privacy attributes are not expressly defined in this schema, but rather inherited from its primitives.

Person Base Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://cs.queensu.ca/PersonBase"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:base="http://cs.queensu.ca/ContextBase"
  xmlns:person="http://cs.queensu.ca/PersonBase">

  <import schemaLocation="ContextBase.xsd"
    namespace="http://cs.queensu.ca/ContextBase"></import>

  <complexType name="PersonType">
    <complexContent>
      <extension base="base:EntityType">
        <sequence>
          <element name="Name" type="person:NameType" maxOccurs="1" minOccurs="0"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="NameType">
    <complexContent>
      <extension base="base:ContextType">
        <sequence>
          <element name="First" type="base:ContextString"
            maxOccurs="1"
            minOccurs="0">
          </element>
          <element name="Middle" type="base:ContextString"
            minOccurs="0">
          </element>
          <element name="Last" type="base:ContextString"
            maxOccurs="1"
            minOccurs="0">
          </element>
          <element name="Nickname" type="base:ContextString"
            minOccurs="0">
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

The Device Base schema below defines an Entity type called “DeviceType”. Devices are defined to have Feature Sets which outline the device’s context and capabilities. The base Feature Set referenced by the Device Type is abstract, requiring that an device that wishes to represent its context must find (or define) a schema to accompany that context. Several common FeatureSet extensions are included in this schema as well, such as “email device”, in part to illustrate how this extension might be done.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema

    targetNamespace="http://cs.queensu.ca/DeviceBase"
    xmlns:device="http://cs.queensu.ca/DeviceBase"

    elementFormDefault="qualified"
    xmlns="http://www.w3.org/2001/XMLSchema"

    xmlns:base="http://cs.queensu.ca/ContextBase"
    xmlns:location="http://cs.queensu.ca/LocationBase"
    xmlns:person="http://cs.queensu.ca/PersonBase"
>

    <import schemaLocation="LocationBase.xsd"
namespace="http://cs.queensu.ca/LocationBase"></import>
    <import schemaLocation="ContextBase.xsd"
namespace="http://cs.queensu.ca/ContextBase"></import>
    <import schemaLocation="PersonBase.xsd" namespace="http://cs.queensu.ca/PersonBase"></import>

    <complexType name="FeatureSetType" abstract="true">
        <complexContent>
            <extension base="base:ContextType"></extension>
        </complexContent>
    </complexType>

    <complexType name="DeviceType" final="#all">
        <complexContent>
            <extension base="base:EntityType">
                <sequence>
                    <element name="FeatureSet"
                        type="device:FeatureSetType" maxOccurs="unbounded"
                        minOccurs="0">
                    </element>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <!-- EXAMPLES:
Following are some examples of common feature set
implemented as XML schema attributes. -->

    <complexType name="PrinterType">
        <complexContent>
            <extension base="device:FeatureSetType">
                <sequence>
                    <element name="Online" type="base:ContextBoolean"

```

```

        minOccurs="0" maxOccurs="1" />
        <element name="Busy" type="base:ContextBoolean"
        minOccurs="0" maxOccurs="1" />
        <element name="ErrorStatus"
        type="base:ContextInteger" minOccurs="0" maxOccurs="1">
        </element>
        <element name="JobsInQueueCount"
        type="base:ContextInteger" minOccurs="0" maxOccurs="1">
        </element>
        <element name="Location"
        type="location:LocationType"
        minOccurs="0"
        maxOccurs="1">
        </element>
    </sequence>
</extension>
</complexContent>
</complexType>

<complexType name="MusicPlayerType">
    <complexContent>
        <extension base="device:FeatureSetType">
            <sequence>
                <element name="Volume" type="base:ContextInteger"></element>
                <element name="Playing" type="base:ContextBoolean" />
                <element name="CurrentSong"
                type="base:ContextString">
                </element>
            </sequence>
        </extension>
    </complexContent>
</complexType>

<!-- A more complete 'emailType' would include attribute values here
instead of in the device:EmailDeviceFeatureSet -->
<simpleType name="emailType">
    <restriction base="string">
        <pattern
        value="[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}">
        </pattern>
    </restriction>
</simpleType>

<complexType name="EmailDeviceFeatureSet">
    <complexContent>
        <extension base="device:FeatureSetType">
            <sequence>
                <element name="EmailAddress" maxOccurs="unbounded"
                minOccurs="0">
                <complexType>
                    <simpleContent>
                        <extension base="device:emailType">
                            <attributeGroup
                            ref="base:ContextAttributes">
                            </attributeGroup>
                        </extension>
                    </simpleContent>
                </complexType>
            </element>
        </sequence>
    </extension>
</complexContent>
</complexType>
</schema>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<schema

    targetNamespace="http://cs.queensu.ca/DeviceBase"
    xmlns:device="http://cs.queensu.ca/DeviceBase"

    elementFormDefault="qualified"
    xmlns="http://www.w3.org/2001/XMLSchema"

    xmlns:base="http://cs.queensu.ca/ContextBase"
    xmlns:location="http://cs.queensu.ca/LocationBase"
    xmlns:person="http://cs.queensu.ca/PersonBase"
>

    <import schemaLocation="LocationBase.xsd"
namespace="http://cs.queensu.ca/LocationBase"></import>
    <import schemaLocation="ContextBase.xsd"
namespace="http://cs.queensu.ca/ContextBase"></import>
    <import schemaLocation="PersonBase.xsd" namespace="http://cs.queensu.ca/PersonBase"></import>

    <complexType name="FeatureSetType" abstract="true">
        <complexContent>
            <extension base="base:ContextType"></extension>
        </complexContent>
    </complexType>

    <complexType name="DeviceType" final="#all">
        <complexContent>
            <extension base="base:EntityType">
                <sequence>
                    <element name="FeatureSet"
                        type="device:FeatureSetType" maxOccurs="unbounded"
                        minOccurs="0">
                    </element>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <!-- EXAMPLES:
    Following are some examples of common feature set
    implemented as XML schema attributes. -->

    <complexType name="PrinterType">
        <complexContent>
            <extension base="device:FeatureSetType">
                <sequence>
                    <element name="Online" type="base:ContextBoolean"
                        minOccurs="0" maxOccurs="1" />
                    <element name="Busy" type="base:ContextBoolean"
                        minOccurs="0" maxOccurs="1" />
                    <element name="ErrorStatus"
                        type="base:ContextInteger" minOccurs="0" maxOccurs="1">
                    </element>
                    <element name="JobsInQueueCount"
                        type="base:ContextInteger" minOccurs="0" maxOccurs="1">
                    </element>
                    <element name="Location"
                        type="location:LocationType"
                        minOccurs="0"
                        maxOccurs="1">
                    </element>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

```

```

<complexType name="MusicPlayerType">
  <complexContent>
    <extension base="device:FeatureSetType">
      <sequence>
        <element name="Volume" type="base:ContextInteger"/></element>
        <element name="Playing" type="base:ContextBoolean" />
        <element name="CurrentSong"
          type="base:ContextString">
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<!-- A more complete 'emailType' would include attribute values here
instead of in the device:EmailDeviceFeatureSet -->
<simpleType name="emailType">
  <restriction base="string">
    <pattern
      value="[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}">
    </pattern>
  </restriction>
</simpleType>

<complexType name="EmailDeviceFeatureSet">
  <complexContent>
    <extension base="device:FeatureSetType">
      <sequence>
        <element name="EmailAddress" maxOccurs="unbounded"
          minOccurs="0">
          <complexType>
            <simpleContent>
              <extension base="device:emailType">
                <attributeGroup
                  ref="base:ContextAttributes">
                </attributeGroup>
              </extension>
            </simpleContent>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
</schema>

```


The Location Base schema illustrates how a single context might actually take multiple forms simultaneously. A device's location might have GPS coordinates, a colloquial term such as "Living Room", or a street address.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:base="http://cs.queensu.ca/ContextBase"
xmlns:location="http://cs.queensu.ca/LocationBase" elementFormDefault="qualified"
targetNamespace="http://cs.queensu.ca/LocationBase">
  <import namespace="http://cs.queensu.ca/ContextBase" schemaLocation="ContextBase.xsd"/>

  <complexType name="LocationType">
    <complexContent>
      <extension base="base:ContextType">
        <sequence>
          <element maxOccurs="1" minOccurs="0" name="GeographicCoordinates"
type="location:GeographicCoordinatesType">
          </element>
          <element maxOccurs="1" minOccurs="0" name="Colloquial"
type="base:ContextString"/>
          </sequence>
        </extension>
      </complexContent>
    </complexType>

    <complexType name="GeographicCoordinatesType">
      <all>
        <element name="Latitude" type="location:DegreesMinutesSecondsType">
        </element>
        <element name="Longitude" type="location:DegreesMinutesSecondsType"/>
        </all>
      </complexType>

    <complexType name="DegreesMinutesSecondsType">
      <complexContent>
        <extension base="base:ContextType">
          <sequence>
            <element name="Degrees" type="base:ContextInteger"/>
            <element name="Minutes" type="base:ContextInteger"/>
            <element name="Seconds" type="base:ContextFloat"/>
            <element name="CardinalDirection" type="base:ContextString"/>
          </sequence>
        </extension>
      </complexContent>
    </complexType>
  </schema>
```

The telephone and Blackberry schemas are used in this thesis to illustration schema extension.

Advanced device features can extend from the definition of a more general schema, such that the original representation is still available, along side the more advanced features.

Telephone Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://cs.queensu.ca/Telephone"
  xmlns:tns="http://cs.queensu.ca/Telephone"
  elementFormDefault="qualified"
  xmlns:device="http://cs.queensu.ca/DeviceBase" xmlns:base="http://cs.queensu.ca/ContextBase">
  <import
    schemaLocation="DeviceBase.xsd"
    namespace="http://cs.queensu.ca/DeviceBase"></import>
  <import
    schemaLocation="ContextBase.xsd"
    namespace="http://cs.queensu.ca/ContextBase"></import>

  <complexType name="PhoneNumberType">
    <complexContent>
      <extension base="base:ContextType">
        <sequence>
          <element name="countrycode"
            type="base:ContextString"
            minOccurs="0"
            maxOccurs="1"
            />
          <element name="areacode"
            type="base:ContextString"
            minOccurs="0"
            maxOccurs="1"
            />
          <element name="number"
            type="base:ContextString"
            minOccurs="0"
            maxOccurs="1"
            />
          <element name="extension"
            type="base:ContextString"
            minOccurs="0"
            maxOccurs="1"
            />
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="telephoneRingVolumeType">
    <simpleContent>
      <restriction base="base:ContextString">
        <enumeration value="Silent" />
        <enumeration value="Vibrate" />
        <enumeration value="Quiet" />
        <enumeration value="Loud" />
      </restriction>
    </simpleContent>
  </complexType>

  <complexType name="TelephoneFeatureSet">
    <complexContent>
      <extension base="device:FeatureSetType">

```

```

    <sequence>

        <element name="TelephoneNumber" type="tns:TelephoneNumberType"
            maxOccurs="unbounded" minOccurs="0">
        </element>

        <element name="VoiceMessages"
            type="base:ContextInteger"
            minOccurs="0"
            maxOccurs="1" />

        <element name="ringVolume"
            type="tns:telephoneRingVolumeType"
            minOccurs="0"
            maxOccurs="1"
            />

        <element name="status"
            minOccurs="0"
            maxOccurs="1"
            >
            <complexType>
                <simpleContent>
                    <restriction base="base:ContextString">
                        <enumeration value="Waiting" />
                        <enumeration value="Ringing" />
                        <enumeration value="Connected" />
                        <attributeGroup ref="base:ContextAttributes"/>
                    </restriction>
                </simpleContent>
            </complexType>
        </element>

    </sequence>
</extension>
</complexContent>
</complexType>
</schema>

```

Blackberry schema

```

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://cs.queensu.ca/Blackberry"

    elementFormDefault="qualified"
    xmlns:device="http://cs.queensu.ca/DeviceBase"
    xmlns:phone="http://cs.queensu.ca/Telephone"
    >

    <import schemaLocation="ContextBase.xsd"
        namespace="http://cs.queensu.ca/ContextBase"></import>

    <import schemaLocation="DeviceBase.xsd"
        namespace="http://cs.queensu.ca/DeviceBase"></import>

    <import schemaLocation="Telephone.xsd"
        namespace="http://cs.queensu.ca/Telephone"></import>

    <complexType name="BlackberryFeatureSet">
        <complexContent>
            <extension base="phone:TelephoneFeatureSet">
                <sequence>
                    <element name="callerID"
                        type="phone:TelephoneNumberType"
                        minOccurs="0"
                        maxOccurs="1">
                    </element>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

```

```
</complexType>  
</schema>
```

The Home Theatre and Ambient Lighting schemas are used in the Use Case examples to illustrate the framework's context sharing protocols.

```

Home Theatre.
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://cs.queensu.ca/HomeTheatre"
  xmlns:tns="http://cs.queensu.ca/HomeTheatre"
  elementFormDefault="qualified"
  xmlns:device="http://cs.queensu.ca/DeviceBase" xmlns:base="http://cs.queensu.ca/ContextBase">

  <import schemaLocation="DeviceBase.xsd" namespace="http://cs.queensu.ca/DeviceBase"></import>

  <import schemaLocation="ContextBase.xsd"
    namespace="http://cs.queensu.ca/ContextBase"></import>

  <complexType name="TelephoneNumberType">
    <complexContent>
      <extension base="base:ContextType">
        <sequence>
          <element name="countrycode"
            type="base:ContextString" minOccurs="0" maxOccurs="1" />
          <element name="areacode"
            type="base:ContextString" minOccurs="0" maxOccurs="1" />
          <element name="number"
            type="base:ContextString" minOccurs="0" maxOccurs="1" />
          <element name="extension"
            type="base:ContextString" minOccurs="0" maxOccurs="1" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="telephoneRingVolumeType">
    <simpleContent>
      <restriction base="base:ContextString">
        <enumeration value="Silent" />
        <enumeration value="Vibrate" />
        <enumeration value="Quiet" />
        <enumeration value="Loud" />
      </restriction>
    </simpleContent>
  </complexType>

  <complexType name="PlayState">
    <simpleContent>
      <restriction base="base:ContextString">
        <enumeration value="Stopped"/>
        <enumeration value="Playing"/>
        <enumeration value="Paused"/>
        <attributeGroup ref="base:ContextAttributes"></attributeGroup>
      </restriction>
    </simpleContent>
  </complexType>

  <complexType name="DVDPlayer">
    <complexContent>
      <extension base="device:FeatureSetType">
        <sequence>
          <element name="PlayState" maxOccurs="1" minOccurs="0" type="tns:PlayState" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>

```

```

</schema>

<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://cs.queensu.ca/AmbientLighting"

  elementFormDefault="qualified"
  xmlns:base="http://cs.queensu.ca/ContextBase"
  xmlns:device="http://cs.queensu.ca/DeviceBase"
  >

  <import schemaLocation="ContextBase.xsd"
    namespace="http://cs.queensu.ca/ContextBase"></import>

  <import schemaLocation="DeviceBase.xsd"
    namespace="http://cs.queensu.ca/DeviceBase"></import>

  <complexType name="AmbientLighting">
    <complexContent>
      <extension base="device:FeatureSetType">
        <sequence>
          <element name="LightLevel">
            <complexType>
              <simpleContent>
                <restriction base="base:ContextString">
                  <enumeration value="Off"/>
                  <enumeration value="Low"/>
                  <enumeration value="Medium"/>
                  <enumeration value="High"/>
                <attributeGroup
ref="base:ContextAttributes"/></attributeGroup>
              </restriction>
            </simpleContent>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
</schema>

```

Appendix B The Situation Document

The following is an example XML document representing a Situation of three devices across two hosts, as is described in Use Case 1: An Adaptive Home Theatre, as would be viewed by any of the devices within the environment. Each device in the environment interprets the context of the others to modify themselves to best suit the circumstances. In this particular instance, Host A is a host with both Bluetooth and WiFi capabilities, while Host B supports only WiFi interactions. Host A is representing

```
<?xml version="1.0" encoding="UTF-8"?>
<situation:Situation
  xmlns:situation="http://cs.queensu.ca/Situation"
  xmlns:host="http://cs.queensu.ca/Host"
  xmlns:base="http://cs.queensu.ca/ContextBase"

  xmlns:device="http://cs.queensu.ca/DeviceBase"
  xmlns:phone="http://cs.queensu.ca/Telephone"
  xmlns:person="http://cs.queensu.ca/PersonBase"
  xmlns:location="http://cs.queensu.ca/LocationBase"
  xmlns:lights="http://cs.queensu.ca/AmbientLighting"
  xmlns:theatre="http://cs.queensu.ca/HomeTheatre"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:entity="http://cs.queensu.ca/Entity"
>
<!-- Host A -->
<host:Host>

  <host:key>a3gGq3hKHAsDLkjg5as231d</host:key>

  <!-- Cellphone -->
  <entity:Entity id="AxngzBaGgL9kDSnruEjpTxfMbN2" xsi:type="device:DeviceType">
    <device:FeatureSet xsi:type="phone:TelephoneFeatureSet">
      <phone:TelephoneNumber>
        <phone:areacode>613</phone:areacode>
        <phone:number>533-6000</phone:number>
      </phone:TelephoneNumber>
      <phone:ringVolume>Quiet</phone:ringVolume>
      <phone:status>Waiting</phone:status>
    </device:FeatureSet>
  </entity:Entity>

  <!-- Home Theatre -->
  <entity:Entity id="lskdrRASqx3fkSJChgdkjky" xsi:type="device:DeviceType">
    <base:Location>
      <location:Colloquial>Living Room</location:Colloquial>
    </base:Location>
    <device:FeatureSet xsi:type="theatre:DVDPlayer">
      <theatre:PlayState>Playing</theatre:PlayState>
    </device:FeatureSet>
  </entity:Entity>
```

```
</host:Host>

<!-- Host B -->
<host:Host>

  <!-- Ambient Lighting -->
  <entity:Entity id="FDzxQC36DgfGQGgFAMpc36FLBUT" xsi:type="device:DeviceType">
    <base:Location>
      <location:Colloquial>Living Room</location:Colloquial>
    </base:Location>
    <device:FeatureSet xsi:type="lights:AmbientLighting">
      <lights:LightLevel lifetime="20">Low</lights:LightLevel>
    </device:FeatureSet>
  </entity:Entity>

</host:Host>

</situation:Situation>
```


The below document is the Entity document representing the cellphone Context Provider device in Use Case 2: Cellphone Desktop Notifications. It illustrates how XML's extensibility lends itself to context modelling by way of type extension, as well as how branches of the XML document may be kept private and shared only with those given access.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity:Entity
  xmlns:situation="http://cs.queensu.ca/Situation"
  xmlns:host="http://cs.queensu.ca/Host"
  xmlns:base="http://cs.queensu.ca/ContextBase"

  xmlns:device="http://cs.queensu.ca/DeviceBase"
  xmlns:phone="http://cs.queensu.ca/Telephone"
  xmlns:bb="http://cs.queensu.ca/Blackberry"
  xmlns:person="http://cs.queensu.ca/PersonBase"
  xmlns:location="http://cs.queensu.ca/LocationBase"

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:entity="http://cs.queensu.ca/Entity"
  xsi:schemaLocation="http://cs.queensu.ca/ContextBase ContextBase.xsd
    http://cs.queensu.ca/LocationBase LocationBase.xsd
    http://cs.queensu.ca/DeviceBase DeviceBase.xsd
    http://cs.queensu.ca/Telephone Telephone.xsd
    http://cs.queensu.ca/Blackberry Blackberry.xsd
    http://cs.queensu.ca/Entity Entity.xsd "

  id="AxngzBaGgL9kDSnruEjpTxfMbN2"
  xsi:type="device:DeviceType">
  <base:Relationship verb="ownership">
    <base:Subject>agOi32NLpLlzY40fC9auZ0kxqLa</base:Subject>
  </base:Relationship>

  <base:Location>
    <location:Colloquial>Office</location:Colloquial>
  </base:Location>

  <base:PermissionGroups>
    <base:PermissionGroup>
      <base:ContextID>AxngzBaGgL9kDSnruEjpTxfMbN2-callerid</base:ContextID>
      <base:EntityID>3gOi32NLpLlzY40fC9auZ0kxqLa</base:EntityID>
    </base:PermissionGroup>
  </base:PermissionGroups>

  <device:FeatureSet xsi:type="bb:BlackberryFeatureSet">

    <phone:TelephoneNumber>
      <phone:areacode>613</phone:areacode>
      <phone:number>533-6000</phone:number>
    </phone:TelephoneNumber>

    <phone:ringVolume>Quiet</phone:ringVolume>

    <phone:status>Ringing</phone:status>

    <bb:callerID contextID="AxngzBaGgL9kDSnruEjpTxfMbN2-callerid" private="true">
      <phone:areacode>613</phone:areacode>
      <phone:number>533-6001</phone:number>
    </bb:callerID>

  </device:FeatureSet>
```

</entity:Entity>

Appendix C Implementation Source and Design

The source code of the software implementation which accompanies this thesis to illustrate it's capabilities is freely available at the following URL:

<https://github.com/cyrusboadway/PervasiveContextSharingHost>









As well, a class diagram, modelled in UML, describing the software layout is depicted in the following three pages. The full image can be downloaded from the following URL:



<https://github.com/cyrusboadway/PervasiveContextSharingHost/blob/master/class%20diagram.PNG>

As well, a UML model in the UCLS XML data format can be downloaded at the following URL:

<https://github.com/cyrusboadway/PervasiveContextSharingHost/blob/master/class%20diagram.ucls>



<<Java Class>>	
 QueryException	
ca.queensu.pervasive.query	
	serialVersionUID: long
	intError: int
	QueryException(int)
	toString(): String









<<Java Class>>	
 Codec	
ca.queensu.crypto	
	
	
	
	
	
	
	






<<Java Class>>	
 ZeroconfListener	
ca.queensu.zeroconf	
	
	
	
	

<<Java Class>>	
 Base64	
ca.queensu.crypto	
	BASE64_CHARS: String
	BASE64_CHARSET: char[]
	
	

<<Java Class>>	
 BluetoothClient	
ca.queensu.pervasive.client	
	
	

<<Java Class>>	
 XMLHandler	
ca.queensu.xml	
	
	
	

<<Java Class>>	
 Key	
ca.queensu.crypto	
	key: KeyPair
	publicKey: String
	privateKey: String
	
	
	
	

<<Java Class>>	
 CodeException	
ca.queensu.crypto	
	serialVersionUID: long
	intError: int
	CodeException(int)
	toString(): String

