The Impress Context Store:
A Coordination Framework for Context-Aware Systems

by

Herman Hon Yu Li

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The dream of weaving technology into our everyday fabric of life is recently being made possible by advances in ubiquitous computing and sensor technologies. Countless sensors of various sizes have made their way into everyday commercial applications. Many projects aim to explore new ways to utilize these new technologies to aid and interact with the general population. Context-aware systems use available context information to assist users automatically, without explicit user input. By inferring user intent and configuring the system proactively for each user, context-aware systems are an integral part of achieving user-friendly ubiquitous-computing environments.

A common issue with building a distributed context-aware system is the need to develop a supporting infrastructure providing features such as storage, distributed messaging, and security, before the real work on processing context information can be done. This thesis proposes a coordination framework that provides an effective common foundation for context-aware systems. The separation between the context-processing logic component and the underlying supporting foundation allows researchers to focus their energy at the context-processing part of the system, instead of spending their time re-inventing the supporting infrastructure.

As part of an ongoing project, Impress, the framework uses the open standard, Jabber, as its communication protocol. The Publish-Subscribe (pubsub) extension to Jabber provides interesting features that match those needed by a context-aware system. The main contribution of this thesis is the design and implementation of a coordination framework, called the Impress Context Store, that provides an effective common foundation for context-aware systems. The separation between the context-processing logic and the underlying supporting foundation allows researchers to focus their energy at the context-processing part of the system, instead of spending their time re-inventing the supporting infrastructure.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor Prof. James P. Black for all his advice and patience during my undergraduate and graduate career at the University of Waterloo. His wisdom and wonderful sense of humor never cease to amaze me.

I want to thank my dear friend Alex Sung. It was his fun-loving, easy-going, and forever optimistic personality that made my time in graduate school full of rewarding times.

Next, I thank my parents for their love and patience for all the years I was in school.

The faculty members and staff, as well as my wonderful friends, all contributed to my success. I want to thank them for their help, support, and interest. I am much obliged to my readers Prof. Johnny Wong, Prof. Michael Terry; my colleagues Michael Kwok, Omar Khan, Hao Chen, Dr. David Evans, Prof. Srinivasan Keshav, Prof. Martin Karsten, Prof. Paul Ward; and our extremely helpful secretary Jessica Miranda.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

New technology has enabled many advances in ubiquitous computing. Originally proposed by Mark Weiser [52] almost 20 years ago, the dream of weaving technology into our everyday fabric of life is recently being made possible by electronic devices that are getting smaller, cheaper, better connected and simpler to deploy. Countless sensors of various sizes have already made their way into everyday commercial applications in factories, airplanes, cars, and home appliances. Many projects aim to explore new ways to utilize these new technologies to aid and interact with the general population. Researchers in artificial intelligence have been trying to find ways for systems to reason based on the environment that these systems are in. The more independent the systems are from the users, the less obstructive and more user-friendly they are [50]. Without explicit input from the users, these systems need more sources of input to improve accuracy of machine reasoning.

Context-aware systems use available context information to assist users automatically, without explicit user input. By inferring user intent and configuring the system proactively for each user, context-aware systems are an integral part of achieving user-friendly ubiquitous-computing environments. Dey defines context as:

> any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves [7].

Context-aware systems can benefit from having a standardized computing infrastructure to provide and organize context information. There have been some ubiquitous

projects which aim to provide a distributed infrastructure for coordinating a range of devices under their control. However, few of these focus on context information or have received wide-spread adoption. These systems are discussed in more detail later.

A common issue with building a distributed context-aware system is the need to develop a supporting infrastructure providing features such as storage, distributed messaging, and security, before the real work on processing context information can begin. When commenting on the obstacles experts faced in building large intelligent systems, Hasha commented accurately:

> I saw all these really smart people spending lots of energy on the same kinds of plumbing-related work just so they could begin focusing on their areas of interest. What a big waste of gray matter! [21]

This thesis proposes a coordination framework that provides an effective common foundation for context-aware systems. The separation between the context-processing logic component and the underlying supporting foundation allows researchers to focus their energy at the context-processing part of the system, instead of spending their time re-inventing the supporting infrastructure. A robust framework for such an infrastructure is easy to extend and maintain, supports the addition and removal of devices from the system without affecting the rest of it, is reliable despite transient failures, is secure, and is available on a wide number of commonly-used platforms.

The work presented in this thesis is part of an ongoing project, Impress [3]. The Impress project aims to explore the convergence of Instant Messaging and Presence in Smart Spaces [3]. Following the direction of the Impress project, our proposed framework uses the open standard Extended Messaging and Presence Protocol (XMPP) [45, 46, 47, 48], more widely known as Jabber, as its communication protocol. Jabber is an open-source, standards-based XML protocol with many extensions. The Jabber protocol is based on the instant messaging (IM) and presence protocol developed by the Jabber Software Foundation. There is a growing adoption of Jabber. Private and public implementations, including Google Talk [20], use Jabber as the communication protocol for their IM services. The Jabber Publish-Subscribe (pubsub) extension provides interesting features that match those needed by a context-aware system.

Our contribution is first to identify a list of requirements for context-aware systems. Then, we propose a coordination framework for context-aware systems using Jabber and its pubsub extension. We further extend the current pubsub semantics

to provide type information for context information, for the purpose of identification and data searching. In addition, we identify the need for a tool to explore the pubsub service. Last, we develop applications to evaluate our framework. Through these applications, we argue that the proposed framework is ideal for context-aware systems. We call our framework the Impress Context Store (iCS). Advantages of our framework are explored in detail in this thesis. Lessons learned from our research experience are discussed and future work is identified.

1.1 Motivating Example

For some time, doctors have known that the amount of motion cognitively impaired patients make in bed is related to the health of the patients [31, 32, 33]. Significant events, such as sitting up, leaving bed, and rolling over in bed, are often hard to capture, or are misreported by the patients themselves or their caretakers [9]. However, these events are strong indicators of the patients' health. A system that automatically and continuously tallies medically significant events can provide tremendous information to assist the doctors in diagnosis. Furthermore, once these medically significant events can be measured, nurses may find that it is useful to be able to have a picture of the area around a patient's bed when certain events occur, such as when the patient leaves the bed. This apprises the nurses of the current situation before they assist the patient. Figure 1.1 illustrates the minimal components involved in such a scenario.

Commercial off-the-shelf sensors can transform a regular bed into a smart bed that detects pressure exerted on it. Rolling and sitting events can be detected by a "smart bed," using strips of sensors, positioned across the bed, that can detect deformation of the strip due to patient movement. Using this technology, we can capture the medically significant events above, such as a patient entering the bed. We can also determine the amount of time a person spends in the bed. Moreover, a camera can be placed in the patient's room to monitor the area surrounding the bed. When the patient leaves the bed, such as in the case of falling off the bed or wandering from the room, the nurses are notified along with a current picture of the room.

Using the iCS framework, all parts of the system are connected using the XMPP/Jabber protocol. Sensors and other components are publishers and subscribers in the Publish-Subscribe extension. Events and notifications are handled by the pubsub service. We refer to this scenario throughout this thesis to reinforce how the framework, sensors, actuators, and other components fit together.

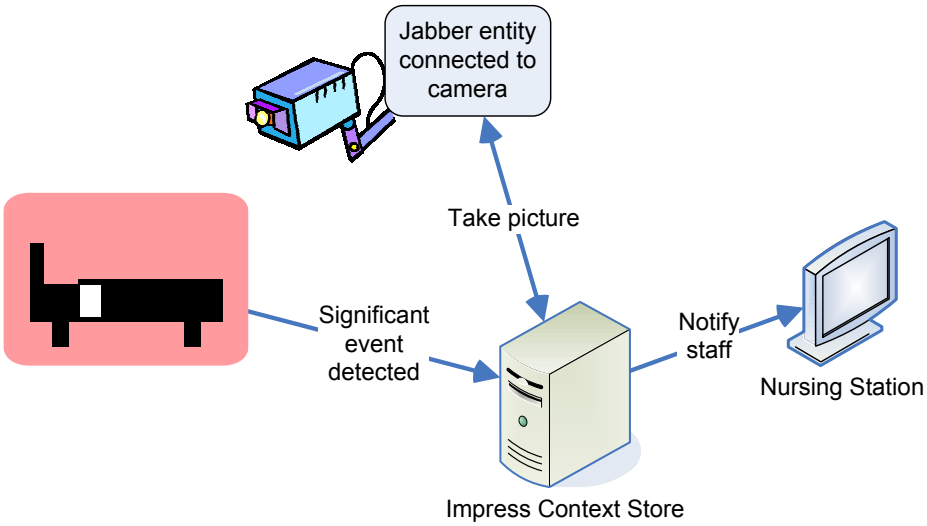

Figure 1.1: Motivating Example: 1) Smart bed detects when the bed is empty; 2) Camera takes a picture of the room; 3) Nurses get notified of the event with a current picture of the room.

1.2 Separation of Context Logic and Communication

Context-aware systems, such as the Context Toolkit [49], usually require various sensors as inputs and some actuators as outputs. In between the inputs and outputs, there are *context processors* that aggregate the inputs into context information, and use the context information to produce output. Sometimes, the context information is combined into higher-level context information. In this thesis, we call sensors and other sources of context information *context producers*. Similarly, we call actuators and other users of context information *context consumers*. Figure 1.2 illustrates the relationship among context producers, context consumers and context processors. Collectively, we call all participants of the context-aware system, including context producers, context consumers and context processors, *entities*. Because the entities of the system are often separate units, communication channels must link them. Security, availability, robustness, and persistence are but a few requirements commonly associated with any communication model.

It is crucial to focus the development of context-aware systems in two separate areas: i) the context processor for processing context information, and ii) the underlying foundation for providing communication and coordination among all

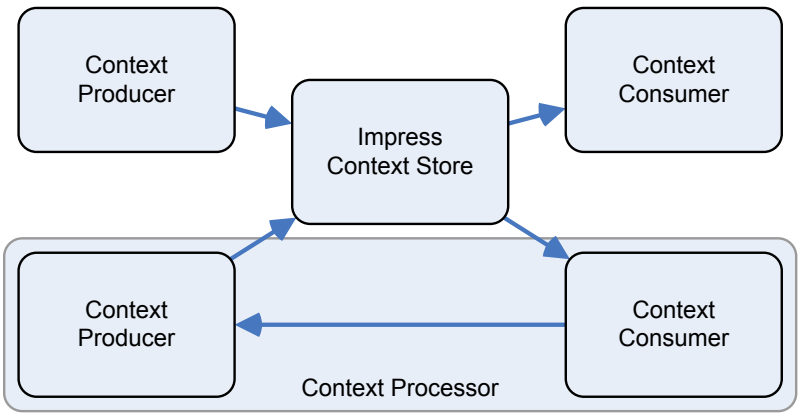

Figure 1.2: Context Producers and Context Consumers

of the devices and entities in the system. This thesis focuses on the latter area. A well-designed coordination framework needs to provide enough features for the context-aware system to process context information through high-level function calls, without worrying about details in the communication foundation. Furthermore, the framework needs to be generic enough to support different context-aware systems. We will see how this framework achieves these goals.

In this thesis, we introduce a coordination framework for context-aware systems. This framework uses the pubsub model to take advantage of its decoupled nature. Furthermore, with the help of the Jabber protocol, the framework is adaptable, fault-tolerant, language-neutral, efficient, secure, and is expected to enjoy industry adoption. Key contributions of our work include identifying a list of requirements for context-aware systems, proposing a system to address these requirements by tagging data with type information for data identification and searching, identifying and filling a need for a tool to explore data stored in the system, and developing proof-of-concept applications to evaluate the proposed system.

Chapter 2 discusses background and work related to this thesis and outlines major differences among the various projects. Chapter 3 introduces the Impress project and its relationship with context-aware systems. Chapter 4 presents our work, the Impress Context Store. Chapter 5 and 6 explain our implementation and performance evaluations respectively. Chapter 7 concludes with a discussion of contributions and future work.

Chapter 2

Background and Related Work

There are many projects that attempt to take the basic Publish-Subscribe model and improve its scalability or efficiency, or augment it with new semantics such as subscribing to contents instead of topics. However, only a handful of them apply the Publish-Subscribe model to large deployments such as context-aware systems in hospitals or university campuses. The Context Toolkit proposes a framework that has the potential to be used in large deployments. However, there is no implementation for such scenarios. The Interactive Workspace project at Stanford and the EDSAC(21) (Event Driven, Secure Application Control for the twenty-first century) project at Cambridge are among the few that envision deployment in real systems.

In this chapter, we provide some background information on three different communication models: i) Message Passing, ii) Tuple Space, and iii) Publish-Subscribe. Each has its own characteristics and they provide the groundwork that leads to the motivation for our work. Then, we present the related work mentioned above in more detail.

2.1 Message Passing

Message passing is the most prevalent method of communication in networks today. Message-passing systems are point-to-point systems consisting of message senders who send items of information directly to the message receivers. The majority of message-passing systems have a one-to-one sender-to-receiver relationship. Message passing is efficient, because only two parties are involved in each message transfer. Although there needs to be a way for senders to find the addresses of the

receivers, there are already mechanisms such as Domain Name Service (DNS) and various service discovery schemes to aid in that task. Despite the fact that message passing dominates network communication, it has some drawbacks. The senders and receivers are completely coupled in both space and time. The senders need to know exactly who they are communicating with. The senders and receivers also need to be running concurrently. This tight coupling characteristic makes faults in communication channels visible to the programs. A lot of effort needs to be put into the software design to make them fault-tolerant.

Next, we introduce the Tuple Space model of communication. It is completely different from message passing and offers a different set of benefits.

2.2 Tuple Space and Linda

In the early days of networked computers, Gelernter proposed a new communication model in distributed systems called, the generative communication model, using the Linda programming language [19]. The model has been incorporated into many languages such as C, Fortran, C++, Scheme, and recently, Java [4, 53]. The heart of the generative communication model is the existence of a Tuple Space shared by Linda programs in a distributed system. To communicate between process A and process B, process A generates tuples, which are ordered lists of data, and adds them to the Tuple Space. Process B later withdraws the tuples. The Tuple Space relays the tuples from A to B. Therefore, the processes are *distributed in space*, as is the case in normal distributed systems, and are *distributed in time*. It is the distributed-in-time nature that sets the generative communication model apart from conventional message-passing models.

Tuples in the Tuple Space can either be i) passive data values, or ii) executing or executable code. The latter gives Linda the ability to run distributed code. In this section, however, we focus on the data tuples.

There are four possible operations in the Tuple Space: `out()`, `in()`, `read()`, and `exec()`. The `exec()` operation is used to execute code stored in a tuple and is not relevant to our discussion here. The operation `out()` adds a tuple to the Tuple Space, `in()` withdraws a tuple, and `read()` reads a tuple without withdrawing it from the Tuple Space.

In its simplest form, a tuple is an ordered list of parameters. Each tuple is tagged with a character-string identifier. For example, to add a tuple with identifier N and parameters $P_2, ..., P_n$, the operation `out(`$N, P_2, ..., P_n$`)` is invoked. To retrieve

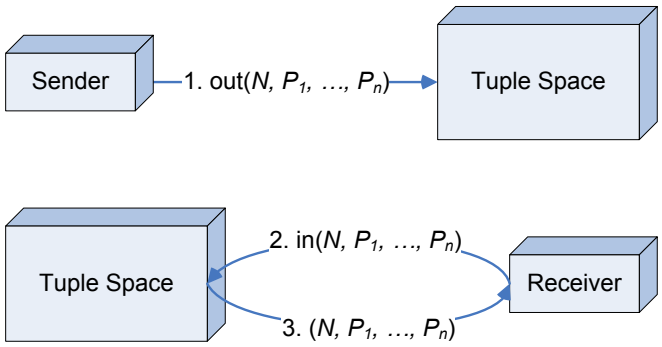

Figure 2.1: Example Tuple Space Operations

the same tuple, one can invoke the operation `in(`$N, P_2, ..., P_n$`)`. If a tuple with identifier N exists, the tuple is removed from the Tuple Space and $P_2, ..., P_n$ are assigned the values in the tuple. If no tuple has the identifier N, the `in()` operation suspends until such a tuple becomes available. The `read()` operation works like the `in()` operation except that the tuple remains in the Tuple Space instead of being withdrawn. Figure 2.1 depicts the `out()` and `in()` operations.

Tuples can be matched and retrieved by `in()` and `read()` using more than the identifier. For example, a tuple added to the Tuple Space using `out("sit up", 14, "yesterday")` will be matched using `read("sit up", ?`i`, "yesterday")` [4]. The variable i gets the value 14 when the call completes.

The generative communication model is significantly different from others due to "orthogonal communication." In orthogonal communication, the process adding tuples to the Tuple Space has no knowledge of the process withdrawing or reading the same tuples. This is significantly different from normal network communication, where the senders need to address the receivers directly. Orthogonal communication leads to space uncoupling, time uncoupling and distributed sharing. Space uncoupling refers to the fact that a tuple may be added by any process sharing the Tuple Space and may be withdrawn or read by any process. Figure 2.2 illustrates the case of space uncoupling. Time uncoupling, on the other hand, refers to the fact that tuples remain in the Tuple Space until explicit removal. Processes do not necessarily have to execute at the same time in order to communicate with one another. Figure 2.2 also shows this relationship. Distributed sharing is the result of the two earlier properties. The Tuple Space provides atomic shared storage space among address-space-disjoint processes. We will see similar characteristics in Publish-Subscribe systems in Section 2.3.

A weakness of Linda and Tuple Space is the lack of security built into the model.

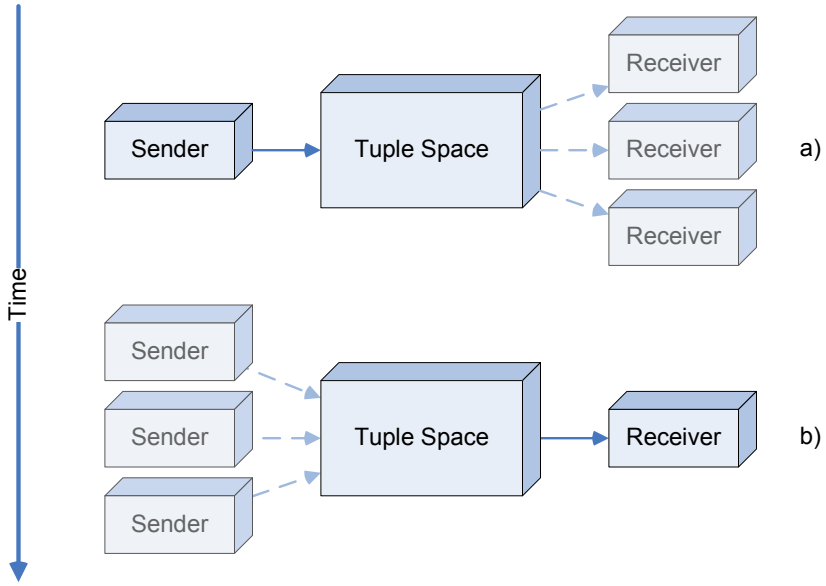

Figure 2.2: Space uncoupling: a) A tuple from a sender can have multiple recipients that may be unknown to the sender; b) A receiver can retrieve tuples from multiple senders that may be unknown to the receiver.
Time uncoupling: a) A sender can send a tuple to the Tuple Space at time t_0; b) A receiver can retrieve a tuple from the Tuple Space at time t_1, where $t_1 > t_0$. Note that the sender and receiver do not have to be running at the same time.

There is no inherent security model to protect tuples in the shared space. The tuples are accessible to all participating processes. Adding authorization mechanisms to Tuple Space can be cumbersome. Security constraints may need to be specified per tuple since there is no concept of a group of tuples. The Tuple Space service must also enforce some client-to-server encryption scheme to ensure data is transmitted securely. Moreover, type checking is an issue in Linda. Tuples with the same identifier but parameters of different types can be added to the Tuple Space. However, the process waiting for a tuple with specific parameter types can be blocked forever because the desired tuple does not exist in the Tuple Space.

Since their introduction, many extensions and improvements have been made to Linda and Tuple Space. For example, fault-tolerant Tuple Space [40] was introduced to minimize the effects of transient failures. With the increasing popularity of mobile and wireless devices, Linda has been extended to the mobile environment [41]. The uncoupling in space and time are suitable for limited bandwidth and frequently disconnected devices. Tuple Space has also been used in coordination systems to communicate among devices participating in the smart space [25, 38]. The uncoupling characteristics again play a crucial role in the success of these systems.

Similar to Linda and Tuple Space, Publish-Subscribe is another messaging model that uncouples participants in the system in both time and space. However, the Publish-Subscribe model has received more refinements by the research community and has enjoyed wider adoption.

2.3 Publish-Subscribe

The Publish-Subscribe (pubsub) model is similar to a message-board system in the way that a user (publisher) publishes his message publicly to everyone viewing the message board. The viewers (subscribers) essentially subscribe to this message board to receive any messages posted. A pubsub service is middleware that provides messaging services to its publishers and subscribers. It has many advantages over traditional point-to-point communication. The pubsub model has been in existence for a long time. The earliest incarnation was in topic-based pubsub services. Newer research directions include content-based, type-based and location-based pubsub services. Pubsub services are middleware that decouple message propagation in time and space. The survey by Eugster et al. [11] provides an excellent treatment of Publish-Subscribe systems in general. We cover some of the major findings in the following section.

Pubsub Models	Characteristics
Topic-based	Messages are published to topics. Subscribers subscribe to topics of interest.
Content-based	Subscribers specify queries to match message contents.
Type-based	Messages are published as data types or objects. Typed-based pubsub services require strong integration with the programming language.

Table 2.1: Different Variations of Publish-Subscribe Models

First, the publishers post messages to the pubsub service. Once the pubsub service determines a message matches the interest of one or more subscribers, it notifies them. Therefore, events are propagated asynchronously from the publishers to the subscribers. Space decoupling is achieved because neither the publishers nor the subscribers necessarily know the identity of one another. They only need to know the existence of the pubsub service. Time decoupling is also achieved because publishers and subscribers do not need to be available at the same time. Events can be dropped if the subscribers are not available, or can be stored until they are delivered. Synchronization decoupling refers to the situation where publishers are not blocked when publishing events. Events are subsequently pushed to the subscribers asynchronously.

There are some variations among pubsub models in how a service accepts messages from publishers and matches messages satisfying the interests of the subscribers. The most common types, topic-based, content-based, and type-based pubsub services, are presented in Table 2.1.

Topic-based pubsub services are well studied and implemented by industry and academia [11]. Topics can be viewed as containers for messages. Messages are published to these topics, and the pubsub service pushes them to subscribers interested in the topics. A topic is therefore very similar to a multicast group. In content-based pubsub services, instead of topics, subscribers subscribe to contents, such as data values. A subscription contains a query that includes a satisfaction condition. Therefore, topic-based pubsub is a special case of content-based pubsub, and can be constructed by using the topic names as query conditions. Finding efficient matching algorithms to match the query to past and future events is an area of active research. Type-based pubsub services have gained attention recently. Messages are published in terms of data types or objects. The pubsub service is tied strongly to the programming language used. An advantage is that these systems provide type-checking at compile time. Lastly, location-based pubsub services have

also been introduced [12]. The subscription queries are constructed in terms of the current location of the subscribers. As such, they receive events that are relevant to their present locations.

Publish-Subscribe is a general term used to describe a wide variety of implementations. Besides the range of topic-based, typed-based and location-based pubsub services, features that make a pubsub service usable in real-life deployments are also different among implementations. For instance, some pubsub services allow persistent storage of messages while others do not. Most implementations allow a configurable persistence model. Another example is security. The security models of different pubsub services are wildly different from one another. Depending on the intended usage of the pubsub service, different authentication and role-assignment methods are used. Furthermore, the set of configurable parameters also varies from product to product. Therefore, when discussing features offered by a pubsub service, it is important to know exactly which pubsub service is under discussion.

2.4 Context Toolkit

The Context Toolkit [7, 8, 49] is some of the earliest work on developing a framework for context-aware systems. It models the context-aware system based on graphical user-interface (GUI) widgets. Dey identified the following differences between the explicit user-input model in GUI and context-aware systems.

1. Distributed source of input: In a traditional GUI, user input is generally from the same computer. However, context-aware systems take context information from multiple, distributed sources.

2. Unlike user input through a GUI, context information is not always in the form required by the application. For example, a context-aware system may have to convert coordinates obtained from global positioning system (GPS) into postal addresses.

3. The application that requires the GUI user input usually runs on the computer to which the input device is connected. However, applications requiring context information usually run on a different computer than the one collecting the information.

Based on these differences, the Context Toolkit incorporates the following components.

1. Widgets: A widget communicates directly with a sensor and encapsulates a single piece of context information. Context information can be retrieved by polling or subscription mechanisms.

2. Aggregators: Aggregators combine context information from multiple widgets and reduce the combined, and often complicated, context information into something applications can use.

3. Interpreters: An interpreter takes context information from a widget or aggregator to form context information that is higher-level and easier for applications to understand.

The goal of the Context Toolkit is to integrate a large number of sensors to form a context-aware system. However, there are some shortcomings. First, although the Context Toolkit supports subscriptions to context information, the aggregators or the applications need to contact the widgets directly. The widgets are responsible for managing the subscription lists and disseminating the context information. The context source and the applications communicate directly with one another. Furthermore, they have to be running concurrently to communicate. Space and time coupling between the context source and applications is strong. While this may be suitable for realtime systems, transient failures are difficult to mask and can lead to the system not being able to scale to large deployments. Second, there is no security mechanism in place to limit access to sensor data. Once a sensor is added to the system, its data is available for all participating applications. For such a decentralized system, implementing and managing security constraints can be difficult. For sensitive sensor data such as the precise location of a person, security mechanisms need to be in place before the system will gain trust from its users.

2.5 Interactive Workspaces

The Stanford Interactive Workspaces project augments rooms with various technology to enable groups to perform collaborative problem solving [13, 26, 27]. The system integrates permanent computers, I/O resources, and portable devices that are brought into the room by its occupants. Based on their experiences with different iterations of the system, the Interactive Workspaces team has defined the following critical requirements for their system.

1. It must assume frequent changes in device configuration. Devices must be able to be added and removed with minimal disturbance to the overall system.

2. It must tolerate frequent transient failures. Due to the nature of wireless networks with frequent connection and disconnection, transient failures are the norm. The overall system must not be affected by transient failures.

3. It must be easy to add new devices. The access methods should be simple enough to allow a large number of devices access to the system.

4. The software infrastructure and applications running on it must be portable across instances of the Interactive Workspaces. Since some devices are mobile, they should be able to move from an instance of the workspace to another without modifying the software.

The group also referred to Human Computer Interaction (HCI) studies and identified the following important technological factors.

1. Heterogeneity: Because the system is built for integrating devices brought together by room occupants, heterogeneity of devices and software is an inherent property of the system. Promoting heterogeneity in the system also encourages evolution of the system to support new devices and software.

2. Changing Environment: It is inevitable that transient failures occur in the Interactive Workspaces. Therefore changes occur in the environment on short timescales as devices fail and recover, and enter and leave the environment. On longer timescales, the environment changes as devices and software are upgraded.

Based on the requirements identified, the Interactive Workspaces group chose to build the coordination mechanism on top of Tuple Space. The Event Heap is their implementation of the coordination mechanism [25]. Interactive Workspaces extends Tuple Space with the following features.

1. Self-describing Tuples: Without self-describing tuples, the semantic meaning of fields in a tuple is not available to new applications. To allow discovery of semantic meaning of the fields, a string is used to identify the type of each field.

2. Flexible Typing: By adding an extra tuple that identifies the minimal set of fields and their semantic meaning, tuples with reordered fields but intended to contain the same information can be matched by the matching algorithm.

3. Standard Routing Fields: A set of pre-defined fields is used to describe routing information in a tuple. For example, a tuple may have the source and target information embedded. Other participants in the Tuple Space can query for tuples that are specifically intended for them.

4. Tuple Expiration: Because accumulation of tuples in the Tuple Space may lead to performance degradation, Interactive Workspaces includes an expiry field in each tuple. Expired tuples are garbage-collected to ensure resources will not be used up by useless tuples.

5. Query Persistence/Registration: Interactive Workspaces allows a process to register queries with the Tuple Space. If a tuple matches a query, the process will be notified. This adds event notification to the existing polling model.

6. FIFO, At Most Once Ordering: To solve the multiple-read problem, where a single tuple may be matched multiple times without a way to identify matches from the same tuple, Interactive Workspaces enforces first-in-first-out ordering from the same source, and ensures a process sees each tuple at most once.

7. Modular Restartability: Failures in one area of the system do not affect the overall functionality of the system. Failed components in the Interactive Workspaces are designed to be able to restart independently while the other parts of the system keep running.

Publish-Subscribe. The Interactive Workspaces project considered using Publish-Subscribe instead of Tuple Space. The group felt that the Publish-Subscribe model is a close match with Tuple Space and fulfills all requirements of the Interactive Workspace except persistent storage. In other words, the Publish-Subscribe model failed the time uncoupling criteria.

Open Issues. The Interactive Workspaces project identified some open issues. Similar to Tuple Space, tuples in a space are shared openly with all participating processes. A security model is needed to restrict access to the tuples if we cannot assume the users of the Interactive Workspaces can be trusted. Second, due to the introduction of type information in tuples, name collision and the use of different names for the same type are problematic. In an environment that aims to integrate heterogeneous devices, these issues are likely to occur. Third, the original targeted frequency of use for the Interactive Workspaces is a few tuples per second at a latency of about 100ms per event. For applications that have higher frequency

or lower latency requirements, the Tuple Space model may not be appropriate. However, the group indicates that there is further evidence to suggest that the latency can stay below 50ms with several hundred tuples per second. Further studies are needed to confirm this claim.

Besides these open issues, the Event Heap lacks flexibility in handling unforseen applications. Their system assumes the application space is known. Therefore, they have complete control over the applications written to run inside the workspace. The types of messages in the pubsub system are for application events only. While this assumption is completely valid, it limits the openness of the system and prevents the inclusion of new applications.

With the help of extended type information, item attributes, and XML payload, our framework allows new applications to integrate with existing applications while providing new functionalities, all without downtime from the system.

2.6 EDSAC(21): Event-Driven, Secure Application Control for the twenty-first century

EDSAC(21): Event-Driven, Secure Application Control for the twenty-first century is a research project at University of Cambridge Computer Laboratory, with a goal of integrating an event-based, wide-area distributed framework with an open access-control architecture [1]. The application of their framework is similar to ours. They use Hermes to provide communication among devices. Hermes is an event-based middleware that uses the attribute and type-based Publish-Subscribe model [42].

We mention EDSAC(21) here because it shows that our research direction is pointing at a new and exciting area. The proposal for this project was published in 2004 and there is much on-going work. Their use of typed-based pubsub model can be used to compare with our topic-based pubsub model at some time in the future.

Throughout these research projects in context-aware systems, we see a common theme emerge: A context-aware system needs to deal with transient failures and adapt to the changing environment, by allowing decoupling between context providers and context consumers. Recent projects suggests that the Publish-Subscribe model holds much potential. In this thesis, we build on results from past research and propose a coordination framework for context-aware systems. We address previously-ignored issues and propose a robust, decoupled, scalable, and secure system suitable for these systems. Next, we highlight key requirements in

context-aware systems. Then, we introduce the Impress project and how it relates to context-aware systems.

Chapter 3

Impress and Context-Aware Systems

Context-aware systems take relevant state information from all participating entities into consideration, and modify the state of the system. The Context Toolkit developed by Dey [7, 8, 49] is one of the first frameworks directed at integrating and managing multiple sensor sources to provide a seamless context-aware system to the users. He suggested many requirements similar to the ones we have identified.

In this chapter, we explore key requirements that are essential to building and using context-aware systems. Then, we introduce the Impress project and its relationship to context-aware systems.

3.1 System Requirements

As explained in Section 1.2, the separation of context logic and communication is essential in building context-aware systems. This thesis focuses on developing an effective supporting infrastructure for context-aware systems. Specific details on processing context information are covered in other literature. In this section, we call all participating devices in the context-aware system *entities*. Following previous work and experience from our own research, we now identify the requirements for such an infrastructure. These requirements address common issues faced in distributed systems, pervasive-computing systems, and context-aware systems. We summarize these requirements in Table 3.1.

Requirements	**Motivating factors**
R1: Distributed communication in an overlay network	Entities are inherently distributed in space. Overlay networks support mobility, easy address lookup, and can cross heterogeneous networks.
R2: Secured communication	Messages contain sensitive information and must be protected. There needs to be clear authentication and authorization mechanisms.
R3: Adaptability in heterogeneous environment	The system consists of widely heterogeneous entities. Entities come and go and the system constantly evolves to satisfy different needs. Minimal human intervention is desired.
R4: Recovery from transient failures	Transient failures are the norm. They need to be contained and cannot cause system-wide failures.
R5: Context information aggregation	The system needs to provide means to retrieve varied context information in order to create higher level context from it.
R6: Continuous context acquisition	For some studies, samples need to be continuous over time.
R7: Persistent context information	Storing messages allows time decoupling and provides context history.
R8: Synchronized timestamps	Clocks of distributed entities should be reasonably well synchronized. The order of messages is important.
R9: Efficient searching of context information	Some context information is in the form of large textual or binary data. Searching this data cannot incur high latency.
R10: Resource discovery	Entities need to discover services and resources on their own to minimize explicit human involvement.
R11: Efficiency	Human-perceived latency should be bounded to a few seconds.

Table 3.1: Requirements for Supporting Infrastructure for Context-Aware Systems

R1: Distributed communication in an overlay network. It is easy to understand why a useful context-aware system needs to be distributed. With various miniature mobile devices gaining popularity, these tiny, wireless or wired, network devices form the basis of the sensing network. The sensors can appear anywhere in the users' space, and make continuous reading of the relevant environmental parameters. The actuators should also exist on a distributed network to provide feedback to the users in a wide area. Because of the potentially large distance between these devices and constraints on physical deployment, such as wire length restrictions, these devices ought to be distributed over a network.

The distributed network should exist in an overlay fashion, to simplify the process of looking up entities and increasing the mobility of entities. An overlay network sits on top of the existing network and routes messages via the existing network. However, the overlay network has its own addressing scheme. For example, an overlay network can exist on top of IP networks or even Bluetooth networks. This is similar to the electronic mail mechanism in that emails are routed from one SMTP (Simple Mail Transfer Protocol) server to another, with an addressing scheme known as an email address. Email users can roam around the globe without worrying about emails not arriving at their inboxes. Senders do not need to know where the receiver is located physically. Operating in the overlay network greatly simplifies issues associated with entity relocation, which happens very often in such a dynamic environment. Using an addressing scheme such as an IP address is not appropriate when entities move in the network. Furthermore, an overlay network can allow addressing schemes that are human-readable. This is especially beneficial in context-aware systems, which are often human-centric.

R2: Secured communication. Referring to our motivating example in Section 1.1, communication among entities and the middleware components needs to be secured. It is especially true in medical or home-monitoring scenarios that messages passing through the network often contain sensitive information. Privacy must be maintained if the system is to receive widespread adoption.

Besides securing the communication medium, authentication and authorization protocols must be enforced. The system must support reasonably fine-grained access restrictions on a case-by-case basis. This ensures that only users who have the proper permission can access the potentially sensitive data in the system.

R3: Adaptability in a heterogeneous environment. As mentioned in Section 2.5, the Interactive Workspaces group suggests that context-aware systems

exist in a heterogeneous environment that is changing constantly. Context-aware systems evolve in functionality over time. In our scenario, this occurred when the nurses discovered the benefit of having a camera in the room once the patient's bed was equipped with the motion detectors. Adding and removing entities from the system is normal and happens frequently. The context-aware system needs to service other parts of the system continuously while entities are being added or removed. Entities in the system each have their own purpose and will always have unique requirements. The context-aware system must be flexible enough to accommodate the differences among these entities. One obvious approach to running heterogeneous entities on a system is to use platform-independent protocols for communication among different hardware and operating systems.

Because our target users are not necessarily computer experts, the system must adapt to changes with minimal human intervention. New entities need to be added to the system with minimal configuration, and hardware that is being replaced because of failures or upgrades cannot bring the whole system down.

R4: Recovery from transient failures. This requirement relates closely to adaptability in a heterogeneous environment. However, it is important enough to warrant mentioning separately. The context-aware system must be robust against transient failures. In a ubiquitous computing environment, where many heterogeneous entities interact with one another, transient failures are normal. Many simple, yet common, situations such as temporary network disconnections, battery power failures, or users restarting the entities may cause transient failures. While the system loses some functionality directly related to the entities having these intermittent failures, the failures need to be transparent to the rest of the entities in the system. A system that shields failures from other parts of the system helps prevent cascading failures and promotes scalability.

R5: Context information aggregation. As introduced in Section 1.2, the context processors exist to transform existing context information into new context information. Very often, many pieces of context information spreading over multiple sources or over time are used to produce a single piece of context information. In our example, bed motion over time can determine whether a patient was rolling and turning while sleeping through the night. Therefore, the framework must allow entities to retrieve many items of context information and add processed context information back into the system. To the high-level users of the system, who are not interested in the low-level details of the context information, the context aggregation is done "under the hood" and is completely transparent.

R6: Continuous context acquisition. In a context-aware system, data from a single context producer may be needed by many entities. Therefore, it makes sense for the sensors to make their data available for sharing, instead of requiring high-level entities to acquire their own low-level context data. This requires the context acquisition entity to run independently of entities that require its data. Hence, continuous context acquisition is needed to promote reuse of context data and simplify the data-acquisition task for high-level entities.

Furthermore, in some studies, such as in Rasch analysis of the HABAM [33], data acquired at set intervals is necessary to produce results with high confidence. The system must ensure all context information collected at predefined intervals is sent to the appropriate receivers. An entity responsible for continuous data acquisition, run independently from the data receivers, is appropriate in this case.

R7: Persistent context information. Sections 2.2 and 2.3 mentioned the benefits of time decoupling between entities. Time decoupling is essential in mitigating the effects of transient failures. In order to achieve time decoupling, messages must be stored. By storing messages, the senders and receivers do not have to be in synchronization and do not need to be running concurrently.

Storing messages is also important for keeping a history of context information. There is a need to keep history of context for studies such as trend analysis and prediction [8]. For example, in our scenario, a doctor may later want to combine bed data for the past year for a long-term trend analysis. It is not always clear what analyses will be done, or who will be doing them in the future. Therefore, message data needs to be stored inside the system itself to facilitate future studies.

R8: Synchronized timestamps. Requirement R1 suggests that entities are distributed over a network. Because of the nature of our universe, entities in the system will never agree on the time. However, the ordering of messages is important because items of context information often have cause-and-effect relationships with one another. Therefore, there need to be mechanisms to synchronize message timestamps within acceptable error margins. One approach might be to timestamp messages at the middleware. This works as long as the middleware runs on a single machine. Another approach might be to synchronize the clocks of all entities in the system using some clock-synchronization mechanism. Local times are then used for message timestamps. Regardless of the mechanism, message timestamps must be accurate to within an acceptable error margin.

R9: Efficient searching of context information. The stored context information is useless unless there are ways to retrieve it. Because much higher-level context information is dependent on lower-level context information over time, the ability to store and search context information is crucial. The first requirement for searching is to know the type of the data stored. It must be known before we can understand its semantics and search for the desired information. Moreover, since much stored context information is from sensors, they may be large text or binary logs. Searching through these large chunks of data can be inefficient and decrease performance of the system. There needs to be ways to tag the large data with smaller, searchable attributes. For example, a video clip of a patient's room can be tagged with the date, time, name of the patient, location of the room, format, and the length of the video clip. Then, we can search the short tag instead of the large binary object.

R10: Resource discovery. Because of the nature of context-aware systems, explicit human intervention must be kept to a minimum. Therefore, it is essential that an entity be able to familiarize itself with other services and resources offered by the system, without excessive human configuration. Automatic resource discovery can be used to promote auto-configuration when new entities are added to the system. The sensors and other resources should make available, at a minimum, their location, communication mechanism and the services they offer.

R11: Efficiency. With today's networking technology, networks with high throughput and low latency can be achieved with very low cost. Furthermore, since the context-aware system interacts primarily with humans, depending on the application, latency as high as a few seconds is usually acceptable. Therefore, we do not need to worry about performance when exchanging short messages. However, low-level context information usually involves detailed sensor data such as long textual logs or large binary files. Therefore, efficient search of context information mentioned above is important in this scenario. Without sacrificing other criteria listed in this section, we should aim to limit the user-perceived latency to within a few seconds.

Requirements R1, R4, R8, and R10 address issues related to distributed systems. Requirements R2, R3, R4, R10, and R11 make sure the system is usable under the dynamic and demanding environments presented by pervasive-computing systems. R5, R6, R7, and R9 are functional requirements of context-aware systems.

Next, we introduce the Impress project and how it can help address these requirements.

3.2 Impress

The Impress project at the University of Waterloo explores the convergence of Instant Messaging and Presence in Smart Spaces [3]. The project envisions an integrated ubiquitous-computing platform built with existing systems and open-source standards.

> We're interested in how to make some of the vision of pervasive and ubiquitous computing a reality. Start with an existing, stable, open-source project with well-proven, standards-based technology and a vibrant community of contributors and users [3].

Leveraging tried and tested technologies, the platform should provide benefits such as extensibility, maintainability, reusability, scalability, and the ability to evolve.

The focus of Impress is to explore the use of the open Jabber standards [45, 46, 47, 48] in building a ubiquitous-computing platform. This thesis is one of the early results of the Impress project, and provides a generic coordination framework for context-aware systems.

Jabber. The Extensible Messaging and Presence Protocol (XMPP), also widely referred to as Jabber, became an IETF Internet standards-track protocol in 2004. It has its roots in the Jabber community dating back to 1999. Jabber protocols have received quick adoption since their inception. This is partly because its open-source, standards-based, XML protocols are easy to learn and use. Developers and users already possess most knowledge necessary to use these protocols. Numerous public and private implementations have been made. The Publish-Subscribe extension supports topic-based publish and subscribe functions, persistent storage, and includes various security features. It is a prime candidate for building a coordination infrastructure for a context-aware system. Being a generic Publish-Subscribe model, we explain next how Jabber's pubsub service can help satisfy some of the requirements outlined in Section 3.1.

Advantages of Publish-Subscribe. The most visible advantage is that the pubsub service decouples the publishers and the subscribers in space [11]. The publishers do not necessarily need to know anything about the subscribers. They only need to know about the pubsub service. Similarly, the subscribers have no

knowledge of the publishers. They are only concerned about the messages relayed by the pubsub service. Space decoupling contributes to requirements R1 and R3. We assume that in a context-aware system, the context producers are publishers and the context consumers are subscribers. Since context processors can aggregate low-level context information into higher-level context information, they are both subscribers and publishers. Adding new hardware to the context-aware system amounts to adding publishers or subscribers to the pubsub service. Since publishers and subscribers are decoupled, the addition of new hardware does not affect the rest of the system. Similarly, the impact of removing any hardware is limited to only publishers or subscribers that directly use the context information produced or consumed. These changes do not require reconfiguration of the pubsub service and are invisible to the rest of the system. This greatly simplifies maintenance since human intervention is kept to a minimum. Therefore, the system is able to scale. Furthermore, the availability of the pubsub service is also increased because hardware and software changes are localized to a small area and do not require bringing down the whole system.

The Publish-Subscribe model further allows messages to be persistent in the pubsub service (R7). If needed, messages can be stored until delivered or deleted explicitly. Combined with decoupling publishers and subscribers in space, the effects of transient failures can be alleviated (R4). Since publishers and subscribers are decoupled in time and are not tied to one another directly, a transient failure in the publisher will only register as a slightly delay in publishing the message. On the other hand, the pubsub service can store the message when a transient failure occurs in the subscriber. The subscriber can still retrieve the messages that would have been dropped otherwise. Since we are not dealing with real-time applications, transient failures do not have a visible effect on the system as a whole. For applications that process failure information, such as performance monitoring tools, extra context producers that monitor the status of the entities in question can be added to publish failure information to the pubsub service. Therefore, relevant failure information is available to be consumed by the context consumers but irrelevant failures are hidden from them. A persistent pubsub service also enables the subscribers to access stored messages to perform searches. This feature is essential for the context processor to aggregate context information (R5, R11).

The Publish-Subscribe model is not as efficient as point-to-point communication. Each message is passed through the pubsub service. Therefore, each message takes a two-hop path. However, with today's affordable high speed networks and our forgiving latency requirements, this additional delay does not pose any significant problem (R11).

The persistent Publish-Subscribe model itself contributes to requirements R1, R3, R4, R5, R7, and R11. The following section introduces Jabber's Publish-Subscribe service and how it contributes to more requirements of context-aware systems.

3.3 Jabber's Publish-Subscribe Extension Protocol

Jabber's Publish-Subscribe service not only provides basic pubsub semantics such as publication and notification of messages, but also features such as message persistence, security, authentication, authorization, and service discovery that are essential to a functional pubsub service. The fact that Jabber is an open-source, standards-based protocol enables heterogeneous entities to communicate with one another with ease (R3).

Jabber's pubsub service is topic-based. Figure 3.1 illustrates key components that make up the pubsub service. Each topic is a leaf node and is usually simply called a node. Publishers publish messages, or items, to these nodes. Subscribers can in turn subscribe to the nodes and receive notifications. Each node operates independently in the sense that each node has its own list of items, publishers, subscribers, and security constraints. "Collection" nodes encapsulate other collection nodes or leaf nodes to form a hierarchical namespace structure and security domain. Subscribers can potentially configure their subscription options to receive notifications from all leaf nodes under a collection. Nodes can be transient or persistent. While a transient node does not store messages published to it, a persistent node stores them until explicit removal. Messages are stored in a node as items. Each item has a unique identifier.

The followings are some security and authorization features provided by Jabber's pubsub service. These features contribute to requirement R2.

Affiliations. The pubsub service supports multiple affiliations of entities to nodes and collections. This essentially enables an administrator to grant read-only access, read-write access, or to deny access altogether. An entity can also initiate subscription requests to a node on its own. This case is discussed next.

Subscription States. The pubsub service allows administrative approval or denial of subscriptions to a node by setting subscription states. Each node has a list

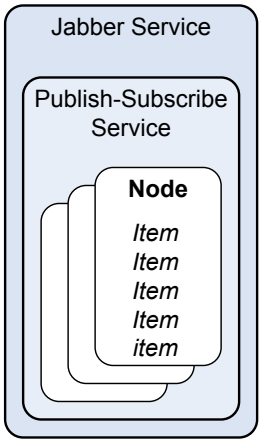

Figure 3.1: Components of a Pubsub Service

of owners who can perform administrative tasks to the node. A node can also be configured to always accept subscription requests. Having approval of subscription requests is essential in maintaining the privacy of sensitive material by allowing access to them by approved entities only.

Encryption. Security is critical for context-aware systems that may contain sensitive information. Jabber is able to use TLS and SSL to encrypt the communication channel between the entity and the Jabber server. However, this approach may not be sufficient to transfer private data, such as the medical history of a patient. Even when an entity uses TLS or SSL, messages are not guaranteed to be encrypted between Jabber servers or between the Jabber server and the pubsub server. The "End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)" protocol [48] specifies a way to encrypt Jabber messages between two entities. Using this encryption scheme, we can achieve end-to-end security between an entity and the pubsub service it is interacting with. Since the publishers generally do not know about the subscribers, messages cannot be encrypted beforehand to achieve publisher-to-subscriber security. However, using entity-to-entity encryption is sufficient because the pubsub service has its own authentication and authorization mechanisms to protect the sensitive data. In the end, only authorized subscribers are able to have access to the payloads. In Section 5.2.4, we discuss complementary work that can strengthen the security of the system. With the necessary features provided by Jabber, the entities do not have to worry about implementing extra security measures or about

the type of medium the messages travel on.

With requirement R2 being provided by the security model in Jabber's pubsub service, we now move onto extending the generic pubsub semantics to satisfy the rest of the requirements, namely R6, R8, R9, and R10. Next, we will refer to the requirements set forth and propose a coordination framework for context-aware systems using Jabber.

Chapter 4

The Impress Context Store

Chapter 3 outlined some critical requirements for context-aware systems. To recap, the devices in the system must be distributed in space over an overlay network and communicate using cross-platform protocols. Communication among entities must be secure to protect the privacy of its users. The environment in which the heterogeneous devices participate is highly dynamic. The system must adapt to new hardware environments and contain transient failures to prevent cascading failures. For the purpose of further study such as trend analysis and future prediction, context information must be collected continuously and stored for retrieval later. Timestamps used to tag context information must be reasonably accurate to preserve message ordering. Logic-processing components must be able to search lower-level context information and aggregate it into higher-level context information. They need to discover and utilize new services with minimal human intervention. While searching large textual or binary data, efficiency can become an issue. User-perceived latency must be bounded to a reasonable time, usually no more than a few seconds.

We propose the Impress Context Store (iCS), a novel coordination framework for context-aware systems. In the next sections, we describe our framework and argue that it meets all criteria listed in the previous section.

Proposed Framework. As part of the Impress project [3], the Impress Context Store (iCS) framework uses the Jabber protocol [45, 46, 47, 48] as the communication protocol. The Publish-Subscribe extension [36] is added on top of Jabber to form the framework. Certain semantics are enforced to further tailor the pubsub service for use by a context-aware system. At this stage of the project, our goal is to augment Jabber without changing its protocol. In time, we may be able to

convince the protocol designers to include some of our recommendations into the protocol.

4.1 Features of iCS

The Impress Context Store aims to augment Jabber's Publish-Subscribe service to enable continuous context acquisition, synchronized timestamps, and resource discovery. New semantics are introduced and are discussed in detail in the following sections. First, we introduce plain nodes and collection nodes. They are provided by Jabber's pubsub service as specified in JEP-0060 [36]. Then, we introduce the typed nodes, which are the augmented nodes designed to fulfill the remaining requirements.

4.1.1 Plain Nodes

The original semantics of pubsub nodes are very generic and can be adapted to many uses. However, they do not support specific features that are important to a context-aware system, such as type identification and efficient content searching. For example, to search through a node that stores large binary data, all data must be transferred to the client for processing. This will no doubt incur high delay in the search request, and may be unacceptable for a human user. We call an original node, or leaf node as specified by Jabber, the plain node. Plain nodes can be used for topics that do not need explicit type information and identifying attributes. However, without this additional information, some requirements listed before, such as efficient content searching, cannot be done. Furthermore, the publishers and subscribers of a plain node can be very much coupled, because they need to agree on the fixed semantics of the items in the plain node. Plain nodes are "light-weight," but lack the richness in features required by context-aware systems. Due to these limitations, we introduce a new kind of node called the *typed node*. First, however, we describe Jabber pubsub's collection-node concept.

4.1.2 Collection Nodes

Besides plain nodes, Jabber's pubsub service also supports collection nodes. A collection node is a container of leaf nodes and collection nodes. Leaf nodes can be plain nodes or typed nodes. A tree of collection and leaf nodes can be created. User

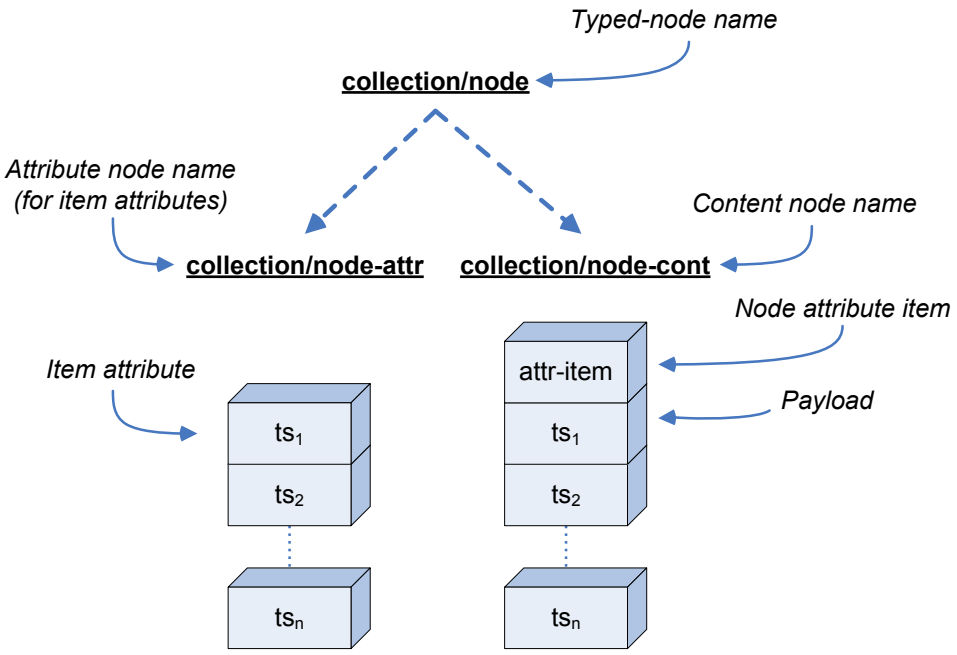

Figure 4.1: Typed Node Structure

permissions and subscriptions, such as affiliations and subscription states discussed in Section 3.3, can be applied to a collection node instead of each individual leaf node in the collection. Jabber indicates that the hierarchy separator in the node-naming scheme is flexible. For simplicity, we follow the UNIX file naming scheme and use "/" as the separator.

4.1.3 Typed Nodes

Section 3.1 mentioned the need for strong type information to facilitate data identification, searching, and retrieval. This subsection explains the new semantics used in typed nodes to incorporate various type information and application-defined custom attributes into the pubsub service. We use Figure 4.1 to illustrates the construction of a typed node.

Nodes

First, type information specific to a node must be stored. Jabber's pubsub service allows several meta-data attributes to be associated with a node. These are: cre-

ation date, creator, description, language, publisher, title and type. We use these, and expand on the pubsub specification to allow additional attributes to be stored. However, these alone are not enough to fulfill the missing requirements.

To facilitate efficient content searching, it is useful to distinguish the various types of data published to the pubsub service. We tag each piece of data with its content type. Since much of the context data stored is sensor data, we propose the use of Multipurpose Internet Mail Extensions (MIME) Media Types as specified by RFC 2045 and RFC 2046 [14, 15] to identify the type of context data. We use the abbreviated name MIME Types in this document. MIME was used originally to identify the types of attachments in Internet mail messages. It has now been extended to the HTTP protocol to identify the type of data stream associated with an HTTP connection. The success of MIME is partly due to the standardization with multiple RFCs [14, 15, 16, 17, 39]. It has been proven to be cross-platform and is used by different email clients and Internet browsers on different platforms. Furthermore, MIME is also extensible. Application developers can create new MIME types if none of the existing ones describe the new content adequately. With the help of MIME content types, new context producers can be added to the system without additional configuration. Context consumers that understand the new content type can use it immediately.

Since many entities are sensors or other devices that publish data of the same type periodically, we limit each node to one content type. This makes searching through a node easier because we know all items in the node are of the same content type. Entities that produce data with varying types can create a different node for each type. These nodes can be created under a common collection so that subscribers can subscribe to the collection and get notifications for all items published by the entity.

The W3C standard [28] suggests that content type should be stored with the encoded binary. This implies that the content type needs to be stored in each item. Tagging the binary with the MIME type allows quick identification of the encoded data. However, since we have established that all items in a node should have the same content type, we believe content type is also a valuable node attribute. Tagging the node with a MIME type allows users to quickly retrieve a list of nodes matching the desired MIME type. Although there is some redundancy, the overhead of storing the MIME type in both the node attribute and item attributes is usually minimal compared to the large binary payload.

Besides the type of data published, the identity of the publisher is also important. We should limit the number of publishers, or context sources, for a node. Ideally, there is a one-to-one publisher-to-node ratio. For example, a node can be

responsible for storing data only from the smart bed in the room. Another node can be responsible for picture data captured by the camera. This separation of responsibility simplifies maintenance and promotes usability. Context sources can have ownership of a node and provide data to that node. This helps realize continuous context acquisition (R6). Furthermore, discovering services (R10) is made easier because items in the node are from the same source. The idea is to make the full node names as descriptive as possible to identify the context source, while using collection nodes to group nodes and form levels of abstraction.

Using this scheme of limiting publishers to each node, new data sources can be added easily by creating a new node. Access control to data in one node is independent of other nodes. Removal of nodes also has minimal impact on the rest of the system. Subscribers only need to subscribe to relevant nodes. Collection nodes can be used to combine multiple leaf nodes into a logical group.

Node attributes. The content type is one of the required items of type information for a node. However, applications using the iCS may want to specify additional, custom attributes for the node. To allow this, we add a special `node-attribute` item in the node. The payload of this item is determined by the application. However, for greater compatibility during the service-discovery phrase, the Data Forms protocol [10] should be used to store custom attributes whenever possible. For example, a `node-attribute` item should have the format listed in Figure 4.2

Because the content type is instrumental in identifying the data, we make it a required node attribute. Therefore, the data form in the `node-attribute` item must contain a field such as listed in Figure 4.3.

Items

Items are individual pieces of data stored in a node. Each item has an identifier and a payload.

Identifier. The timestamp of the item is important in many instances. We propose to use timestamps as item identifiers. There are many representations of a timestamp. In order for a timestamp to become an effective identifier, there are a few criteria: i) it must be sortable in its string representation, ii) it must be human-readable and easy to understand, and iii) it must be cross-platform. W3C has proposed a dateTime format in its "XML Schema Part 2: Datatypes Second Edition" [2]. Section 3.2.7 defines the dateTime format in lexical representation as:

```
<item id="node-attr" xmlns="{node-attr-namespace}">
  <!-- All custom attributes wrapped in a Data Form. -->
  <x xmlns="jabber:x:data"
     type="result">
    <field type="{field-type}"
           var="{field-name}">
      <value>{field-value}</value>
    </field>
    <!-- Other fields -->
    ...
  </x>
  <!-- Other custom XML stanza. -->
  ...
</item>
```

Figure 4.2: Node-Attribute Item XML Format Example

```
<field type="text-single"
       var="mime-type">
    <value>{data-mime-type}</value>
</field>
```

Figure 4.3: Content-Type Field XML Format Example

```
<item id="{item-timestamp}">
  <base64 xmlns="http://impress.uwaterloo.ca/ics/base64"
          xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
          xmime:contentType="{mime-type}">
        <!-- Base64-encoded binary goes here. -->
        ...
  </base64>
</item>
```

Figure 4.4: Content-Node Item XML Format Example

```
'-'? yyyy '-' mm '-'dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)?
```

In fact, Jabber Date and Time Profiles [44] also use this dateTime format. We use this format as it satisfies all requirements listed above.

Although Jabber's pubsub service does not support range queries, range queries based on time can be emulated by implementing the logic in an entity. The pubsub service can discover all item identifiers for a given node. With the list of all item identifiers, an entity can apply a filter to these identifiers and retrieve individual items of interest based on the time of publication.

The identifiers are timestamped by the entities. Therefore, their clocks need to be in close synchronization with one another. Fortunately, there are many solutions for accurate clock synchronization. A few examples include highly accurate GPS clocks [34] for outdoor entities, and the NTP or SNTP protocols for reasonable accuracy [30]. Using these mechanisms, synchronized timestamps (R8) can be achieved.

Payload encoding scheme. Because the Jabber protocol is XML-based, only XML stanzas can be exchanged between the pubsub service and a Jabber entity. For publishing binary data, such as a video clip, the content of the file needs to be encoded as text. Referring to W3C's "Describing Media Content of Binary Data in XML" [28], Base64 encoding is used to encode the binary. A content type is provided for each item. The format of an item payload is shown in Figure 4.4. The corresponding schema is listed in Appendix B.

Item attributes. Some binary data or large textual data published to the pubsub service can benefit from being accompanied with application-customizable, textual,

searchable attributes. For a piece of text-encoded binary data in an item, the only attributes defined so far that are specific to the item are its timestamp and content type. For instance, an audio clip could be tagged with the duration of the clip, and a video clip could be tagged with its location. Having this summary data can aid a human user to scan quickly through the node and select only items of interest. It also decreases bandwidth and processing requirements of an entity by not having to download the entire item to search the payload after decoding. Efficient searching of content information (R9) depends on these searchable attributes.

Due to the item discovery semantics in Jabber's pubsub service, the attributes should not be stored in the same item as the payload. We cannot retrieve the attributes without retrieving the entire item if they are stored together. We define two nodes, named the attribute node and the content node, for each typed node. While the attribute node and content node are actual nodes in the pubsub service, the typed node is a concept realized by these two nodes. Figure 4.1 illustrates the structure of a typed node. The payload of each item in the attribute node contains attributes of the published data. The published data is stored as the payload of an item in the content node. These two items are linked by the same timestamp identifier. In the database sense, this is a vertical partition of the typed node. The attribute node name and content node name are derived from the typed node name appended with the `-attr` and `-cont` suffixes respectively. For example, the typed node `sensor/bed` is represented by the attribute node `sensor/bed-attr` and content node `sensor/bed-cont`.

The node-attribute item is placed in the content node. Therefore, this item can be hidden from users when browsing the item attributes.

Some attributes of interest are the publisher of the item and the location of the publisher. The location can be either a geographical location or a physical location, defined by JEP-0080 and JEP-0112 respectively, or both. Depending on higher-level needs, other attributes can also be added by the application designer. Although the format of the payload is ultimately determined by the application, the custom attributes should be stored using the Data Forms protocol whenever possible. The format of an item in the attribute node is shown in Figure 4.5.

Garbage collection. The Impress Context Store does not enforce any garbage collection rules on its own. However, if needed, a garbage-collecting context processor can be made to monitor nodes that need garbage collection. For example, items can be published with expiry times as one of the custom attributes. It is up to the application policy to decide how to tag and clean unwanted items.

```
<item-attr xmlns="{item-attr-namespace}">
  <x xmlns='jabber:x:data' type='result'>
    <field type='{field-type}'
           var='{field-name}'>
      <value>{field-value}</value>
    </field>
    ...
  </x>
  <physloc xmlns='http://jabber.org/protocol/physloc'
           xml:lang='en'>
    <country>{country}</country>
    <locality>{locality}</locality>
    <building>{building}</building>
    <room>{room}</room>
    <text>{text}</text>
  </physloc>
  ...
</item-attr>
```

Figure 4.5: Attribute-Node Item XML Format Example

4.2 Access Methods

In order for a context-aware system to be functional, there must be application programming interfaces (API) available for the developers. Because context-aware systems exist in dynamic heterogeneous environments, the APIs must also be able to adapt quickly to new platforms.

Figure 4.6 shows some access methods as examples. As the pubsub service uses XML for its messages, APIs for different languages and platforms can be developed without platform-compatibility issues. The bottom of the hierarchy is Jabber. The pubsub service sits on top and exposes its XML interface. From there, there are multiple options. One can use any programming language that supports XML. In this day and age, that means virtually all modern languages. We choose to illustrate our example with Java and .NET. These are two of the few languages that are potentially portable across a wide range of operating systems. They also have easy migration paths for wrapping libraries written in these languages as web interfaces. Because many handheld devices have limited processing power, web APIs can be especially useful. For devices that can run customized programs but do not have the processing power or storage space to handle XML parsing or XML packages, URL rewriting, where one embeds parameter values in the URLs, can be used to issue iCS commands securely over HTTPS. Alternatively, web services can be used for the same purpose. For devices that cannot run customized applications but have a built-in web browser, one can write a web application to expose specific iCS functions for use with the devices' browsers. Callback-type asynchronous events, which make the Publish-Subscribe model so powerful, can also be imitated in web applications using the new Asynchronous JavaScript + XML (AJAX) paradigm [18].

By using an open-source and standard XML protocol, Jabber and its pubsub service enable support from a wide variety of access methods. This flexibility is welcome because the highly dynamic heterogeneous environments that the context-aware systems are in require heterogeneous access methods to fit different client needs.

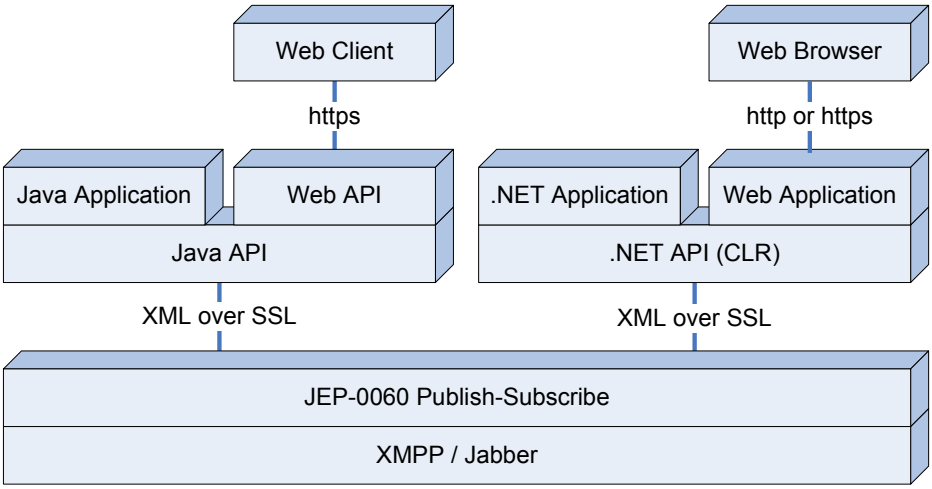

Figure 4.6: API Access Methods

Chapter 5

Implementation

An implementation of the framework has been made to verify the feasibility of the Impress Context Store and its benefits discussed so far. The iCS framework must first have a simple API to aid rapid deployment of entities. Then, we discuss applications built using the framework and how they perform.

5.1 iCS on XMPP / Jabber

We have chosen the open-source jabberd 2 [24] as our Jabber server. On the pubsub service side, Idavoll 2 [35] is used since it is one of the few working open-source Jabber pubsub services available for jabberd 2. Other open-source and commercial implementations of the Jabber and pubsub server are available and are listed on the Jabber web site [23]. Since our goal is to evaluate the iCS semantics and not to optimize its performance, using third-party software such as jabberd 2 and Idavoll serve to demonstrate benefits iCS brings.

To access Jabber and its pubsub service, we use the client-side Java library called Muse [5]. Although Muse does not support the Jabber pubsub service, it is an extensible library and we have extended it to include basic pubsub features, as well as iCS features. We call our extended library the *iCS API*.

The iCS API provides interfaces for traditional pubsub features, as well as the extended iCS features. A reference implementation using Muse is developed. The API has been tested on jabberd 2 and Idavoll 2. For a complete list of methods provided by the iCS API, please refer to Appendix A.

In the next section, we demonstrate proof-of-concept applications that use this combination of software. However, any Jabber server, pubsub service, and client library conforming to the required standards should work.

5.2 Applications

A few applications have been developed to test the feasibility and usability of the iCS framework. First and foremost, we needed a tool to explore the data stored in the pubsub service. We could not find such a tool elsewhere, so we developed the iCS Browser. It understands basic pubsub semantics, as well as the extended semantics proposed in this thesis. Next, we developed a security-monitoring application with a context producer, a context consumer, and a context processor. This application uses both the plain and typed nodes. Then, we demonstrate how different context sources can be reconfigured easily to work together, by integrating the smart-bed sensors with the security-monitoring application. Finally, drawing on the experiences with the iCS framework, we reiterate the Impress vision.

5.2.1 iCS Browser

The iCS Browser is a Jabber entity that explores a Jabber pubsub service. It is a research tool that allows us to conveniently explore the pubsub service. Besides basic functions as specified in the pubsub specification, the iCS Browser also understands iCS semantics such as the typed node. It uses the Java iCS API with the Echomine Muse library, as discussed above. In this section, we call the person using the browser the *user*. Figure 5.1 illustrates a sample iCS Browser user interface.

The browser exposes many of the API functions in GUI form. This ensures the API contains functions commonly used by the entities. Many of the browser features discussed below have direct mapping to API functions outlined in Appendix A.

At the top of the window is the Quick Connect panel. It accepts user credentials for security purposes. Once a user is authenticated, she can navigate the context store. The left side of the browser lists all the nodes available in the context store, grouped into a tree structure. Plain nodes and typed nodes can be created. There are certain operations that can be applied to a node. These include subscription, unsubscription, publication, deletion of a node and purging of items in a node. When a node is selected in the Nodes panel, the node attributes and the list of items in the node are retrieved automatically. The Node Information panel shows

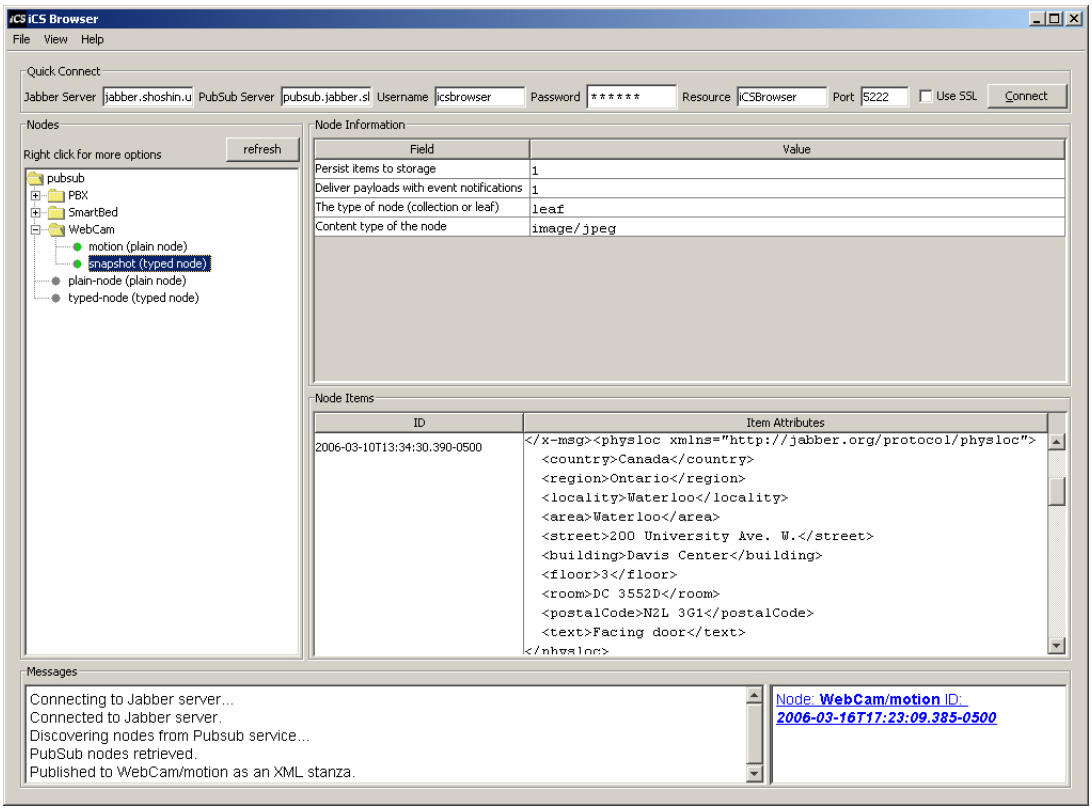

Figure 5.1: iCS Browser User Interface

node attributes. The Node Items panel shows the list of items in the selected node. For a typed node, the item attributes are shown and the payloads are hidden. Finally, the Messages panel in the bottom displays status messages on the left and incoming events on the right.

Creating Nodes. The iCS Browser can create both plain nodes and typed nodes. Only a node name is required to create a plain node. For a typed node, the node name and the content type are required. Custom attributes are optional. Upon creation of the plain or typed node, the user logged onto the browser becomes the owner of the node and can therefore perform administrative tasks.

Subscribing and Unsubscribing. An iCS Browser user, including the owner of the node, needs to subscribe to a node before viewing the items (pull-based) or receiving events (push-based). As the current Idavoll implementation has no mechanism for approving or denying subscriptions, all subscription requests are approved automatically. This feature will be available once the pubsub service is compliant to all requirements in the JEP-0060 specification. A user can unsubscribe at any time to stop receiving events.

Publishing. For a plain node, a user can publish a piece of text or an XML stanza. The difference is that certain characters (such as "<" and "&") are escaped when publishing as text. On the other hand, a valid XML stanza is published as is. The browser will try to publish the given text as XML. If the text is not a valid XML stanza, it will escape necessary characters and publish it as normal text.

For a typed node, a user can publish data in the form of a file. The browser asks for optional custom attributes to accompany the data. The contents of the file is encoded with Base64 and published along with the content type as an item in the content node. The attributes are published as an item in the attribute node.

Browsing. The browser presents a list of nodes available in the pubsub service through service discovery. Subject to sufficient permission (such as the user being subscribed to the node), the browser displays the list of node attributes and the list of items. For a typed node, only the item attributes are displayed in the Node Items panel. The payload in encoded form is assumed to be of no direct use to a human user for viewing.

Viewing and Saving. After locating an item in a typed node, the user can launch an external application that matches the content type to view its content. The content-type-to-application mapping is specific to the operating system. For example, on Microsoft Windows, the `text/html` MIME type is associated with Internet Explorer by default. When viewing items in a node with the MIME type `text/html`, the decoded data is handed to Internet Explorer and it, in turn, displays the data. For a plain node, the text is assumed to have the content type `text/plain`.

The user can also save the payload of a typed node to a file instead of opening it with an application. In this case, the browser will attempt to find the correct file extension for the user. Again, this is operating-system-specific.

Receiving Events. When an event is received by the browser, it logs the event in the Messages panel as a link. If the message is of interest to the user, he can click the link and locate the message in the browser. This is similar to many instant-messaging clients. Once the item is located in the browser, the user can then view or save the content as described above.

5.2.2 Security-Monitoring Application

Our first proof-of-concept application is a security-monitoring application. It monitors motion in an area using a motion detector and notifies a person with a picture of the area when motion is detected. The relationships among the different components are shown in Figure 5.2.

Motion Detector. The motion detector is a commercial off-the-shelf X10 device [54] that informs the host computer whenever motion is detected in the area it views. When motion is detected, the host publishes an item to an iCS plain node (for example, the `WebCam/motion` node). A typical payload is shown in Figure 5.3.

Referring to Figure 1.2, the motion detector is clearly a context producer.

Event Interpreter. The event interpreter is a context processor that subscribes to the `WebCam/motion` node and waits for events published by the motion detector. Once the event interpreter determines that sufficient motion exists in the area, it takes a picture. Issuing the capture command is outside the scope of iCS. The event interpreter can contact the camera entity via Jabber Ad-Hoc Commands [37] or

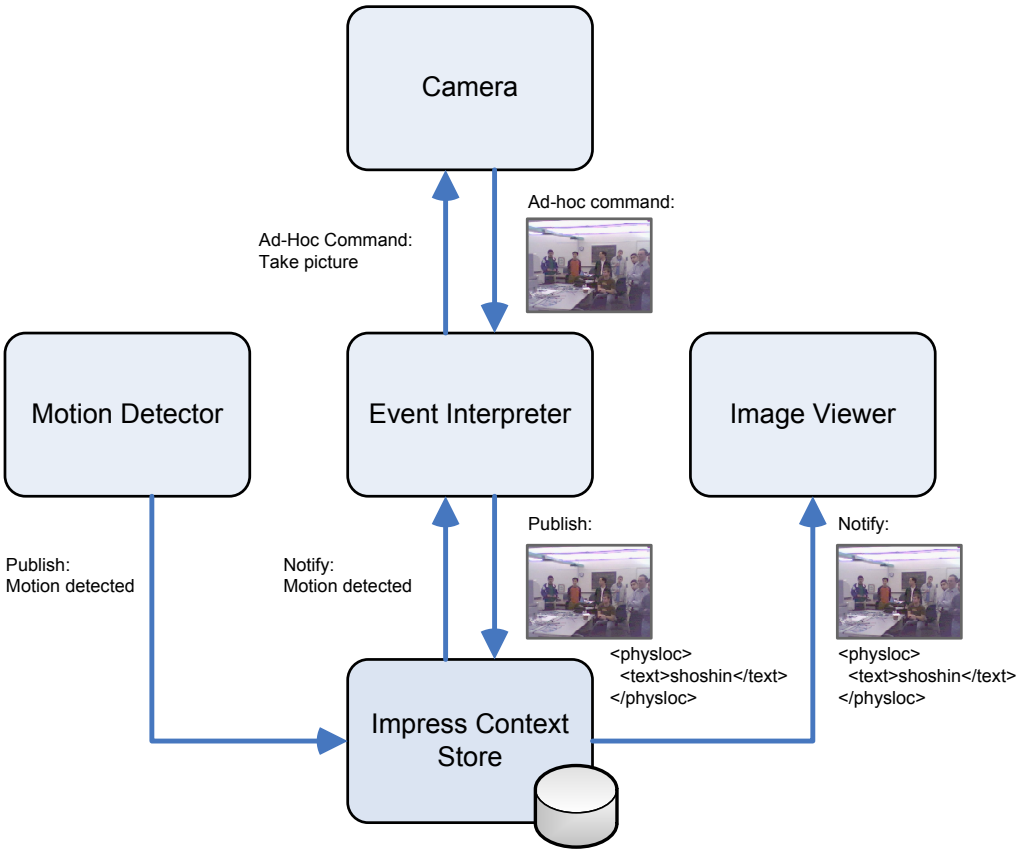

Figure 5.2: Components of the Security-Monitoring Application

```
<sensor location="DC3552D">Motion detected</sensor>
```

Figure 5.3: Item Payload Published by the Motion Detector

```
<x xmlns="jabber:x:data" type="result">
  <field xmlns="jabber:x:data"
         var="publisher"
         type="text-single"
         label="Publisher of this item">
    <value>ics@jabber.uwaterloo.ca/WebCam</value>
  </field>
</x>
<physloc xmlns="http://jabber.org/protocol/physloc">
  <country>Canada</country>
  <region>Ontario</region>
  <locality>Waterloo</locality>
  <area>Waterloo</area>
  <street>200 University Ave.  W.</street>
  <building>Davis Center</building>
  <floor>3</floor>
  <room>DC 3552D</room>
  <postalCode>N2L 3G1</postalCode>
  <text>Facing door</text>
</physloc>
```

Figure 5.4: Example Item Payload in Attribute Node

other out-of-band methods. Upon receiving the picture from the camera, the event interpreter publishes it to an iCS typed node (for example, the `WebCam/snapshot` node). The picture is tagged with attributes including publisher, textual reason for taking a picture, and the location of the camera. An example payload is shown in Figures 5.4 and 5.5.

The event interpreter is both a context consumer and context producer. It transforms low-level context into high-level context.

Image Viewer. The image viewer is the last piece of the application. It subscribes to the snapshot node and waits for pictures published by the event interpreter. Upon receiving a picture, signifying motion in the area, the image viewer notifies the computer user and displays the picture. Currently this application runs on a desktop computer. However, it is simple to redeploy this application on a mobile device such as a Pocket PC or cellular phone, to act as a pager.

```
<base64 xmlns="http://impress.uwaterloo.ca/ics/base64"
        xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
        xmime:contentType="image/png">
  <!-- Base64 encoded binary goes here -->
</base64>
```

Figure 5.5: Example Item Payload in Content Node

Summary. The security-monitoring application demonstrated typical usage and key benefits of iCS. The motion detector, event interpreter, and image viewer are components that communicate directly with iCS and not one another. With the additional level of indirection, any one the three components can fail for a short amount of time without causing a cascading failure. By the same token, components can be updated with improved versions without the need to take down the whole system or to reconfigure other components. The communication channels among all components are secured with SSL. Therefore, data between the Jabber server and clients is safe from prying eyes. Because messages are XML-based, a component can be re-written for another operating system and this is transparent to those other components. With sufficient user-defined attributes, such as time and location information in the case of the pictures published, another context consumer can search and locate stored pictures efficiently, given a specific location and a certain time range.

5.2.3 Smart Bed Application

Returning to our motivating example in Section 1.1, we now illustrate how we can construct such a system easily with the iCS framework.

First, we need the smart bed that captures the occupant's motion. At Dalhousie University as part of a larger project, we equipped a bed with sensors that can detect the magnitude and location of bending from rest. Using these sensors, placed at strategic locations, we can infer the location of the occupant and some other meaningful activities such as sleeping, sitting up, and leaving the bed. Figure 5.6 shows the actual smart bed equipped with the sensors. Although the details of setting up such a smart bed are interesting, in this thesis, we are only concerned with the events that it publishes to the iCS.

Among the voluminous data that the bed provides to generate reports for the doctors, one piece of information stands out in our example. When an Alzheimer's

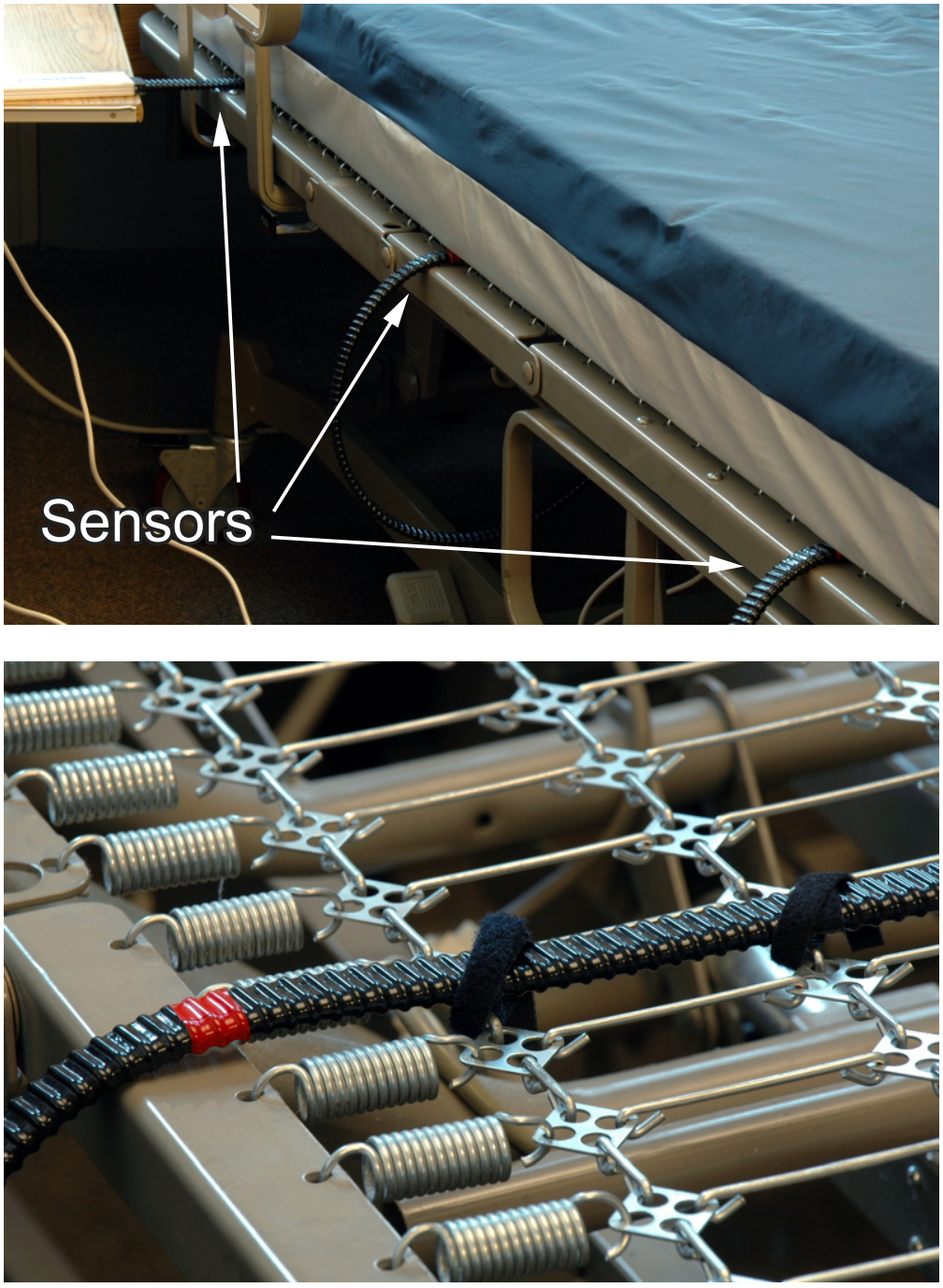

Figure 5.6: Motion-Sensing Smart Bed: Sensors underneath the mattress in parallel formation. Each sensor can detect the magnitude and location of the deformation from its resting position.

patient leaves her bed in a patient-care environment, the nurses may want to know about it because the patient may have fallen, or may wander off and get lost. Therefore, it is important for the bed, or other context processors that aggregate context information produced by the bed, to publish a message to the iCS when a patient leaves her bed. With the appropriate monitoring application subscribing to that event, the nurses can be notified in near realtime when the patient leaves the bed.

We mentioned one of the benefits of using Publish-Subscribe systems is to allow the system to evolve while minimizing maintenance efforts. Imagine that after the nurses have the ability to know immediately when a patient leaves the bed, they want to know more about the state of the room before determining a course of action. The system needs to evolve to satisfy that new requirement. At CASCON 2005 [22], we integrated the smart bed at Dalhousie University and the security-monitoring application at the University of Waterloo to satisfy the request from the nurses. Figure 5.7 illustrates the setup. Notice how similar this figure is to Figure 5.2. In fact, we replaced the motion detector component with a bed component and a bed-motion interpreter that collectively publish events when the patient leaves the bed. The event interpreter needs to know how to interpret the new empty-bed message. However, the camera entity and image viewer can be left untouched. We did the integration on the day of the conference and the system was up and running within half an hour.

5.2.4 Towards the Big Picture

As part of a larger project entitled "Intelligent System Requirements for Cognitively Impaired Individuals," [43] we have worked with medical doctors to further investigate use cases for collecting data that is indicative of a patient's health. Besides the patient's motion in bed, as described above, appliance and telephone-usage patterns also contain valuable information. With the help of the Impress team, we have expanded our implementation from the scenario depicted in Figure 1.1 to Figure 5.8. All sensors and actuators communicate through the iCS. Different context processors are in place to aggregate and transform low-level context in iCS to higher-level context. Using a private branch exchange (PBX), we are able to monitor telephone usage, such as the frequency of local and long-distance calls made. With this aggregated usage data, we can compile a trend report that can aid doctors in diagnosing patients' health more accurately. For example, doctors believe that a decrease in making long-distance calls or an increase in dialing wrong numbers are early signs of dementia. Moreover, using X10 technologies, we are able to monitor electric-

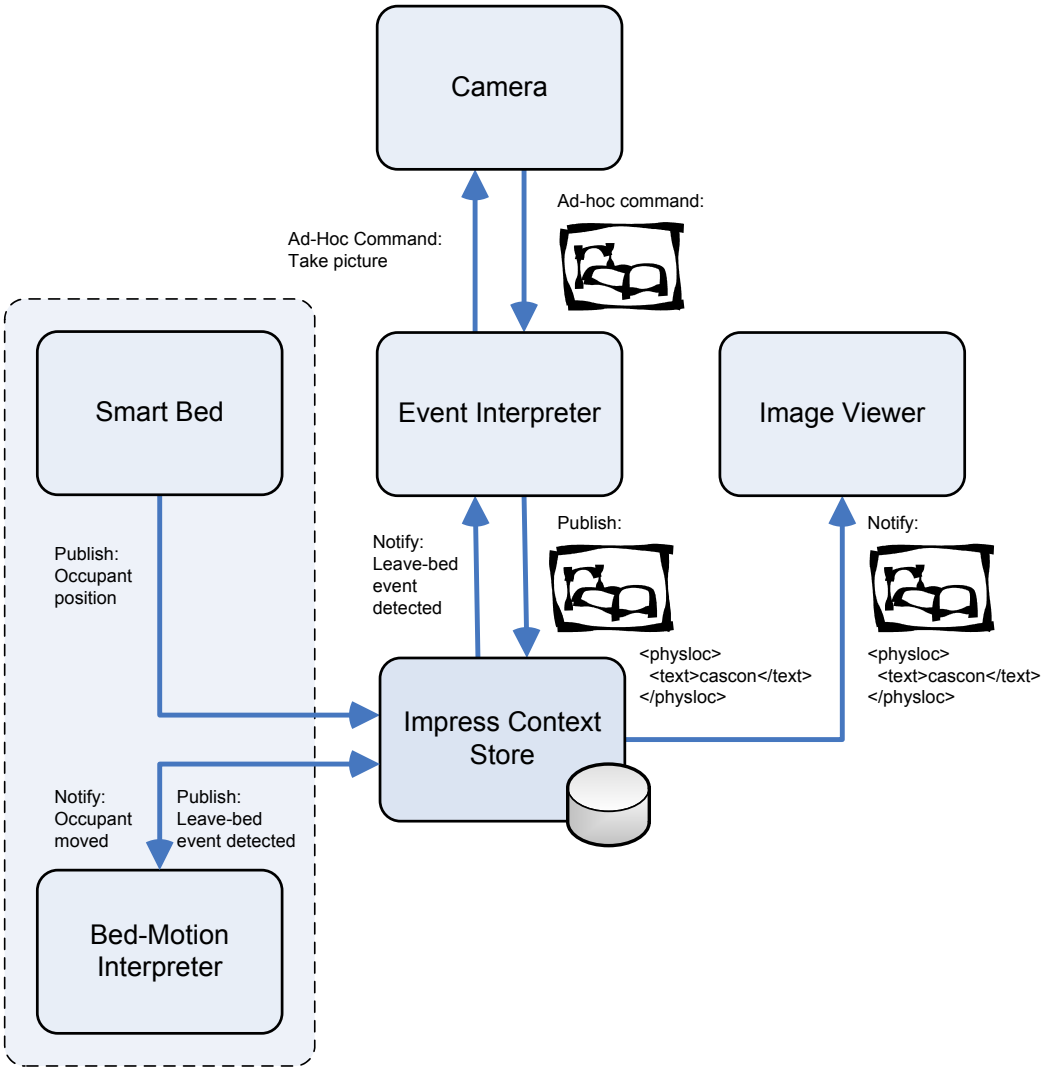

Figure 5.7: Components of the Smart Bed Application

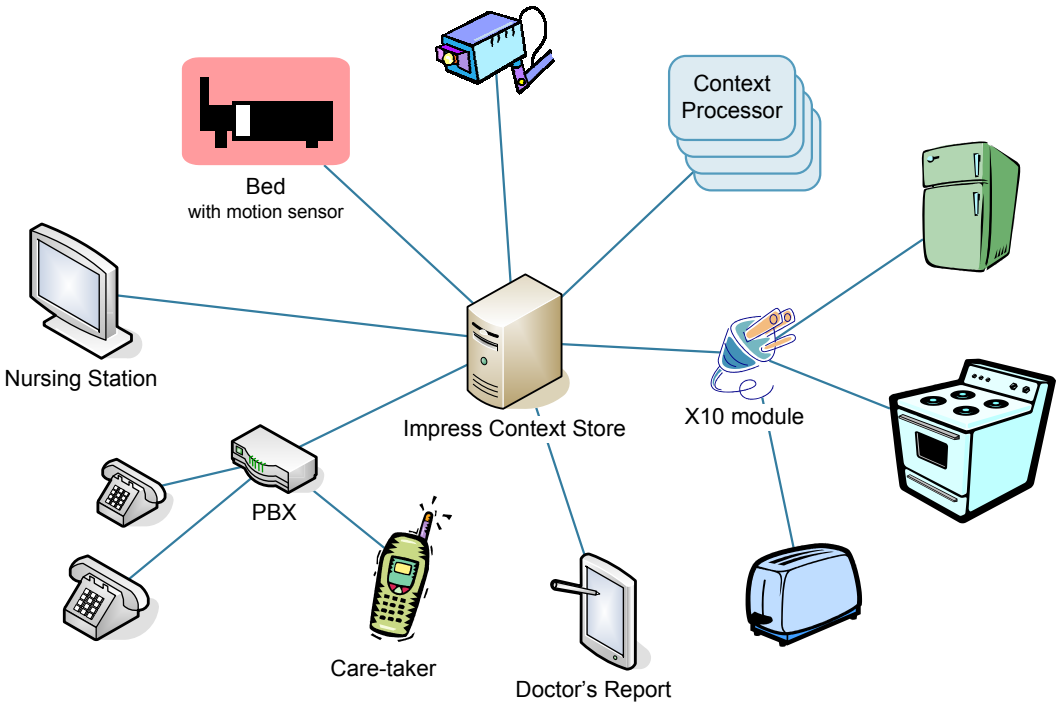

Figure 5.8: Big-Picture Scenario

appliance usage such as the time the stove or toaster was turned on and off. From this data, we can provide our system with even more context information that can aid trend analysis and pattern prediction. Furthermore, integrating the PBX with the appliance monitoring allows us to build an early-warning system that notifies the care-takers via the telephone, should a dangerous event occur. Utilizing the iCS framework, we are able to identify and exchange context information securely and efficiently among entities with minimum effort.

To this point, we have treated the iCS as an independent service located within the home or the hospital. This is desirable because the data stays close to where it will be used most frequently. Privacy concerns are also reduced because sensitive data can be contained in a relatively small area. Again in the context of the larger project, we have used TPIC, the "Token-passing Peer-to-Peer Interaction Coordination" protocol [6], to decide to disseminate, securely and with a fine grain of control, part of the data stored in one context store to other application entities.

> [TPIC] allows distributed users and data warehouses to transfer information to/from those who need/store it without requiring any knowl-

edge of global state, database architecture, or even implementation details [51].

The properties provided by TPIC are favorable for a large context-aware system because they are in line with the requirements we laid out in Section 3.1. Imagine having a wide-area context-aware system where individual context stores scattered around the service area manage minute-to-minute context around local areas, and these context stores communicate aggregated context with one another, based on predefined security constraints specified by the TPIC protocol. This is the vision that the Impress project, along with our sister projects, aspire to.

Next, to ensure iCS can scale, we evaluate its performance by comparing response times between plain-node operations with typed-node operations using the iCS API.

Chapter 6

Performance Evaluation

Scalability is a crucial requirement for context-aware systems. In order to support a large number of context producers and context consumers, the additional complexity required by iCS semantics has to be evaluated. We claim that iCS semantics can scale because the added complexity, compared to plain nodes, is constant. That is, the response times of typed-node operations are a small constant multiple of their counterparts. In this chapter, we discuss various experiments performed to justify this claim.

The focus of our performance evaluation investigates the performance and scalability of our framework. We do that through exercising the iCS API and comparing results between the plain-node and typed-node operations. Third-party software used by the API, such as jabberd 2 and Idavoll 2, are also contributing factors when interpreting the actual performance results. However, optimizing these third-party software is out of the scope of this thesis.

Throughout this chapter, we use the setup of Figure 6.1 to conduct our experiments. All machines - `jabber1` to `jabber4`, are dual Intel Pentium III 550 MHz machines with 256 MB of RAM, running Fedora Core 4. As we are interested in testing the performance of iCS semantics, to isolate performance bottlenecks and remove networking issues, the machines are connected via a Gigabit network with a Gigabit switch. While `jabber1` and `jabber4` run jabberd 2 and Idavoll 2 respectively, `jabber2` and `jabber3` are used for driving the requests. In these tests, messages are sent to the server asynchronously. This means that at any time, there may be multiple outstanding requests whose responses have yet to be received. Response time is defined as the time between when the test client sends a request and when it receives the corresponding response from the server.

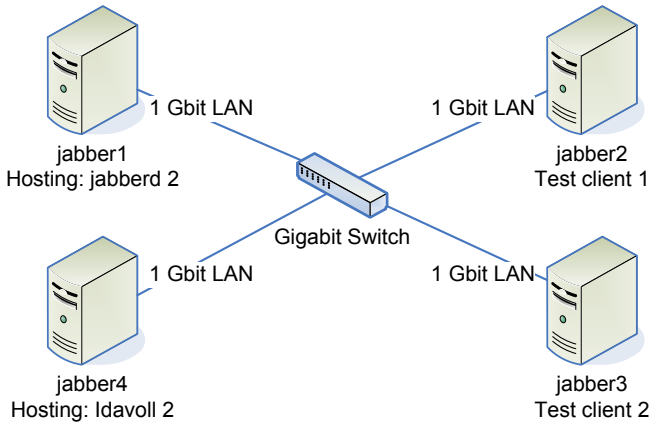

Figure 6.1: Load Test Setup

For the graphs below, each data point shows the mean of at least 1000 measurements. Error bars mark the 95% confidence interval about the means. For tests that involve varying the arrival rate of requests, a Poisson process is used and the mean response times of the operations are recorded on the graphs.

6.1 Node Creation

Depending on the application, creating a node may be a one-time or frequent operation. We expect node creation to be a low frequency operation in most cases. For example, it may only happen when a sensor or a context producer is added to the system. However, in Khan's work [29], node creation (and deletion) is a very frequent operation. Therefore, we want to make sure this operation is reasonably efficient even in the extreme case. Referring to Section 4.1.3, creating a typed node involves three messages to the pubsub service (i.e., creating the attribute node, creating the content node, and publishing the node attribute item). The publish request can only be sent upon receiving a successful response to a creation request. Therefore, the minimum response time for creating a typed node is $max((t_c+t_i), t_a)$, where t_a = time to create the attribute node, t_c = time to create the content node, and t_i = time to publish the node attribute item. Assuming $t_c = t_a$, this minimum time is approximately $t_c + t_i$. The server still observes thrice the load because of the need to service three messages.

We compare the response times for creating a plain node and a typed node with varying request rates. Figure 6.2 shows that the response time for creating a typed

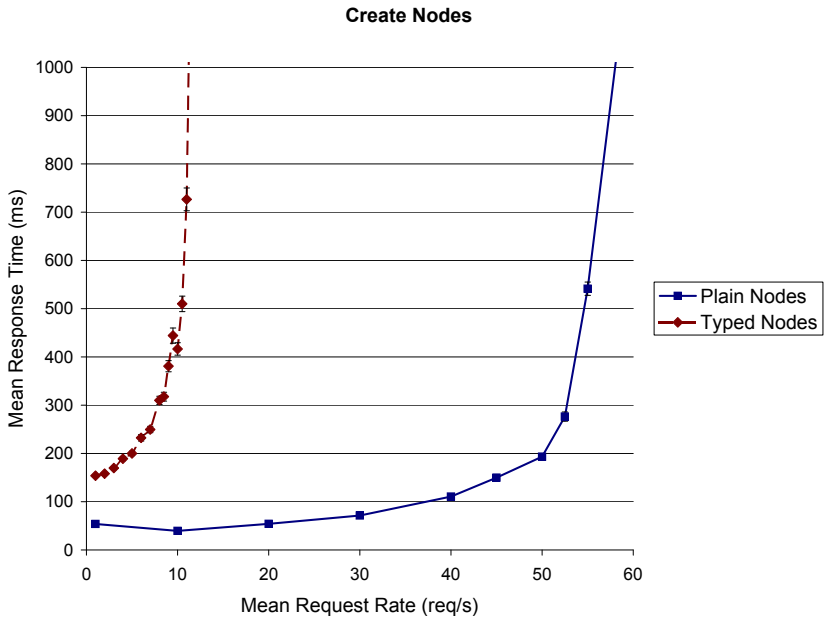

Figure 6.2: Node Creation Response Time vs. Mean Request Rate

node is about three times that of the plain nodes when the request rate is low. The additional time is due to the relatively higher service time for publishing an item. Also, the load of the server is relatively higher because of the additional steps needed in creating a typed node.

We see that the saturation point of the typed node, compared to the plain node, has reduced by a factor of five. We believe that the five-fold increase may be caused by inefficiencies in the server software. For example, we observed that the time it takes to create two nodes concurrently exceeds two to three times the time to create one. This may be contributed by inefficient thread management or database design. Further investigation and optimization of the third-party server implementation and the underlying PostgreSQL database, which is out of the scope of this thesis, should be able to pinpoint the issue.

6.2 Node Deletion

We expect that deleting a typed node takes twice as long as deleting a plain node. This is due to the fact that there are two nodes to delete in a typed node. We compare node deletion response times with varying request rates.

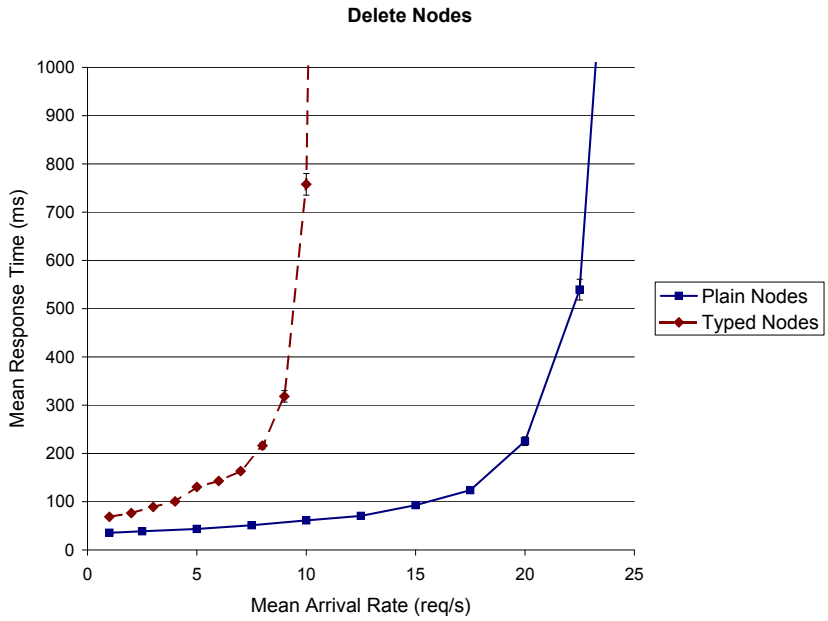

Figure 6.3: Node Deletion Response Time vs. Mean Request Rate

Figure 6.3 shows that the response times of the typed nodes are indeed roughly twice those of the plain nodes. The saturation point is slightly below half of the plain nodes, because while the plain node requests come one by one, the pubsub server is momentarily overwhelmed by the need to service two simultaneous requests for typed nodes.

6.3 Node Subscription

Subscribing to a node is another regular operation that needs to be performed efficiently. We expect the performance pattern to be similar to that of node deletion. In other words, response time for typed nodes should be doubled and the saturation point halved. In this test, we compare node subscription response times with varying request rates.

As expected, Figure 6.4 shows that the trend is similar to node deletion. The mean response time of subscribing to a typed node is up to twice the time of the plain node and the saturation point is reduced by about half.

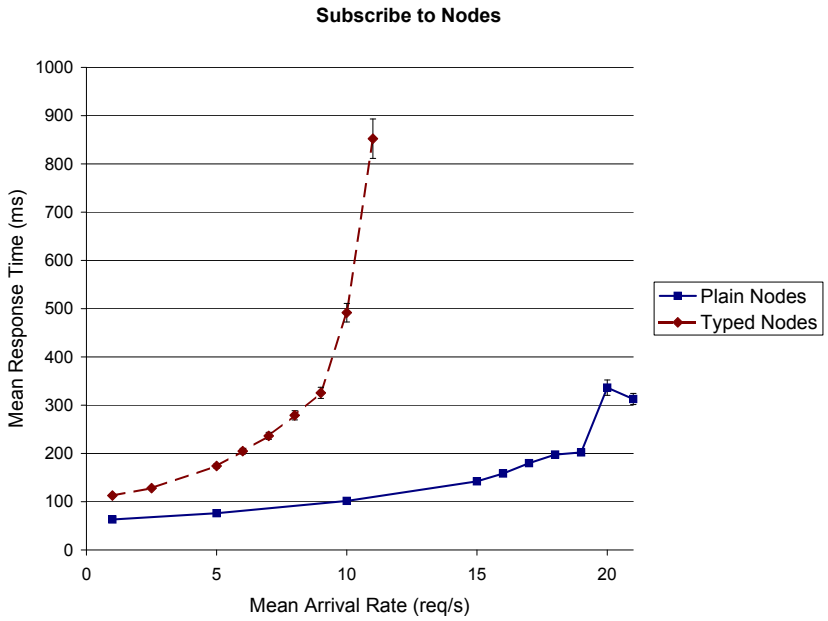

Figure 6.4: Node Subscription Response Time vs. Mean Request Rate

6.4 Subscribers and Publishers

Publishing to the pubsub service is one of the core operations. Its performance has serious implications on the usability of iCS. We expect a small increase in response time when publishing to typed nodes. Furthermore, we wish to see how the number of active, connected subscribers to a node affects the performance of the publisher of that node. In this test, we publish to plain nodes and typed nodes with a varying number of subscribers. Because this is not a load test of the pubsub service, the inter-arrival time is adjusted so that the CPU is not fully loaded.

Figure 6.5 shows that publishing to a typed node takes slightly longer than plain nodes and is linear. Given our test setup, the fluctuations we see in response time, around 50ms, is relatively small. More importantly, we see that the number of subscribers does not significantly affect the response time of the publisher. In reality, it is worth noting that CPU utilization increases as the number of subscribers increases.

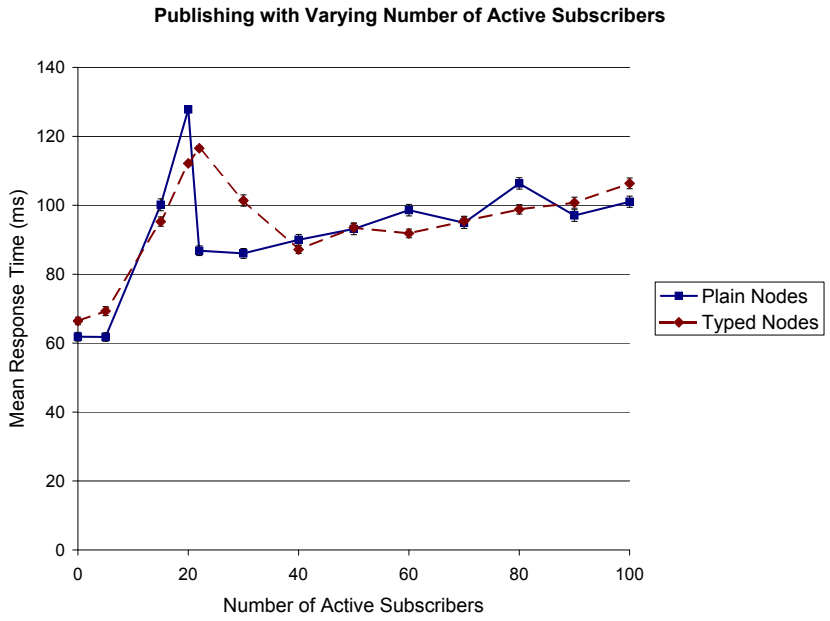

Figure 6.5: Publish Response Time vs. Number of Active Subscribers

6.5 Searching

To highlight how plain nodes and typed nodes differ in searching capabilities, we constructed a test to retrieve published items of varying sizes. In this test, the size of the custom attributes is set conservatively at 10KB. As we will see later, the actual size does not affect the conclusion of this test.

Figure 6.6 shows that the mean retrieval time of a typed node is constant. This is because only the attributes are retrieved in order to perform searches. However, using a plain node, the whole payload is retrieved when performing a search. The time needed for searching after retrieving the payload will also likely be less if meta-data exists. In this test, because the attribute node is 10KB in size, we see benefit when the payload is larger than 10KB. In general, as long as the size of the attributes is less than the actual payload, which is typical for most binary data or textual logs, typed nodes offer time-saving benefits. Moreover, similar effects can be seen when publishing large items. Since attribute items are used as event notifications, each subscriber only receives a fraction of the published data. On the other hand, when a large item is published to a plain node, all subscribers receive this large item regardless of whether the item is useful to them. The pubsub service may well grind to a halt while disseminating these items.

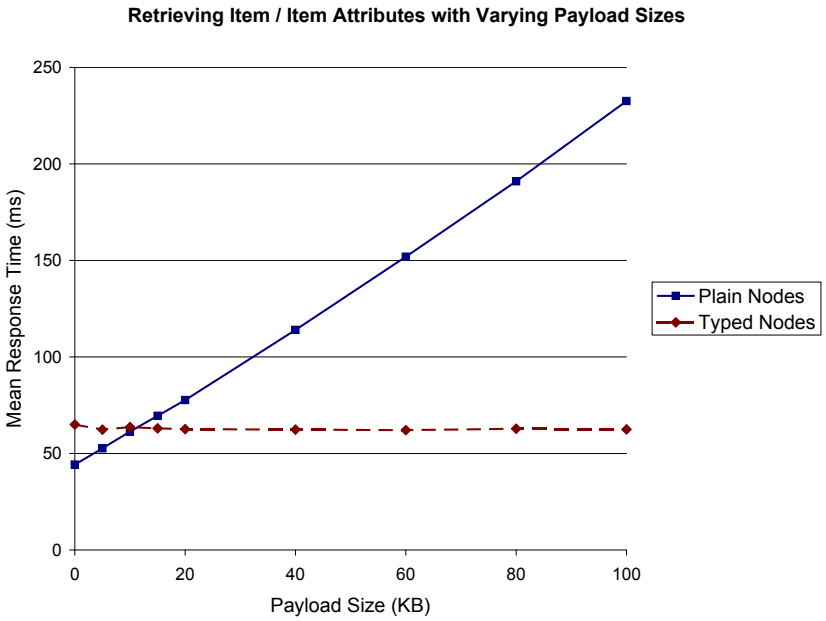

Figure 6.6: Retrieval Response Time vs. Payload Size

Summarizing these tests, we see that response times of the typed-node operations are at most two to three times that of their counterparts. Creating typed nodes is three times more expensive. However, the node-creation operation is an infrequent operation in general. Frequently-used operations such as publishing an item to multiple subscribers show little increase in time. When searching large binary data or textual logs, typed nodes are more efficient than plain nodes. The server load is kept down when publishing large payloads as well.

Next, we conclude the thesis and identify future work.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Thanks to rapid advances in miniaturized computers and electronic devices, sensors and actuators will be pervasive in our everyday lives in the foreseeable future. Intelligent systems are needed to manage these devices and process their data without human intervention. An underlying supporting infrastructure needs to be in place for context-aware systems, so that the system designers can focus their energy on the context part of the system, instead of reinventing the supporting infrastructure for each system.

The Impress Context Store (iCS) addresses the need for the separation of responsibility and provides a coordination framework for context-aware systems. It uses the open-source, standards-based XML protocol Jabber to ensure interoperability and compatibility. Using a persistent Publish-Subscribe model for iCS allows decoupling of context producers and context consumers. The effects of transient failures can be mitigated, and new devices can be added and removed effortlessly because of the decoupled nature of these devices. The system can scale easily due to its self-configuration character. Finally, security to ensure that sensitive data is kept private is provided naturally by Jabber.

This thesis proposes semantics to enhance the Jabber pubsub service to allow storing type information, and to enable searching using node and item attributes. Both are necessary to identify the items stored in the pubsub nodes. Bandwidth requirements are reduced, and processing time for binary data and long textual data is improved by providing searchable attributes along with the payload.

The work presented in this thesis takes a slightly different approach to building context-aware system than the concurrent work done by Omar Khan. In his thesis [29], Khan discussed an incremental approach to allow the context-aware system to self-learn using a set of ontologies. Underneath the context and the ontologies, however, is the iCS detailed in this thesis. The ontology approach is more heavy-weight and is suitable for higher-level context. On the other hand, the iCS typed-node approach is suitable for low-level context that is often larger in size. The iCS framework provides the groundwork to build robust context-aware systems. The iCS API and the iCS Browser provide accessible tools to speed up future research on these projects.

The Impress Context Store provides the context-system designers with a functional framework to manage context producers, context consumers, and context information. Development time and maintenance costs for feature-rich context-aware systems can be reduced as a result.

7.2 Future Work

In the future, we will develop more context-aware applications using iCS as the underlying coordinating framework. For example, we can publish daily walking patterns to iCS for analysis by a context consumer. Each application has its unique functional and performance requirements. It will be interesting to explore the maximum extent of the benefits iCS brings to these new applications. Our vision is to build a complete context-aware system to enhance the lives of elderly people.

There is one limitation of iCS that can be best addressed by modifying the Publish-Subscribe specification [36]. So far, in order to retrieve the content type of a typed node, the entities must, at least, subscribe to the attribute node. However, we believe some node attributes, including the content type, should be available to the public during service discovery. In order to achieve this, the specification will have to allow user-customized meta-data to be made available during service discovery. Node attributes can then store sensitive attributes and be viewed only by approved entities.

For the moment, we have concentrated on events that can tolerate high latency, up to the order of a few seconds. However, it may be possible to open up a wide range of use cases by considering low-latency communication, or even perhaps enabling initiation of out-of-band communication. We have seen such a case in the security-monitoring application. Last, incorporating a scheme to specify relationships between published events may be beneficial to the users. For instance, we

may be able to specify an item in iCS is caused by the aggregation of several other items. The cause-and-effect relationship enables a user to trace and search for related events.

During the development of the pilot applications, we found two defects in the libraries we use. First, Muse is not able to handle "x messages" (XML element with the name "x") in a pubsub item payload. Without loss of generality, we altered the "x message" format slightly to continue with our work. We notified the authors of Muse and a fix is underway. Second, Idavoll 2 has problem parsing the xmime XML stanza. The service corrupts the XML item. The author has issued a fix in the development tree that will be available to the public in the near future. We have found workarounds to mitigate the effects of these defects without loss of generality.

Bibliography

[1] J. Bacon and K. Moody. EDSAC(21): Event-driven, secure application control for the twenty-first century. `http://www.cl.cam.ac.uk/users/jmb/edsac21.ps`, April 2004.

[2] P. V. Biron, K. Permanente, and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. `http://www.w3.org/TR/xmlschema-2`, October 2004.

[3] J. Black. The Impress Project. `http://www.cs.uwaterloo.ca/~jpblack/research/Impress.html`, July 2005.

[4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[5] C. Chen. Muse home. `http://open.echomine.org/confluence/display/MUSE/Muse+Home`, July 2005.

[6] T. Chiasson. *Token-based Peer-to-Peer Interation Coordination*. PhD thesis, Dalhousie University, 2003.

[7] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[8] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2, 3 & 4):97–166, 2001.

[9] S. E. Doble, J. D. Fisk, and K. Rockwood. Assessing the ADL functioning of persons with Alzheimer's Disease: Comparison of family informants' ratings and performance-based assessment findings. In *International Psychogeriatrics*, volume 11, pages 399–409, 1999.

[10] R. Eatmon, J. Hildebrand, J. Miller, T. Muldowney, and P. Saint-André. JEP-0004: Data Forms. http://www.jabber.org/jeps/jep-0004.html, January 2006.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[12] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Middleware 2003*, volume 2672, pages 103–122, Rio de Janeiro, Brazil, 2003.

[13] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications*, 20(3):54–65, 2000.

[14] N. Freed and N. Borenstein. RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. http://www.ietf.org/rfc/rfc2045.txt, November 1996.

[15] N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt, November 1996.

[16] N. Freed and N. Borenstein. RFC 2049: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. http://www.ietf.org/rfc/rfc2049.txt, November 1996.

[17] N. Freed, J. Klensin, and J. Postel. RFC 2048: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. http://www.ietf.org/rfc/rfc2048.txt, November 1996.

[18] J. J. Garrett. AJAX: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, February 2005.

[19] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[20] Google. Google Talk. http://www.google.com/talk, August 2005.

[21] R. Hasha. Needed: A common distributed-object platform. In *Intelligent Systems and Their Applications*, volume 14, pages 14–16, Mar/Apr 1999.

[22] IBM. IBM Centers for Advanced Studies: CASCON - CASCON 2005. https://www-927.ibm.com/ibm/cas/cascon, 2005.

[23] Jabber Software Foundation. Jabber Servers. http://www.jabber.org/software/servers.shtml, 2005.

[24] Jabber Studio. jabberd 2. http://jabberd.jabberstudio.org/2, 2004.

[25] B. Johanson and A. Fox. The Event Heap: A coordination infrastructure for Interactive Workspaces. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 83–93, Washington, DC, USA, 2002. IEEE Computer Society.

[26] B. Johanson and A. Fox. Extending tuplespaces for coordination in interactive workspaces. *Journal of Systems and Software*, 69(3):243–266, 2004.

[27] B. Johanson, A. Fox, and T. Winograd. The Interactive Workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[28] A. Karmarkar and Ümit Yalçinalp. Describing media content of binary data in XML. http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504, May 2005.

[29] O. Z. Khan. Incremental deployment of context-aware applications. Master's thesis, School of Computer Science, University of Waterloo, 2006.

[30] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*, pages 70–78, Washington, DC, USA, 1999. IEEE Computer Society.

[31] C. MacKnight and K. Rockwood. A hierarchical assessment of balance and mobility. *Age and Ageing*, 24(2):126–30, 1995.

[32] C. MacKnight and K. Rockwood. Mobility and balance in the elderly: A guide to bedside assessment. In *Journal of Postgraduate Medicine*, volume 99, pages 269–271, 275–276, 1999.

[33] C. MacKnight and K. Rockwood. Rasch analysis of the hierarchical assessment of balance and mobility (HABAM). In *Journal of Clinical Epidemiology*, volume 53, pages 1242–1247, 2000.

[34] J. Mannermaa, K. Kalliomäki, T. Mansten, and S. Turunen. Timing performance of various GPS receivers. In *Joint Meeting of the European Frequency and Time Forum and the IEEE International Frequency Control Symposium*, volume 1, pages 287–290, Besançon, France, April 1999.

[35] R. Meijer. Idavoll. `http://idavoll.jabberstudio.org/2`, April 2005.

[36] P. Millard, P. Saint-André, and R. Meijer. JEP-0060: Publish-Subscribe. `http://www.jabber.org/jeps/jep-0060.html`, March 2003.

[37] M. Miller. JEP-0050: Ad-Hoc Commands. `http://www.jabber.org/jeps/jep-0050.html`, July 2005.

[38] J. C. Moon and S. J. Kang. A multi-agent architecture for intelligent home network service using tuple space model. *Consumer Electronics*, 46(3):791–794, August 2000.

[39] K. Moore. RFC 2047: Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text. `http://www.ietf.org/rfc/rfc2047.txt`, November 1996.

[40] L. I. Patterson, R. S. Turner, and R. M. Hyatt. Construction of a fault-tolerant distributed tuple-space. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 279–285, New York, NY, USA, 1993. ACM Press.

[41] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 368–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[42] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[43] Precarn. Intelligent system requirement for cognitively impaired individuals. `http://www.precarn.ca/IRIS/PrecarnUnivLedPrgm/prjILNWDJhoBi_en.html`, April 2003.

[44] P. Saint-André. JEP-0082: Jabber Date and Time Profiles. `http://www.jabber.org/jeps/jep-0082.html`, May 2003.

[45] P. Saint-André. RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core. http://www.ietf.org/rfc/rfc3920.txt, October 2004.

[46] P. Saint-André. RFC 3921: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. http://www.ietf.org/rfc/rfc3921.txt, October 2004.

[47] P. Saint-André. RFC 3922: Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM). http://www.ietf.org/rfc/rfc3922.txt, October 2004.

[48] P. Saint-André. RFC 3923: End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP). http://www.ietf.org/rfc/rfc3923.txt, October 2004.

[49] D. Salber, A. K. Dey, and G. D. Abowd. The Context Toolkit: Aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[50] T. Selkar and W. Burlesson. Context-aware design and interaction in computer systems. *IBM Systems Journal*, 39(3-4):880–891, 2000.

[51] M. Smit, M. McAllister, and J. Slonim. Privacy of electronic health records: Public opinion and practicalities. In *Networking and Electronic Commerce Research Conference*, Lake Garda, Italy, 2005. http://torch.cs.dal.ca/~smit/publications/public_opinion_electronic_health_records_solutions.pdf.

[52] M. Weiser. The computer for the 21st century. *Human-computer interaction: Toward the year 2000*, pages 933–940, 1995.

[53] G. C. Wells. New and improved: Linda in Java. In *PPPJ '04: Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 67–74, Las Vegas, Nevada, 2004. Trinity College Dublin.

[54] X10. X10 Home Security, Wireless Security Camera, Home Automation, Electronics and More! http://www.x10.com, 2005.

Appendix A

iCS API Methods

The iCS API contains the methods listed in Tables A.1 and A.2. The `PubSubService` interface provides basic pubsub features while the `ICSService` interface provides additional features offered by iCS. Complete API documentation is available along with the API library.

Methods	**Details**
`connect`	Connects to a pubsub service via a Jabber server.
`createNode`	Creates a node.
`deleteNode`	Deletes a node.
`disconnect`	Disconnect from the pubsub service.
`discoverItems`	Discovers items in a node.
`discoverNodeInfo`	Discovers node info and metadata.
`discoverNodes`	Discovers a list of available nodes.
`getItem`	Retrieves an item by the node and item identifier.
`getItems`	Gets a list of items for a node.
`publish`	Publishes an item to the pubsub service.
`purgeNode`	Purges items in a node.
`subscribe`	Subscribes to a node.
`unsubscribe`	Unsubscribes from a node.

Table A.1: Main Methods in the `PubSubService` interface

Methods	Details
`createTypedNode`	Creates a typed node with node attributes.
`getContentType`	Retrieves the content type of a typed node.
`getItem`	Retrieves the payload and item attributes by typed node and item identifier.
`getNodeAttributes`	Retrieves the node attributes of a typed node.
`publishBinary`	Publishes data as binary to typed node.
`publishText`	Publishes data as text to plain node.

Table A.2: Main Methods in the `ICSService` interface

Appendix B

Content-Node Item Schema

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://impress.uwaterloo.ca/ics/base64"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
    targetNamespace="http://impress.uwaterloo.ca/ics/base64">

  <xs:import namespace="http://www.w3.org/2005/05/xmlmime"
      schemaLocation="http://www.w3.org/2005/05/xmlmime"/>

  <xs:complexType name="BinaryType">
    <xs:simpleContent>
      <xs:restriction base="xmime:base64Binary" >
        <xs:attribute ref="xmime:contentType"
            use="required"/>
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>

  <!-- This element designates the range of values
       that the element definition will accept -->
  <xs:element name="base64" type="tns:BinaryType"
      xmime:expectedContentTypes="*"/>

</xs:schema>
```