Distribution of Context Information using the Session Initiation Protocol (SIP)

CARLOS ANGELES PIÑA

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2008

COS/CCS 2008-12

Distribution of Context Information using the Session Initiation Protocol (SIP)

Carlos Angeles Piña

angeles@kth.se

School of Information and Communication Technology

Royal Institute of Technology

Master of Science Thesis

Project conducted at Appear Networks

June 23, 2008

Academic Supervisor and Examiner, KTH

Professor Gerald Q. Maguire Jr.

Industrial Supervisor, Appear Networks

Alisa Devlic

ABSTRACT

Context-aware applications are applications that exploit knowledge of the situation of the user (i.e. the user's context) to adapt their behavior, thus helping the user achieve his or her daily tasks. Today, the transfer of context information needs to take place over unreliable and dynamically changing networks. Moreover context information may be produced in different devices connected to different networks. These difficulties have limited the development of context-aware applications. This thesis presents a context distribution method exploiting the event notification mechanisms of the Session Initiation Protocol (SIP), aiming to provide access to context information regardless of where it is produced.

The context distribution component presented in this thesis uses SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE) to enable context sharing by using a SIP presence server, specifically the SIP Express Router (SER) and its presence module.

This context distribution component allows distribution of context information in both synchronous and asynchronous mode. The distribution mode depends on the application requirements for context distribution, as well as the nature and characteristics of the context-information. In this thesis, based on system scalability, the user's mobility, and latency - recommendations are given about in which situations each mode is more suitable for distributing context information.

The system was evaluated using a load generator. The evaluation revealed that the server is highly scalable. The response time for synchronous retrieval of context information is nearly constant, while in asynchronous mode the time to process a subscription increases with the amount of information in the database regarding previous subscriptions. Notifications are sent at a regular rate (≈2800 notifications per second); however there is a purposely random delay (0 to 1 second), between an update of context information (i.e. receipt of a publish message) and the start of notifications to subscribed users.

The requirements of the context-aware applications using the distribution component, such as response time, have to be taken into account when deciding upon the mode of context distribution for each application. This thesis provides some empirical data to help an application developer make this selection.

Sammanfattning

Kontext-medvetna (eng. Context-aware) applikationer är applikationer som utnyttjar information om användarens situation (d.v.s. användarens kontext) och förändrar applikationens beteende i syfte att hjälpa användaren i dennes vardagliga arbetsuppgiften. Idag överförs kontextuell-information (eng. context information) i nätverk som är opålitliga och dynamiskt föränderliga. Därtill tillkommer komplexiteten att kontextuell-information är ibland producerad i olika noder anslutna till olika nätverk.Utvecklingen av kontext-medvetna applikationer har hittills begränsats av ovannämnda svårigheter. Denna avhandling presenterar en metod för att distribuera kontextuell-information genom användning av mekanismer för händelsemeddelande (eng. event notification mechanisms) inbyggda i Session Initiation Protocol (SIP). Målet är att undersöka hur metoden kan användas för att möjliggöra tillgång till kontextuell-information oavsett vart den är producerad.

Komponenten för distribution av kontextuell data, som presenteras i denna uppsats, använder SIP för direktmeddelanden (eng. Instant Messaging) och tekniken "Presence Leveraging Extensions (SIMPLE)" för datadelning av kontextuell data (eng. Context sharing). För detta ändamål används SIP närvaroserver (eng. SIP presence server), mer specifikt modulen för närvaroinformation tillhörande SIP Expressroutrar (SER).

Komponenten för distribution av kontextuell information möjliggör både synkront och asynkront distribution. Valet mellan de två beror delvist på applikationens kravspecifikation för distribution av kontextuell information, delvist på typen av den kontextuella informationen. Baserat på systemet skalbarhet (eng. Scalability), användarens rörlighet och latens (eng. latency) kan man ge rekommendationer vilken av de två distributionssätten, synkront eller asynkront, som är lämpligast för distributionen av kontextuell information.

Systemet utvärderades med hjälp av ett program som genererar belastning (eng. load generator). Resultaten visar att systemet är mycket skalbart. Responstiden för synkront åtkomst av kontextuell information är nästan konstant, medan responstiden för asynkront åtkomst ökar med informationsmängden i databasen, i respekt till den föregående prenumerationen av kontextändringar. Händelsemeddelande skickas regelbundet (≈2800 meddelande per sekund). Vi har dock medvetet valt att skapa en slumpmässigt dröjsmål (0 till 1 sekund) mellan varje uppdatering av kontextuell information (t.ex. en kvitto på en Publish-meddelande) och den tidpunkten då händelsemeddelande skickas till de användare som prenumererar på ändringarna.

För utvecklingen av varje kontext-medveten applikation, som distribuerar kontextuell information måste man ta hänsyn till responstid vid beslut huruvida man ska välja synkront eller asynkront sätt för distribution. Denna uppsats ger empirisk data som hjälper applikationsutvecklare i detta val.

Acknowledgements

I would like to express my gratitude to my academic supervisor and examiner, Profesor Gerald Quentin Maguire Jr, thank you for all the times you reviewed my thesis, all the valuable comments you gave me help me to learn and to accomplish a better work. I would also like to thank my industrial supervisor, Alisa Devlic, for her invaluable assistance, support and guidance. Thank you for the innumerable times we had discussions, for all the comments and advices you gave me, and for being a really good friend.

Thanks also to all my colleagues at Appear Networks, especially to the EU Team: Attakorn, Giorgios, Hossein, Kai, and Vedran, for creating a very nice working environment, I really had fun these last months. Also I should thank all my friends for always being there.

Last, I would like to thank my parents for their support, encouragement, and endless love, through the duration of my studies.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

3GPP	Third Generation Partnership Project
CODEC	Coder-Decoder
CONTEXTITY	Context Entity
CPP	Common Profiles for Presence
DUMMBO	Dynamic Ubiquitous Mobile Meeting Board
HTTP	Hyper Text Transfer Protocol
IETF	Internet Engineering Task Force
IST	Information Society Technology
MADAM	Middleware for Adaptive Applications
MIDAS	Middleware for the Deployment of Mobile Services in ad-hoc Networks
MUSIC	Self-Adapting Applications for Mobile Users In Ubiquitous Computing Environments
PAN	Personal Area Network
PIDF	Presence Information Data Format
PRESENTITY	Presence Entity
RFC	Request For Comments
RPID	Rich Presence Information Data Format
SCTP	Stream Control Transmission Protocol
SER	SIP Express Router
SIMPLE	SIP for Instant Messaging and Presence Leveraging Extensions
SIMS	Semantic Interfaces for Mobile Services
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

1. Introduction

The popularity of mobile devices, such as smarthphones or handhelds, has increased in recent years, and along with their increased adoption have come great challenges to developers. Applications in mobile distributed environments should adapt according to the situation and needs of the users. Schilit [1], defines such applications as context-aware applications. This class of applications should exploit knowledge of the user's context and take advantage of this information in order to adapt their behavior accordingly. The result is both more useful and less cumbersome applications which truly support the users.

The context information associated with a user includes all the information that may affect the interaction between this user and a system. Contextual information might include the location of the user, the temperature of their environment, date, time, ambient brightness, ambient noise level, the available network bandwidth, the remaining battery power of the device, the user's personal profile, etc.

Today, computers can be found everywhere, in mobile phones, music players, cars, etc. In addition to difference in programmability (ranging from fixed function to general purpose computers) theses device support different communication and networking capabilities. Mark Weiser popularized the idea of having computing anywhere and anytime [2], today this is known as ubiquitous computing. Because of the decreasing prices and increasing adoption of computing and communications systems, computers are becoming ubiquitous in our daily lives. Today a person commonly employs several different devices, often highly specialized for specific applications or use settings. Unlike a ubiquitous computing environment, where users would not have to bring these devices with them - today users must bring much of their computing and communications with them. This often causes problems as the user needs to configure the device differently for use in different locations, via different access networks, etc. In order to reduce this configuration and management burden from the user, many believe that users could benefit from making more effective use of information about their context, for example by allowing context-aware application to help the user in their tasks.

The "Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environment" (MUSIC) project [3] is a project funded by the European Commission under the Information Society Technology (IST) priority under the 6th Framework Programme as an Integrated Project. The MUSIC project aims to provide an open platform that makes it technically and commercially feasible for the wider IT industry to develop mobile applications that are context-aware (understand user's context), self adapting (dynamically adapt to changes in context), and inherently distributed – while supporting interactions between multiple users. More specifically the project's proposal states:

> "MUSIC will provide a design methodology and distributed system architecture for the design and implementation of self-adapting applications in ubiquitous computing environments. This will be complemented with enhanced modeling languages for the specification of context dependencies and adaptation capabilities, supported by model specification, validation and simulation tools. This platform will be used to develop trial services, based on a set of challenging application scenarios with real market potential, having a central role: as sources of requirements, to assess technical adequacy of the results, and to promote the results." [4]

Applications developed using the MUSIC framework will be capable of adapting in highly dynamic user and execution contexts while maintaining a high level of usefulness across context changes. The MUSIC architecture [5] includes both context and adaptation middleware. The context middleware is responsible for monitoring, managing, and detecting changes in the user's context. The **context middleware** also has the task of distributing contextual information to the relevant nodes. The **adaptation middleware** controls, tunes, and monitors the adaptation of applications according to context changes.

1.1 Problem Statement

Context-aware systems should not be limited to a specific physical space or network. Due to the wide variety of network technologies (e.g. WLAN, 3G, GPRS, Bluetooth, etc.) context information may need to be shared among different context-aware applications running on nodes (possibly) connected to different networks. Wei Li [40] identified that the difficulty of distributing context information among context-aware applications has limited the spreading of context-aware systems, especially because context information providers may be widely dispersed throughout the Internet. This thesis studies the distribution of contextual information for enabling applications (that may be in a distributed environment) to take advantage of such information. The MUSIC context middleware includes a context-distribution manager that provides access to context information regardless of where it is produced. The context distribution component is responsible for supporting both synchronous and asynchronous access to context data, as well as for sharing context information among different networked instances of the middleware available on different nodes.

1.2 Objectives

Due to differences in the nature of the different elements of context information, its distribution may need to be done in a synchronous fashion, as well as in an asynchronous fashion in order to provide the relevant information to applications at the proper time. This thesis has the objectifo of stydying how the Session Initiation

Protocol (SIP) may be used in order to share context information synchronously and asynchronously. SIP offers a great variety of powerful features that are suitable for the transmission of context information among devices. SIP is an open standard protocol that is widely used by the networking industry and by the research community.

The proposed approach for distributing context information used in this thesis is based in the SIP SIMPLE (SIP for Instant Messaging and Presence Leveraging Extensions). This protocol provides instant messaging and presence functionalities, but can also be used for sharing context information in a SIP-enabled network. The context distribution system presented in this thesis uses the SIP Express Router (SER), as a presence/context server, and a SIP user agent for enabling context information delivery.

The main goal of this thesis is to analyze and compare the synchronous and asynchronous modes of context distribution in order to give recommendations about when to use each mode, based on the desired scalability of the system and the response time, while also considering the context dynamics.

1.3 Thesis Structure

The remainder of this thesis is organized as follows: **Chapter 2** presents a usage case, via a scenario showing how context information distribution assists in coping with an emergency situation at an airport. **Chapter 3** explores and briefly describes related protocols and the background needed to understand the proposed system. **Chapter 4** describes the architecture and the methodology of the system for our proposed approach for distributing context information. An evaluation of the system in terms of scalability and response time is presented in **Chapter 5**. In **Chapter 6** we provide some suggestions about how to distribute context information based in the evaluation of SIP/SIMPLE as a protocol for distributing context information. Finally **Chapter 7** presents some conclusions and suggestions for future work in the area.

2. An Emergency Scenario at an Airport

This Chapter describes a usage case showing the need for distributing context information through an emergency scenario in an airport. A context-aware application is proposed in order to assist in coping with an emergency.

Aerodrome emergency planning is the process of preparing an airport to cope with an emergency occurring at the airport or in its vicinity. The main goal of emergency planning is to minimize the negative effects of an emergency particularly with respect to saving lives and maintaining aircraft operations. The following user scenario is based on the actual airport process from the Southwest Florida International Airport and is an extension of a user case from Appear Networks [7].

In order to assist the airport personnel in coping with an emergency at or near an airport a context-aware application may be beneficial. A major requirement of this application is that it dynamically reacts to the demanding emergency and security situations that arise in the airport everyday by directing relevant information to the most suitable response team, where suitability is in terms of abilities, equipment, roles, location, etc.

The National Fire Alarm Code (NPFA72) [6] norms state that "emergency teams need to be able to correctly respond to a security situation within a critical three minute window" before starting massive evacuation of premises. Within this three minute window the emergency staff should discover an activated fire alarm and determine the alarm's location, conclude whether the alarm's activation was legitimate or a "false alarm", handle the possible fire and reset this alarm in order to ensure passenger safety and minimal disruption to the functioning of the airport [7]

A context aware application may be helpful for such an emergency scenario at an airport because properly handling the emergency involves making decisions in a complex environment with hundreds of providers of context information. Moreover, the system has to cover large areas of the airport and an emergency event may involve coordination of dozens of different types of workers (guards, maintenance workers, airline, police, fire, medical personnel, etc).

A fire emergency at the airport requires the coordination between several different units at the airport: command center, public relations, airlines, maintenance, police, and fire department. Different tasks both must and should be performed in order to properly deal with the emergency; some of these tasks should be performed in parallel, while others are dependent on the completion of prior tasks.

Figure 1 shows the complete flow of tasks and events for handling a fire at the airport. It can be seen that when a fire is detected by sensors an alert is sent to the command center, to the public relations, fire, police, and maintenance departments. The fire department is responsible for finding the source of the alarm, handling the possible fire, and resetting the fire alarm panel. At the same time the police department should control the crowds and manage the opening and close of doors. After the handling of

the fire the maintenance department should open the air conditioning baffles and close the water valves. Finally the airline employees should restart the baggage conveyer belt. In order to maintain airport operations all these tasks should be performed within a 3 minutes window in order to avoid initiating an evacuation of the whole airport.

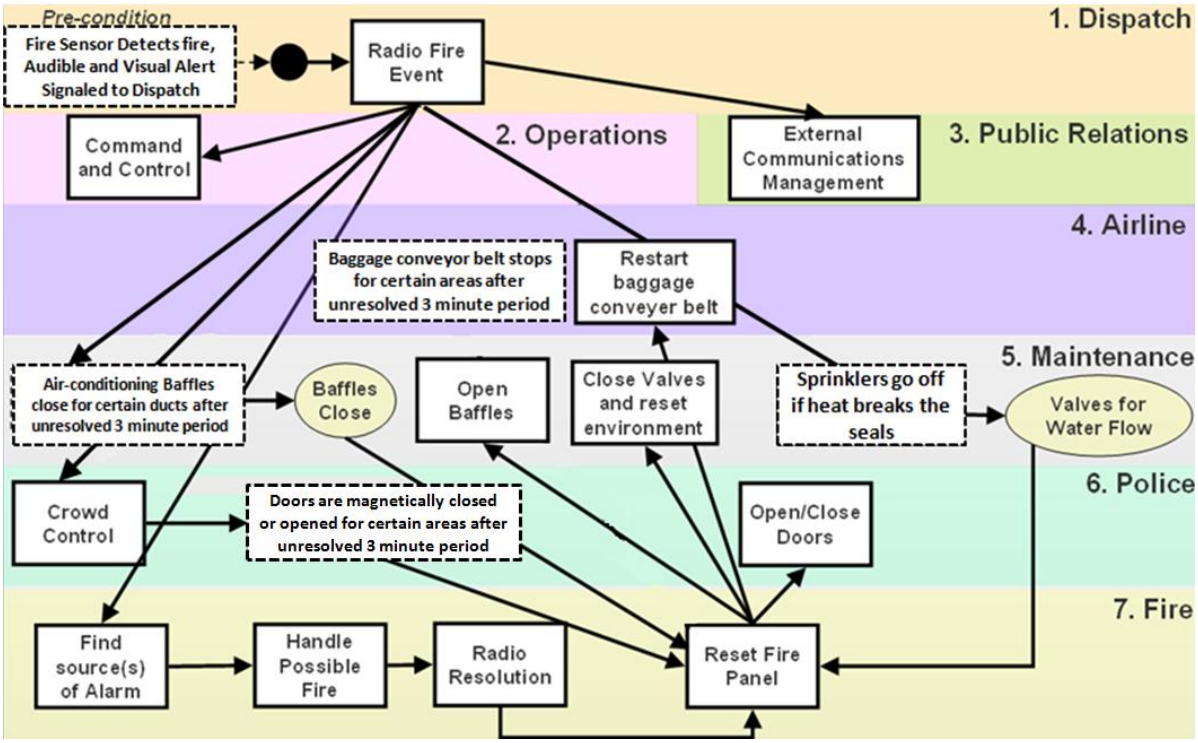


Figure 1. Task flow for a possible fire emergency at an airport

The context aware application, possibly running in a mobile device, may assist in the coordination of the personnel; providing the relevant information for facilitating the users making good decisions and increase the speed of their response. For instance when a fire is detected by the sensors, an alert containing the location of the sensor(s), this enables the visual display of this location on a map (or floor plans) – along with displaying pertinent information about the emergency can be communicated to the relevant personnel. The relevance of a worker is computed based on this person's abilities, their current location, their equipment, and their completion of other tasks. A fire alert will be sent to available fire department personnel close to the incident; provided that they have the necessary equipment to deal with the problem. This alert may also contain a map (including documentation about the water valves). At the same time an alert containing information about the incident will be sent to the guards of the airport closest to the emergency to inform them so that they can perform crowd control.

After handling of the fire, some tasks have to be scheduled to be performed by the maintenance unit. The context-aware application should assign tasks to different maintenance workers according to their capabilities, location, presence information, current activity, task completion progress, etc. The application will also track the progress of the many different tasks, based on time (task assignment and planned completion) and feedback received from each worker.

In the emergency scenario several types of context information are involved: each user's geographic location, location of the fire, presence information, each user's profile, device type and capabilities, current task, task completion progress, available network bandwidth, temperature, and presence of smoke, toxic fumes, etc.

Table 1. Distribution of Context information

Context	With Whom?
Location of user	Between user and control center and peers
Location of fire	Fire detector sending location to other users and control center
User´s Profile	Between users (workers, firefighters, etc.) and the control center
Presence Information	Between users and the control center
Current Task	Between users and the control center
Task completion progress	Between users and the control center
Available bandwidth	With the control center
Temperature	With the control center
Presence of smoke, fumes, etc.	With the control center

The context information produced by temperature, sensors, smoke detectors, external applications, positioning systems, and a worker's profile should be distributed among the different personnel involved in dealing with the emergency. This context information may be distributed with different periodicity according to the characteristics of the information and the current situation. Table 1 summarizes which information may be distributed and among whom. Within this scenario the distribution of context is important for coordinating the activities of workers and assisting then in their decision making in order to solve their part of the problem. It is important to note that this context application is not designed to be suitable for major emergencies, but rather is focused on dealing with minor emergencies that can arise frequently in an airport. The system is not designed to be suitable for large scale emergencies as this is not its target market; thus it does not have program logic to marshal large numbers of responders nor to interact with new entities (such as national or regional emergency personnel). However, by focusing on the most frequently occurring minor and small scale emergencies it should have greater applicability and produce savings for the typical operations of an airport. Based on the evaluation of our context distribution approach we will try to find which distribution mode is better for each type of context information, this will be presented in Chapter 6.

3. Background

This Chapter provides an overview of topics that are useful for understanding the work performed in this thesis. Also related work in the area is presented.

3.1 The MUSIC Project

Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environment (MUSIC) is an integrated project funded by the European Commission under the Information Society Technology (IST) priority. The main objective of MUSIC is to develop an open technology platform for software developers in order to facilitate the development of context aware and self-adapting mobile applications, capable of adapting in highly dynamic user and execution context -while maintaining a high level of usefulness across context changes [3]. Self-adapting applications are capable to dynamically adapt their functionality and internal implementation mechanisms to changes in context.

MUSIC is an integrated project because it integrates and extends the results of other European research and development projects, such as, Middleware for Adaptive Applications (MADAM), Semantic Interfaces for Mobile Services (SIMS), and Middleware for the Deployment of mobile services in ad-hoc networks (MIDAS).

The layered view of the MUSIC architecture [5] is shown in Figure 2. As can be seen from the figure, the MUSIC architecture is divided into two main blocks: The MUSIC Studio and the runtime environment. The MUSIC Studio represents a set of tools that provide support to developers. These tools are needed in order to build the adaptation model for applications. The runtime environment block is divided into three different layers: system services, middleware, and applications.

The Application layer is responsible for managing the different MUSIC applications running on top of the MUSIC middleware. The MUSIC middleware is divided into the context middleware and the adaptation middleware. The context middleware collects, organizes, manages, and shares context information - making this information available to the adaptation middleware. The context middleware may also be used directly by context-aware applications. The adaptation middleware analyzes the changes of the context and its impact on the application(s) in order to adapt the set of running applications to the current circumstances (e.g. resource situation).[5].

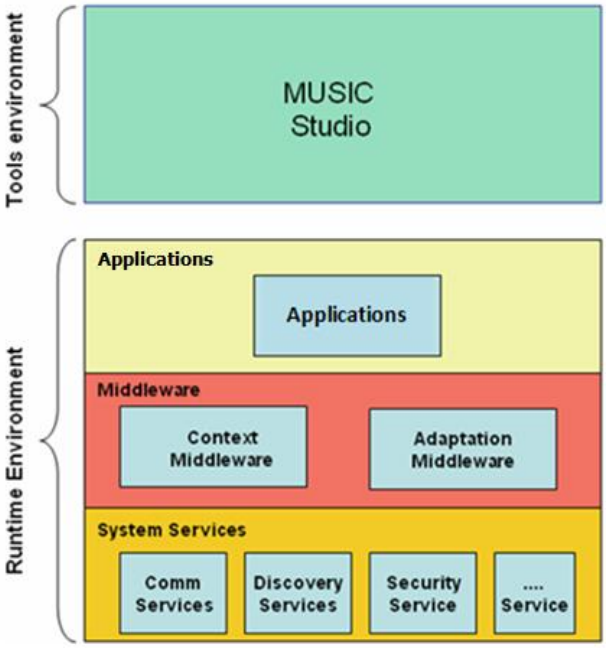


Figure 2. The layered MUSIC architecture

The context middleware is composed of several blocks, as shown in Figure 3. The context manager is the central hub of the context middleware. It is responsible for coordinating the rest of the middleware components and interoprating with them. The context providers manager communicates any data generated by the plugged-in sensors to the context manager, and enables external context sensors to dynamically connect and disconnect from the middleware.

The context query processor component reads parses and executes database-style queries passed to the context middleware. This component has access to the Context Cache (which that acts as a temporary storage for context information). The context cache stores only the most recent historical context in order to provide quick access to context clients. The context cache component interacts with the context history component, so that the later can store historical context data for longer periods of time. The ontology manager is responsible of the management of a set of ontologies that can be used by other components of the context middleware or by applications.

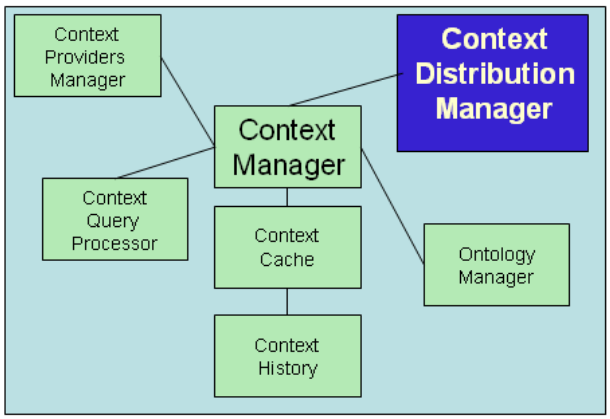


Figure 3. The MUSIC context middleware

The MUSIC Distribution manager is the component within the MUSIC architecture responsible for distributing context data among devices. The main function of this component is to distribute context information within distributed context spaces. Within the MUSIC project a context space is a collection of context types and values. This context space is populated with information from context sensors. The context distribution manager helps to avoid hardware replication and enables the ubiquitous computing paradigm by providing methods for distributing context among different devices and instances of the middleware.

The MUSIC network architecture is depicted in Figure 4. It includes three different types of nodes:

i) Personal Area Network (PAN) nodes with a Bluetooth interface that can have direct communication with other devices in the PAN.

ii) Nodes with Bluetooth and Wi-Fi interfaces linking the PAN devices with a gateway node.

iii) Gateway nodes enabling communication between different PANs, WLANs, and throughout the Internet.

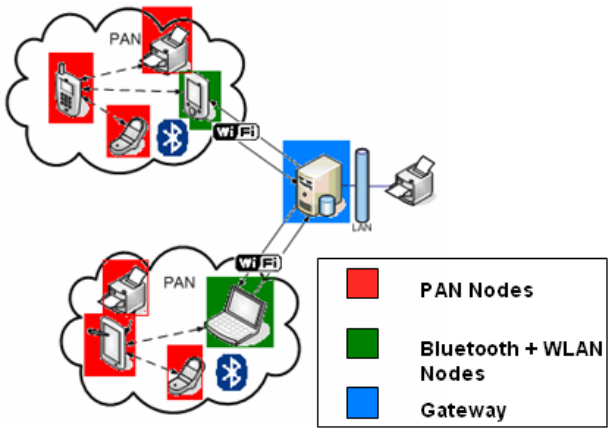


Figure 4. MUSIC Network Architecture

In this architecture the distribution manager distinguishes between two different types of context distribution:

1) Context distribution in an ad hoc environment based on a WLAN service discovery protocol presented in [8]. The main goal of this protocol is to discover peers and exchange context information. For a deeper insight into this protocol and context distribution scheme see [5] and [8].

2) Context Distribution in infrastructure-based environments using SIP. The idea behind this context distribution method is that each device locally stores discovered context information, which is distributed, synchronously and/or asynchronously to remote devices that request this information. Whether the synchronous or asynchronous mode of distribution is used depends on the characteristics of the

context information, such as how often it changes, the number of devices interested in specific information, as well as the user's mobility. A SIP proxy server is used in the Distribution Manager in order to receive queries and respond to them as shown in Figure 5. The MUSIC distribution manager will take advantage of the SIP addressing scheme (i.e. utilize URIs). All the devices with access to the SIP proxy can make synchronous or asynchronous queries to the SIP proxy in order to retrieve information concerning potential remote peers.

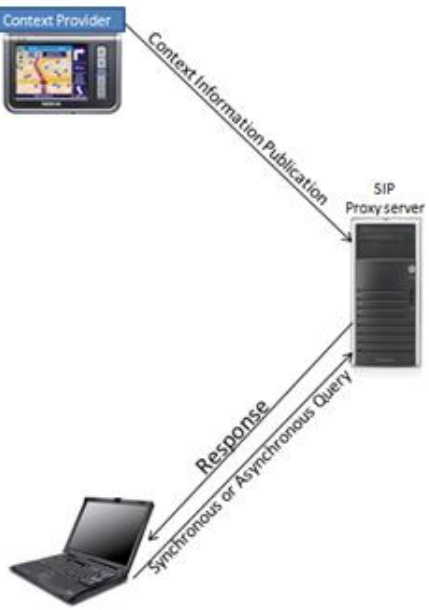


Figure 5. Distributing context information using a SIP proxy server

When a synchronous query occurs the peer must immediately responds to this query. An asynchronous query occurs when an application subscribes to an event. Subsequently, when a change (relevant to the subscribed event) occurs (while the subscription is valid), the user will be notified of this change, thus decoupling the query from the response. It is important to note that in the SIP event notification framework, after a subscription, the user also gets an immediate response with the current status of the peer within the subscribed event.

3.2 What is Context?

Throughout the Ubiquitous Computing research community we can find several definitions of context. Schilit and Theimer [9], were the first to introduce the term *context-aware*, and they define context as *"location, identities of nearby people and objects and changes to those objects"*. For Schilit the important components of the contest are: where you are, who you are with, and what resources are nearby.

Context is also defined by Schilit et al. [1] as the constantly changing execution environment. This environment includes the computing environment (available processors, devices capabilities, etc.), the user environment (location, social situation, etc.), and the physical environment (light intensity, noise level, etc.).

Dey and Abowd. [10] define 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"*.

Based in some eariler definitions, the MUSIC project [11] uses context to denote the circumstances and conditions under which services provided by applications and other software systems (e. g. middleware) are being used. This context may be subdivided into three main groups:

- **User context** related to the user of a service;
- **System context** that includes the properties of the execution environment of an application and;
- **Environmental context** that reflects information concerning the object's surroundings (e.g. location, weather, etc).

3.3 Context-Awareness

The first attempt to create a context-aware application occurred in 1992 with the Olivetti Active Badge location system [12]. This system used wall-mounted sensors responsible for collecting infrared IDs that were broadcasted by tags worn by building's occupants. The first application that used this system routed telephone calls to the extension nearest the intended recipient.

The first definition of context-aware computing was given by Schilit and Theimer [9] as "software that adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time". This first definition of context-aware applications limits the software to only adapt to context, however applications may exploit context information not only for adaptation, but for executing services, displaying information, etc.

A more general definition of context-aware computing is given by Pascoe [13] as *"the ability of computing devices to detect and sense, interpret and respond to aspects of a user's local environment and the computing devices themselves"*.

Dey [10] gives the following definition "*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task"*. According to Dey, context-aware applications use context to perform some behavior that can be: displaying context, automatically executing/adapting services, or tagging captured information for easier retrieval.

An extension to the previous definition is presented by Wei [14] describing the goal of context-aware computing as *"to provide computers with an awareness of user's situation by feeding them with various background (contextual) information, based on that computers can take some actions on behalf of the users without their explicit interference, thus making their attention more focused and interaction more efficient".*

The definition of context-awareness used in the MUSIC [11] project follows these earlier ideas. *"Context-awareness is the ability of an application (possibly middleware) to be conscious of the context and to act on its knowledge about the context".* This definition emphasizes that context-aware applications should exploit the knowledge of context as an integral part of their functionality.

3.4 Context-Aware Applications

Throughout the Ubiquitous Computing research literature we can find a wide variety of context-aware applications. The following subsections describe some of the research that has contributed to greater understanding and exploitation of context-awareness.

3.4.1 The PARCTab Mobile Computing System

The Xerox PARCTab [15] used a palm-sized tablet computer capable of communicating via a network of infrared (IR) transceivers. In this system the PARCTab controlled the applications, but these were running in remote hosts, with the results displayed on the tablet. Some of the applications running in the PARCTab system used location information provided by the microcellular IR network, becoming the first context-aware applications running on a handheld device.

The main focus of the PARCTab was to function as a mobile personal digital office assistant. Many of the PARCTab applications used context for adapting the user interface, configuring the system, or as a criteria for extracting and presenting data to the user. Some of the context elements used by the PARCTab are: location, the presence of other mobile machines, the presence of people, time, nearby stationary machines, and the state of the network file system. One of the simplest context aware features was to automatically invert the display if the user was left-handed versus right-handed. This was important because the buttons to interact with the device were along one side - thus the user wanted to place these buttons under their thumb.

Some of the applications that implemented context-awareness in the PARCTab system are:

- Presenting information about the location of the user. This information could be shown automatically or on request (e.g. when the user is at the library, information about the library is shown.)
- Helping the user to find the most convenient local resource (e.g. the nearest printer to the user.)

- Attaching a certain UNIX directory to a certain room, so when a user enters the room all of the files of this directory are shown.
- For controlling the environment (e. g. for controlling the lights and temperature of the current location.)

3.4.2 Cyberguide

One of the most popular types of context-aware applications are systems dealing with information about the surrounding environment. The Cyberguide [16] developed at Georgia Tech is a context-aware tour guide in which the user's current location, as well as a history of past location, are used to provide customized information or services. The long-term vision of the Cyberguide includes knowing the location of the user and his or her preferences in order to predict and answer question he or she might pose and to facilitate interaction with other people and the environment.

The Cyberguide is divided into four different components. The Map component displays a map or maps of the physical environments that the tourist is visiting. The information component provides information about the sights and area the tourist is visiting. The positioning component is responsible for positioning the user within the physical surroundings (i.e., computing the position of the user). Finally the communications component enables the user to send and receive information.

3.4.3 The DUMMBO Meeting Board and the Context Toolkit

As an effort for creating interoperable context-aware applications, Anind K. Dey from Georgia Tech proposed an architecture for supporting the creation of context-aware applications, known as the context toolkit [17]. The context toolkit was the first research focused on providing generic support for context acquisition.

The context toolkit is based on three main abstractions: Widgets, Aggregators, and Interpreters, as shown in Figure 6. A context widget is a "software component that provides applications with access to context information from their operating environment" [17]. The context widgets encapsulate information about a single piece of context (e.g. location) and provide a uniform interface to applications, hiding all the underlying context sensing mechanisms. Context widgets also provide reusable building blocks of presentation to be defined once and reused, combined, and/or tailored for use in many applications. Context widgets provide context information to applications via polling and subscribing methods.

The context aggregators are software entities responsible for aggregating context information coming from different context widgets. A context aggregator can be seen as a meta-widget acting as a gateway between applications and elementary context widgets.

Finally the interpreters are responsible of abstracting context information into higher level context information. For instance identity, location, and sound level could be use to deduce that a meeting has started. The context toolkit makes the distributed

nature of context transparent to applications. The context toolkit has been built in java and is downloadable from http://www.cs.cmu.edu/~anind/context.html.

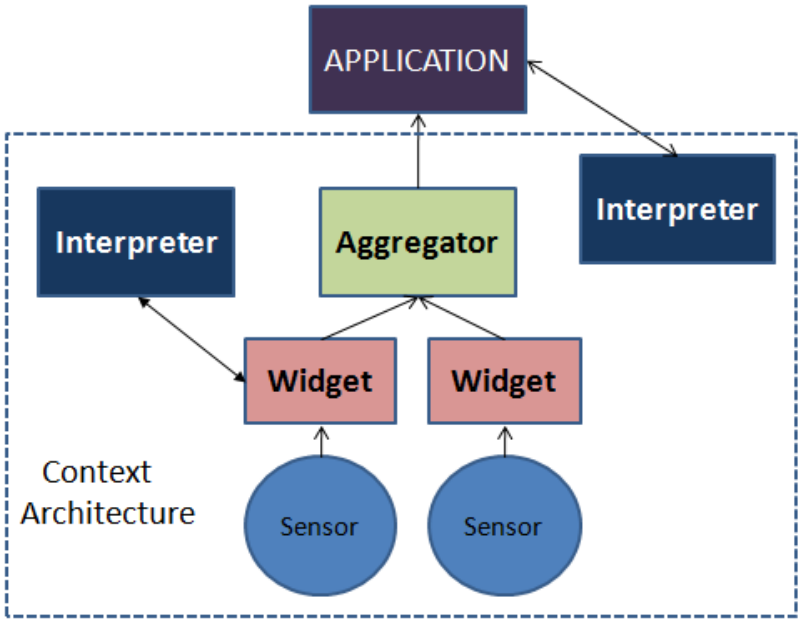


Figure 6. The context toolkit architecture

Based on the context toolkit several applications have been deployed, an example of one of these applications is the Dynamic Ubiquitous Mobile Meeting Board (DUMMBO) [17] for capturing and providing access to informal and spontaneous meetings. The DUMMBO system captures notes from the whiteboard and the audio of a discussion. This application is context-aware because the recording of the meeting is triggered when two or more persons are gathered close to the whiteboard. Also information about the time and identities of the persons are captured in order to ease the retrieval of the captured meetings.

3.5 Alternatives for distributing Context Information

Different approaches may be taken for distributing context information among applications in a network. Some of these approaches rely on a central server, while others are built on top of peer to peer architectures. Moreover, different protocols may be used for context information distribution, such as HTTP, SIP, or JXTA. HTTP has been a popular choice, however its main drawback is the increased latency and traffic load related to the overhead of TCP; also HTTP is a stateless protocol without native session support. In a peer to peer environment the JXTA protocol may be utilized for disseminating context information, having all the advantages of peer to peer architectures, especially avoiding having a single point of failure. However; the cost of using JXTA, is the high latency relative to its startup [52] (i.e. local cache management, rendevouz connection, etc.), which may be a critical issue for context aware applications. Another popular approach is SIP/SIMPLE, which is a general purpose

communication protocol supporting interactive session establishment. SIP relies on UDP and TCP, when using UDP it implements its own reliability mechanisms. The current standardization of SIMPLE considers a centralized architecture; however a peer to peer version of SIP is under standardization and can possibly be used in peer to peer environments.

3.6 Session Initiation Protocol (SIP)

SIP is an application-layer control protocol for creating, modifying, and terminating sessions with one or more participants. One of the strengths of SIP is that it provides user mobility, thus a SIP proxy can decide where to direct a SIP request at the time of the request – hence the target of the request can be in one or more locations and these locations can dynamically change. Hence as long as the target has updated its proxy as to its current location(s) it can be reached. The use of SIP URI enables the target to be identified based either upon its own "name" (user@domain) or via a specific device user@130.237.15.248. SIP allows the negotiation of any type of session between end points.

SIP is a text based protocol similar to HTTP and SMTP, hence SIP messages are human readable and the protocol is structured as a request-response protocol. SIP can utilize UDP, TCP, TLS, or SCTP as transport protocols and typically port 5060 is used for establishing a connection from a SIP user agent client to a SIP user agent server or SIP proxy server.

3.6.1 Elements of a SIP Network

There are three main elements in a SIP network [18]:

1) **User Agents** executed in the end devices in a SIP network. These devices originate SIP requests in order to send and receive data. A wide variety of devices may act as a user agent (SIP phones, SIP softphones, etc.). Typically the user agent is divided into two logical parts, a User Agent Client (UAC) responsible for initiating requests and a User Agent Server (UAS) which generates responses to received requests.

2) **Servers** are devices that assist user agents in session establishment and other functions such as redirection. RFC 2543 [20] defines three different types of SIP servers: proxy servers, redirect servers, and registrar servers. Proxy servers play a central role in the SIP network, as they route SIP messages and can implement complex decision logic [19]. A proxy receives SIP requests and forwards them to another location. This proxy can be stateful (and remain part of the SIP signaling) or stateless. The second type of server is the redirect server which simply redirects requests or indicates where a request should be retried. Finally

the third type of server is the Registrar server, which is responsible for updating the registering user agent's information in a location server.

3) **Location Servers** provide a database that contains location (current IP address) information of user agents.

3.6.2 SIP Methods and Responses

The main methods defined in SIP are presented in Table 2; many of these methods are defined in their own IETF RFCs.

Table 2 SIP Methods

Method	Description
INVITE	Session setup
ACK	Acknowledgement to INVITE
BYE	Session termination
CANCEL	Session cancellation
REGISTER	Registration of the user's URI
SUBSCRIBE	Request notification of an event
PUBLISH	Advertise and event
NOTIFY	Transport of subscribed event notification

The responses in SIP include a number using a scheme inherited from HTTP. Table 3 shows the SIP response code classes.

Table 3 SIP responses

Class	Description	Examples
1xx	Provisional or Informational	180 Ringing, 100 Trying
2xx	Success	200 OK, 202 Accepted
3xx	Redirection: (Request should be tried at another location)	301 Moved permanently
4xx	Client error	400 Bad Request, 404 Not found
5xx	Server Error	500 Server Internal error, 501 Not Implemented.
6xx	Global failure	600 Busy Everywhere

Figure 7 shows the normal flow of messages for setting up and terminating a SIP call between two UACs. We note that messages 1, 4, and 5 are essential for establishing the session. While messages 2 and 3 are purely informational and need not be sent or received. : The "Bye" message can be sent by either party to terminate the session. The other party responds with an OK to indicate that it has received this message.

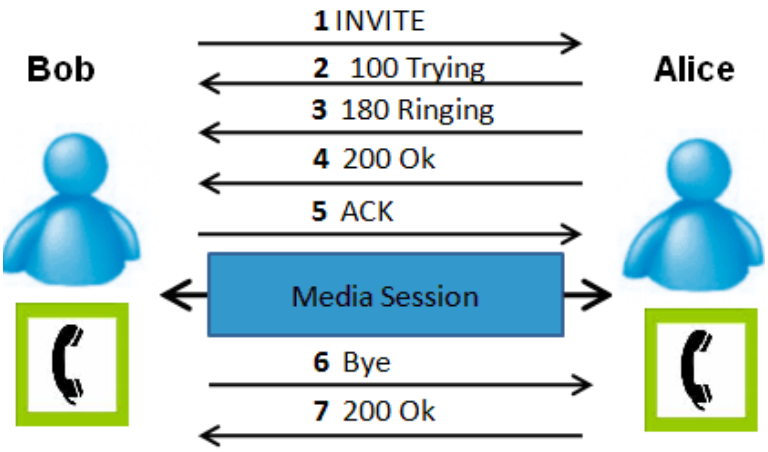


Figure 7. SIP message flow for establishing and terminating a call

An important feature of SIP is that it supports user, device, and session mobility, hence users may utilize a wide variety of devices (phones, fax, handhelds, etc.), in different locations, and can potentially switch between devices and change location while in a session. Additionally, these devices can be attached to different types of networks. A SIP connection is independent of the type of network or type of device the parties may use at a given time [18]. Note that SIP messages may contain information about where the device is, which ports are to be used, what CODECs are supported, etc. In general, the INVITE message carries the media communication parameters proposed by the caller, these can be modified by the callee in the OK response. Additionally, the SIP signaling path is independent of the path which the session uses and even asynchronous from the session (except that the SIP signaling to create the session must occur before the session starts).

3.6.3 Why use SIP for Distributing Context?

SIP is a good candidate for provisioning context for several reasons. One of the most important reasons is that SIP allows both synchronous and asynchronous events. The usage of SIP URIs allows for symbolic addressing; this decouples the logical address from the network address (i.e. IP address) – thus allowing devices to change their network address. However, the most important reason to consider SIP for distributing context is that it utilizes both a protocol and a communication infrastructure that will be widely deployed in future mobile devices. SIP has been adopted by several organizations (i.e. 3GPP), and is the foundation for session initiation and presence support in desktop, mobile, and server platforms. We believe that this wide adoption will make SIP events ubiquitously available [21].

3.6.4 Reliability in SIP

As stated before SIP can use many different transport protocols. Due to the fact that some of these (such as UDP) are unreliable transport protocols, SIP must provide its own reliability. SIP implements such reliability when using UDP by performing its own retransmissions. RFC 2543 [20] states:

*A SIP client using UDP SHOULD retransmit a BYE, CANCEL, OPTIONS, or REGISTER (also Notify) request with an exponential backoff, starting at a T1 second interval, doubling the interval for each packet, and capping off at a T2 second interval. This means that after the first packet is sent, the second is sent T1 seconds later, the next 2*T1 seconds after that, the next 4*T1 seconds after that, and so on, until the interval hits T2.*

The RFC also states that these retransmissions cease after sending eleven packets or when the sender receives a definite response (i.e. 200 OK). The default values for T1 and T2 are 500 ms and 4 s respectively; however, larger values may be used. In the SIP Express Router (SER) [29], an open source SIP proxy server, the values of these timers may be modified if needed (See chapter 4).

3.7 SIP SIMPLE

Currently the IETF SIMPLE (SIP for Instant Messaging and Presence Leveraging Extensions) working group [31], [23] is working on standardization of SIP Presence. Presence is defined as the willingness and ability of a user to communicate with other users on the network [32]. SIP SIMPLE was designed, as an interoperable and scalable protocol for presence. One important feature of this standardization is that the presence service may be used for other communication applications beyond short text messaging, such as applications for alerting users about stock trading events, travel itinerary changes, inventory events, supplies status, etc. [18]

The IETF model for presence presented in [26] is shown in Figure 8, this model defines a presentity (an abbreviation for "presence entity"), as a software entity that provides information to the presence service. The model also defines an entity known as a watcher. The watcher requests information from the presentity through the presence service. The presence service or presence server is in charge of distributing presence information concerning the presentity to the watchers, via a message, called a *Notification*. A user agent is usually divided into two logical software entities. The presence user agent is responsible of sending messages to the presence server for **publishing** information and the watcher user agent is in charge of initiating request for **fetching** presence information. In this model the presence protocol is any protocol capable of enabling the exchange of presence information in close to real time, between the different entities defined by the model.

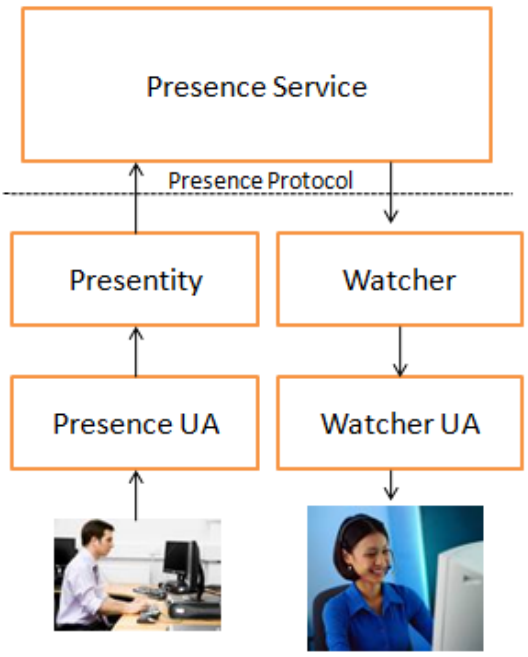


Figure 8. IETF Model for presence

In this IETF model three different types of watchers are defined:

Subscriber	Asks to be notified of changes to one or more presentities.
Fetcher	Makes a request for presence information.
Poller	A fetcher that makes repeated requests to update presence information.

3.7.1 SIP SIMPLE Messages

SIP SIMPLE provides the means for synchronous and asynchronous fetching of information. This information is distributed using three SIP methods:

- SUBSCRIBE
- PUBLISH
- NOTIFY

The synchronous fetch of information can be initiated by setting the expiration time of the SUBSCRIBE message to zero seconds [22]. In order to avoid confusions, in the remainder of this document the subscription with expiration equals to zero, will be called a *request* and its notification as *reply*. A subscription always returns the current state, resulting in a synchronous request, however if the expiration time is set to zero it will cancel an outstanding subscription.

SIP is a text based protocol and, hence the messages are readable to humans. SIP messages contain a message header and in some cases a message body. The message body for the NOTIFY and PUBLISH method utilizes the Presence Information Data Format (PIDF), using an eXtensible Markup Language (XML) schema. PIDF will be described in the next subsection. The syntax of the SIP-SIMPLE messages is shown in Table 5, Table 6, Table 7, and Table 8.

An important field in the PUBLISH message is the SIP-If-Match. This field is used in order to refresh, remove, or modify an existing PUBLISH message. The initial PUBLISH message does not contain this field. The value of this field is a random number created by the presence server that can be found in the OK message that acknowledges the initial PUBLISH message, as a SIP-Etag. In order to refresh a message (i.e. to extend its validity) the presentity should send a new PUBLISH message with the SIP-If-Match identifying the publication, with an expiration value greater than zero and no message body (because the contents of the body will be the same as in the initial publication). If the presentity wants to modify the contents of the publication, then it should send a new PUBLISH message, with the SIP-If-Match identifying the publication, an expiration value greater than zero, and the modified body. Finally the presentity can remove the message by sending a PUBLISH message with the SIP-If-Match, an expiration time of zero, and without a message body. Table 4, summarizes how this SIP-If-Match should be used in order to refresh, remove, or modify a PUBLISH message [25].

Table 4. SIP IF-Match for publishing messages

Message Type	Body	SIP-If-Match	Expiration Value
Initial	Yes	No	>0
Refresh	No	Yes	>0
Modify	Yes	Yes	>0
Remove	No	Yes	=0

Table 5. PUBLISH message

Message Header	Description
PUBLISH sip:Alice@192.168.100.153 SIP/2.0	The word PUBLISH indicates that this is a PUBLISH message. The URI of the user publishing the message is also stated. (The domain of the URI is the SIP Proxy server domain)
Via: SIP/2.0/UDP 192.168.100.53:52768;branch=z9hG4bK-d87543-3a35b0441f1d2b5c-1--d87543-;rport	This line indicates that UDP is used as transport protocol. The second variable is the IP address and port that shows the path the request has taken in the SIP network. The branch is a random number used to detect loops.
Max-Forwards: 70	This field is a limit in the number of hops that the message can traverse for arriving to its destination.
Contact: <sip:Alice@192.168.100.53:1885>	This field states the URI for direct communication between UAS.
To: "Alice"<sip:Alice@192.168.100.153>	The To and From fields are the same in the PUBLISH message, containing the server's address.
From: "Alice"<sip:Alice@192.168.100.153>;tag=a1390c7b	
Call-ID: MmYzMGY4OTJiZmZjMDAxODE0NmJhM2JiYTlhN2E2MDY.	The call-id is an identifier of the message. This call-id should be the same in the OK message acknowledging the PUBLISH message.
CSeq: 1 PUBLISH	This Command Sequence number is incremented for each subsequent request, it is used to distinguish a retransmission from a new request.
Expires: 3600	It states the validity of the PUBLISH message in seconds.
Content-Type: application/pidf+xml	Indicates the type of message body attached.
Event:presence	It indicates the category of the information published in the body.
Content-Length: 578	Indicates length of the message in bytes

Message body

```
<?xml version='1.0' encoding='UTF-8'?>
        <presence
           xmlns='urn:ietf:params:xml:ns:pidf'
           xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
           xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
           xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
           entity='sip:Alice@192.168.100.153'>
           <tuple
              id='t8a130d03'>
              <status>
                 <basic>
                    open
                    </basic>
                 </status>
              <note>
                 idle
                 </note>
              <contact
                 priority='0.8'>
                 Carlos
                 </contact>
              </tuple>
           </presence>
```

Table 6. **NOTIFY message**

Message Header	Description
NOTIFY sip:Bob@192.168.100.53 SIP/2.0	The word NOTIFY indicates that it is a notification.
Via: SIP/2.0/UDP 192.168.100.153;branch=z9hG4bKa957.04ab5e95.0	Same as in PUBLISH
To: "Bob"<sip:Bob@192.168.100.53>;tag=a1390c7b	Identifies the recipient of the Notification
From: "Alice"<sip:Alice@192.168.100.153>;tag=a6a1c5f60faecf035a1ae5b6e96	Identifies the sender of the Notification.
CSeq: 1 NOTIFY	Same as in PUBLISH.
Call-ID: MmYzMGY4OTJiZmZjMDAxODE0NmJhM2JiYTlhN2E2MDY	Same as in PUBLISH
Content-Length: 542	Same as in PUBLISH
Event: presence	Same as in PUBLISH
Content-Type: application/pidf+xml;charset="UTF-8"	Same as in PUBLISH
Subscription-State: active;expires=600	Indicates the state of the Notification and the time left for expiration.

Message Body

```
<?xml version='1.0' encoding='UTF-8'?>
<presence
   xmlns='urn:ietf:params:xml:ns:pidf'
   xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
   xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
   xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
   entity='sip:Alice@192.168.100.153'>
   <tuple
      id='t8a130d03'>
      <status>
         <basic>
            open
            </basic>
         </status>
      <note>
         idle
         </note>
      <contact
         priority='0.8'>
         Carlos
         </contact>
      </tuple>
   </presence>
```

Table 7. SUBSCRIBE message

Message Header	Description
SUBSCRIBE sip:Bob@192.168.100.153 SIP/2.0	The Word SUBSCRIBE identifies the message as a subscription.
Via: SIP/2.0/UDP 192.168.100.53:1886;branch=z9hG4bK-d87543-3a35b0441f1d2b5c-1--d87543-;rport	Same as in PUBLISH
Max-Forwards: 70	Same as in PUBLISH
To: "Alice"<sip:Alice@192.168.100.153>	It indicates to whom the user is subscribing
From: "Bob"<sip:Bob@192.168.100.53>;tag=a1390c7b	It indicates the sender's URI.
Call-ID: MmYzMGY4OTJiZmZjMDAxODE0NmJhM2JiYTlhN2E2MDY.	Same as in PUBLISH
CSeq: 1 SUBSCRIBE	Same as in PUBLISH
Expires: 3600	It indicates the time in which the subscription will be valid. No notifications will be sent after the expiration of the subscription**. A value of 0 introduces a synchronous fetching of information**.
Event: presence	It indicates the category of information that is of interest for the watcher.
Content-Length: 0	This message doesn't contain a body

Table 8. 200 OK message

Message Header	Description
SIP/2.0 200 OK	This field identifies the message as an OK
Via: SIP/2.0/UDP 192.168.100.153;branch=z7.04ab5e95.0	Same as in PUBLISH.
To: "Alice"<sip:Alice@192.168.100.153>;tag=a6a1c5f60faecf035a1ae5b6e96	Identifies the recipient of the OK
From: 192.168.100.153	Identifies the sender of the message
Call-ID: MmYzMGY4OTJiZmZjMDAxODE0NmJhM2JiYTlhN2E	It carries the Call-ID of the message that is acknowledging.
CSeq: 1 NOTIFY	It carries the CSeq of the message Acknowledging.
SIP-Etag: 0xb58341kjl1345	It identifies the PUBLISH message for refresh, modification of removal. (only for OKs after PUBLISH) This SIP-Etag is used as SIP-If-Match as described in Table 4.
Contact: 192.168.100.153	Same as in PUBLISH
Content-Length: 0	This message doesn't contain a body.

The message flow for a synchronous fetch is shown in Figure 9. As it can be seen in the figure the request/response sequence results in four messages being exchanged between the watcher and the SIP presence server (assuming that there are no messages lost) Note that a SUBSCRIBE with expiration equal to zero was used for the request and a NOTIFY for the reply.

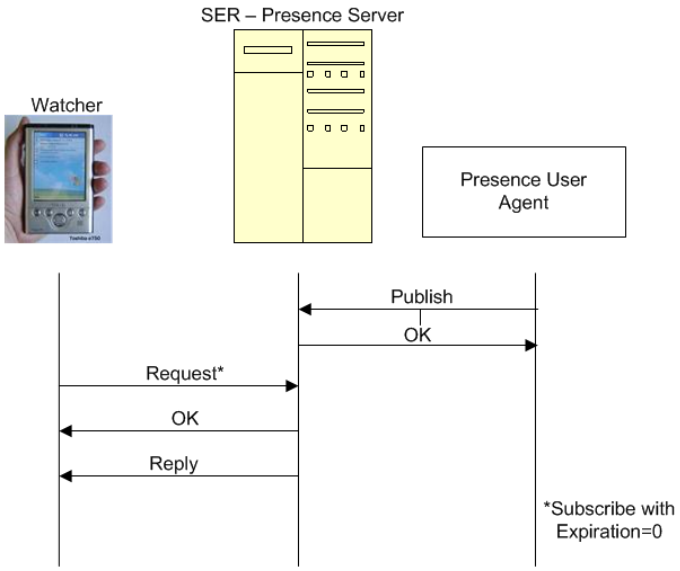


Figure 9. Message flow for synchronous exchange of information

The message flow using a Subscribe/Notification scheme is shown in Figure 10. When a user subscribes to an event, he or she immediately receives a notification with the status and presence information of the presentity. When the presentity modifies the published information or the publication expires, then the watcher is notified immediately.

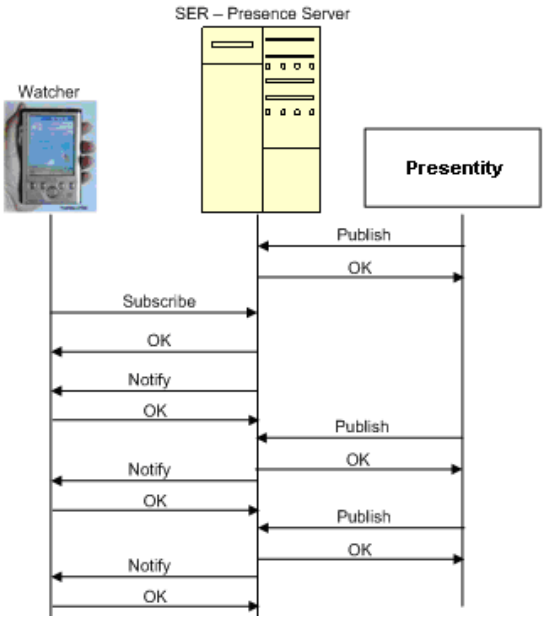


Figure 10. Message flow for asynchronous exchange of information

3.8 Context Modeling

Context data, as with other types of information, requires modeling mechanisms to guarantee efficient and interoperable functionality. The main objective of context modeling "is to develop uniform models, representation and query languages as well as reasoning algorithms that facilitate context sharing and interoperability of applications." [46] A context model should provide an unambiguous definition of the context artifacts, their representations, semantics, and usage. A context model should also take into account the general characteristics of the context data, such as its temporal nature, ambiguity, impreciseness, and incompleteness [11]. Furthermore, context models have to address the requirements of pervasive computing, such as heterogeneity of context sources, distribution, and mobility.

In the following subsections several context models will be described and summarized, following the taxonomy of Thomas Strang [46]. Additionally, the context model designed for the MUSIC project is summarized.

Key-value models

Key-value pairs are the simplest data structure used for modeling context. The context data is stored as a key-value pair; the key will refer to the environment variable and the value will hold the actual context data. Key-value models are easy to manage, but lack capabilities for sophisticated structuring. An example of a key-value pair is (status, busy) where status is the key and busy is the value.

Graphical models

Graphical models for context data are often based on the Unified Modeling Language (UML) due to its generic structure. Various research projects have proposed different approaches for modeling context through graphical models. An example is ContextUML [48] that provides a model-driven approach to the development of context-aware web services. The syntax of ContextUML includes a metamodel and a notation. The metamodel defines the abstract syntax of the language and the notation defines the concrete format used to represent the language.

Markup Scheme models

Markup based models use a hierarchical structure consisting of markup tags with attributes and content. Typical markup scheme models are profiles based upon a serialization of a derivative of the Standard Generic Markup Language (SGML), such as XML. An example of these profiles is: the *Composite Capabilities/Preference Profile* (CC/PP) [47] used for defining the capabilities and preferences of user agents. CC/PP is a vocabulary extension of the Resource Description Framework (RDF).

Object Oriented models

Context data modeling using object oriented methods offers the possibility to use the full benefit of object orientation, such as encapsulation, inheritance, and

reusability. "The detail of context processing is encapsulated on object level and hence hidden to other components. Access to context information is provided through specific interfaces" [46].

Logic based models

In logic based models, facts or concluding expression may be derived from other expressions of facts through logic conditions. Context information is modeled as facts and a logic-based system is used to manage the terms allowing: adding, updating, or removing new facts.

Ontology based models

Ontologies are an instrument for specifying concepts and interrelations. An ontology is described by concepts, relations, and rules for combining concepts and relations. "Context models based on ontologies provide a vocabulary for representing and sharing context knowledge in a pervasive computing domain, including machine-interpretable definitions of basic concepts in the domain and relations among them" [11]. Ontologies enable context data to be described semantically and independent of the underlying operating system, programming system, or middleware. Due to all the strengths of context models based on ontologies, several context-aware frameworks make use of them. An example is the CoBra [49] system that provides a set of ontological concepts to characterize entities within their contexts.

The MUSIC Context Model

The context model defined for MUSIC [11] is divided into three different layers: the conceptual layer, the exchange layer, and the functional layer. The conceptual layer is defined for developers to enable the definition of context elements, scopes, and representations based on standard specification languages, such as the Ontology Web Language (OWL). The exchange layer is used for interoperability between devices and for distribution of context information. In this layer, the context data can be expressed in eXtendible Markup Language (XML), Java Object Script Notation (JSON), and Comma-Separated Values (CSV). Finally the functional layer is the actual implementation of the context model representation based in object-oriented modeling.

The conceptual layer includes an ontology described in OWL and a context meta-model specified in UML. Ontologies are included for enabling interoperability, for defining the internal structure of context data, and for modeling a wide range of relationships between context elements, enabling flexible context reasoning.

The distribution of context information is done at the exchange layer in which an ontology server may be contacted for mapping between the context scopes received and corresponding concepts in the ontology. This enables the correct interpretation of the information. The XML representation was chosen because it is a widely adopted standard; furthermore several libraries and tools exist for parsing and mapping XML to object models in different programming languages (C, JAVA, C#). Another advantage

is its extensibility. The exchange layer converts the information from XML, JSON, or CSV to appropriate data-structures at the functional layer.

Finally, the functional layer contains a set of data structures for storing the context information. The internal structure of the context elements is contained in the ontology, so the data structures can easily be filled with the information represented at the exchange layer.

3.9 Presence Information Data Format (PIDF)

The Presence Information Data Format (PIDF) is defined in IETF's RFC 3863 [27] as a common presence data format for Common Profiles for Presence (CPP) compliant presence protocols [35]. The main objective of PIDF is to achieve interoperability between different instant messaging and presence protocols meeting the "Instant Messaging/Presence Protocol Requirements", described in RFC 2778 [26]. PIDF encodes the presence information using XML

According to RFC 3863, presence information consists of one or more PRESENCE TUPLES. A presence tuple consists of a mandatory status element and other optional extension elements, such as a contact element, note element, or timestamp element. The status in the presence tuple has at least the values OPEN and CLOSED. RFC 3863 defines these two values in the context of instant messaging. The status of OPEN means that the associated contact element is an instant mailbox ready to accept an instant message. In the other hand, the status CLOSED means that the contact is unable to accept an instant message. In a wider context these two values express that the user is available for (near) real-time interactive exchange of information. The message body presented in the PUBLISH and NOTIFY messages in follows the PIDF format. A PIDF object is a well formed XML document. The first element of the XML document is an encoding declaration of the form:

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

The root of a PIDF+xml object is a <presence> element that contains any number of <tuple> elements. The <presence> element must have an entity attribute that is the URI of the presentity publishing this presence document (e.g. entity="pres:Alice@example.com">). The presence element must also contain a namespace declaration indicating the namespace on which this presence document is based (e.g. 'urn:ietf:params:xml:ns:pidf').

The <tuple> element must contain an 'id' attribute which is used to distinguish the tuple. The <contact> element is optional and contains the URI of the contact address. The <status> element in the tuple contains one <basic> element that contains the "open" or "closed" values.

The <note> element contains a string value, which is usually used for a human readable comment. This note element may be child of a presence element or a tuple element. In the first case the comment is about the presentity; while in the second case the comment concerns a particular tuple (the parent tuple of the note element). Finally the optional <timestamp> element contains a string indicating the date and time of the status change of this tuple.

An example of a PIDF document is shown in Figure 11:

```
<?xml version='1.0' encoding='UTF-8'?>
<presence
   xmlns='urn:ietf:params:xml:ns:pidf'
   xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
   xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
   xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
   entity='sip:Alice@192.168.100.153'>
   <tuple
      id='t8a130d03'>
      <status>
         <basic>
            open
            </basic>
         </status>
      <note>
         idle
         </note>
      <contact
         priority='0.8'>
         Carlos@appearnetworks.com
         </contact>
      </tuple>
   </presence>
```

Figure 11. PIDF example

3.10 Rich Presence Information Data Format (RPID)

Presence information is not limited to availability, for many applications the Presence Information Data Format (PIDF) as defined in RFC 3863 [27] is not sufficient to represent presence information accurately. IETF's RFC 4480 [33]defines extensions to the PIDF document format for conveying richer presence information. The Rich Presence Information Data Format (RPID) defines extensions providing features common in many presence systems. It also defines elements that can be derived automatically from existing applications such as the calendar or from other sensors that can provide information about the user's current physical environment.

RFC 4479 [34] defines a model for representing presence information based on the status of a service, a device, or a person. According to this model, a (communication) service is a system for providing interaction between users and provides certain modalities or content. A device is a "physical component that a user interacts with in order to make or receive communications". Finally a person is the end user, which is characterized (in terms of presence) by states that impact this person's ability and willingness to communicate. RPID was defined for representing more accurately this model for presence data. In RPID documents the presence information from services is encoded using the <tuple> element defined in PIDF; devices and persons are represented in extended XML elements: <device> and <person>.

RPID also defines additional presence attributes beyond the <basic> status element. These attributes are XML elements that extend the <tuple>, <device>, and <person> elements. It is important to note that RPID is backward compatible with PIDF. RPID seeks to derive presence information from different information sources, such as personal calendars, the status of communication devices, typing activity, and physical presence detectors. The additional attributes that extend the <tuple>, <device>, and <person> elements are defined in [33] as:

Activities	What the person is doing.
Class	An Identifier that groups similar persons, devices or services.
deviceID	A device identifier in a tuple references a <device> element, indicating that this device contributes to the service described by the tuple.
Mood	It indicates the mood of the person.
Place-is	Reports the properties of the place the presentity is currently at.
Place-type	It reports the type of place the person is located
Privacy	It states whether the communication service is likely to be observable by other parties.
Relationship	This element is used when a service is likely to reach a user besides the person associated with the presentity and states how this user relates to the person.
Service-class	This element describes how the service will be delivered.
Sphere	The <sphere> element characterizes the overall current role of the presentity.
Status-icon	This element depicts the current status of the person or service.

Time-offset	Used for quantifying the time zone the person is in. This offset is expressed as the number of minutes away from UTC.
User-input	This element records the state of the service or device based on human user input.

Table 9 shows (marked with an 'x') which elements may have the from/until attributes for expressing a period of time, which elements may contain a note containing additional information, and which elements may be child of a <tuple>, <device>, or <person> element.

Table 9. Extended Attributes of RPID

Element	From/Until	Note	<person>	<tuple>	<device>
activities	X	X	X		
Class			X	X	
deviceID				X	X
Mood	X	X	X		
place-is	X	X	X		
place-type	X	X	X		
privacy	X	X	X	X	
relationship		X		X	
service-class		X		X	
sphere	X		X		
status-icon	X		X	X	
time-offset	X		X		
user-input			X	X	X

As in PIDF the root element of a RPID document is the <presence> element having an entity attribute containing the URL of the presentity publishing the information (i.e. someone@example.com). This element also specifies the namespace declarations in which the document is based. An example of a RPID document is the following:

```
<presence xmlns="urn:ietf:params:xml:ns:pidf"
  xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
  xmlns:lt="urn:ietf:params:xml:ns:location-type"
  xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
  entity="pres:someone@example.com">
```

The following example document contains three different tuples (representing three different services). The first one is a SIP contact that can be contacted at someone@mobile.example.net and is ready to accept communication. This tuple contains notes regarding presence information:

```
<tuple id="bs35r9">
      <status>
        <basic>open</basic>
      </status>
      <dm:deviceID>urn:device:0003ba4811e3</dm:deviceID>
      <rpid:relationship><rpid:self/></rpid:relationship>
      <rpid:service-class><rpid:electronic/></rpid:service-class>
      <contact priority="0.8">im:someone@mobile.example.net</contact>
      <note xml:lang="en">Don't Disturb Please!</note>
      <note xml:lang="fr">Ne derangez pas, s'il vous plait</note>
      <timestamp>2005-10-27T16:49:29Z</timestamp>
    </tuple>
```

The second tuple shows that the presentity has an assistant (secretary@example.com) who happens to be available for communications via email.

```
    <tuple id="ty4658">
      <status>
        <basic>open</basic>
      </status>
      <rpid:relationship><rpid:assistant/></rpid:relationship>
      <contact priority="1.0">mailto:secretary@example.com</contact>
    </tuple>
```

The third and last tuple shows an email box service:

```
    <tuple id="eg92n8">
      <status>
        <basic>open</basic>
      </status>
      <dm:deviceID>urn:device:0003ba4811e3</dm:deviceID>
      <rpid:class>email</rpid:class>
      <rpid:service-class><rpid:electronic/></rpid:service-class>
      <rpid:status-icon>http://example.com/mail.png</rpid:status-icon>
      <contact priority="1.0">mailto:someone@example.com</contact>
    </tuple>
```

A RPID document may also have a note for additional information regarding to the three tuples, such as:

```
<note>I'll be in Tokyo next week</note>
```

The second part of the RPID document contains information about the devices which can be used to support the services, represented in the tuples. The device represented in the document supports both SIP and email service. Note that the deviceID corresponds to the one used in the <tuple> elements.

```
<dm:device id="pc147">
      <rpid:user-input idle-threshold="600"
        last-input="2004-10-21T13:20:00-05:00">idle</rpid:user-input>
      <dm:deviceID>urn:device:0003ba4811e3</dm:deviceID>
      <dm:note>PC</dm:note>
    </dm:device>
```

Finally the third part of the RPID document contains information about the person, such as mood, location, environment, etc. We can see that the <activities> element indicates the range of time in which the presentity will be in the activities using the from/until attributes.

```
<dm:person id="p1">
      <rpid:activities from="2005-05-30T12:00:00+05:00"
        until="2005-05-30T17:00:00+05:00">
        <rpid:note>Far away</rpid:note>
        <rpid:away/>
      </rpid:activities>
      <rpid:class>calendar</rpid:class>
      <rpid:mood>
        <rpid:angry/>
        <rpid:other>brooding</rpid:other>
      </rpid:mood>
      <rpid:place-is>
        <rpid:audio>
          <rpid:noisy/>
        </rpid:audio>
      </rpid:place-is>
      <rpid:place-type><lt:residence/></rpid:place-type>
      <rpid:privacy><rpid:unknown/></rpid:privacy>
      <rpid:sphere>bowling league</rpid:sphere>
      <rpid:status-icon>http://example.com/play.gif</rpid:status-icon>
      <rpid:time-offset>-240</rpid:time-offset>
      <dm:note>Scoring 120</dm:note>
      <dm:timestamp>2005-05-30T16:09:44+05:00</dm:timestamp>
    </dm:person>
  </presence>
```

3.11 SIP Express Router

The SIP Express Router (SER) [29] is a SIP server licensed under the GNU General Public license. This SIP server has high performance and is configurable for acting as a SIP registrar, proxy, or redirect server. Recently, a presence module supporting the presence event package has been developed by IPTEL. The implementation for the proposed system in this thesis uses this SIP proxy server with this presence module.

The “main strength of SER is its performance, SER runs well even under heavy load caused by large subscriber populations” [29].

SER was initially developed by Fraunhofer Gesellschaft, a German research institute with 56 institutes spread throughout Germany. Part of the team that initially developed the SER created a new company, iptel.org in 2004. In parallel, other developers started a new open source project named OpenSER [45]. SER and OpenSER have followed different development paths. For our implementation we are using the SER development, rather than OpenSER because of the previous work done by others [30] within context distribution using SIP [29] [45].

3.12 Related Work in Context Distribution

Within the context-awareness research community several researchers have identified the need for sharing context among different system components. Several different solutions to this problem have been proposed; this section will review the related work that is relevant to this thesis.

3.12.1 A Location-Aware Content Delivery Service

Alisa Devlic and Ivana Podnar [36] proposed a location-aware content delivery service enabling the delivery of personalized content to users based on their location and preferences. This service uses Publish/Subscribe mechanisms to transport content between publishers and subscribers. In this architecture the users define personal profiles specifying their subscriptions and preferences. These subscriptions are location-aware; each subscription contains a list of locations for which it is valid. For example in the delivery of weather information, the user may only be interested in information related to his or her current location or other specific location of interest.

The architecture of the system uses a presence component which tracks the user device employed in order to support personal mobility. A location management component is used to track the location of the user for updating/removing subscriptions from the system. A profile handler stores the subscriptions, device capabilities, and user preferences.

Their proposed content delivery service can also be used for delivering context information. During their work, they identified the need for an asynchronous mode of retrieving information. Moreover, the requirement to support user mobility was also identified. Although the design and the implementation of their system did not consider the SIP event communication framework, we have determined that most of the features and requirements for delivering content may be supported by SIP and related protocols. Their work provided the motivation for this thesis project and its goals.

3.12.2 MIDAS

The Middleware Platform for Developing and Deploying Advanced Mobile Services (MIDAS) [38] is a European research project concerning 3G and beyond, the main goal of the project is to implement and develop a platform for rapid development of mobile applications.

The context engine for this project is described in [37]. It provides mechanisms to retrieve, synthesize, and distribute context information in a mobile distributed environment. Context information distribution in MIDAS utilizes different sources of context information and makes this information available to applications on remote nodes through the replication of the information contained in their context database.

The context engine recognizes two different forms of context retrieval. The first form is based on context queries that are used for stateless retrieval of context information. The second form is based on context triggers that are queries for stateful retrieval of context information; thus an action is triggered when a specific context occurs.

The context information retrieval can be as simple as an access to the context database or it can be very complex, for example requiring context synthesis through operators. Operators are functions or programs that take as input certain context information, and produce as output higher level context information.

The MIDAS project has identified a need of distributing context among distributed and mobile applications using both stateless and stateful context retrieval mechanisms. The MIDAS project implements context synthesis through context operators for providing high-level and meaningful information to context-aware applications, this idea will be extended and used in MUSIC. As MUSIC, is an integrated project, it should consider the ideas, experiences, and results from other European projects working within the same research area, such as MIDAS.

3.12.3 Context Sharing in SIP-based Telephony Systems

When humans communicate, both the perceived information and the context of those participating in this communication play an important role; however, distant interpersonal communication does not provide means to know the situation or context of the communicating partners.

Enhancements of the SIP protocol for IP telephony systems [39] has been proposed by Görtz et al. in order to support context sharing between communication participants to enhance the communication process. The approach proposed for sharing context information may occur directly between two SIP User Agents or between a client and a server that relays the information. The SIP event framework provides an asynchronous communication mechanism.

One of the advantages identified for using a centralized approach for sharing context information is the capability of aggregating context information generated by many different devices or sensors, as shown in Figure 12.

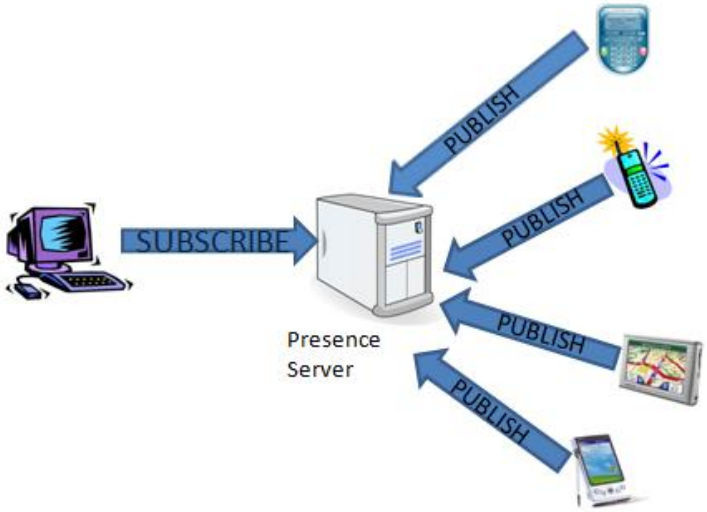


Figure 12. Aggregating context information through a presence server

The proposed enhancements to SIP for supporting direct queries among SIP peers are based upon use of the OPTIONS method. Today this method is used for querying another SIP entity about its capabilities; however it may also be suited for querying a peer for context information. In the proposed solution the OPTIONS method is extended with a *context* header, this additional header is ignored by unmodified SIP entities - but parsed by a modified SIP user client that will respond with its context information in the OK response message. In this prototype the context information is a simple ASCII string. It would be better to encode the context information in an XML schema, instead of using a simple ASCII string for allowing easy extensibility

In the previous enhanced IP telephony system, the use of the SIP event mechanisms were proposed for supporting asynchronous means of context sharing. The SUBSCRIBE, PUBLISH, and NOTIFY methods were used for distributing context information. This context distribution method is proposed when the system is used in conjunction with a context/presence server.

In this thesis project SIP has also been selected as the protocol to be used to share context information among applications. Just as was the case for Görtz et al. we seek to support both synchronous and asynchronous means for sharing context information, hence we will build upon their ideas.

3.12.4 A SIP infrastructure for Adaptive and Context-Aware Wireless Services

As part of a Person-Centric Context Aware System [14] Wei Li proposed a service-oriented context infrastructure [40] for exchanging context among services. The Session Initiation Protocol and related protocols were adopted for transferring context information. The SIP protocol has been identified as an appropriate protocol for transporting context, because it supports all the different requirements identified for disseminating context information. The SIP presence framework was adopted because it “provides a light support for short-term communication where a session only exists

within a few rounds of messages exchanges, supporting the timely and intermittent nature of context exchange" [14]. Furthermore, the use of SIP URI enables naming and addressing mechanisms for identifying and locating context entities. Another advantage of using a SIP-based infrastructure for distributing context information is that it can be tied seamlessly with any other SIP-based network.

His context provisioning infrastructure extends the SIP Presence framework for delivering context information. The architecture of the proposed system is based on a Presence Agent and a Watcher. The first is responsible for handling the subscriptions to presence and context information events and to notify the Watcher of any context changes. Notifications are encoded in a XML format and sent to the registered Watchers. A Watcher is defined as a consumer of context that registers with the Presence Agent with an indicating interest in a specific context, then waits for notifications from the Presence Agent.

In his licentiate thesis [14] Wei Li has identified several of the requirements for distributing context information and how SIP may support these requirements. His context data communication is based on asynchronous mechanisms provided by the SIP presence framework.

3.12.5 A Presence Server for Context-Aware Applications

Mohammed Zarifi's master thesis [30] describes an adapted SIP presence server, which acts as a context server in order to create a context-aware (middleware) infrastructure for different types of context-aware applications. His design and implementation of a context server is based on the SIP Express router (SER) [29], its presence module and its MySQL database. The resulting context-server was evaluated in terms of service time (How long does it take for the server in each scenario to respond to each of the different messages) with 60 watchers subscribed to an event. The proposed context server: (i) obtains the updated context information, (ii) reads, processes, and stores this information in the local database, and (iii) notifies interested watchers about context information.

The context delivery mechanisms are based on SIP-SIMPLE for supporting asynchronous distribution of information. The context information is distributed using the Presence Information Data Format.

However, the initial presence module of the SER only supports the presence event package standardized by the IETF. In order to support different types of events (e.g. location) he extended the source code of the SER. His context server implementation extends the standard PIDF tags, by creating new tags inside the <status> element. These new elements were designed in order to describe a location. The location element includes a description, room, floor, and coordinates element. Inside the coordinates elements for latitude, longitude, and height were defined. The new schema for the location event is shown in Figure 13:

```
<location>
    <description>Appear</description>
    <room>Meeting_room</room>
    <floor>4</floor>
    <coordinates>
       <latitude>59 23'</latitude>
       <longtitude> 18° 00'</longtitude>
       <height></height>
    </coordinates>
</location>
```

Figure 13. A PIDF schema for the location event

In this schema the description, room, and floor elements should have at least one value, the remaining elements can be empty. For further information and details of this implementation the reader should refer to [30].

4. Context Distribution using SIP

This chapter describes the approach we used for distributing context information. Also, the different components of the testbed we used for evaluating SIP/SIMPLE as a protocol for distributing context information are described.

As Li Wei observed, the development of context-aware applications in distributed environments has been limited because of the difficulties faced in distributing context information between applications and peers. One of the main difficulties is that the transfer of context information needs to take place over unreliable and dynamically changing networks. Moreover, context information may be produced in different devices connected to different networks. This chapter presents a proposed context distribution component for distributing context information via a SIP infrastructure. This component uses SIP to distribute context information, both synchronously and asynchronously. As shown earlier in Figure 5 on page 10, different devices may fetch information synchronously or asynchronously via a SIP proxy server. In synchronous mode a user can fetch context information once or poll periodically to retrieve the most recent values of the desired information.

This distribution component consists of three subcomponents: a presence/context server, a watcher, and a presentity/context entity (contextity); as shown in Figure 14. The term context-server denotes a SIP presence server that can support context information in the body of the PUBLISH and NOTIFY messages. On the other hand the term **contextity** defines the entity that publishes not only presence information, but also context information. Figure 14 shows different kinds of queries and responses that can exist within the system. It is important to note that the watcher and the contextity are logical entities that could reside in the same user agent.

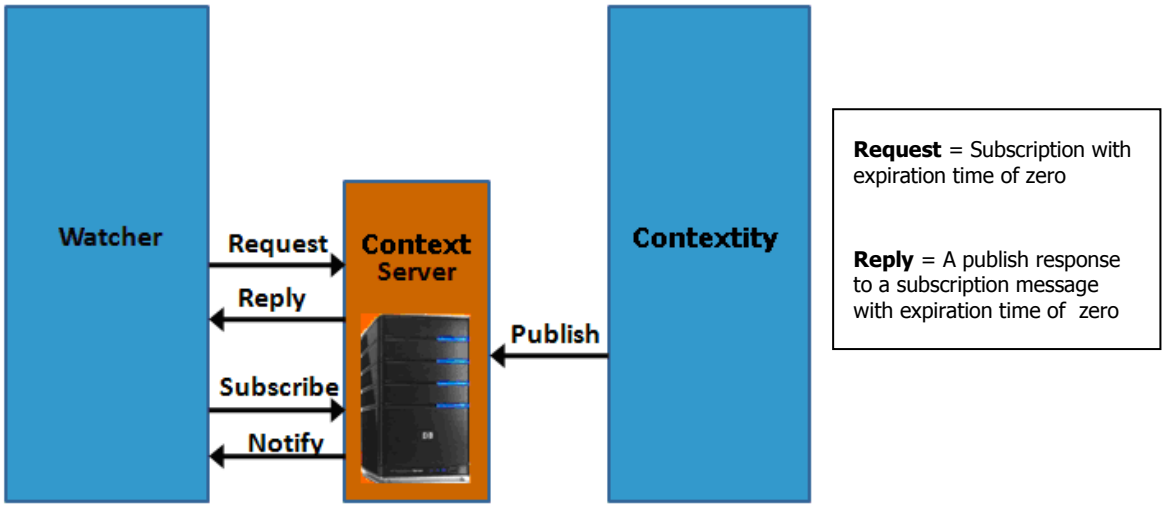


Figure 14. Elements of the Context Distribution component

4.1 Context Server

The context distribution component uses the SIP Express Router (SER) [29] for the context/presence server. SER complies with the SIP RFC 3261 specifications [41]. SER version 0.10.99 was used for the context distribution component. Details of installing and running this software can be found in Appendix C.

SER is implemented around a processing core that receives messages and enables the basic functionality needed for handling messages, this processing core is very small, fast, and stable [42]. Other functions are provided by SER modules. Another advantage of SER is its flexibility. A configuration file controls which modules shall be loaded and defines the module's behavior (In terms of configuration parameters).

An example of a SER configuration file for supporting presence can be found in appendix A. The SER configuration file can be seen as a script that is executed for every SIP message received. The structure of the SER configuration file (ser.cfg) is shown in Table 10.

Table 10. SER Configuration file structure

Section	Description
Global definitions	Configuration of IP address and port for listening, debug level, etc.
Modules	List of external libraries that are needed to expose functionality not provided by the core.
Module configuration	Configuration of parameters for the different modules.
Main route block	The entry point of processing SIP messages for controlling how each received message is handled.
Secondary route block	This routing block can be called from the main route block.
Reply route block	Utilized for handling replies to SIP messages
Failure route block	Route block used to handle failure conditions (e.g. busy)

Many parameters within SER are configurable through the configuration file in the module configuration block. The syntaxes for setting the values for the different parameter follow the following format:

```
modparam("module", "parameter", value)
```

An example for modifying the max_publish_expiration parameter of the presence module to have a value of 120 seconds is:

```
modparam("pa", "max_publish_expiration", 120)
```

Table 11 summarizes some important parameters related to retransmission timers and presence module parameters:

Table 11. Some SER configurable parameters

Parameter	Value	Module	Description
retr_timer1	Millliseconds	Tm	Initial retransmission value. The T1 timer defined in RFC 2543 [20]. Default value 500 ms.
retr_timer2	Milliseconds	Tm	Maximum retransmission period. The T2 timer defined in RFC 2543 [20]. Default value 4000 ms.
fr_timer	Milliseconds	Tm	Timer which goes off if no final reply for a request arrives. Default value 30 seconds.
default_expires	Seconds	pa	Default expiration time for SUBSCRIBE and PUBLISH message when the client does not supply one. Default value 3600 s.
max_subscription_expiration	Seconds	Pa	Maximal subscription expiration value. Default value 3600 s.
max_publish_expiration	Seconds	Pa	Maximal expriation for PUBLISH messages. Default value 3600 s.
use_db	Integer	Pa	If set to 1, PA module stores all subscription data into the database
db_url	String	Pa	Database connection URL.
timer_interval	Seconds	Pa	Interval when the timer runs and clears expired watchers and send NOTIFY for changed presentities.
maxbuffer	Kbytes	Core Options	Maximum Receive buffer size, default size 256 Kbytes.

4.1.1 SER Presence Module (pa)

This SER presence module [43] implements a presence server, which is responsible for receiving SUBSCRIBE requests and sending NOTIFY when the presence status of a user changes. The SER presence server works in conjunction with a MySql database which stores information about publications and subscriptions. This module also receives PUBLISH requests for publishing presence status information. The presence server supports PIDF, CPIM-PIDF (last version differs from PIDF only in namespace and MIME type name), and PIDF extensions (e.g. RPID) as document formats.

In order to distribute context information a RPID [33] formatted document was chosen because it is a standardized data format and is supported by SIP proxy servers, specifically the SER. Another reason for using RPID is that context information at the exchange layer of the MUSIC context model is represented using XML. An example of an RPID document carrying information regarding location (place type, coordinates, floor, sphere, etc) is shown in Figure 15.

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:Publisher1@192.168.100.234">
<tuple id="0xb581e834x4046d8a5x47bc3b1b">
<status><basic>open</basic></status>
<contact priority="0.00"></contact>
</tuple>
<dm:person xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid" id="p1f5e1369">
<rpid:place-type><appear/>
    <rpid:note>longitude 18° 00'</rpid:note>
    <rpid:note>latitude 59 23'</rpid:note>
</rpid:place-type>
<rpid:sphere>meeting room
    <rpid:note>4th floor</rpid:note>
</rpid:sphere>
</dm:person>
</presence>
```

Figure 15. RPID carrying location information

To support an XML schema different from that defined in RPID or PIDF, the context distribution component needs to be modified. To create new tags a new XML schema has to be defined in SER's source code, specifically in the "pidf.c" file, as proposed by Mohammad Zarifi in his master thesis [30]. This file is used by the system to parse the tags specified in the PIDF and RPID standards [27][33].

4.2 Context entity and Watcher

The contextity and watcher entities were implemented in a load generator in order to perform scalability and latency tests of the proposed context server. This load generator has been implemented in Java version 1.6.0_02 and the source code may be found in Appendix B. The main objective of this implementation is to simulate multiple watchers and contextities in order to load the server with SUBSCRIBE and PUBLISH message and to test its performance.

The load generator implementation is based on non-blocking sockets of the java New I/O (NIO) API. For each user (watcher or presentity) simulated in the load generator a non-blocking socket is created. This kind of socket allows communications between applications without blocking the processes using the sockets. An alternative implementation could be a multithreaded load generator; however, this caused concurrency and scheduling conflicts, affecting the performance of the system. The non-blocking technology is based on a selector that monitors the recorded socket channels and serializes the requests that the application has to satisfy.

The implementation is divided into three different threads simulating watchers and contextities:

1) The **main thread** responsible for creating sockets for each user (watcher or presentity) and registering them with their selectors. After initialization this the thread goes into an infinite loop waiting for events on the sockets. When information arrives to a presentity or watcher socket a callback function is triggered. These callback functions are responsible for handling the incoming messages to the watchers and presentities. In the case of watchers, when a NOTIFY message arrives, a corresponding OK is generated. On the other hand, when an OK message arrives to the presentity, acknowledging a PUBLISH, it is parsed in order to extract the SIP-ETag for further PUBLISH messages.

2) The **Subscriber thread** is responsible for creating and sending SUBSCRIBE messages to the server.

3) The **Publisher thread** is responsible for creating and sending PUBLISH messages to the server.

Figure 16 shows how the main thread handles watchers. Messages arrive to a selector through socket channels, in the selector the requests are serialized, each request contains a key representing the client (watcher or publisher). This key does not represent the entire information stream a client sends to a server, but simply a part. The selector divides the client-data into sub-requests identified by the keys [44], the entire message may be processed by the server processes using the key.

The process followed for responding to messages sent to presentities is analogous. After a presentity sends a PUBLISH message, OK messages coming from the server are serialized and parsed in order to extract the SIP-Etag that will be used for modifying, refreshing, or removing the published information.

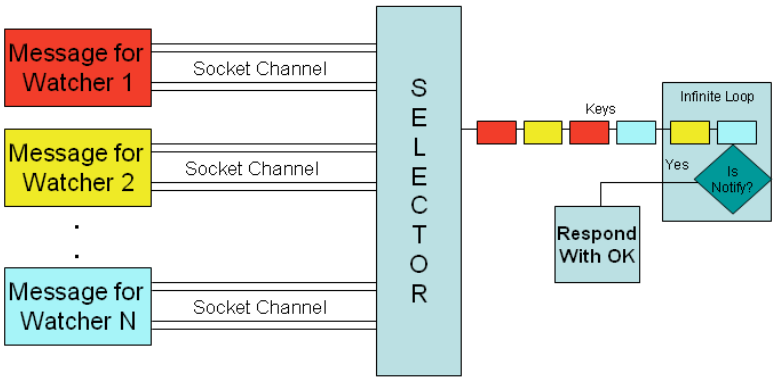


Figure 16. Main Thread for the Load Generator

5. Evaluation

Different types of context information have different characteristics, resulting in different demands for distribution. For example, in the emergency scenario presented in Chapter 2, the location of an emergency only has to be retrieved once, while information about the current activity of a worker should be retrieved after a change occurs, while the available bandwidth may be retrieved periodically for monitoring it.

The main goal of this chapter is to investigate when it is more suitable to use a synchronous or asynchronous mode of context distribution when using SIP/SIMPLE. The decision about whether to distribute context synchronously or asynchronously mainly depends on: 1) how many users will be interested and subscribed to a certain context information, 2) how many context providers will be publishing information to the context server, 3) how often context information updates need to be retrieved by the users, and 4) what is the intensity of context information updates compared to the distribution time needed for information to reach the watcher(s).

The evaluation of SIP/SIMPLE as a protocol for distributing context information will be based on how the number of users (watchers and contextities) and the number of messages they send (with varying intensity) to the server affect the scalability and latency of the system.

Scalability refers to the number of messages (e.g. PUBLISH or SUBSCRIBE) sent by watchers or contextities that can be handled by the system in a short period of time. It is important to consider multiple watchers and multiple contextities, because certain information will be popular and may be of interest to many watchers, as well as some applications will need to retrieve information from several contextities. The scalability tests focus on how the context server performs in different load situations (multiple watchers and/or multiple contextities) in terms of the packet acceptance rate, this is the number of packets that are correctly processed (e.g. acknowledged with an OK response). The packet acceptance rate is measured as a percentage of the total packets sent.

The latency or response time refers to the time elapsed from a message triggering the distribution of context information (REQUEST, SUBSCRIBE, and PUBLISH) and the NOTIFY message arriving at the watcher. In asynchronous mode the notification of new context information is triggered after a subscription and after a publication of information. While, in synchronous mode the notification is only sent in response to the REQUEST message (a SUBSCRIBE with expiration equals to zero). It is important to consider the response time of each mode and compare it with the dynamics of the context information to see if the information is still valid within the user's context. Highly dynamic context information arriving late to a watcher will often be useless, so the distribution time has to be compared with the rate of change of the context

information. For example if someone has subscribed to the location of a user moving at the speed of 300 km/h, and it takes 5 seconds to get the information, then the received location information may have a deviation of 400 meters. Acceleration or other parameters may be used in order to predict the exact position, however it may be inaccurate.

In order to measure how fast the context information distribution is, the response time is defined as the period of time elapsed from when a watcher expects to get the notification (i.e., when the send of a REQUEST or an information update occurs) until the Watcher receives the information. In asynchronous mode, two different response times can be identified. The first one (i.e. subscription response time) between the SUBSCRIBE and the immediate NOTIFY messages and the second one (i.e. notification response time) between a PUBLISH sent by a contextity and the notification being received by the Watcher. On the other hand, for the synchronous mode we have only one response time (i.e. request response time) – which is simply the time between the REQUEST and the REPLY. These response times are shown in Figure 17.

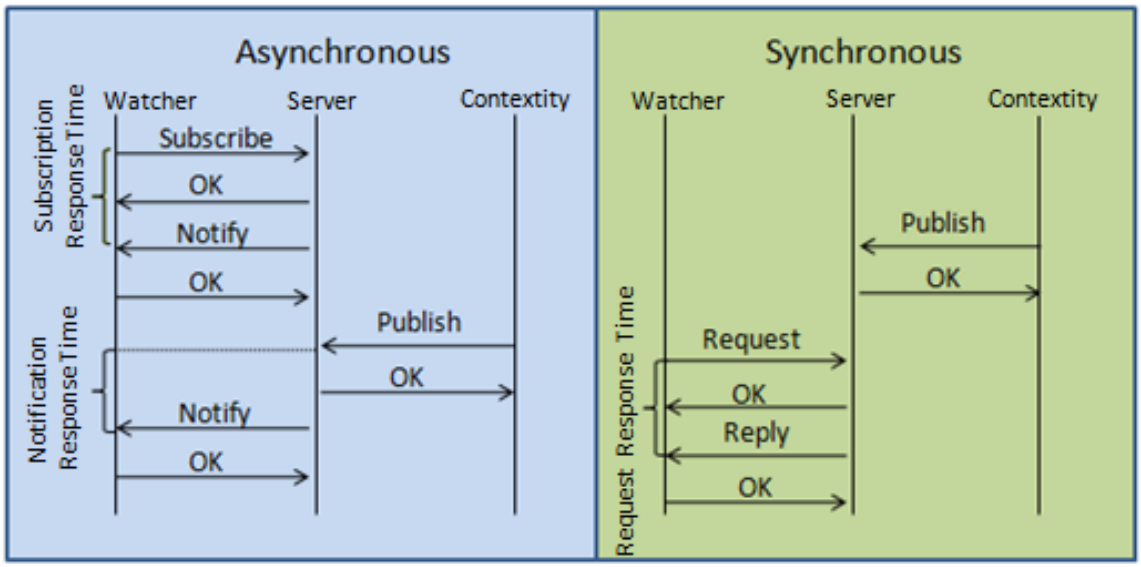


Figure 17. Response Time for the Asynchronous and Synchronous mode

For performing different measurements Wireshark [50] was used. Wireshark is a protocol analyzer used for analyzing and monitoring network traffic available for different platforms (Windows, Linux, OS X, and Solaris). The test bed used for the different tests is described in Table 12. The tests were performed in an isolated wired local area network having one switch (a Netgear fast Ethernet switch model FS108) between the context server and the load generator. The test bed configuration is shown in Figure 18.

The packet length of the SIP messages is not fixed because the message body is variable; however, for the tests performed we use the same message body, the values of the packet length for the different SIP messages are shown in Table 13

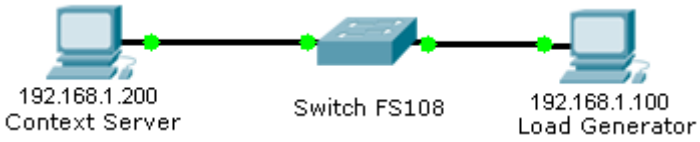


Figure 18. Test Bed Configuration

Table 12. Testbed

	SIP Context Server (SER)	Load Generator for Watchers and presentities
Device	Fujitsu Siemens Celsius M420	HP Compaq dc5100 MT
Operating System	Ubuntu (Linux)	Microsoft Windows XP Professional
Processor	Intel Pentium 4 @ 2.60 GHz	Intel Pentium 4 @3.00 GHz
RAM Memory	1 GB	2 GB
Network Adapter	Broadcom NetXtreme Gigabit Ethernet	Broadcom NetXtreme Gigabit Ethernet
IP Address	192.168.1.200	192.168.1.100
Maxbuffer	256 Kbytes (default value)	
Timer_interval	1 second	

Table 13. Packet length in bytes for the tests

Message	Packet length (Bytes)
PUBLISH	1071
OK	710
SUBSCRIBE	510
NOTIFY	817

5.1 One watcher and one contextity

The simplest case is when we have only one watcher and one contextity interacting with the server; moreover, the response times measured for this case are useful as a reference for more complex cases. This test aims to measure the different response times, specifically the **subscription** and **notification time** for the asynchronous mode and the **request response time** for the synchronous mode. In the asynchronous mode the watcher sends a SUBSCRIBE message, then the contextity sends a PUBLISH message. The subscription and notification response times are measured. In synchronous mode the contextity publishes information, then the watcher

sends a REQUEST message and we measure the time until the reception of the REPLY message. For both cases the test was repeated 100 times in order to reduce the random error. The main goal of this test is to obtain reference response times that are a base for comparison with more complex situations (when utilizing multiple watchers and/or multiple contextities).

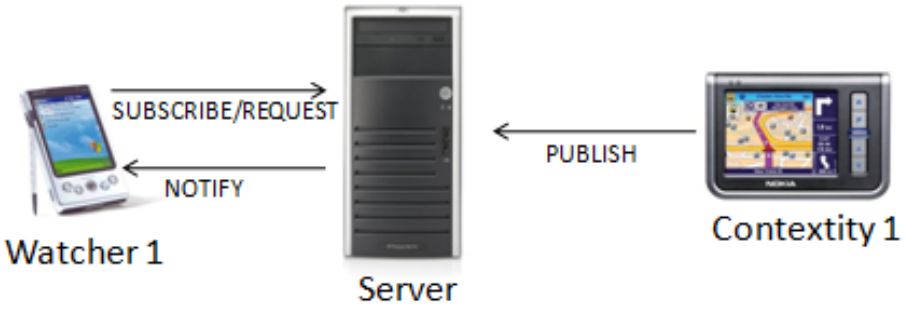


Figure 19. One watcher and one contextity

The results from the previous test are summarized in Table 14. As it can be seen for the asynchronous mode, the average subscription response time is of 1.65 milliseconds and the notification response time is of 0.537 seconds. On the other hand, for the synchronous mode the average request response time is of 1.45 milliseconds. Statistically, the average subscription and request response times are equal, however in asynchronous mode we got in some repetitions response times much higher than in synchronous mode (maximum value was 5.1 ms), mainly because in this mode database operations need to be performed. The notification response time depends on the timer_interval parameter. As mentioned in section 4.1, when this timer runs out the server sends NOTIFY messages of changed contextities. For this test this timer was set to one second, so it is expected to have a notification response time randomly distributed between zero and one second, this because the PUBLISH message may arrive at any moment before the timer running out. This is the reason for having such a considerable standard deviation for this response time.

Table 14. Time response for one watcher and one contextity

	Asynchronous		Synchronous
	Subscription Response Time (ms)	Notification Response Time (ms)	Request Response Time (ms)
Minimum	1.4	56.4	1.1
Average	1.7	537.4	1.5
Maximum	5.1	932.9	1.8
Standard Deviation	0.5	384.4	0.07
Standard Error	+/- 0.05	+/- 171.9	+/- 0.007

A second test tries to determine the minimum interarrival time between REQUEST messages that can be responded to by the server (i.e., with the server sending a NOTIFY message). The main goal of this test is to find the maximum speed at which a watcher can synchronously retrieve information from the server, in order to

find the absolute maximum polling rate (note that this rate is based upon the server not having any other requests or tasks to perform). After a publication of information, the watcher starts sending 30 REQUEST messages, first with 2 seconds of interarrival time, and then with decreasing interarrival times (decreasing by 50 ms each) between REQUEST messages. Figure 20 depicts the evaluation scenario. For each round we measure how many packets were responded to by the server with a NOTIFY message.

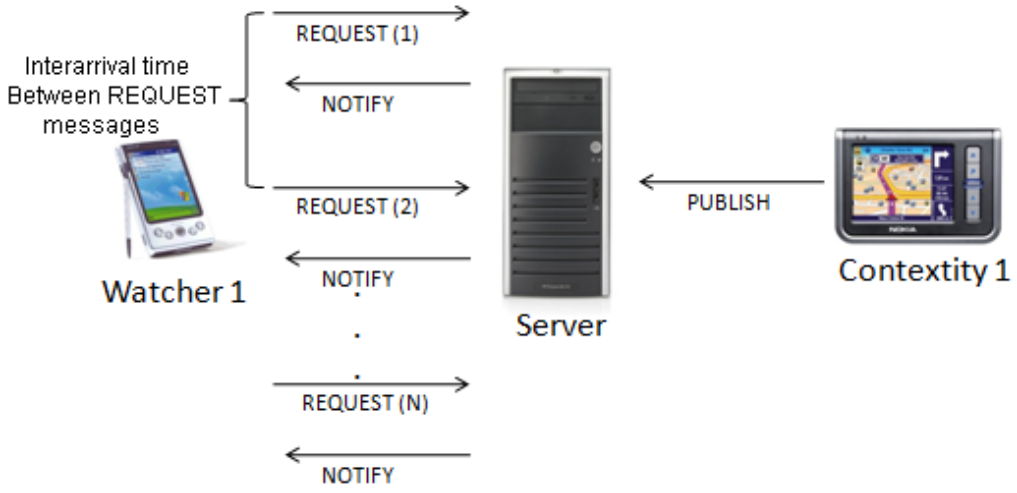


Figure 20. Polling with one watcher and one contextity

The results obtained are shown in Figure 21. We have found that the minimum interarrival time between REQUEST messages is one second, when the watcher starts sending REQUEST messages faster, some of these messages are not responded to with a NOTIFY message, although they are acknowledged with an OK response. When a watcher sends more than one REQUEST message per second, the context server emits only one notification. This feature is inherent in the performance and behavior of the server and is not related to the timer_interval parameter (is not parameterized in the configuration file of the context server). Note that the arrows in the figure represent the fraction of messages which were not responded to by a NOTIFY message.

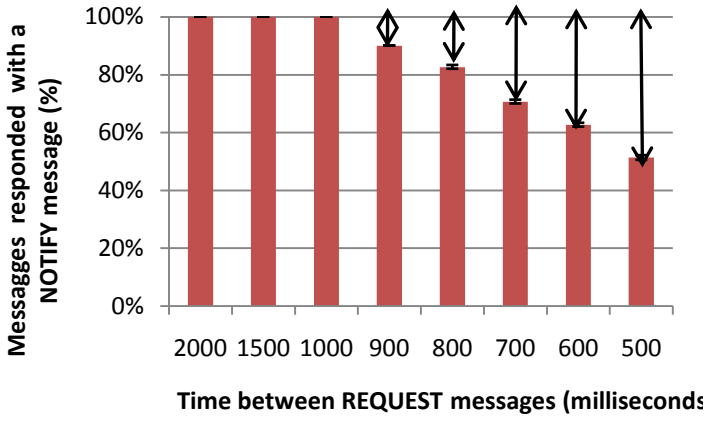


Figure 21. Results from multiple requests messages (one contextity and one watcher in synchronous mode)

5.2 Multiple Watchers and one contextity

In this evaluation scenario we want to simulate the case when several watchers are interested in retrieving information from one contextity. This could occur when a large number of users are interested in retrieving or subscribing to the location information of a single user. We will like to explore how the server performs when handling several subscription or request messages within a short period of time in terms of response time and scalability. Furthermore, we want to determine if the service time of the server is affected by the number of watchers using the system.

The first test addresses scalability and tries to measure how many packets can be handled by the server in a short period of time (acknowledged with an OK response), the messages are sent at a rate of approximately 2000 messages per second. For the asynchronous mode the load generator starts by sending 100 SUBSCRIBE messages from different watchers (this is emulated by a single process sending these request in a FOR loop), then we measure how many packets were processed by the server (i.e. acknowledged by an OK response), the number of SUBSCRIBE messages increases in steps of 100 messages in subsequent rounds, for each round we repeated the test 5 times. For the synchronous mode the same test is performed, but instead of SUBSCRIBE messages, REQUEST (subscription with expire equals to zero) messages from different watchers are sent. In Figure 22 this test scenario is shown.

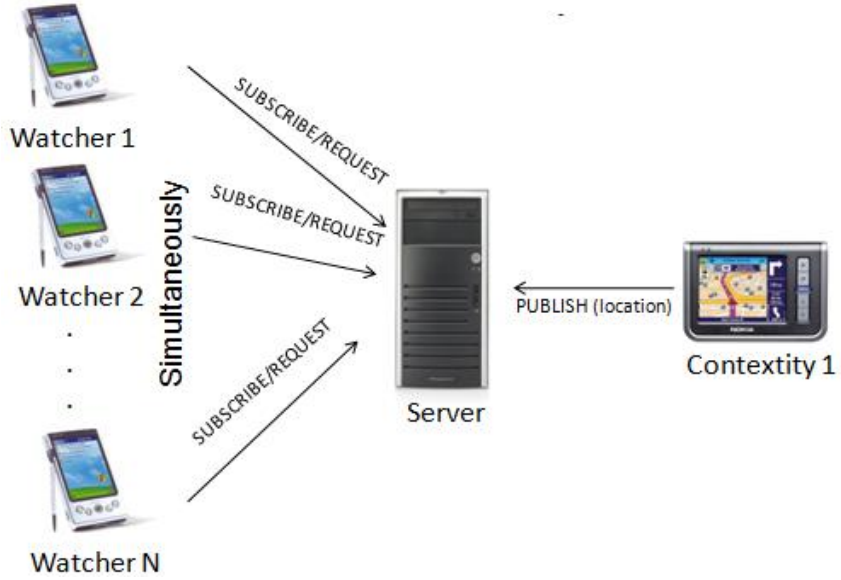
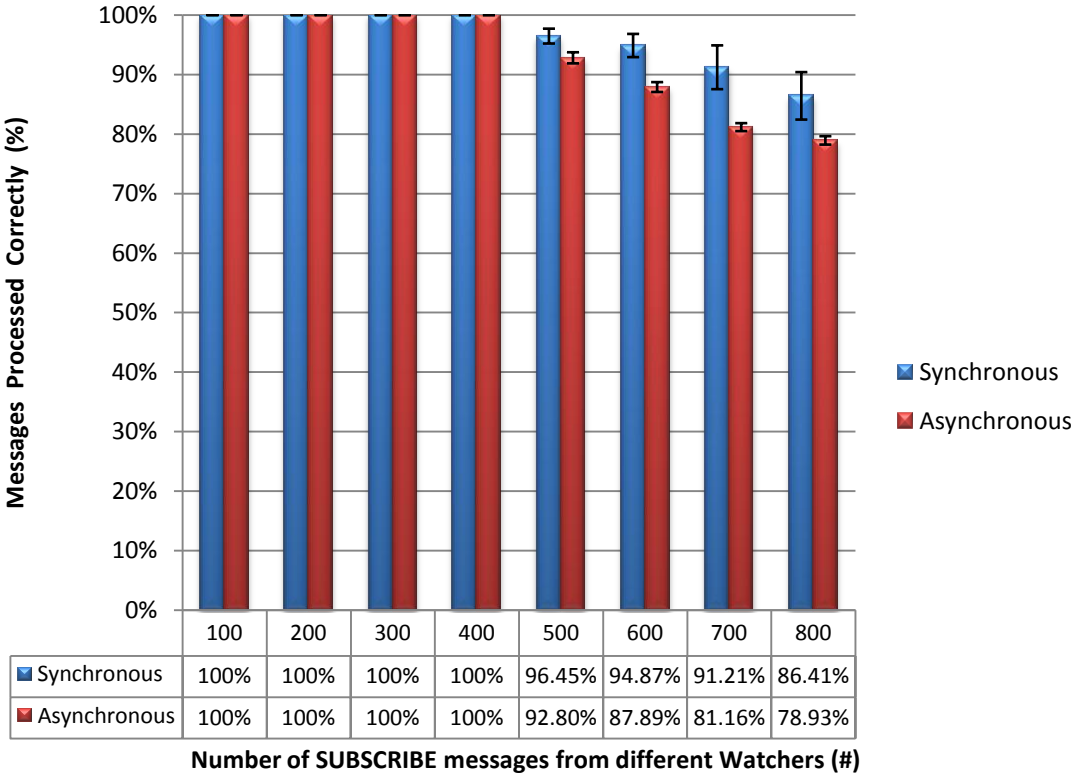


Figure 22. Multiple watchers subscribing/requesting from one contextity

The results obtained are shown in Figure 23. As it can be seen in the graph, for both distribution modes the server can handle correctly up to 400 SUBSCRIBE or REQUEST messages sent with an interarrival time approximately of 500 μs. Sending more than 500 messages in such short period of time results in a loss of messages, mainly because the arrival rate of SUBSCRIBE/REQUEST messages is faster than the service rate of the server and the server buffers have a size of only 256 KBytes. After comparing the synchronous and asynchronous case, we can see that in asynchronous

mode, when sending more than 400 messages, even more packets are lost. Mainly, this is because the service time of the server is larger for asynchronous mode, so the server buffers got full faster, leading to more messages being discarded. The loss of packets is addressed by the SIP reliability mechanisms- which will cause the entity to retransmit them. By considering the length of the server buffers (256 Kb) and the length of SUBSCRIBE messages (510 bytes), the expected number of accepted SUBSCRIBE messages by the server is around 500 messages.



	100	200	300	400	500	600	700	800
Synchronous	100%	100%	100%	100%	96.45%	94.87%	91.21%	86.41%
Asynchronous	100%	100%	100%	100%	92.80%	87.89%	81.16%	78.93%

Figure 23. Results from Scalability Test – One contextity and Multiple Watchers (messages sent in a burst)

A second test examines how the server notifies watchers in asynchronous mode. 400 different watchers sent a SUBSCRIBE messages in a burst, 400 messages are sent because in the previous test we found that when 400 messages were sent in a burst they could be handled successfully by the server. After getting all the OK responses for these messages, a PUBLISH message is sent, in order to trigger the notifications. We examine in which order the server notifies the watchers and also how quickly the NOTIFY messages are sent to the different watchers. We are interested in learning how long it takes to get a notification; in order to compare it with the case when a user is synchronously retrieving the information – in order to decide which method is faster.

In Figure 24 we can observe how the notification process to 400 subscribed watchers occurs. After the first notification all the subscribers are notified at a nearly constant rate, with around 350 μs of interarrival time between each NOTIFY message (i.e. a rate of 2850 notifications per second). Also it is important to notice that the notification order follows a last to subscribe first to be notified manner. In this case the

watcher$_{400}$ is the first one to receive a notification and the watcher$_1$ is the last one. From all the repetitions, the average time for notifying 400 watchers was 139 ms. In general the time for a watcher to get a notification after a PUBLISH message depends on a random period of time which depends upon the timer_interval parameter and the number of subscriptions received after his or her subscription. For the example depicted in Figure 24, the most favored watcher is watcher$_{400}$ (the last one subscribed), while watcher$_1$ is the last watcher receiving the notification, receiving the notification 140 ms after the watcher$_{400}$. It is important to note that the notification process starts after a period randomly distributed between 0 and 1 second due to the timer_interval parameter, in this graph we only show the process since the first notification.

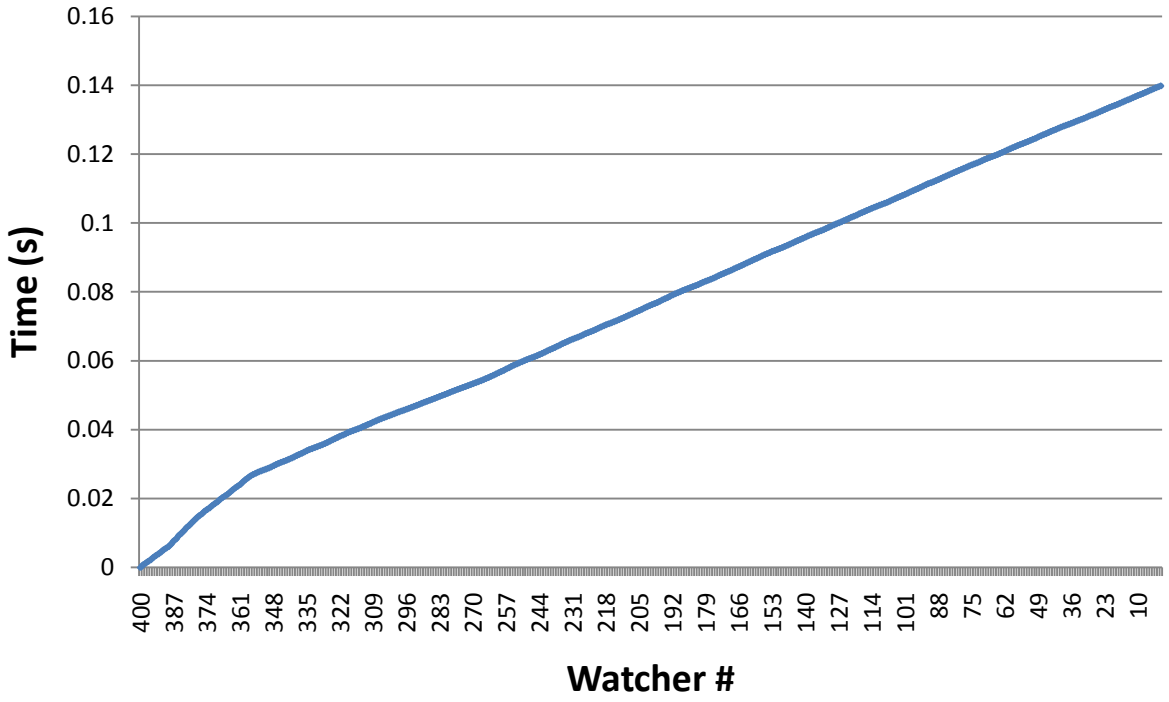


Figure 24. Notifications to 400 Watchers

In order to study if the number of watchers subscribed to the same event affects the notification process and the time between NOTIFY messages, the previous test was repeated for different numbers of watchers subscribed (100, 500, 1000, 2000, 3000, 4000, and 5000) to the contextity. The behavior of the notification process was the same, notifications at a regular rate; the average time between notifications was between 300 and 400 μs. The results are shown in Figure 25. As it can be seen the time between notifications does not vary very much; having information stored in the database does not affect the time between notifications.

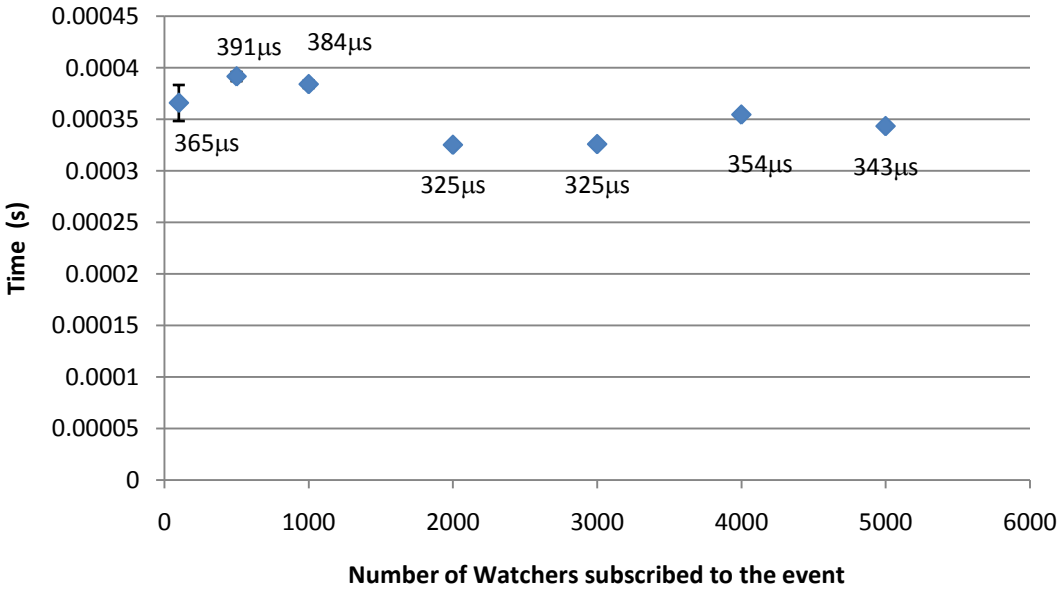


Figure 25. Average time between notifications, varying number of watchers

The next test measures how the subscription and request response times of the server are affected when multiple users are subscribing or requesting information from the server. In asynchronous mode we measure the subscription response time. For this 400 SUBSCRIBE messages (from different watchers) are sent in a burst, we measure the response time to get the NOTIFICATION for each watcher in order to quantify the subscription response time. For the synchronous mode the same procedure is followed, but with REQUEST messages, in order to measure response times between a pair of REQUEST and REPLY messages. We choose to send 400 messages, because from previous tests we have found that 400 SUBSCRIBE messages sent in a burst (in 0.2 seconds) can be handled by the server. The averaged results for the **subscription** response time and **request** response time from 10 repetitions is shown in Figure 26. The black shadows in the figure represent the error bars derived from the measurments. After comparing the response times for both distribution modes, we can observe that for both modes the time response increases from one watcher to the next one(s). Also, we can note that for the first 150 watchers the response time between the two modes is almost the same; however for the next watchers the response time between watchers increases faster in asynchronous mode. For the watcher400 in asynchronous mode the subscription response time is 27.26 ms, while in synchronous mode the request response time is 18.85 ms. One of the reasons for the increase in the response time from one watcher and the previous ones, is that each message has to wait for longer in the buffer (queues) of the server.

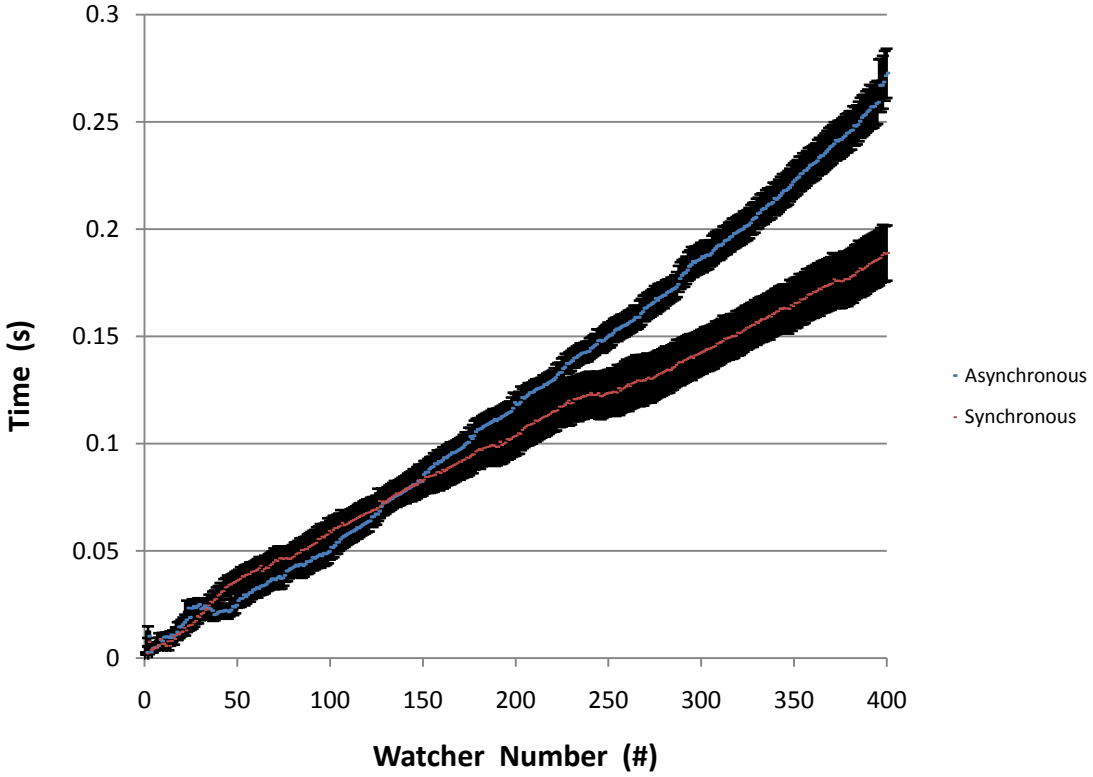


Figure 26. Responses time for Synchronous and Asynchronous mode (i.e. Request vs Subscription response time) - 400 messages sent in a burst

In order to examine if the response times increases due to other factors (besides the waiting time in the queues), such as database operations, we evaluate the system by sending 14 bursts of 300 SUBSCRIBE messages from different watchers. The time between each burst is 5 seconds, enough for empting the buffers. In this test we measure the time required to serve a complete burst, in order to investigate if the time to serve the following bursts is dependent on the number of previous bursts already subscribed in the system. For the synchronous mode the same procedure will be followed but with REQUEST messages.

In Figure 27 the subscription and request response times are shown for both modes of distribution for the 4200 watchers (14 bursts of 300 watchers). We can see that the elapsed time between each burst was sufficient for emptying the server buffers. Comparing both modes of distribution we can see that for the first 3 bursts the response times for both modes is similar, however for the following bursts we can observe a significant increase in the subscription response time for asynchronous mode. Clearly we can observe that in asynchronous mode, the subscription request time increases from one burst to the next one, while in synchronous mode the request response time is quite constant for all the bursts. Analyzing the last burst, we can see that the subscription response time in asynchronous mode is about 3 times larger than the request response time in the synchronous mode. In Figure 28, we compare the time needed to serve each

burst, and evidently it can be seen how in asynchronous mode the time for serving a burst increases for each subsequent burst.

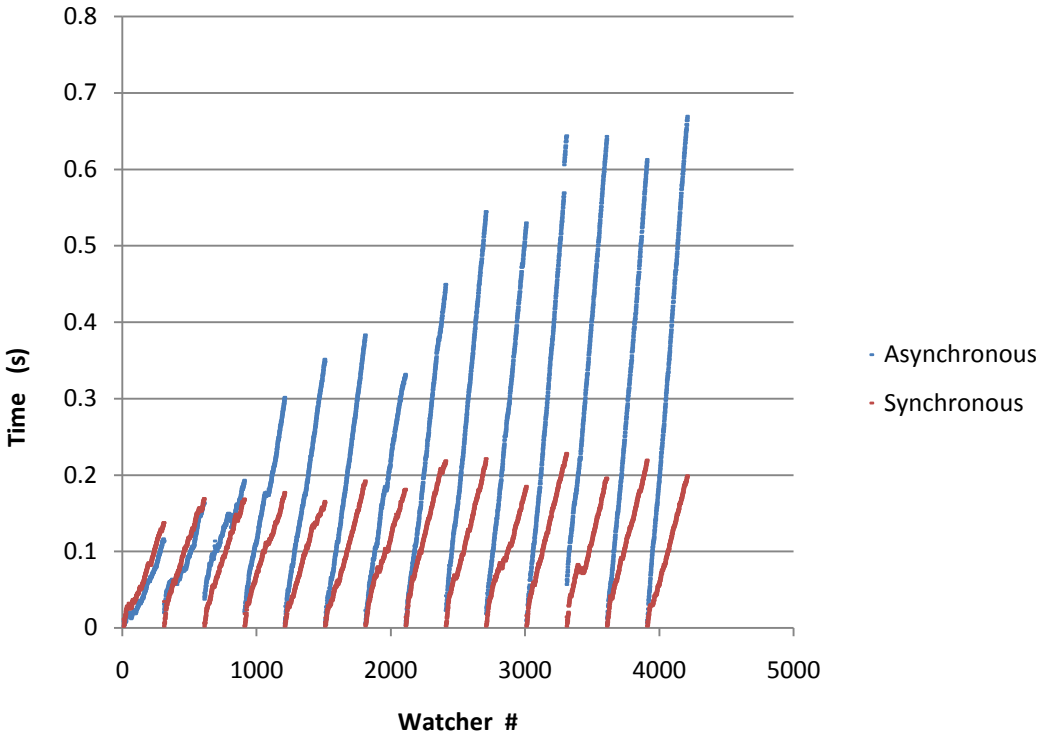


Figure 27. Bursts of 300 SUBSCRIBE/REQUEST messages

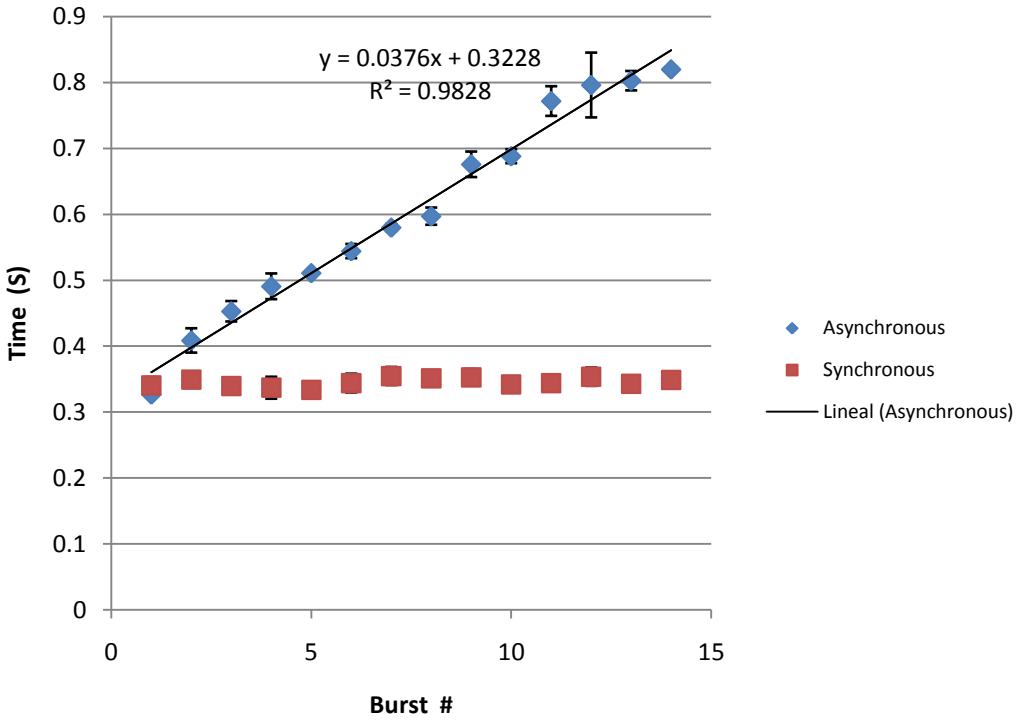


Figure 28. Time for serving bursts of 300 watchers, Synchronous and Asynchronous

In asynchronous mode the time difference between one burst and the next one is on average 37 ms. After this experiment we can recognize that the subscription response time in asynchronous mode is dependent on the number of previous subscriptions, mainly because of the database operations that have to be performed when receiving a SUBSCRIBE message, primarily storing information from the watcher in the database (URI, IP address, expiration value, etc.).

In the next test for multiple watchers and one contextity 4200 SUBSCRIBE or REQUEST messages from different watchers will be sent at a sustained rate for examining how the response times are affected because of the number of watchers already subscribed to the server. The interarrival time between messages was around 5 ms. In Figure 29, we can see that in synchronous mode the request response time is nearly constant for all the watchers, while in asynchronous mode, the subscription response time increases, for the last watchers the subscription time increases even faster because the subscription response time increases and becomes larger than the interarrival between SUBSCRIBE messages, so messages have to wait in the queue.

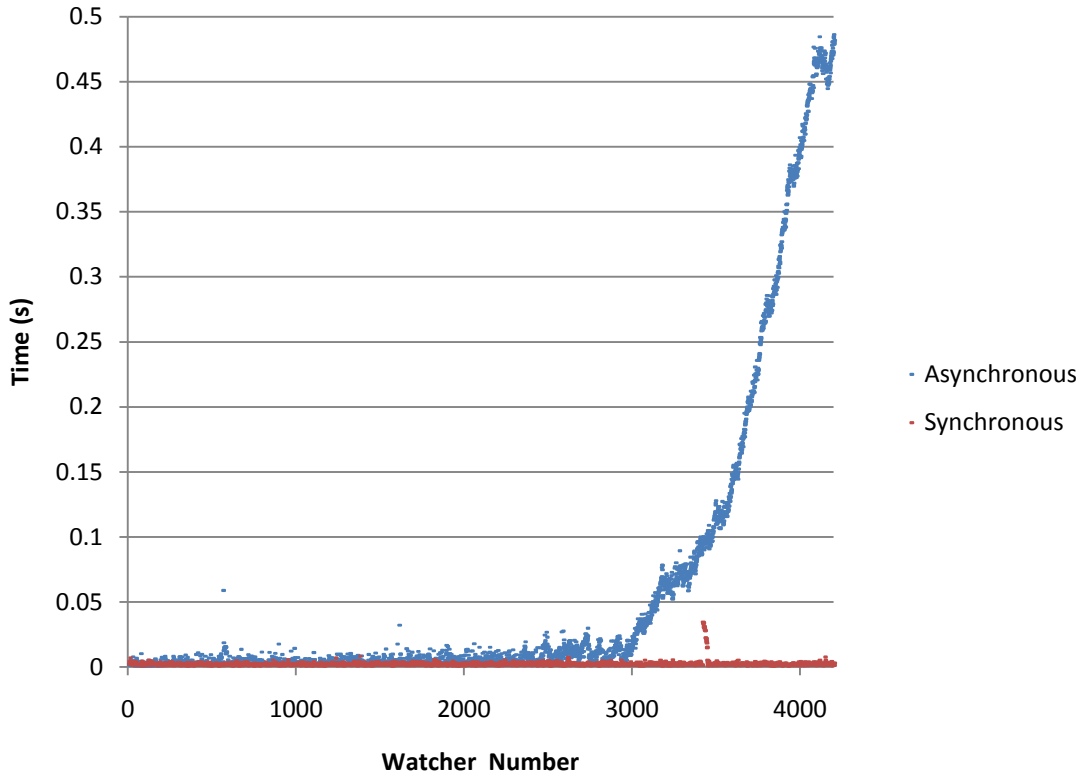


Figure 29. subscription/request response time - sustained rate.

We have found that the subscription time is affected by the previous users already subscribed to the system; mainly because of reading and writing information from subscriptions in the watcher table of the server's database. In order to examine if the subscription and request response times also increase because of having information in the database from PUBLISH messages, we tested the server when the server's database is loaded with information from the contextities (i.e. information from

PUBLISH messages). For this, first we load the database by sending PUBLISH messages, the test was repeated for different numbers of PUBLISH messages (100, 500, 1000, 2000, 3000, 4000, 5000, and 10000 messages), then 100 different watchers start to send REQUEST or SUSCRIBE messages to one contextity at a constant rate (3 messages per second). We measure, then average the SUBSCRIBE and REQUEST response time from these 100 watchers.

The results are shown in Table 15. As it can be seen the REQUEST response time and SUBSCRIBE response time for both distribution modes are nearly constant, independent of the amount information in the database from publications. This is because the information from PUBLISH messages and SUBSCRIBE messages are stored in different tables within the server's database. Thus information stored in the contextity tables does **not** affect the time required for performing operations in the watcher table.

Table 15. SUBSCRIBE and REQUEST response times when the database is loaded with information PUBLISH messages

	Synchronous	Asynchronous
# of Publish messages stored at database	REQUEST Response Time (ms)	SUBSCRIBE Response Time (ms)
10000	1.46	1.68
5000	1.48	1.68
4000	1.48	1.68
3000	1.45	1.66
2000	1.47	1.60
1000	1.44	1.64
750	1.45	1.66
500	1.45	1.64
100	1.44	1.62

5.3 One watcher, one contextity and multiple PUBLISH messages

This evaluation scenario will simulate different intensities of updates of context information. As different types of context information change with different rates, for instance location information may change quickly when compared to the temperature of a city that may change more slowly. It is important to find out at what rate the server can handle updates of context information. For this test we have one watcher subscribed to the information from the contextity, then a PUBLISH message is sent, after receiving the OK response, an update is sent. After 100 PUBLISH updates are sent periodically, for the first round the interarrival time between these updates is 1.5 seconds, for later rounds the interarrival time decreases by 250 ms. In this test we measure how many of the PUBLISH messages are responded with a NOTIFY messages to the watcher. The scenario is shown in Figure 30.

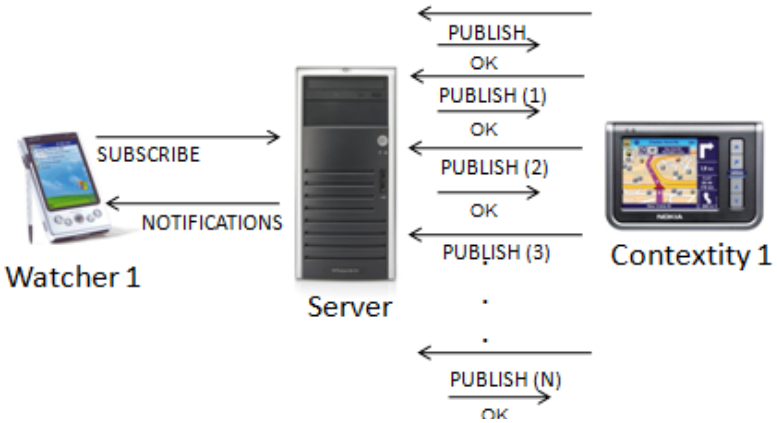


Figure 30. One watcher, one contextity, multiple PUBLISH messages

In Figure 31 the results from the test are shown. As it can be seen only when the time between PUBLISH messages is 1.5 seconds are all of the messages followed by a NOTIFY message. It is clear than the server can only send a NOTIFY message each second because of the timer_interval parameter (set to 1 second), that is why a smaller number of PUBLISH messages result in notifications when PUBLISH messages are sent with an interarrival time below 1 second. Although not all the messages were followed by a NOTIFY, all were acknowledged by an OK response, this means that they were handled correctly by the server (i.e. the information is updated correctly in the database). Even when the PUBLISH updates are sent with a separation of 10 ms, they were all acknowledged by the server (with an OK response). From this experiment we can observe that one limitation of the context server is that it is not suitable for distributing context information when the information is changing faster than one message per second.

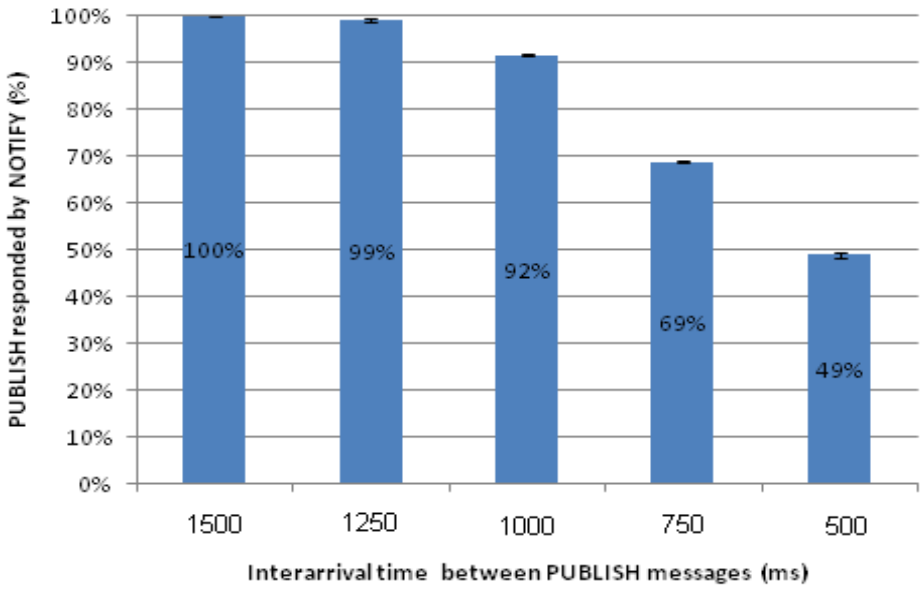


Figure 31. Sending PUBLISH updates periodically (One contextity and one Watcher)

5.4 One watcher and multiple contextities

Certain applications need to retrieve information from several contextities, this evaluation scenario simulates this case. For example in the emergency scenario presented in Chapter 2, the system needs to retrieve information about the location, current activity, and capabilities of a worker in order to decide if she or he is a relevant worker for a certain task. In order to see how scalability is affected in relation to the number of contextities publishing information to the server we perform tests having multiple contextities, as shown in Figure 32.

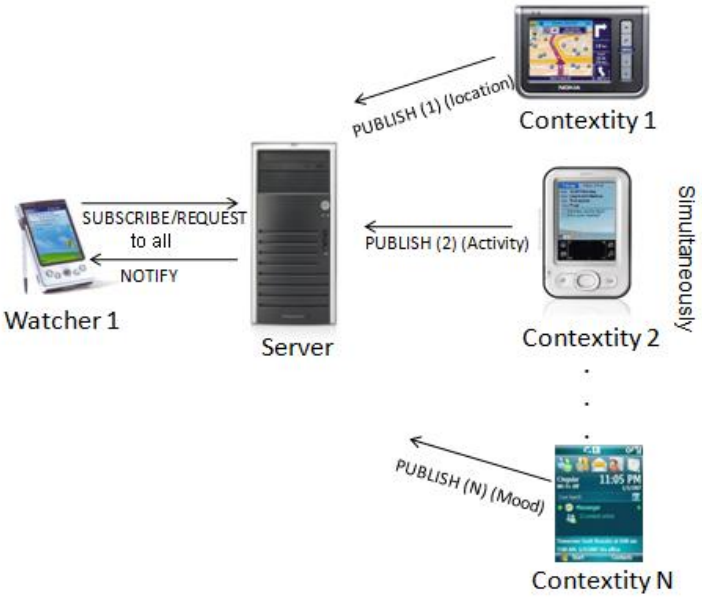


Figure 32. One watcher and multiple contextities

In order to study how many PUBLISH messages can be handled by the server in a short period of time. We start by simulating 100 contextities each sending a PUBLISH message, the messages are sent in a burst, the number of contextities sending a PUBLISH message will increase in steps of 100 for each round. We measure how many PUBLISH messages can be handled by the server, by quantifying how many PUBLISH messages are responded to with an OK response.

In Figure 33, the results show that when we sent SUBSCRIBE messages in a burst, some of the messages were dropped. In the case of PUBLISH messages 18% of the messages were lost when sending 300 PUBLISH messages (from different contextities) in a short period of time. As in the case of SUBSCRIBE messages, this is because the server's buffers got full, so these loss PUBLISH messages need to be retransmitted by the contextities. The server can accept fewer PUBLISH messages in a short period of time, than SUBSCRIBE messages, because PUBLISH messages are larger and the buffer size is unchanged.

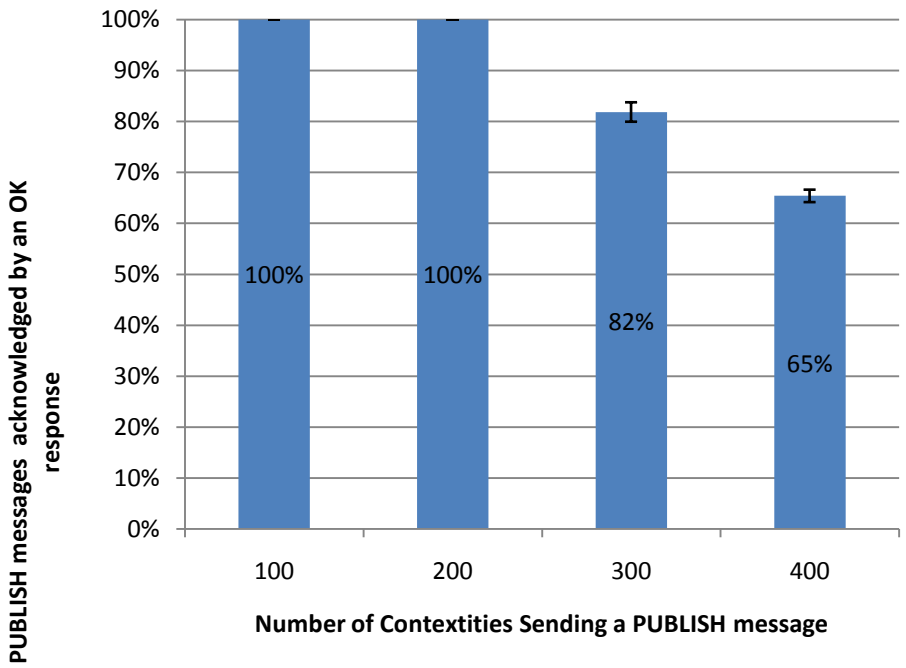


Figure 33. Scalability for PUBLISH messages sent in a burst

A more interesting case is when PUBLISH messages from the different contextities are sent at a sustainable rate. To measure the greatest sustainable rate for PUBLISH messages coming from different contextities that can be handled by the server (replied with an OK) we send 2000 PUBLISH messages, starting with a period of 50 ms between PUBLISH messages, we decrease this period of time for the next rounds, until 4ms. For all the rounds performed the 2000 contextities got the OK response, this means that the information from the PUBLISH messages is stored in the database.

In the previous test, we have found that the time between the PUBLISH message and OK response, increased slightly, in order to get a deeper insight into this we test the server by varying the number of contextities sending a PUBLISH message at a sustainable rate (3 messages per second). After capturing all the packets in Wireshark, we measure the time needed for handling the publication, this is the time between the PUBLISH message and its OK response. We have found that the time for handling the publication messages increases slightly. Figure 34 shows the case when we send 100 PUBLISH messages. As can be seen the time for handling the PUBLISH message in general increases with respect to the previous messages, on average this increase in time is 5μs for each previous PUBLISH message. The average time for handling the PUBLISH message for the different cases is shown in Table 16. As it can be seen the average time for handling the publication increases slightly, however this increase is quite small and may be insignificant for most of the applications using the context server. The raise in the time needed for handling the PUBLISH messages is due to the database operations that have to be performed in order to store the information from the contextities in the database, as when there is already information in the database's table, these operations take a little bit more time.

Table 16. Time for Handling PUBLISH messages

# of Publishers	Average Time for handling PUBLISH message (ms)	Standard Error (ms)
100	3.405	+/- 5.55E-02
500	3.593	+/- 3.19E-02
750	3.66	+/- 3.52E-02
1000	3.664	+/- 2.38E-02
2000	3.764	+/- 2.26E-02
3000	3.82	+/- 2.83E-02
4000	3.913	+/- 3.53E-02
5000	3.954	+/- 4.26E-02
10000	4.494	+/- 4.92E-02

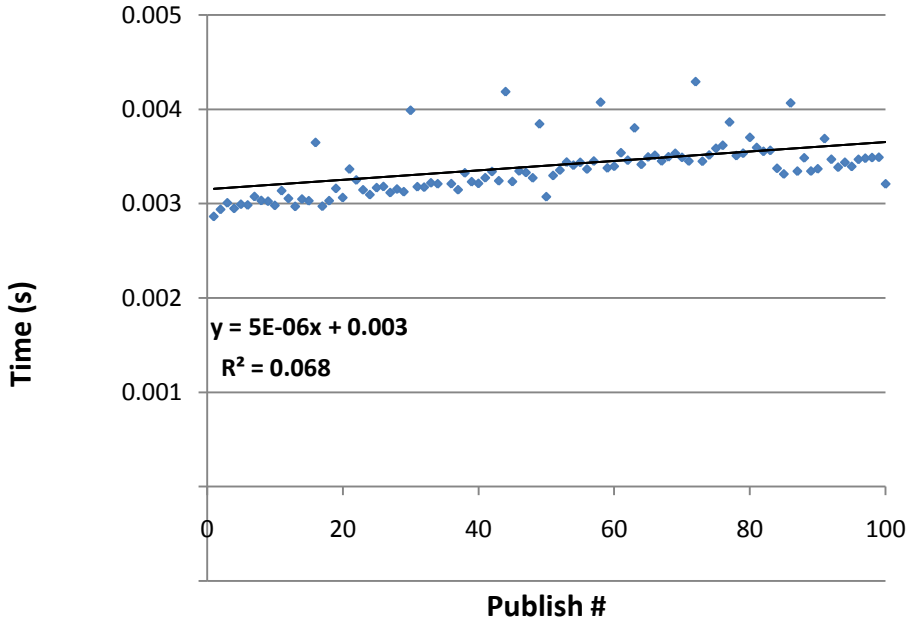


Figure 34. Time for Handling PUBLISH messages with 100 contextities

5.5 Multiple watchers and multiple contextities

Finally the last evaluation scenario considers the most complex scenario: multiple watchers retrieving or subscribing to information and multiple contextities publishing different types of information to the server. This evaluation scenario considers a more realistic case where several context providers are publishing different kinds of context information to the server and also several applications are retrieving synchronously and asynchronously these information. As in the previous tests we are interested in studying how the server scales with multiple users (watchers and contextities) in the system, as well as how the response times are affected.

The first test will focus on the scalability of the server, starting with 50 contextities sending PUBLISH messages, at the same time 50 watchers subscribe to different contextities by sending 50 SUBSCRIBE/REQUEST. So there will be 100

messages sent at almost the same time. The number of contextities and watchers sending messages increase in steps of 50 contextities and 50 watchers. In this test we measure how many PUBLISH and SUBSCRIBE/REQUEST messages can be handled correctly in a short period of time. This test scenario is shown in Figure 35.

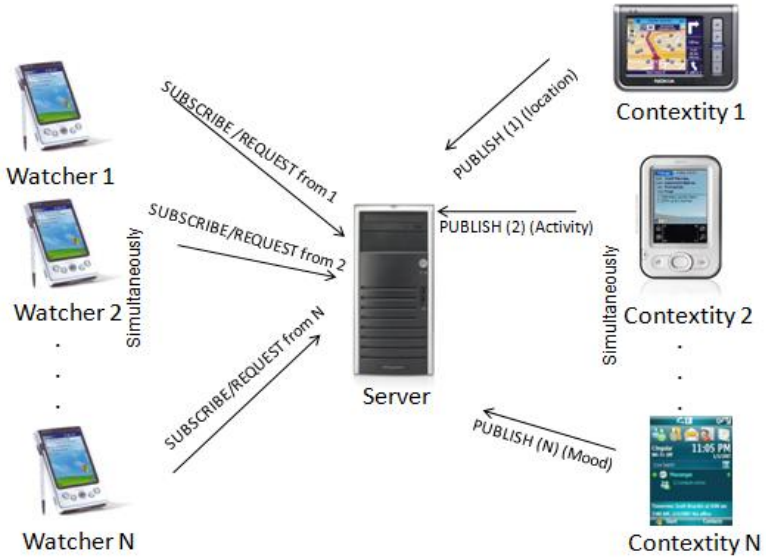


Figure 35. Multiple watchers and multiple contextities

Figure 36. Scalability when having multiple watchers and multiple contextities

As in the previous scalability tests messages were dropped, for this case the maximum number of messages handled in a burst were 150 SUBSCRIBE and 150 PUBLISH messages all from different watchers and different contextities, after this

amount of messages some were dropped mainly because of buffer overflow. The results are shown in Figure 36.

The last test examines how the subscription and notification behavior (for the asynchronous case) and the request behavior (for the synchronous mode) are affected by having multiple watchers subscribing to or requesting information from the server and multiple contextities publishing it. In the first round we begin with 1 watcher subscribed to/requesting information from 1 contextity, this is 100 different watchers retrieving from 100 different contextities. Later rounds will consider more watchers subscribed to a same contextity, 100 groups of {5, 10, 50, or 100} watchers subscribing to or requesting from 100 different contextities, all the watchers within a group will be subscribing to or requesting context data from the same contextity. For the asynchronous mode we measure the subscription response time, then we will send PUBLISH updates for triggering notifications and examine how the notification process occurs. In the synchronous case we will measure the request response time. This evaluation scenario is shown in Figure 37.

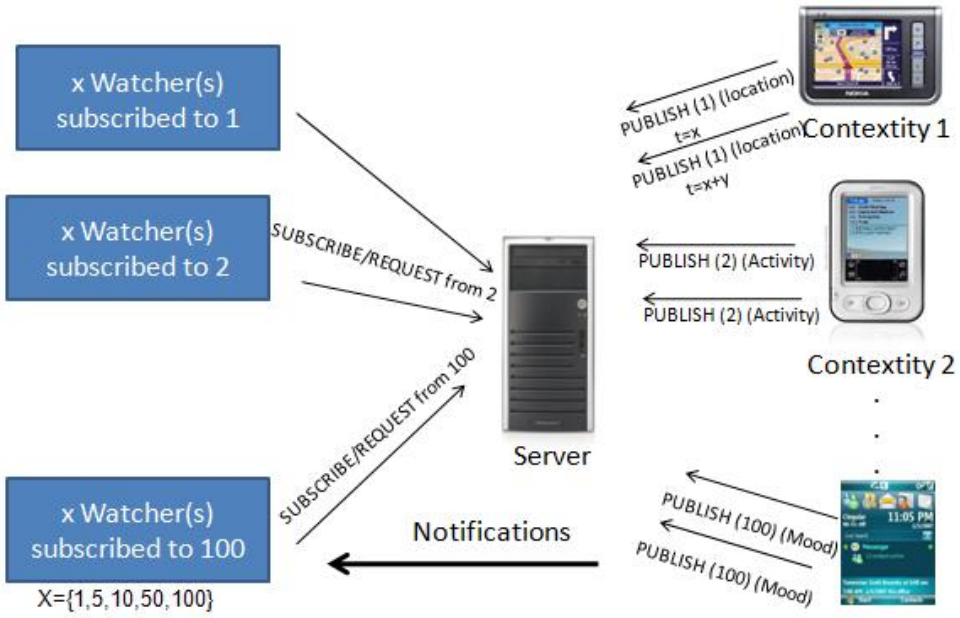


Figure 37. Groups of watchers subscribing/requesting to multiple contextities. X represent number of watchers that were subcribed to the same contextity in each round

After performing the tests we found that the subscription and request response time for the both modes of distribution was the same as the one observed when having multiple watchers. The only significant changes occurred in the behavior of the system when notifying watchers after a PUBLISH message, more precisely in the scheduling of notifications to watchers. We have chosen the particular case of having 5000 watchers, divided in 100 groups of 50, each group subscribed to a different contextity in order to show this scheduling behavior. As in the case of multiple watchers, the notification process occurred at a regular rate, in average 326 μs between each NOTIFY message; however, the schedule of notifications changed compared to the case with only one contextity. This scheduling is shown in Figure 38 and Figure 39. As it can be seen

within each group 50 notifications occurred subsequently and it followed the same scheme as in the case with one contextity, last to subscribe, first to receive notification. The difference lies in the order in which the system notified watchers subscribed to different contextities. The pattern followed by the server is the one shown in Figure 38, each rectangle in the graph is a group of 50 watchers that are subscribed to the same contextity, the number besides the group denotes to which contextity the group is subscribed. The pattern followed to notify watchers subscribed to different contextities is presented in Appendix D. Figure 39 shows a snapshot of a group of 50 watchers subscribed to the same contextity. In terms of accuracy, all the notifications were addressed to the correct watcher (correct URI and correct UDP socket).

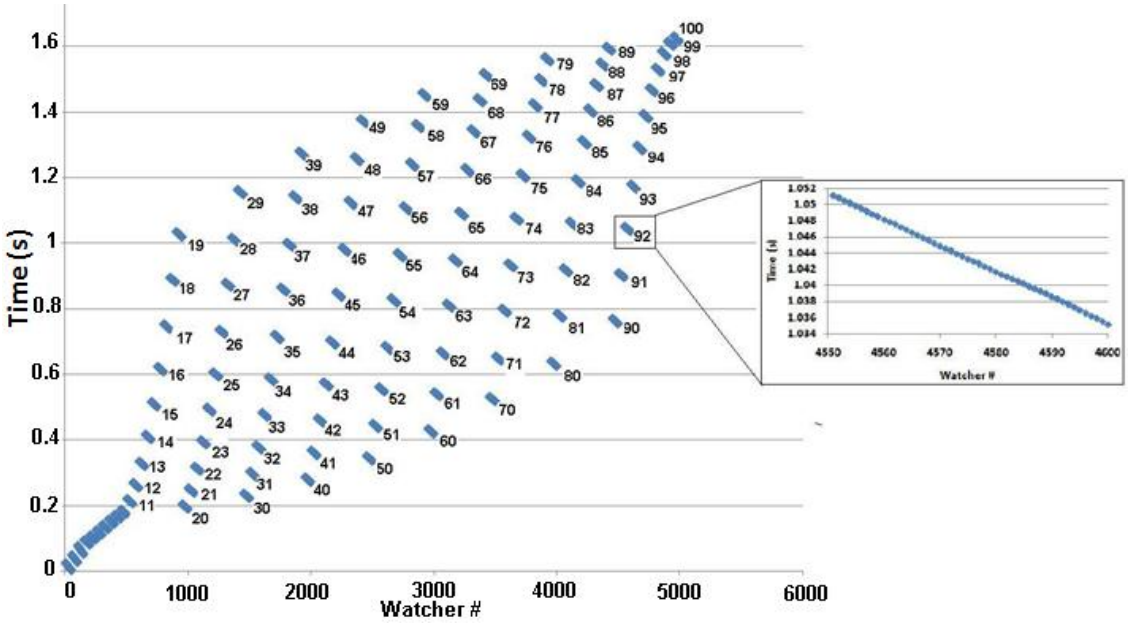


Figure 38. Notifications with Multiple Watchers and Multiple Contextities

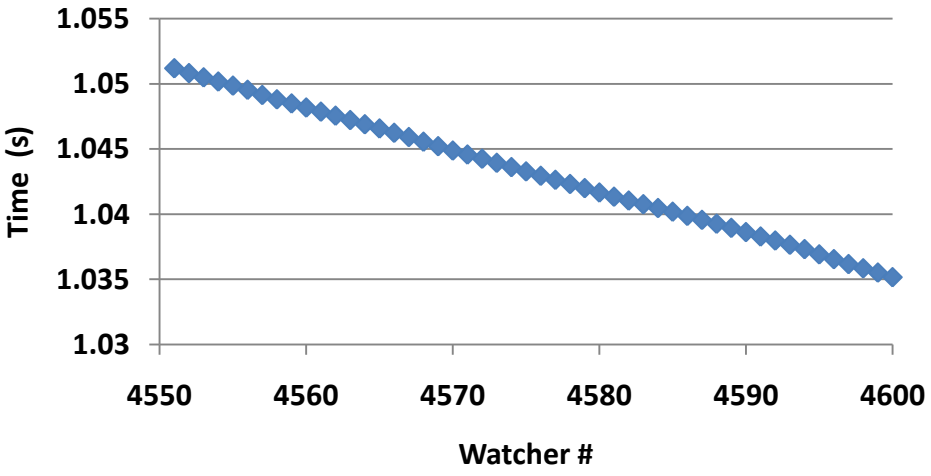


Figure 39. Notifications to Watchers subscribed to Publisher 92

5.6 Summary of the tests performed

We performed different tests considering: (i) one watcher and one contextity, (ii) (iii) multiple watchers, (iv) multiple contextities, and (v) multiple watchers and multiple contextities. From the tests dealing with sending several messages (PUBLISH; SUBSCRIBE, and REQUEST) at a high rate (2000 messages/sec), we have found that after 200 PUBLISH, 400 SUBSCRIBE, or 400 REQUEST messages some messages get lost due to the limited buffer size (default value 256 KB) -and thus need to be retransmitted by the user agents. The buffer size is a configurable parameter in the server's configuration file, however this is configured at startup, so it cannot be adaptively configured. Increasing the value of the buffer size allows receiving more packets in a short period of time, but more memory will be reserved.

As observed by Mohammad Zarifi [30], we also found that the schedule of notifications follows a fixed scheme. For watchers subscribed to the same contextity, the last ones to subscribe are the first ones to be notified. Notifications are delivered at a uniform rate of 2850 notifications per second. In the case where watchers are subscribed to different contextities the system follows the notification scheme illustrated in Figure 38. Due to this fixed scheduling scheme for notifying watchers, some watchers will always get the notified first, while the others will receive notifications with some delay (while their subscriptions do not expire.)

In terms of response times, we have found that the subscription response time in asynchronous mode is dependent on the amount of information stored in the watcher's table of the Mysql database. On the other hand the request response time in synchronous mode is constant because no database operations need to be performed. Both response times are not affected where there is information in the tables regarding the contextity in the database. In order to decouple the subscription response time from the database operations, it will be valuable to respond messages before performing database operations.

In the asynchronous mode the server can quickly procces PUBLISH messages or PUBLISH updates, however the notification to subscribed watchers is limited by the timer_interval parameter (granularity in seconds). The server only sends notifications to watchers when the timer_interval timer runs out. Finally for synchronous mode we found that the maximum polling rate is 1 second. REQUEST messages arriving with an interarrival time smaller than one second do not result in notifications, this behavior cannot be parameterized in the server's configuration file. A summary of the results obtained in the scalability evaluation tests is shown in Table 17.

The evaluation of the system was designed in order to test the server in terms of (1) how many users will be interested and subscribed to a certain context information, (2) how many context providers will be publishing information to the context server, (3) how often context information updates need to be retrieved by the users, and (4) what is the rate of context information updates compared to the distribution time needed for

information to reach the watcher(s). After the evaluation we have gained some insights into how the above factors affect the performance of the system.

In asynchronous mode the number of watchers subscribed to an event affects the subscription response time, because this means more information stored at the database. The more information about watchers that is stored in the server's database, the greater the increase in the subscription response time (on average 9.25 ms per 100 watchers subscribed). As well, in asynchronous mode the notification response time is affected by the number of watchers subscribed at the context server. Notifications are sent to all the watchers subscribed to an event sequentially, so the whole notification process takes about 35 ms more for every 100 watchers subscribed to the same context information. In synchronous mode the number of watchers affects the system, mainly when requests arrive at a faster rate than the processing time of the server (1.44 ms per request), because messages have to wait in the queue to be processed.

Table 17. Summary of test results

	SUBSCRIBE messages	REQUEST messages	PUBLISH messages	SUBCRIBE/REQUEST and PUBLISH messages
Maximum number of messages that can be processed in a short period of time	400	400	200	150 SUBSCRIBE/REQUEST and 150 PUBLISH
Maximum Polling Rate accepted by the server (Synchronous)	1 Request per Second			
Server's Notification rate (Asynchronous)	2850 Notifications per second			

The number of context providers publishing information affects the time the server needs to process and store information from PUBLISH messages. This time increases linearly with respect to the amount of information about publications in the database, on average 500 μs per 100 PUBLISH message already stored. As mentioned earlier, the server notifies subscribed watchers about PUBLISH messages only every second, so the increase in processing PUBLISH messages may be not significant for applications.

In terms of how often context information can be distributed to watchers, in synchronous mode the maximum polling rate is 1 second. In asynhronous mode the maximum rate at which the watchers can receive context updates is also one second. However, in the asynchronous case this parameter is configurable through the timer_interval parameter, but 1 second is the minimum time supported. Most context information changes slower, however SIP/SIMPLE is not suitable for distributing context information when it changes at high rates (>1 change per second).

Finally, comparing the context information distribution time with the dynamics of context information, we can see that the context information distribution time in asynchronous mode may be an issue for some applications. In asynchronous mode, the notification of a change in context information may take more than 4 seconds (when there are 10000 or more users subscribed to a given item of context information), in this cases if this delay is an issue, it may be convenient to switch to a polling (synchronous) mode that is faster, but of course it will involve the transfer of more messages. The switching threshold will depend mainly in the application latency requirements.

6. Suggestions for Distributing Context Information

After evaluating the context server using the implemented load generator, this chapter focuses on offering some suggestions about how to distribute context information, especially when it is better to use the synchronous mode or the asynchronous mode. These recommendations are based in the insights gained during the evaluation phase of the thesis.

The evaluation of the system showed that the SIP Express router is highly scalable and stable; it can handle hundreds of SIP messages per second running on Linux Ubuntu on a PC with Intel Pentium 4 at 2.6 GHz CPU and 1 GB of RAM. Different SIP messages are processed and responded to in a short period of time; however when the processing involves database operations, this time depends on the amount of information stored in the database.

The asynchronous and synchronous mode of context information distribution have different characteristics, both methods have their advantages and disadvantages. Their usage depends primarily on the characteristics and nature of the context information to be transmitted, on the requirements of specific applications, and user's mobility.

The main advantage of the asynchronous mode is that after the subscription and its immediate notification, further notifications are decoupled from the watcher and depends only on changes in the status of events. In this mode after the subscription, the delivery of context information when a change occurs involves the exchange of only a NOTIFY message and its OK response, every time the status of an event changes. In contrast the synchronous mode involves the exchange of context information each of which requires 4 messages (REQUEST, OK, NOTIFY, and OK).

The synchronous mode's main advantage is the handling of REQUEST messages, which does not require operations in the database, so the response time is constant and independent of the number of watchers and contextities using the system.

Different types of context information have different requirements for its distribution, mainly because each type of context information has different dynamics. Context information with a high update rate has to be delivered in a short period of time; otherwise this information may be inaccurate.

Different context-aware applications have different requirements in terms of the context information that should be retrieved. Thus, for some applications it will be critical to receive the information in a short period of time; while other applications may tolerate bounded delays in the distribution of context information. Some applications will need to know the status of the context of the users nearly all the time, while others will only need this knowledge just at certain moments.

Finally, we have to consider the user's mobility; when a user is moving rapidly and the context information retrieved has to do with certain position, then the user wants to get the information as fast as possible, otherwise this information might not be useful.

Using the context distribution method proposed in this thesis, context information can be retrieved asynchronously or it can be fetched or polled synchronously. The mode of context distribution mainly impacts the network's traffic and the context retrieval response time, advantages and disadvantages with respect to these factors are summarized in the next subsections.

6.1 Network Traffic

For certain types of networks, especially those where the service provider charges per byte transmitted (e.g. GPRS), the traffic in the network is an important factor for deciding how to distribute context information. Also in low bandwidth networks it is important to consider the traffic in the network to avoid network congestion.

In terms of the traffic generated when delivering context information, the asynchronous mode involves the transmission of fewer packets and fewer bytes. In asynchronous mode the distribution of context information only involves a NOTIFY message and its OK response (≈1527 bytes). On the other hand, in synchronous mode each time an application fetches context information 4 packets (1 SUBSCRIBE, 1 NOTIFY and 2 OK messages, for a total of ≈2747 bytes.) need to be exchanged between the user agent and the context server. In order to reduce the number of bytes and messages exchanged in synchronous mode it will be beneficial to piggyback the context information in the body message of the OK response (The message acknowledging the REQUEST message). The piggybacking of context information will reduce one message exchanged per request.

In asynchronous mode the notification is sent only after a change in the status of the state of an event, so the watcher always has the current state of the contextity. In synchronous mode, to have information up to date, the watcher has to poll for information frequently and this will mean more traffic in the network.

6.2 Latency

For certain applications, especially when the context information is highly dynamic, the time for receiving the context information is an important factor. In terms of the time response, the main constraints in asynchronous mode are the timer_interval parameter and the scheduling algorithm of the server. The timer_interval timer determines how often the server notifies about changes in the status of subscribed events, having the minimum value of one second. After a publication, the notification to watchers may take up to one second, due to this timer. Another drawback of the

asynchronous mode is the fixed notification scheduling of the server. The watchers are always notified in the same order; this means that always some watchers will be more penalized in terms of latency. In order to avoid this, it may be possible to find the section in the code of SER that handles the notifications and change it to randomize the notification scheduling.

The synchronous mode for distributing context information has the main advantage that retrieving information in this mode does not require the performance of database operations, so the request response time is independent of the number of entries stored at the database. The request response time, however depends mainly on the number of messages waiting in the queue. When messages arrive in a short period of time, the response time of messages will increase due to the time that messages have to wait in the buffer.

In terms of response time, the request response time in synchronous mode is mostly shorter than the notification response time in asynchronous mode. Comparing the case with only one watcher in the system, in synchronous mode the average request response time was 1.65 milliseconds, while in the asynchronous mode the average notification time was 537 milliseconds, however the minimum and maximum notification times were 56.416 ms and 932.854 ms respectively. The main reason for the notification time variability is the timer_interval parameter, a random delay of 0 to 1 seconds occurs between the PUBLISH message and the notification process. When we consider multiple watchers the notification time in asynchronous mode may be faster for some watchers, but slower for others when compared with the request response time in synchronous mode, depending on the delay between the PUBLISH message and the first notification. Taking into account the average values from our previous tests, we can conclude that the synchronous mode is faster. As an example, we show in Table 18 the values of the response time when we had 400 watchers using the system. As it can be seen the synchronous mode is faster especially when the requests occurred at a sustained rate (350 requests per second). When dealing with the notification of a big amount of users (around 10,000) the notification response time may be in the order of 4 or 5 seconds for some users, this time may be not tolerated by certain applications, because context information may be out of date. In comparison in synchronous mode there is no random delay related and the request response time is in the order of milliseconds.

Having multiple watchers affects primarily the notification response time because the notifications are sent sequentially to all subscribed watchers. Also the subscription response time in asynchronous mode increases in relation with the number of watchers already subscribed in the server. This may not be significant for many applications, especially if the subscriptions have a large expiration time. The subscription process occurs just once and it has to be repeated only after the expiration of the subscription (if the user is still interested in the context information). In the synchronous mode the number of requests and publications affects the time the messages have to wait in the queues before being processed, especially if the requests

and/or publications arrive within the same period of time. Having multiple contextities publishing context information mainly affects the time required to store and process this information that is dependent on the amount of information stored in the contextities' tables of the database.

Table 18. Comparison between request and notification response time with 400 watchers

	Asynchronous (ms)	Synchronous (ms) (400 requests in a burst)	Synchronous request (ms) (Sustained Rate - 350 requests per second)
Average	569	101	1.911
Minimum	501	2.15	0.852
Maximum	640	188	6.96

Each mode of context distribution has advantages and disadvantages, in the following subsections we summarize when is more suitable to use one or another.

6.3 Asynchronous mode

The asynchronous mode for distributing context information is especially suitable when applications need to get the context information just after a change in the status of such information. This method is particularly beneficial when the bandwidth and network traffic are critical, the delivery of context information in this mode contains less overhead than the delivery in synchronous mode.

The SUBSCRIBE/NOTIFY scheme is appropriate when context information changes sporadically and an application needs to be aware of context changes; this may be the case of the user's profile or the presence of smoke in the airport emergency scenario presented in Chapter 2. Using asynchronous context distribution in this case is optimal in terms of bytes transmitted; a frequent polling in this case will result in an increase of the network's traffic.

Due to the connection between a notification and a change in the status of the user's context, applications will have knowledge of context changes almost in real time; unless in events with a large amount of watchers subscribed (more than 5000), when the delivery of context information may take several seconds.

6.4 Fetching Context Information

Many applications only need to retrieve the user's context information, only once before performing an action. For this application the synchronous mode will be appropriate. For example, considering the airport's emergency scenario presented in

Chapter 2, a communication application will only need to retrieve the available bandwidth before startup; in order to decide if voice or text based communications will be preferred.

The main benefit of distributing context information using a synchronous mode is the short time period between the request and getting a response, so the received information will be up to date and the system can perform the pertinent actions.

6.5 Polling for Context Information

A variant in synchronous mode is periodically fetching the information, known as polling. Polling for retrieving context information may be suitable when the context information changes very quickly and applications do not need updates so often. This may be the case of the location of the users in the airport's emergency scenario. The location of the different workers may be changing every second, however for the application it is sufficient to know the approximate location of the users, especially if there is no emergency. If the application polls every 30 seconds for the location of the workers, the system will have a very good idea of their location. Moreover, when there is an emergency the polling rate can be increased in order to get their "real time" position.

An advantage of using a REQUEST/REPLY method for retrieving context information is that the time response is shorter than the one in asynchronous mode. When the time response has higher value, even at the expense of bandwidth and network traffic, polling at a high rate (1 request per second) may deliver context information quicker than the asynchronous mode.

How this fits in our Emergency Scenario?

Taking into account the recommendations we have derived from the evaluation of SIP/SIMPLE and the scenario we proposed in Chapter 2, now we can decide how to distribute context information. The distribution mode for each type of context information that needs to be shared is summarized in Table 19.

For sharing the location of a user with the control center an asynchronous distribution will be suitable, if the control center desires to track the position of workers with a high accuracy. By using asynchronous mode the control center will be notified of all the changes in the worker's position. In the case that the control center or other workers only want to get the position of a user at certain time, then a synchronous request may be appropriate.

For the location of fire, user's profile, presence information, current task, task completion, progress, and the presence of smoke or fumes an asynchronous distribution of information will be beneficial, mainly because the distribution of information will be

decoupled from the receivers and will only depend on changes in the context, this will result in having less network traffic.

Finally retrieving the available bandwidth of a worker may be useful to the system in order to decide which kind of communication software (voice, video or text based) should be started for enabling communication between the control center and a user. For this purpose the available bandwidth should be retrieved on a request basis just before deciding which communication software to start.

Table 19. Distribution mode for the Emergency Usage Case

Context	How Often?
Location of user	When it changes or on request (synchronous and asynchronous)
Location of fire	When fire is detected (asynchronous)
User´s Profile	When a user makes a change (asynchronous)
Presence Information	When a change in presence status occurs (asynchronous)
Current Task	When a change in task occurs (asynchronous)
Task completion progress	When a threshold is reached (such as every 10%) (aysnchronous)
Available bandwidth	On Request (synchronous)
Temperature	When a sudden increase of temperature occurs (asynchronous)
Presence of smoke, fumes, etc.	When smoke/fumes are detected (asynchronous)

7. Conclusions and Future Work

7.1 Conclusions

Context-aware applications aim to exploit the user's context information, in order to adapt their behavior to the user's current situation and assist him/her in the daily tasks. Most of the applications developed until now use only the context implicitly sensed by the device; however, context information may be produced outside of the local device causing the need for distributing context information among applications (that may be running on different devices distributed on a network).

This thesis studied context distribution based on the Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions (SIP SIMPLE). This context distribution method enables the delivery of context information, both synchronously and asynchronously. In synchronous mode, this method uses a SUBSCRIBE message with expiration equals to zero, resulting in an immediate notification with the current context information. The asynchronous mode exploits the event notification mechanisms of SIP-SIMPLE. In both distribution modes context information is delivered via a NOTIFY message. Context information is contained in the NOTIFY message body in a RPID document. RPID was chosen for representing context information in order to distribute context in a standard format.

In synchronous mode a NOTIFY message containing the context information is sent just after the request message. In asynchronous mode, the user also receives the current status of the context information immediately, however he or she will aslo be notified about every change about the status of the subscribed context information. For both cases, the context information status is reported to the server using a PUBLISH message.

The context distribution component is based on the SIP Express Router and its presence module. A load generator was developed in order to evaluate the performance of the server in terms of response time and scalability. The evaluation of the context distribution component included different evaluation scenarios involving multiple users interested in context information (i.e. watchers) or/and multiple users publishing context information (i.e. contextities).

The evaluation revealed that the server is highly scalable and can respond to hundreds (up to 600) of synchronous and asynchronous requests per second. The tests performed showed that the time required for processing subscriptions in asynchronous mode grows linearly in relation with the amount of information in the database from previous subscriptions. On the contrary, the time required for responding synchronous requests is constant, unless the messages have to wait in the buffer queues, when several packets arrive in a short period of time. The time required for processing a PUBLISH message also increases linearly in relation to the amount of information stored in the database from previous publications. The notification scheduling in asynchronous mode

follows a fixed pattern and all notifications are sent at a constant rate (on average 2850 notifications per second). Considering average times for different load conditions, the delivery of context information using the synchronous mode is faster than when using asynchronous mode, mainly because in asynchronous mode a timer determines when to send notifications to subscribed users (watchers).

One important limitation of using SIP/SIMPLE for distributing context information is that the maximum polling rate and the maximum update rate supported by the server is 1 second, so this approach is not suitable when context information is changing more than one time per second.

Based on application's requirements (e.g. accepted latency, how often they need context information, and the nature of context information), for some it will be better to use synchronous mode, while for others it will be better to use asynchronous. In order to decide which mode is more suitable for certain application we needed to analyze the application's needs and requirements, as well as the dynamics of the context information.

This thesis was written as part of the MUSIC project, resulting in recommendations to middleware developers on how to implement the context distribution component in order to fulfill with the needs and requirements of the project. The evaluation performed in this thesis showed that SIP is an appropriate protocol for transporting context information; moreover SIP infrastructure is the foundation for session initiation and presence support in desktop, mobile, and server platforms and will be widely deployed in future mobile devices. However, in order to fulfill the requirements and features proposed in the architecture design of the MUSIC project several open issues still need to be addressed. In the next section I emphasize in some suggestions for future work that can build upon this thesis.

7.2 Future Work

Some of the suggestions for extending and enhancing the work of this thesis are:

1) Distributing Context information within a Peer to Peer environment

The MUSIC middleware aims to run in infrastructure based environments, as well as in *ad hoc* environments, where the distribution component cannot rely on a SIP context server. Exploiting the distributed nature of MUSIC networks it would be interesting to develop a distribution component based on Peer to Peer SIP [51] (P2PSIP). Recently, an IETF group has started working in the standardization of this Peer to Peer version of SIP, however this work is not related to context information, but it might be adopted in the MUSIC project for sharing context. The work on this protocol does not focus on SIP SIMPLE; however, we see a need for developing SIP SIMPLE on top of peer to peer networks for enabling event notification mechanisms.

2) Design of a layer facilitating interaction between applications and the context distribution component

In order to ease the communication between the context distribution component and applications a layer in charge of controlling the distribution of context information is needed. This layer, based on applications requirements and the nature of context, will decide how to retrieve context. This software entity will keep track of the system infrastructure conditions, such as response time, network traffic, etc. in order to select between context distribution modes in order to comply with the mentioned requirements. As well, this layer may keep track of the status of context information subscriptions in order to renew expired subscriptions if it is needed.

3) Privacy Issues

When dealing with the information and context of a user, such as the user's location, privacy may be critical. In order to deal with this, some authorization mechanisms may be implemented in order to set policies determining how to distribute context information. For instance, a user may only want to disclose his or her location to close relatives. Policies may be established for distributing different granularities of the same context information to different types of users; these policies may be based on social relations. For example a user may set that his or her family will receive the exact location, friends will receive a less granular location, and the information may be disclosed to other group of users. It will be important to consider the delay caused by processing these constraints and analyze if this is a feasible feature in order to fulfill with the latency requirement of applications.

4) A Graphical User Interface for the load generator

In order to make the load generator easier to use and configure parameters of the evaluation a graphical user interface may be developed.

5) Changes to Context server implementation

In order to improve the performance of context information distribution using SIP/SIMPLE it is required to change the implementation of SER. In synchronous mode changes in order to piggyback the context information in the body message of the OK message are required. On the other hand, in asynchronous mode the SER has to be change for responding messages before performing database operations, with this modifications the time response will be decoupled from the database.

References

[1] Bill N. Schilit, Norman Adams, and Roy Want, Context Aware Computing Applications, IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, *CA*, US, December, 1994.

[2] Mark Weiser, Some computer science issues in ubiquitous computing, ACM SIGMOBILE Mobile Computing and Communications, 1999, Review, Vol. 3, page 12.

[3] IST project 035166, Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environment project, available at www.ist-music.eu, last accessed: March 27, 2008.

[4] The MUSIC Consortium, MUSIC - Annex 1 – Description of work, September 12, 2006.

[5] The MUSIC Consortium, D4.2 System Design of the MUSIC Architecture, November 2007, available at http://www.ist-music.eu/MUSIC/results/music-deliverables/techreportreference.2007-12-10.3633513999

[6] National Fire Protection Association (NPFA), NPFA 72: National Fire Alarm Code, 2007, Quincy, Massachusetts, USA.

[7] Appear Networks, Southwest Florida International Airport, Context-Aware Wireless Emergency Response, available at http://www.appearnetworks.com/IMG/pdf/Southwest_Florida_International_Airport.pdf, Last accessed March 25, 2008.

[8] Alisa Devlic, Alan Graf, Alessandro Barone, Paolo, Mamelli, and Athanasios Karapantelakis, Evaluation of context distribution methods via Bluetooth and WLAN: Insights gained while examining Battery Power Consumption, accepted in Mobiquitous 2008, Dublin Ireland.

[9] B. Schilit and M. Theimer, Disseminating Active Map Information to Mobile Hosts,. IEEE Network, IEEE Computer Society, 22-32, October 1994.

[10] Anind Dey and Gregory Abowd, Towards a Better Understanding of Context and Context-Awareness, GVU Technical Report GIT-GVU-99-22, College of Computing, Georgia Institute of Technology, 1999, Atlanta, Georgia.

[11] The MUSIC consortium, Initial research results on methods, languages, algorithms and tools to modeling and management of context, Deliverable 2.2, December, 2007, available at http://www.ist-music.eu/MUSIC/results/music-deliverables/techreportreference.2008-02-21.2969892968 last accessed: March 30, 2008.

[12] R. Want, A. Hopper, V. Falcao and J. Gibbons, The Active Badge Location System, ACM Transactions on Information Systems, 1992, Vol 10, pp. 91-102.

[13] J. Pascoe, Adding Generic Contextual Capabilities to Wearable Computers, 2nd.International Symposium on Wearable Computers, October, 1998, Pittsburgh, Pennsylvania, U. S. , pp. 92-99.

[14] Wei Li, Towards a Person-Centric Context Aware System, Licentiate Thesis, Department of Computer and System Sciences, Stockholm University and Royal Institute of Technology, March 2006, Kista, Sweden.

[15] Roy Want, Bill Schilit, et al., The PARCTAB Ubiquitous Computing Experiment, Technical Report CSL-95-1, Xerox Palo Alto Research Center, March 1995.

[16] Gregory Abowd, et al. Cyberguide. A mobile context-aware tour guide, Mobile Computing and networking: selected papers from MobiCom 96, ACM, Vol. 3, pp. 421-433, October 1997.

[17] Daniel Salber, Anind Dey, and Gregory Abowd, The Context Toolkit: Aiding the Development of Context-Enabled Applications, Proceedings of Conference fon Human Factors in Computing Systems (CHI´99), ACM press, Pittsburgh, Pennsylvania, U. S., May 15-20, 1999.

[18] H. Sinreich and A. Johnston, Internet Communications Using SIP: delivering VoIP and multimedia services with session initiation protocol, John Wiley and Sons, New York, U.S., 2001.

[19] J. Kuthan and D. Sisalem, SIP: More Than You Ever Wanted to Know About, IPTEL, March 2007, available at http://www.iptel.org/files/sip_tutorial.pdf.

[20] M. Handley, et al., SIP: Session Initiation Protocol, RFC 2543, IETF, March, 1999 available at http://tools.ietf.org/html/rfc2543

[21] Dana Pave and Dirk Trossen, Context Provisioning and SIP Events, Workshop on Context Awareness at ACM SIGMOBILE, Boston Massachusetts, U. S., June 6, 2004.

[22] A. Roach, SIP-Specific Event Notification RFC 3265, IETF, June, 2002, available at http://www.ietf.org/rfc/rfc3265.txt.

[23] J. Rosenberg, A Presence Event Package for the Session Initiation Protocol (SIP), RFC 3856, IETF, August, 2004, available at http://www.ietf.org/rfc/rfc3856.txt.

[24] Gerald Q. Maguire Jr., Lecture notes for Practical Voice Over IP (VoIP): SIP and related protocols, School of Information and Communication Technology (ICT), Royal Institute of Technology, March, 2007.

[25] A. Niemi, Session Initiation Protocol (SIP) Extension for Event State Publication, RFC 3903, IETF, October 2004, available at http://www.ietf.org/rfc/rfc3903.txt.

[26] M. Day, et al., A model for Presence and Instant Messaging, RFC 2778, IETF, February 2000, available at http://www.ietf.org/rfc/rfc2778.txt.

[27] H. Sugano, et al., Presence Information Data Format (PIDF), RFC 3863, IETF, August 2004, available at http://www.ietf.org/rfc/rfc3863.txt.

[28] SIP Express Router (SER) in Wikipedia available at http://en.wikipedia.org/wiki/SIP_Express_Router_%28SER%29.

[29] IPTEL's SER Homepage, available at www.iptel.org/SER, last accessed: March 25, 2008.

[30] M. Zarifi, A Presence Server for Context-aware Applications, Master thesis, School of Information and Communication Technology, KTH, December 17, 2007.

[31] SIMPLE RFC´s, SIP for Instant Messaging and Presence Leveraging Extensions, available at http://www.voip-telephony.org/rfc/simple.

[32] A. Georgescu, Presence, SIP beyond VoIP, AG-Projects, available at http://www.mediaplaza.nl/uploaded/FILES/seminars/2007/OS%20VoIP%20telefonie/20070531-SIPSIMPLE.pdf, Last accessed March 25, 2008.

[33] H. Schulzrinne, et al., RPID: Rich Presence Extensions to the Presence Information Data Format (PIDF), RFC 4480, IETF, July 2006, available at http://www.ietf.org/rfc/rfc4480.txt

[34] J. Rosenberg, A Data Model for Presence, RFC 4479, IETF, July, 2006, available at http://www.ietf.org/rfc/rfc4479.txt

[35] J. Peterson, Common Profile for Presence (CPP), RFC 3859, IETF, August, 2004, available at http://www.ietf.org/rfc/rfc3859.txt

[36] A. Devlic and I. Podnar, Location-aware Content Delivery Service Using Publish/Subscribe, Telecommunications and Mobile Computing (tcmc 2003), Workshop on Nomadic Data Services and Mobility, OVE-Mediacenter, Austrian Electrotechnical Associations (OVE), Graz, Austria, March, 2003.

[37] A. Devlic and E. Klinstkog, Context retrieval and distribution in mobile distributed environment, Third Workshop of Context Awareness for proactive Systems, Guildford, UK, June, 2007.

[38] IST project MIDAS, Middelware Platform for Developing and Deploying Advanced Mobile Services, available at www.ist-midas.org last accessed April 2, 2008.

[39] Manuel Görtz, Ralf Ackermann and Ralf Steinmetz, Enhanced SIP Communication Services by Context Sharing, Euromicro Conference 2004, IEEE, February 13, 2004, La Coruña, Spain.

[40] Wei Li, A Service Oriented SIP infrastructure for Adaptive and Context-Aware Wireless Services, Proceedings of the 2nd International Conference on Mobile and Ubiquitous Multimedia, Norrköping, Sweden, Dec 10-12, 2003.

[41] J. Rosenberg, et. al. SIP: Session Inititiation Protocol, RFC 3261, IETF, june, 2002, available at http://www.ietf.org/rfc/rfc3261.txt

[42] Paul Hazlett, Simon Miles, and Greger Teigre, SER- Getting Started, IPTEL, available at http://siprouter.teigre.com/doc/gettingstarted/, last accessed April 3, 2008.

[43] Vaclav Kubart, SER presence handbook, IPTEL, available at http://www.iptel.org/~vku/presence_handbook/index.html#id2514614, last accessed April 3, 2008.

[44] Naccarato Guisseppe, Introducing Nonblocking sockets, O'Reilly ON Java.com, available at http://www.onjava.com/pub/a/onjava/2002/09/04/nio.html, last accessed April 4, 2008.

[45] Openser, The Open Source Sip Server, available at www.openser.org, last accessed April 10, 2008.

[46] Thomas Strang and Claudia Linhoff, A Context Modeling Survey, UbiComp 1st International Workshop on Advanced Context Modeling, Reasoning and Management, Nottingham, September 7-10, 2004, pp 34-41

[47] WWW Consortium, CC/PP Information Page, available at www.w3.org/Mobile/CCPP, last accessed April 11, 2008.

[48] Q. Sheng and B. Benatallah, ContextUML: a UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services, International Conference on Mobile Business (ICMB'05) July 11-13, 2005, Sydney Australia.

[49] H. Chen, T. Finin, and A Joshi, A Using OWL in a Pervasive Computing Broker, Proceeding of Workshop on Ontologies in Open Agent systems, Melbourne, Australia, July, 2003.

[50] Wireshark Network Protocol Analyzer available at www.wireshark.org last accessed April 15, 2008.

[51] D. Bryan, et al, Concepts and Terminology for Peer to Peer SIP, Internet Draft, November 15, 2007, expires May 18, 2008, IETF, available at http://www.p2psip.org/drafts/draft-ietf-p2psip-concepts-01.txt

[52] E. Halepovic and R. Deters, The costs of using JXTA, Third International Conference on Peer-to_Peer Computing, Linkoping, Sweden, IEEE Computer Society, (2003).

Appendix A

SER Configuration file ser.cfg

```
# Ser.cfg file
# configured as a Context server
debug=3         # debug level (cmd line: -ddddddddd)
check_via=no # (cmd. line: -v)
dns=no          # (cmd. line: -r)
rev_dns=no # (cmd. line: -R)
#listen=192.168.1.103
port=5060
children=2
# ------------------ module loading ----------------------------------
# Uncomment this if you want to use SQL database
loadmodule "/base/ser/directory/lib/ser/modules/sl.so"
loadmodule "/base/ser/directory/lib/ser/modules/avp.so"
loadmodule "/base/ser/directory/lib/ser/modules/avpops.so"
loadmodule "/base/ser/directory/lib/ser/modules/tm.so"
loadmodule "/base/ser/directory/lib/ser/modules/rr.so"
loadmodule "/base/ser/directory/lib/ser/modules/maxfwd.so"
loadmodule "/base/ser/directory/lib/ser/modules/usrloc.so"
loadmodule "/base/ser/directory/lib/ser/modules/registrar.so"
loadmodule "/base/ser/directory/lib/ser/modules/textops.so"
loadmodule "/base/ser/directory/lib/ser/modules/mysql.so"
loadmodule "/base/ser/directory/lib/ser/modules/dialog.so"
loadmodule "/base/ser/directory/lib/ser/modules/rls.so"
loadmodule "/base/ser/directory/lib/ser/modules/pa.so"
loadmodule "/base/ser/directory/lib/ser/modules/presence_b2b.so"
loadmodule "/base/ser/directory/lib/ser/modules/uri.so"
loadmodule "/base/ser/directory/lib/ser/modules/uri_db.so"
loadmodule "/base/ser/directory/lib/ser/modules/domain.so"
loadmodule "/base/ser/directory/lib/ser/modules/fifo.so"
loadmodule "/base/ser/directory/lib/ser/modules/xmlrpc.so"
loadmodule "//base/ser/directory/lib/ser/modules/xlog.so"
# Uncomment this if you want digest authentication
# mysql.so must be loaded !
loadmodule "/base/ser/directory/lib/ser/modules/auth.so"
loadmodule "/base/ser/directory/lib/ser/modules/auth_db.so"
loadmodule "/base/ser/directory/lib/ser/modules/msilo.so"
# ----------------- setting module-specific parameters ---------------
modparam("msilo","use_contact",0)
modparam("msilo","expire_time",7200)
# -- auth params --
# Uncomment if you are using auth module
modparam("auth_db", "calculate_ha1", yes)
# If you set "calculate_ha1" parameter to yes (which true in this config),
# uncomment also the following parameter)
modparam("auth_db", "password_column", "password")
# -- rr params --
# add value to ;lr param to make some broken UAs happy
modparam("rr", "enable_full_lr", 1)
modparam("rls", "min_expiration", 200)
modparam("rls", "max_expiration", 300)
modparam("rls", "default_expiration", 300)
modparam("rls", "auth", "none")
modparam("rls", "xcap_root", "http://localhost/xcap")
modparam("rls", "reduce_xcap_needs", 1)
modparam("rls", "db_mode", 1)
modparam("rls", "db_url", "mysql://ser:heslo@localhost:3306/ser")
modparam("pa", "use_db", 1)
# allow storing authorization requests for offline users into database
modparam("pa", "use_offline_winfo", 1)
# how often try to remove old stored authorization requests
modparam("pa", "offline_winfo_timer", 600)
# how long stored authorization requests live
modparam("pa", "offline_winfo_expiration", 600)
modparam("pa", "db_url", "mysql://ser:heslo@localhost:3306/ser")
```

```
# mode of PA authorization: none, implicit or xcap
modparam("pa", "auth", "none")
modparam("pa", "auth_xcap_root", "http://localhost/xcap")
# do not authorize watcherinfo subscriptions
modparam("pa", "winfo_auth", "none")
# use only published information if set to 0
modparam("pa", "use_callbacks", 1)
# dont accept internal subscriptions from RLS, ...
modparam("pa", "accept_internal_subscriptions", 0)
# maximum value of Expires for subscriptions
modparam("pa", "max_subscription_expiration", 600)
# maximum value of Expires for publications
modparam("pa", "max_publish_expiration", 120)
# how often test if something changes and send NOTIFY
modparam("pa", "timer_interval", 1)
# route for generated SUBSCRIBE requests for presence
modparam("presence_b2b", "presence_route", "<sip:127.0.0.1;transport=tcp;lr>")
# waiting time from error to new attepmt about SUBSCRIBE
modparam("presence_b2b", "on_error_retry_time", 60)
# how long wait for NOTIFY with Subscription-Status=terminated after unsubscribe
modparam("presence_b2b", "wait_for_term_notify", 33)
# how long before expiration send renewal SUBSCRIBE request
modparam("presence_b2b", "resubscribe_delta", 30)
# minimal time to send renewal SUBSCRIBE request from receiving previous response
modparam("presence_b2b", "min_resubscribe_time", 60)
# default expiration timeout
modparam("presence_b2b", "default_expiration", 3600)
# process internal subscriptions to presence events
modparam("presence_b2b", "handle_presence_subscriptions", 1)
modparam("usrloc", "db_mode", 1)
modparam("domain", "db_mode", 1)
modparam("domain|uri_db|acc|auth_db|usrloc|msilo", "db_url",
"mysql://ser:heslo@localhost:3306/ser")
modparam("fifo", "fifo_file", "/tmp/ser_fifo")
# ------------------------ request routing logic -------------------
# main routing logic
route{
   # XML RPC
   if (method == "POST" || method == "GET") {
      create_via();
      dispatch_rpc();
      break;
   }
   # initial sanity checks -- messages with
   # max_forwards==0, or excessively long requests
   if (!mf_process_maxfwd_header("10")) {
      sl_send_reply("483","Too Many Hops");
      break;
   };
   if (msg:len >= max_len ) {
      sl_send_reply("513", "Message too big");
      break;
   };
   # we record-route all messages -- to make sure that
   # subsequent messages will go through our proxy; that's
   # particularly good if upstream and downstream entities
   # use different transport protocol
   if (!method=="REGISTER") record_route();
   # subsequent messages withing a dialog should take the
   # path determined by record-routing
   if (loose_route()) {
      # mark routing logic in request
      append_hf("P-hint: rr-enforced\r\n");
     route(1);
     break;
  };
  # if the request is for other domain use UsrLoc
  # (in case, it does not work, use the following command
  # with proper names and addresses in it)
    if (uri=~"192.168.1.103") {
    if (!lookup_domain("To")) {
```

```
        xlog("L_ERR", "Unknown domain to: %tu from: %fu\n");
        route(1);
        break;
     }
        if (method=="SUBSCRIBE") {
            log(1,"Subscribe\n");
            if (t_newtran()) {
                   log(1,"Register\n");
                   handle_subscription("registrar");
               log(1,"Done\n");
               };
            break;
           };
        if (method=="PUBLISH") {
            log(1,"Publish\n");
             if (!t_newtran()) {
                 log(1,"newtran error\n");
               sl_reply_error();
               };
            handle_publish("registrar");
               log(1,"publish handled\n");
            break;
           };
     # get user (common for all other messages than SUBSCRIBE)
     if (!lookup_user("To")) {
        # log(1, "Unknown user - message should be forwarded?");
       # break;
       append_hf("P-hint: unknown user\r\n");
       route(1);
       break;
    }
    if (method=="NOTIFY") {
        if (!t_newtran()) {
           log(1, "newtran error\n");
           sl_reply_error();
          break;
        };
       # handle notification sent in internal subscriptions (presence_b2b)
        if (!handle_notify()) {
            t_reply("481", "Unable to handle notification");
        }
        break;
     };
    if (method=="MESSAGE") {
       if (authorize_message("http://localhost/xcap")) {
           # use usrloc for delivery
           if (lookup("location")) {
              log(1, "Delivering MESSAGE using usrloc\n");
              t_on_failure("1");
              if (!t_relay()) {
                  sl_reply_error();
              }
              break;
           }
           else {
              # store messages for offline user
              xlog("L_ERR", "MSILO: storing MESSAGE for %tu\n");
              if (!t_newtran()) {
                 log(1, "newtran error\n");
                 sl_reply_error();
                 break;
              };
              # store only text messages NOT isComposing... !
              if (search("^(Content-Type|c):.*application/im-iscomposing\+xml.*"))
{
                  log(1, "it is only isComposing message - ignored\n");
                t_reply("202", "Ignored");
                break;
             }
             if (m_store("0", "sip:127.0.0.1")) {
  #               log(1, "MSILO: offline message stored\n");
```

```
                if (!t_reply("202", "Accepted")) {
                   sl_reply_error();
                };
             } else {
                log(1, "MSILO: error storing offline message\n");
                if (!t_reply("503", "Service Unavailable")) {
                   sl_reply_error();
                };
             };
             break;
          }
          break;
       }
       else {
          # log(1, "unauthorized message\n");
          sl_reply("403", "Forbidden");
       }
       break;
    }
    if (method=="REGISTER") {
       # uncomment this if you want to authenticate REGISTER request
       if (!www_authenticate(" 192.168.1.103", "credentials")) {
           www_challenge( "192.168.1.103", "0");

          break;
       };
       save("location");
       # dump stored messages - route it through myself (otherwise routed via DNS!)
       if (m_dump("sip: 127.0.0.1")) {
          xlog("L_ERR", "MSILO: offline messages for %fu dumped\n");
       }
       break;
    };
    # native SIP destinations are handled using our USRLOC DB
    if (!lookup("location")) {
       sl_send_reply("404", "Not Found");
       break;
    };
   };
# append_hf("P-hint: usrloc applied\r\n");
   route(1);
}
route[1]
{
   # send it out now; use stateful forwarding as it works reliably
   # even for UDP2TCP
   if (!t_relay()) {
      sl_reply_error();
   };
}
failure_route[1] {
   # forwarding failed -- check if the request was a MESSAGE
   if (!method=="MESSAGE") { break; };
   log(1, "MSILO: MESSAGE forward failed - storing it\n");
   # we have changed the R-URI with the contact address, ignore it now
   if (m_store("0", "")) {
      t_reply("202", "Accepted");
   } else {
      log(1, "MSILO: offline message NOT stored\n");
      t_reply("503", "Service Unavailable");
   };
}
   #   if ( pthread_join ( SendNotifyClient, NULL ) )
    #  {
     # printf("error joining thread.");
      #abort();
      #}
#exit(0);
#}
```

Appendix B

Load Generator

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;  // bug? redundant with previous one??
import java.util.*;
import java.nio.charset.Charset;


public class NonBlock {
      static String test;
      public static void main(String[] args){
            int port=54000;
            int portpub=64001;
            int SIPport=5060;
            String server="192.168.100.234";
            String client="192.168.100.53";
            int watchers=1500;
            int publishers=1;
            String subs_exp="0";
            String req_exp="0";
            String location="<?xml version='1.0' encoding='UTF-
8'?><presence xmlns='urn:ietf:params:xml:ns:pidf'
xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
entity='sip:Alice@192.168.100.153'><tuple
id='t8a130d03'><status><basic>open</basic><location><description>Appea
r</description><room>Eulab</room><floor>1</floor><coordinates><latitud
e>123213</latitude><longtitude>47382145</longtitude></coordinates></lo
cation></status><note>location</note><contact
priorit='0.8'>Carlos</contact></tuple></presence>";
            String location2="<?xml version='1.0' encoding='UTF-
8'?><presence xmlns='urn:ietf:params:xml:ns:pidf'
xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
entity='sip:Alice@192.168.100.153'><tuple
id='t8a130d03'><status><basic>open</basic><location><description>Kista
</description><room>Galleria</room><floor>1</floor><coordinates><latit
ude>1</latitude><longtitude>4</longtitude></coordinates></location></s
tatus><note>location</note><contact
priorit='0.8'>Carlos</contact></tuple></presence>";
            String location3="<?xml version='1.0' encoding='UTF-
8'?><presence xmlns='urn:ietf:params:xml:ns:pidf'
xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
entity='sip:Alice@192.168.100.153'><tuple
id='t8a130d03'><status><basic>open</basic><location><description>KTH</
description><room>Esal</room><floor>1</floor><coordinates><latitude>12
3213</latitude><longtitude>47382145</longtitude></coordinates></locati
on></status><note>location</note><contact
priorit='0.8'>Carlos</contact></tuple></presence>";
```

```
            String presence="<?xml version='1.0' encoding='UTF-8'?>
<presence xmlns='urn:ietf:params:xml:ns:pidf'
xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
xmlns:lt='urn:ietf:params:xml:ns:location-type'
xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
entity='pres:someone@example.com'><dm:person id='p1'><rpid:activities
from='2005-05-30T12:00:00+05:00' until='2005-05-30T17:00:00+05:00'>
<rpid:note>Far away</rpid:note> <rpid:away/>
</rpid:activities><rpid:mood><rpid:angry/></rpid:mood><rpid:place-
type><lt:residence/></rpid:place-type><rpid:sphere>bowling
league</rpid:sphere><dm:note>Scoring
120</dm:note></dm:person></presence>";
            String presence2="<?xml version='1.0' encoding='UTF-
8'?><presence xmlns='urn:ietf:params:xml:ns:pidf'
xmlns:dm='urn:ietf:params:xml:ns:pidf:data-model'
xmlns:rpid='urn:ietf:params:xml:ns:pidf:rpid'
xmlns:c='urn:ietf:params:xml:ns:pidf:cipid'
entity='sip:watcher1@192.168.1.103'><tuple
id='t3f127a27'><status><basic>open</basic></status></tuple><dm:person
id='p1f5e1369'><rpid:place-type><lt:appear/><rpid:note>longitude
35</rpid:note><rpid:note>latitude 45</rpid:note></rpid:place-
type><rpid:sphere>meeting room<rpid:note>2nd
floor</rpid:note></rpid:sphere></dm:person></presence>";
            Selector selector = null;
            Selector pselector=null;
            String subscribeto="Publisher1";
            String[] users=new String[watchers];
            String[] pubs=new String[publishers];
            for(int i=0;i<watchers;i++){
                  users[i]="Watcher"+(i+1);
            }
            for(int i=0;i<publishers;i++){
                  pubs[i]="Publisher"+(i+1);
            }

          try {
              // Create the selector
              selector = Selector.open();
              pselector = Selector.open();
              // Create  non-blocking sockets.

              DatagramChannel[] sChannel=new
DatagramChannel[watchers];
              DatagramChannel[] pChannel=new
DatagramChannel[publishers];
              for(int i=0;i<watchers;i++){
                  sChannel[i]=createDatagramChannel(client,port);
                  sChannel[i].register(selector,
sChannel[i].validOps());
                  port++;
              }
              for(int i=0;i<publishers;i++){

      pChannel[i]=NonBlock.createDatagramChannel(client,portpub);
                  pChannel[i].register(pselector,
pChannel[i].validOps());
                  portpub++;
              }

              Publisher publish;
```

```
            publish=new
Publisher(publishers,server,client,pubs,pChannel,presence2);
            publish.start();

            Subscriber subscrib;
            subscrib=new
Subscriber(watchers,server,client,subs_exp,subscribeto,users,sChannel)
;
            subscrib.start();



        } catch (IOException e) {

        }

        // Wait for events
        while (true) {
            try {
                // Wait for an event
                selector.select();
                pselector.select();
            } catch (IOException e) {
                // Handle error with selector
                break;
            }

            // Get list of selection keys with pending events
            Iterator it = selector.selectedKeys().iterator();
            Iterator itp= pselector.selectedKeys().iterator();
            // Process each key at a time
            while (it.hasNext()) {
                // Get the selection key
                SelectionKey selKey = (SelectionKey)it.next();

                // Remove it from the list to indicate that it is
being processed
                it.remove();

                try {
                    processSelectionKey(selKey,server,client);
                } catch (IOException e) {
                    // Handle error with channel and unregister
                    selKey.cancel();
                }
            }
            while (itp.hasNext()) {
                // Get the selection key
                SelectionKey selKeyp = (SelectionKey)itp.next();

                // Remove it from the list to indicate that it is
being processed
                itp.remove();

                try {
                    processSelectionKey2(selKeyp);
                } catch (IOException e) {
                    // Handle error with channel and unregister
                    selKeyp.cancel();
                }
```

```
            }
        }
    }


    // Creates a non-blocking socket channel for the specified host
name and port.
    // connect() is called on the new channel before it is returned.
    public static DatagramChannel createDatagramChannel(String
hostName, int port) throws IOException {
        // Create a non-blocking socket channel
        DatagramChannel sChannel = DatagramChannel.open();
        sChannel.configureBlocking(false);
        DatagramSocket socket=sChannel.socket();
        socket.bind(new InetSocketAddress(hostName,port));
        System.out.println("new Socket binded to "+port);
        return sChannel;
    }
    public static void processSelectionKey(SelectionKey selKey,String
server,String client) throws IOException {
      if (selKey.isValid() && selKey.isReadable()) {
            // Get channel with bytes to read
            DatagramChannel sChannel =
(DatagramChannel)selKey.channel();
            ByteBuffer buf = ByteBuffer.allocateDirect(3000);
            String message;

            try {
                  String ID=null;
                  String via=null;
                  String to=null;
                  String from=null;
                  String user=null;
                  String cseq=null;
                  // Clear the buffer and read bytes from socket
                  buf.clear();
                sChannel.receive(buf);
                buf.flip();

message=Charset.forName(System.getProperty("file.encoding")).decode(bu
f).toString();

if(message.contains("NOTIFY")&&message.contains("</presence>")){
                        StringTokenizer tokens=new
StringTokenizer(message,"\n");
                        String linea=tokens.nextToken();
                        while(tokens.hasMoreTokens()){
                              if(linea.contains("Call-ID")){
                                    ID=linea.trim()+(char)13+(char)10;
                                    linea=tokens.nextToken();
                              }
                              else if(linea.contains("Via")){

      via=linea.trim()+(char)13+(char)10;
                                    linea=tokens.nextToken();
                              }
                              else if(linea.contains("To:")){
                                    to=linea.trim();

      user=linea.substring(linea.indexOf('"'),linea.lastIndexOf('"'));
```

```
                                    to=linea.substring(4)+(char)10;
                                    linea=tokens.nextToken();
                              }
                              else if(linea.contains("From:")){
                                    from=linea.trim();
                                    from=linea.substring(6)+(char)10;
                                    linea=tokens.nextToken();
                              }
                              else if(linea.contains("CSeq:")){
                                    cseq=linea.trim();
                                    linea=tokens.nextToken();
                                    }
                              else{
                                    linea=tokens.nextToken();
                              }
                        }

      Ok(sChannel,server,client,ID,via,to,from,user,cseq);
                  }
            }

            catch (IOException e) {
                // Connection may have been closed
            }
        }
    }



    public static void processSelectionKey2(SelectionKey selKey)
throws IOException {
            String etag=null;
      if (selKey.isValid() && selKey.isReadable()) {
            // Get channel with bytes to read
            DatagramChannel sChannel =
(DatagramChannel)selKey.channel();
            ByteBuffer buf = ByteBuffer.allocateDirect(1024);
            String message;;

            try {
                  buf.clear();
                sChannel.receive(buf);
                buf.flip();


message=Charset.forName(System.getProperty("file.encoding")).decode(bu
f).toString();
                if(message.contains("200 OK")){
                        StringTokenizer tokens=new
StringTokenizer(message,"\n");
                        String linea=tokens.nextToken();
                        String cseq;
                        String info[]=new String[2];
                        String ID=null;
                        while(tokens.hasMoreTokens()){
                              if(linea.contains("Call-ID")){
                                    ID=linea.trim();
                                    ID=linea.substring(9);
                                    linea=tokens.nextToken();
                              }
```

```
                                    else if(linea.contains("CSeq:")){
                                          cseq=linea.trim();
                                          linea=tokens.nextToken();
                                          }

                                    else if(linea.contains("SIP-ETag:")){
                                          etag=linea.trim();
                                          etag=linea.substring(10);
                                          linea=tokens.nextToken();
                                          test=etag;
                                    }

                                    else{
                                          linea=tokens.nextToken();
                                    }
                              }
                        }
                  }
                   catch (IOException e) {
                         // Connection may have been closed
                   }
                   }
      }

    public static void Subscribe(DatagramChannel local_socket,String
server,String client,String expires,String subscriber, String
subscribeto,int code){
            Date ts=new Date();
      String mensaje="SUBSCRIBE sip:"+subscribeto+"@"+server+"
SIP/2.0"+(char)13+(char)10+"Via: SIP/2.0/UDP
"+client+":"+local_socket.socket().getLocalPort()+";branch=b9hG4cd-
d87543-3a35b0441f1d2b5c-1--d87543-;rport"+(char)13+(char)10+"Max-
Forwards: 70"+(char)13+(char)10+"To:
\""+subscribeto+"\"<sip:"+subscribeto+"@"+server+">"+(char)13+(char)10
+"Contact:
<sip:"+subscriber+"@"+client+":"+local_socket.socket().getLocalPort()+
">"+(char)13+(char)10+"From:
\""+subscriber+"\"<sip:"+subscriber+"@"+server+">;tag=b3412c8b"+(char)
13+(char)10+"Call-ID: "+ts.toString()+"-
"+code+(char)13+(char)10+"CSeq: 1
SUBSCRIBE"+(char)13+(char)10+"Expires:
"+expires+(char)13+(char)10+"Content-Type:
application/pidf+xml"+(char)13+(char)10+"Event:
presence"+(char)13+(char)10+"Content-Length:
0"+(char)13+(char)10+(char)13+(char)10;
            send(local_socket, mensaje);
      }
      public static void Ok(DatagramChannel local_socket,String
server,String client,String callid,String via,String to, String
from,String user,String cseq){
            String mensaje="SIP/2.0 200
OK"+(char)13+(char)10+via+"Contact:
<sip:"+user+"@"+client+":"+local_socket.socket().getLocalPort()+">"+(c
har)13+(char)10+"To: "+to+"From:
"+from+callid+cseq+(char)13+(char)10+"User-Agent: Music-Client
V1.0"+(char)13+(char)10+"Content-Length:
0"+(char)13+(char)10+(char)13+(char)10;
            send(local_socket, mensaje);
      }
```

```
    public static void Publish(DatagramChannel local_socket,String
server,String client,String pubuser,String body,String callid){
            int length=body.length()+2;
            String mensaje="PUBLISH sip:"+pubuser+"@"+server+"
SIP/2.0"+(char)13+(char)10+"Via: SIP/2.0/UDP
"+client+":"+local_socket.socket().getLocalPort()+";branch=a9hG4bK-
d87543-3a35b0441f1d2b5c-1--d87543-;rport"+(char)13+(char)10+"Max-
Forwards: 70"+(char)13+(char)10+"Contact:
<sip:"+pubuser+"@"+client+":"+local_socket.socket().getLocalPort()+">"
+(char)13+(char)10+"To:
\""+pubuser+"\"<sip:"+pubuser+"@"+server+">"+(char)13+(char)10+"From:
\""+pubuser+"\"<sip:"+pubuser+"@"+server+">;tag=a2390c7b"+(char)13+(ch
ar)10+"Call-ID: "+callid+(char)13+(char)10+"CSeq: 1
PUBLISH"+(char)13+(char)10+"Expires: 60"+(char)13+(char)10+"Content-
Type: application/pidf+xml"+(char)13+(char)10+"Event:
presence"+(char)13+(char)10+"Content-Length:
"+length+(char)13+(char)10+(char)13+(char)10+body;
            send(local_socket, mensaje);
      }

    public static void rePublish(DatagramChannel local_socket,String
server,String client,String pubuser,String etag,String body,String
callid,int cseq){
            int length=body.length()+2;
            String mensaje="PUBLISH sip:"+pubuser+"@"+server+"
SIP/2.0"+(char)13+(char)10+"Via: SIP/2.0/UDP
"+client+":"+local_socket.socket().getLocalPort()+";branch=a9hG4bK-
d87543-3a35b0441f1d2b5c-1--d87543-;rport"+(char)13+(char)10+"Max-
Forwards: 70"+(char)13+(char)10+"Contact:
<sip:"+pubuser+"@"+client+":"+local_socket.socket().getLocalPort()+">"
+(char)13+(char)10+"To:
\""+pubuser+"\"<sip:"+pubuser+"@"+server+">"+(char)13+(char)10+"From:
\""+pubuser+"\"<sip:"+pubuser+"@"+server+">;tag=a2390c7b"+(char)13+(ch
ar)10+"Call-ID: "+callid+(char)13+(char)10+"CSeq: "+cseq+"
PUBLISH"+(char)13+(char)10+"Expires: 30"+(char)13+(char)10+"Content-
Type: application/pidf+xml"+(char)13+(char)10+"SIP-If-Match:
"+etag+(char)10+"Event: presence"+(char)13+(char)10+"Content-Length:
"+length+(char)13+(char)10+(char)13+(char)10+body;
            send(local_socket, mensaje);
      }

    public static void send(DatagramChannel sChannel,String message){
      ByteBuffer bufout = ByteBuffer.allocateDirect(10024);
      byte[] mess=new byte[10024];
      mess=message.getBytes();
      bufout.put(mess);
        bufout.flip();
        try{
            sChannel.connect(new
InetSocketAddress("192.168.100.234",5060));
            int numBytesWritten = sChannel.write(bufout);
            sChannel.disconnect();
        }
        catch(IOException e){

        }
    }
}
class Subscriber extends Thread{
      protected int watcherss;
```

```
	protected String servers;
	protected String clients;
	protected String expirs;
	protected String subscribetos;
	protected String[] users;;
	protected DatagramChannel[] sChannels;


	Subscriber(int watcher,String server, String client, String
expir,String subscribeto,String[] user,DatagramChannel[] sChannel){
		this.watcherss=watcher;
		users=new String[watcherss];
		sChannels=new DatagramChannel[watcherss];
		this.servers=server;
		this.clients=client;
		this.expirs=expir;
		this.subscribetos=subscribeto;
		for(int i=0;i<watcherss;i++){
			this.users[i]=user[i];
			this.sChannels[i]=sChannel[i];
		}
	}
	int aux=watcherss;
	public void run(){
		int aux=0;
		for(int i=0;i<watcherss;i++){

	NonBlock.Subscribe(sChannels[i],servers,clients,expirs,users[i],
subscribetos,i);
			//for(int j=0;j<650000;j++){
			//}
		}
	}
}

class Publisher extends Thread{
	protected int publisherss;
	protected String servers;
	protected String clients;
	protected String[] users;;
	protected DatagramChannel[] sChannels;
	protected String body;


	Publisher(int publishers,String server, String client,String[]
user,DatagramChannel[] sChannel,String mess){
		this.publisherss=publishers;
		users=new String[publishers];
		sChannels=new DatagramChannel[publishers];
		this.servers=server;
		this.clients=client;
		this.body=mess;
		for(int i=0;i<publishers;i++){
			this.users[i]=user[i];
			this.sChannels[i]=sChannel[i];
		}
	}

	public void run(){
	//	Wait.manySec(30);
```

```
            Date[] ts=new Date[publisherss];
            for(int i=0;i<publisherss;i++){
                  ts[i]=new Date();
                  String timestamp=ts.toString();
                  String id=timestamp+"-"+i;

      NonBlock.Publish(sChannels[i],servers,clients,users[i],body,id);
                        for(int j=0;j<500000;j++){

                  }
        }

            /*String x;
            while(NonBlock.test==null){

            }
            Wait.manySec(5);
            for(int i=0;i<publisherss;i++){
                  ts[i]=new Date();
                  String timestamp=ts.toString();
                  String id=timestamp+"-"+i;

      NonBlock.rePublish(sChannels[i],servers,clients,users[i],NonBloc
k.test,body,id,2);
                  for(int j=0;j<620000;j++){
                  }

        }*/


      }
}
class Wait{
      public static void oneSec() {
            try {
                  Thread.currentThread().sleep(1000);
            }
            catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }
      public static void manyMilli(long s) {
            try {
                  Thread.currentThread().sleep(s);
            }
            catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }
      public static void manySec(long s) {
            try {
                  Thread.currentThread().sleep(1000*s);
            }
            catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }
}
```

Appendix C

Installing SER as a presence/context server

The SER version 0.10.99 was used for the context distribution component and the source code is available at http://ftp.iptel.org/pub/ser/presence/ser-0.10.99-dev35-pa-4.2_src.tar.gz. This server runs in a Linux environment. Before the installation of the server the following libraries should be installed: the standard C libraries, libxml2, libcurl3, flex, bison, and libmysqlclient-dev.

The libpresence dependencies libcds also need to be compiled before installing SER. This libraries are distributed with SER. These libraries are compiled from the lib directory of the downloaded with the make command:

```
make –f Makefile.ser install prefix=”/base/ser/directory”
```

The SER is compiled with:

```
Make install group_include=”standard,presence,standard-dep” prefix=/base/ser/directory
```

Finally the following commands are used for running SER:

```
export LD_LIBRARY_PATH=/base/ser/directory/lib/ser

/base/ser/directory/sbin/ser -f /base/ser/directory/etc/ser/ser.cfg
```

The next step is to initialize the SER database, scripts for initializing the database are in SER's source tree in directory scripts, in the case of MySql it can be created with:

```
scripts/mysql/ser_mysql.sh create
```

After database creation data can be added using the ser_ctl utility also included in the SER distribution.

add domain: ./ser_domain add domain domain_id
add user: /ser_user add user_name
add uri: ./ser_uri add user_name uri

For further information about installing and running SER the reader may consult the SER presence handbook available at http://www.iptel.org/~vku/presence_handbook

Appendix D

Scheduling Pattern when having watchers subscribed to different contextities

Order	Contextity
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	20
12	11
13	30
14	21
15	12
16	40
17	31
18	22
19	13
20	50
21	41
22	32
23	23
24	14
25	60
26	51
27	42
28	33
29	24
30	15
31	70
32	61
33	52
34	43

Order	Contextity
35	34
36	25
37	16
38	80
39	71
40	62
41	53
42	44
43	35
44	26
45	17
46	90
47	81
48	72
49	63
50	54
51	45
52	36
53	27
54	18
55	91
56	82
57	73
58	64
59	55
60	46
61	37
62	28
63	19
64	92
65	83
66	74
67	65

Order	Contextity
68	56
69	47
70	38
71	29
72	93
73	84
74	75
75	66
76	57
77	48
78	39
79	94
80	85
81	76
82	67
83	58
84	49
85	95
86	86
87	77
88	68
89	59
90	96
91	87
92	78
93	69
94	97
95	88
96	79
97	98
98	89
99	99
100	100

COS/CCS 2008-12

www.kth.se