Exploiting Presence

KE WANG

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2008

COS/CCS 2008-27

Exploiting Presence

Ke Wang
kewang@kth.se

Supervisor & Examiner:

Professor Gerald Q. Maguire Jr.

Submitted in partial fulfillment of
the requirements for the degree of

Master of Science (Information Technology)

Department of Communication Systems
School of Information and Communication Technology
Royal Institute of Technology
Stockholm, Sweden

December 05, 2008

I will always remember the days sitting by the little window of the dark laboratory with the June sun shining on my shoulder

Like a movie scene.

I love it.

Abstract

By exploiting context awareness, traditional applications can be extended to offer better quality or new functions. Utilizing a context-aware infrastructure, a variety of context information is merged and processed to produce information that is useful to applications. Applications exploiting such context can provide more intelligent and creative services to end users.

This thesis examines two ways to make use of a user's location along with room occupancy information in context aware applications: a Context Agent and a Call Secretary. In the former case, the application subscribes to room occupancy information via a context server, and provides a Meeting Room Booking System with "real-time" information about the utilization of the rooms which the room booking system is to manage. The Call Secretary acquires both location and room occupancy information from a server. When the user is in one of the meeting rooms and multiple people are present, then this is interpreted as the user being in a meeting -- therefore it triggers a CPL module in a SIP proxy to redirect the user's incoming call to their voice mail box. A description of the implementation of these two applications will be presented along with an evaluation of these applications' performance.

The evaluation of the Context Agent showed that it was straightforward to integrate information from a presence source and to extend the meeting room booking system to use this information. The Call Secretary needs a more reliable source for the user's location. However, given this location the Call Secretary provides the service which the user expects.

Sammanfattning

Genom att utnyttja sammanhang medvetenhet, traditionella tillämpningar kan utvidgas till att erbjuda bättre kvalitet eller nya funktioner. Använda en kontextmedvetna infrastruktur, en rad olika kontextuppgifter är sammanslagna och bearbetas för att producera information som är användbar för tillämpningar. Tillämpningar som utnyttjar sådana sammanhang kan ge mer intelligenta och kreativa tjänster till slutanvändare.

Denna avhandling undersöker två sätt att använda sig av ett användaren befinner sig längs med rummet beläggningen information i samband medveten program: ett sammanhang av ombud och en uppmaning Sekreterare. I det förra fallet skall ansökan under på rumspris information via ett sammanhang server, och ger ett mötesrum bokningssystem med "realtid" information om användningen av de rum som lokalbokning system är att hantera. Ring sekreterare förvärvar både plats och rumspris information från en server. När användaren är i en av konferenslokaler och flera människor är närvarande, så är det tolkas som att användarna är i ett möte - därför det utlöser en CPL-modul i en SIP-proxy för att dirigera om användarens inkommande samtal till deras telefonsvarare fält. En beskrivning av genomförandet av dessa två program kommer att presenteras tillsammans med en utvärdering av dessa ansökningar resultat.

Utvärderingen av det sammanhang ombud visade att det var enkelt att integrera information från en närvaro källa och att utvidga mötesrum bokningssystem att använda denna information. Ring sekreterare behöver en mer tillförlitlig källa för användarens plats. Med tanke på denna plats för samtal sekreterare tillhandahåller tjänster som användaren förväntar sig.

Acknowledgements

First and foremost, I would like to express my deepest appreciate to Professor Gerald Q. Maguire Jr. His warm encouragement and stimulating suggestions helped me to overcome the difficulties throughout my research for and writing of this thesis. Not only did I learn technical skills from doing this project, but also I learned how to analyze problems.

My gratitude also goes to all those who gave me support and encouragement to complete this thesis, as well as the lovely coffee machine at Wireless@KTH which keeps me awake every afternoon. Tack Så Mycket!

Table of Contents

List of Figures

List of Tables

Glossary

3D	Three Dimension
ACK	Acknowledge
API	Application Program Interface
CODEC	Encoder/Decoder
CoBrA	Context Broker Architecture
CPIM	Common Profile for Instant Messaging
CPL	Call Process Language
DNS	Domain Name System
DTD	Document Type Definition
GPS	Global Position System
GUI	Graphic User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IMPP	Instant Messaging and Presence Protocol
JDK	Java Developer's Kit
JRE	Java Run Environment
MMS	Multimedia Messaging Service
MRBS	Meeting Room Booking System
OSI	Open System Interconnect Reference Model
PA	Presence Agent
PDA	Personal Digital Assistant
PIDF	Presence Information Data Format
PUA	Presence User Agent
QoS	Quality of Service
RFID	Radio-frequency identification
SER	SIP Express Router
SGML	Standard for General Markup Language
SIP	Session Initiation Protocol
SIMPLE	SIP for Instant messaging and presence leveraging extensions protocol
SMS	Short Message Service
SQL	Structured Query Language
UAProf	User Agent Profile
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

1. Introduction

1.1 Problem Statement

Due to rapid development of computing and communications technology, people are enjoying a high-technology life. Experts and other specialists are trying to introduce ever more devices into people's daily activities, to increase both personal convenient and efficiency. Smarts card are being used to identify us, Global Position System (GPS) can be used to track where we have been or guide us to where we want to go, and voice mail can record incoming calls when you are not available. However, sometimes these intelligent services and devices are not smart enough. For example, a poorly designed system would enable thieves to enter your office, simply because they stole your wallet and have your smart card. The last recorded GPS coordinates might worry your spouse if you lost in the mountains your GPS equipped cellular phone runs out of battery power. After returning from the mountains and recharging your cellular phone, of course you forget to manually switch your mobile phone into Meeting mode (i.e., silent mode), thus the ringing of your phone (with some very inappropriate ring tone) due to an incoming call interrupted your (now former) boss just as she is deciding upon your promotion or dismissal (as someone has to go and some one gets to stay -- but too bad that due to your phone ringing it is not you who will be staying). Looking around, we realize that most of these devices do not have suitable mechanisms to communicate with the users (nor with other devices belong to their user). While the individual device may work correctly, they lack information which would enable them to adjust their behavior to better assist their user. For this reason a context-aware system should be introduced to provide more suitable services to users according to their current context (i.e., situation). An additional benefit of exploiting context is to reduce the burden on the user - thus the user can spend less time configuring their device and more time doing what they actually want to do.

Today each device works separately in a context-aware architecture by sending or retrieving context information to/from a context server. For example, a room occupancy sensor can enable an online room booking system to provide better service, thus if a booked room has not detected anybody inside the room during the initial 15 minutes of its booked time, then this room could be re-signed to others. If the user's incoming call can be automatically redirected to voice mail as soon as the user is detected as being in a meeting, then the user does not have to remember to manually tell their phone to go into meeting (i.e., silent) mode. In this way, devices not only can provide more accurate and efficient services, but they can provide more intelligent functions.

1.2 Objectives

In this thesis, I am interested in developing two context-aware applications based on an existing context-aware architecture developed by earlier thesis students at the KTH Center for Wireless Systems (Wireless@KTH). One application, a Context Agent, mainly implements three functions to update room occupancy information within a Meeting Room Booking System: (1) it adapts the Session Initiation Protocol (SIP) for Instant messaging and presence leveraging extensions protocol (SIP SIMPLE) to subscribe to presence information (i.e., room occupancy information) via a context server. (2) it listens for Notify messages from the context server with updated presence information which encoded using the Presence Information Data Format (PIDF) and (3) to read and parse these messages to extract presence information and to update the associated database entry within the Meeting Room Booking System.

The Call Secretary application, automatically redirects the mobile subscriber's incoming calls to their voice mail box when they are in a meeting. (1) The mobile user's subscription to this service triggers the Call Secretary to send a Subscribe request to the context server requesting presence information about both the user's location and the occupancy of the room (at and near the user's location). (2) If the updated presence information contained in Notify message from the context server matches the pre-defined "*in a meeting"* predicate, then the Call Secretary uploads a specific Call Processing Language (CPL) script to the user's SIP proxy to redirect incoming calls to this subscriber to his or her voice mail. (3) This CPL script will be removed by the Call Secretary after receiving a Notify message from the context server which indicates that this user is no longer in a meeting.

A number of different technologies are required to implement these two applications. In chapter 2, some of the underlying core technologies and knowledge are introduced in order to provide the necessary background for our readers.

2. Background

2.1 Context-aware Systems

2.1.1 Introduction

During a conversation with your colleague, you tell her that you saw her teenage son smoking yesterday. You present some details to describe what you saw to this angry mother, such as the street name, the people he was with, the brand of cigarette he was smoking, etc. Subsequently, the mother asks her son about his behavior -- which he can not deny as he has nicotine on his breath, cigarette smoke on his clothes, and the empty packet in his shirt pocket. All of these bits of information form evidence that indicate that it was very likely that he was in fact smoking. In general, the collection of information used for characterizing the boy's context provides information to the mother about her son's behavior. Although we will not consider sensors for cigarette smoke, nicotine, etc. in this thesis we will examine how context information is gathered and how it can be used. We will focus on the use of context information in an office setting where users share a number of meeting rooms and where most users have cellular phones which they routinely carry with them and almost never turn off the phone (excepted when required to do so, for example on airplanes).

It is apparently that context information could concern information about almost everything and everyone who is somehow "relevant" to a given user. This context information can provide very important clues to application that can make use of this context information to ease their user's daily life. This is especially true of mobile personal devices as they provide mobility and are increasingly co-located with the user. There is a wide diversity of possible uses of context ranging from very simple to very deep & rich uses of context. For example, when you are in your office, you may have several desktop computers, printers, and projectors around you. When you want to print a file, do you as the user have to figure out which of the printers would be the most suitable or should the system somehow take care of this for you [59]? When you move outside, you take your Personal Digital Assistant (PDA), laptop, and one or more Radio-frequency identification (RFID) tags along with you. The increasing presence of computers with and near you is evolving toward what Mark Weiser referred to as 'pervasive computing'. This term refers to the integration of computing devices into the user's environment. The context information supports the application and the acquisition & use of this information should occur in the background (i.e., the user should not have to interact with the context elements directly). For example, when a student enters a school building, the electronic ID card reader by the front door identifies this student via his or her access card, the lab computer logs the user

on to his or her account automatically as the user approaches the console, and when the user walks into a meeting room, his or her presentation slides are already to be presented via the projector in the room [11].

In these examples, context-aware systems can be implemented in many ways, depending on the application's specific requirements and the user's environment & usage of the application. With such a system, sensors of different types and in different locations provide different types of context information, thus the applications making use of this context could provide a wide variety of different services via the available devices

2.1.2 Definition

There are many definitions of context which are used in different environments and situations. When the concept of context-awareness was first introduced in Schilit and Theimer [1], they focused on location information, and described context adaptation by the system in terms of its location, the identities of nearby people and objects, and changes to those objects over time. Later, Dey changed the definition of context information to focus on the user's emotional state, focus of attention, location, date and time, etc. [2] The differences in each of these definitions arise from the aspects of context which the researcher believed were relevant. In this thesis, I will use the definition given in Dey's work:

> *"Context is 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" [3].*

This definition was not created for certain situations or environments. The generalization over the earlier definitions enlarged its scope and gave it greater extensibility, hence allowing it to be used in different applications. The applications developed in this thesis build upon a number of earlier projects; each of these projects will be introduced in section 2.1.4. As all of these also used Dey's definition, this thesis also adopts this definition.

For example, in Yu Sun's location based remainder scenario [4], locations of users are used as context information. Users set their PDA to remind them not only at an approximate time but at a specific location. After the subscription to the context server for this user's location changes, notifications about the user's location can be utilized to trigger the reminder. The user's location can be detected using RFID, GPS, WLAN based positioning, etc. -- it is important to note that by subscribing to the context server for this information the application is now independent of the actual means used to locate the user or to detect that they have moved to a new location. The context server receives updates from one or more location systems and updates its

knowledge of where the user is and notifies any of the entities which have properly subscribed to learn of changes in this user's location. The application running in the user's device receives location updates and if the user's current location and the current time match the specified constraint, then, the remainder will be triggered - which will generate some type of notification to the user.

2.1.3 Architectures

There can be various architectures for context-aware systems. Each architecture has its own advantages and disadvantages. The selection of an architecture for an implement is mainly based on considerations of the specific application's requirements and physical environment(s). For example, certain types of sensors may have limitations in use, the extensibility of the system may be the crucial issue, or advance features may increase the complexity of the whole system.

The direct sensor access model is an easy and low-complexity approach to implement a context-aware system [5]. This model integrates software and hardware. The sensor is directly controlled by the applications. When context information is collected by the sensors, it was converted into a specific data format to be directly used by the application. Because the control of the sensor and the application are so tightly coupled, the data gathered is only suitable for this specific application. This makes it difficult to use the information for other applications, and makes it difficult to add different types of sensors to this system (in order to extend its functions). Therefore, this model only survives in a very narrow niche.

With the help of a context widget component, we could easily add an extra layer between the sensor and the application, which increases the reusability of the context information. A context widget is a software component that mediates between sensors and applications. It encapsulates the context information collecting procedure on the sensor side, and provides applications with methods to access to this information. Finally, it provides a uniform interface allowing multiple applications to utilize it. Moreover, it can work with two other components, an Interpreter and an Aggregator to abstract and process data before transferring the data to an application [6].

The use of an ontology extends the value of context information through out the whole system. The core activity is information sharing. In practice, the system is divided into two parts, input and output. Inputs from different devices collect context information, while the outputs are utilized by applications. Consequently, context information from different devices could be used by different applications to provide a variety of services to the users, enabling pervasive computing. For example, if a group of sensors are at the entrance to each meeting room, they can monitor the movements of the people in/out of the room. This raw context information can be utilized by many different applications. The application counting the number of people in the meeting room provides a signal which can be used to turn lights on and

off (or to invoke other applications). If the number of people inside the room is zero, then the lights could be switched off automatically. At the same time, the same context information can be used to provide input to a smart Meeting Room Booking System. If the room has been booked, but has nobody inside within the first 15 minutes of the booked period, then the system could automatically cancel this booking and reset this room's status to be "UNBOOKED", therefore, other users could book it.

By using an ontology, the context information can be shared by several independently developed context-aware systems. A Context Broker Architecture (CoBrA) is a good example of an architecture that provides context information to all kinds of devices, services, and agents in the same environment [7]. A middleware infrastructure is a popular architecture supporting this trend, and it is the layered architecture that I can implement in the thesis project.

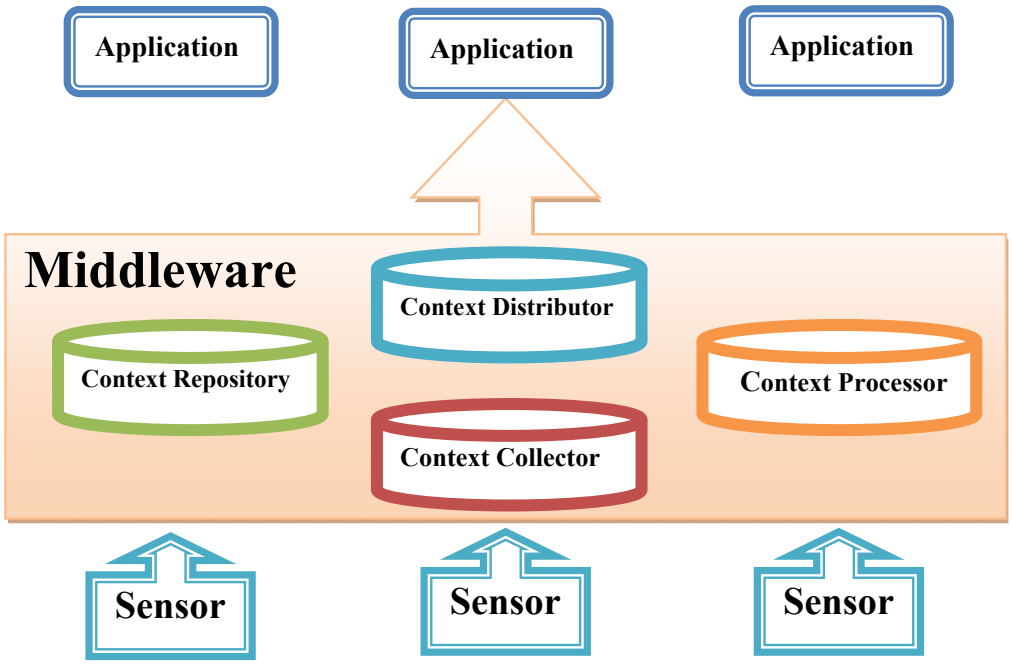

Figure 1: Middleware Architecture

You can see the entire architecture of this context-aware system in Figure 1. It starts from the bottom layer which is the sensor layer. A collection of sensors with various functions monitor specific conditions in the environment around them. The raw context data is collected and encapsulated using a strict encoding scheme, then forwarded to the next layer which is so-called middleware. This layer includes four potential entities to forward context data to application layer, which is the top layer of this architecture. Within the middleware layer all data are initially accepted by a context collector, then stored in a context repository using a uniform format. Such context information can be provided to applications either directly through a context distributor, or interpreted (for example, using artificial intelligence techniques) by an entity named a context preprocessor before it is provided to applications. This context

preprocessor provides applications with more highly abstracted context information.

Based on this middleware architecture, a remote access management component can be added to the context server architecture. Using this component, context data can be accessed remotely by multiple applications. When building a context-aware system based upon a client-server model, the mobile device's processing, storage, and energy limitations can be relieved by using a context server. As described in the next section, the SIP Express Router (SER) is used as the basis of this context server.

2.1.4 Context-aware system overview

This thesis project builds upon an existing context-aware system. The major parts of this system are shown in Figure 2.

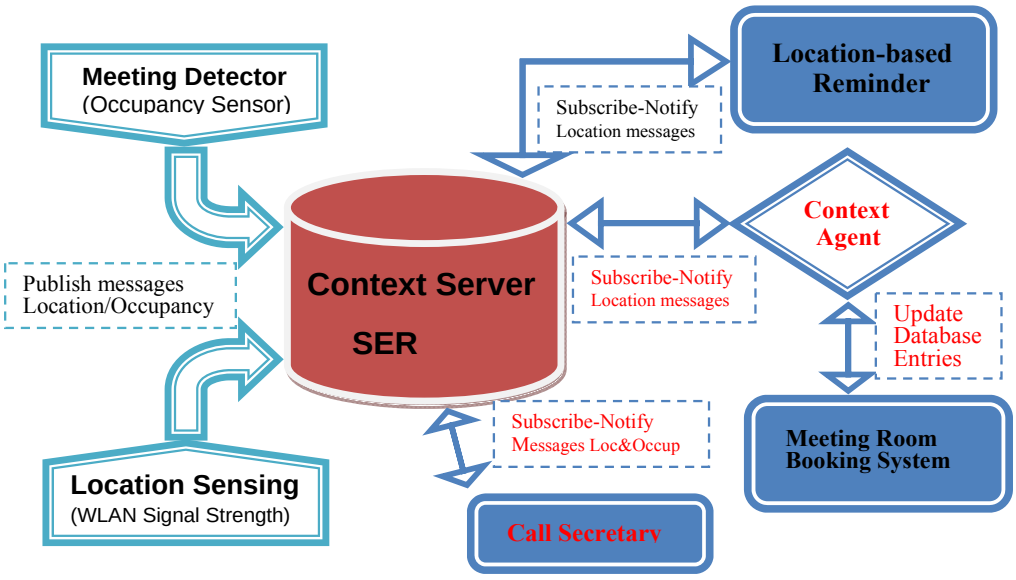

Figure 2: Blueprint of a Context-aware System

From left to right, the raw and processed context information is collected by a number of sensing systems. The Meeting Detector [8] and Location Sensing Systems [9] monitor their environment to collect the desired context information. This context data is formatted in a uniform format (PIDF, see section 2.5.2), then published to a context server which is located in the middle. This server functions as the middleware. Mohammad Zarifi Eslami introduced the use of the SIP Express Router (SER) [10] as a context server. This server communicates with the sensors to the left and the context-aware applications to the right, as well as processing and storing the context information. These applications fetch context information from the context server - some of this information is stored in its database and some is passed immediately to applications when there is new context information. Between the server and applications, context information transmissions utilize SIP SIMPLE (see section 2.3). This protocol allows an application to send a Subscribe request to the server, later the

applications' receives asynchronous Notify messages containing the context information which they subscribed to. SIP SIMPLE Publish request messages convey context information from the sensors to the context server. The Location-based Reminder application enables a user to schedule a location based reminder; this is based on the application subscribing to a source for location information [4]. Therefore, whenever the user's location changes, the updated location information would be forwarded to the user's calendar application device (here assumed to be running on the user's PDA) as a Notify message - if this matches the location of a reminder, then this reminder is triggered. In this thesis project I will develop Context Agent that will act as a bridge between a context source which provides room occupancy information and room booking system. This application will update the room booking system's database when a Notify message containing updated occupancy data for a room is received. The room occupancy information indicates how many people are in a particular room. The Meeting Room Booking System Based (MRBS) is extended to accept room occupancy sensor information to enable the user to find an unoccupied, but booked room. Additionally, the room booking system could automatically unbook rooms that are not in use within a given period of time or which are no longer being used. We can see that the Context Agent is used in this setting to transform a simple web based database system with manual updates into a dynamic system. Similar extensions could be used to enhance building management systems and other applications.

Another application, the Call Secretary utilizes both location and room occupancy context information to predict if the subscriber is in a meeting. When the subscriber is in a meeting, an incoming call to this subscriber should be redirected to his or her voice mail box. Mobile users will subscribe to this service using a web-based interface. The Call Secretary will install a CPL (Call Processing Language, see section 2.6.3) script on the subscriber's SIP proxy (in this case assumed to be running on the SER server) to redirect the subscriber's incoming calls to their voice mail box. The prediction that the subscriber is in a meeting will be based upon both the user being inside a meeting room and there are more than a certain number of people in this room. The Call Secretary is an example of an application that utilizes a variety of context information to provide an intelligent service - which simplifies the user's life as they no longer need to manually turn on and off the "meeting" mode for their cellular phone nor will they face the embarrassment of having their phone ring during a meeting.

2.2 SIP

Communication is one of major features of Internet connectivity. This communication can take many forms, but one of the most common is interactive conversations. Several signaling protocols have been developed and used to

implement new applications providing different forms of communication. These signaling protocols enable the creation and management of sessions between participants [12].

The Session Initiation Protocol (SIP) is a widely used signaling protocol for conversational communication sessions, along with H.323 [13], MGCP, and some proprietary protocols (Skype and Cisco's Skinny, etc) these protocols provide the basis for Voice over IP (VoIP) services. The current SIP is standardized by Internet Engineering Task Force (IETF) in RFC 3261 [14]. SIP works at the application-layer of the Open Systems Interconnect (OSI) model, and is used for creating and tearing down sessions of one or more participants, and for specifying & manipulating the media data transmission associated with such a session. Such session manipulation includes adding or deleting media streams, specifying or modifying the participant address or port, and sending invitations to potential new session participants during a session.

However, SIP does not transport the session's media, i.e., the data within a session (such as video, audio, real-time text, media streams) are not handled by SIP itself. Instead, SIP utilizes other IETF protocols to create a complete multimedia communication architecture. Two of these protocols are the Session Description Protocol (SDP) [17] for describing multimedia sessions (choice of media, choice of CODEC(s) [16], sampling rate, etc.) and the Real-time Transport Protocol (RTP) [18] for transporting real-time data, providing Quality of Service (QoS) feedback, etc. Applications built on SIP (and the related protocols) provide multimedia communication services over Internet Protocol (IP). These services include Internet telephony calls, video conferences, interactive gaming, etc.

Similar to the HTTP protocol, SIP is a text-based, human readable protocol. In addition these HTTP & SIP have many features in common. They are both based on a request/response transaction model, which means that within a session, each request invokes a particular function though a method, and there will be one or more responses to this request. Moreover, SIP developed an addressing scheme using a SIP URI - to play a similar role to a URL in the HTTP protocol. This SIP URI follows the guideline for Uniform Resource Identifiers (URI) [19]. A SIP URI is a globally unique address with a format similar to an email address:

sip:user:password@host:port;uri-parameters?headers

The SIP proxy server can perform a DNS look up to retrieve the current IP addresses of participants' SIP proxies based upon their SIP URI. This DNS lookup is limited to a lookup based upon the host portion of the SIP URI. However, this does not actually provide the IP address of the user's SIP user agent, but rather it provides the IP address of the user's (incoming) SIP proxy. To pass a SIP INVITE request (to create a new session), the user's (incoming) SIP proxy must consult a database containing information about the current IP addresses of the user's SIP user agents.

These SIP user agents are expected to have earlier registered this information with a location server. (Details of this process can be found in [63])

Consider the case of a call, where Ivan tries to start a session with LoLo by sending a SIP request message INVITE to her. This INVITE is a SIP method for initiating a session. Ivan and LoLo each have User Agents (UA) with their own SIP URIs. These user agents can work both as a server and a client depending on who initiates the session (the client initiates a session, while the server responds to this request). The INVITE contains a group of header fields that provide the necessary information to process this request, such as the URI addresses of Ivan and LoLo indicating where the response and request are going to be routed, a unique Call-ID identifying this session, and a Content-type header indicating the type of the session information that is to be passed in the body of the request (or response), and other header information that may be relevant to this session.

Table 1 lists the most widely used SIP methods.

Table 1: Widely used SIP Request Message

Method	Reference
INVITE	RFC3261[14]
ACK	RFC3261
BYE	RFC3261
CANCEL	RFC3261
REGISTER	RFC3261
OPTIONS	RFC3261
INFO	RFC2976[66]

In a typical communication scenario, SIP messages reach the callee via intermediaries between them. These proxies are not shown in Figure 3 but are present implicitly. Thus between Ivan's UA (acting as a user agent client (UAC)) and LoLo's user agent (acting as a user agent server (UAS)) there is one or more SIP Proxy Server(s) that will receive the INVITE message from Ivan. This message is destined to LoLo's UAS, as indicated by the destination URI contained in this message. Each of these SIP proxies either knows LoLo's address or forwards it on to another proxy "closer" to LoLo. In the later case the proxy performs a DNS lookup to locate LoLo’s IP address or her domain’s incoming SIP proxy (or proxies) address, i.e., the SIP proxy responsible for incoming calls to LoLo’s SIP provider. In practise, there are two possibilities: one is that Ivan knows (via some other means) the IP address of LoLo's SIP UA - thus he can directly deliver the INVITE message to LoLo’s device, otherwise Ivan will route this INVITE to the next SIP proxy server which will in turn repeat the lookup-forward action until the INVITE message reaches LoLo’s device. If the INVITE message successfully reaches LoLo's UA, then two messages back to Ivan. 180 Ringing indicats the server receiving the INVITE is trying to alert the user.

If LoLo answers the call then her UAS will send a 200 OK response shows that the request has succeeded and containing LoLo's session description information.

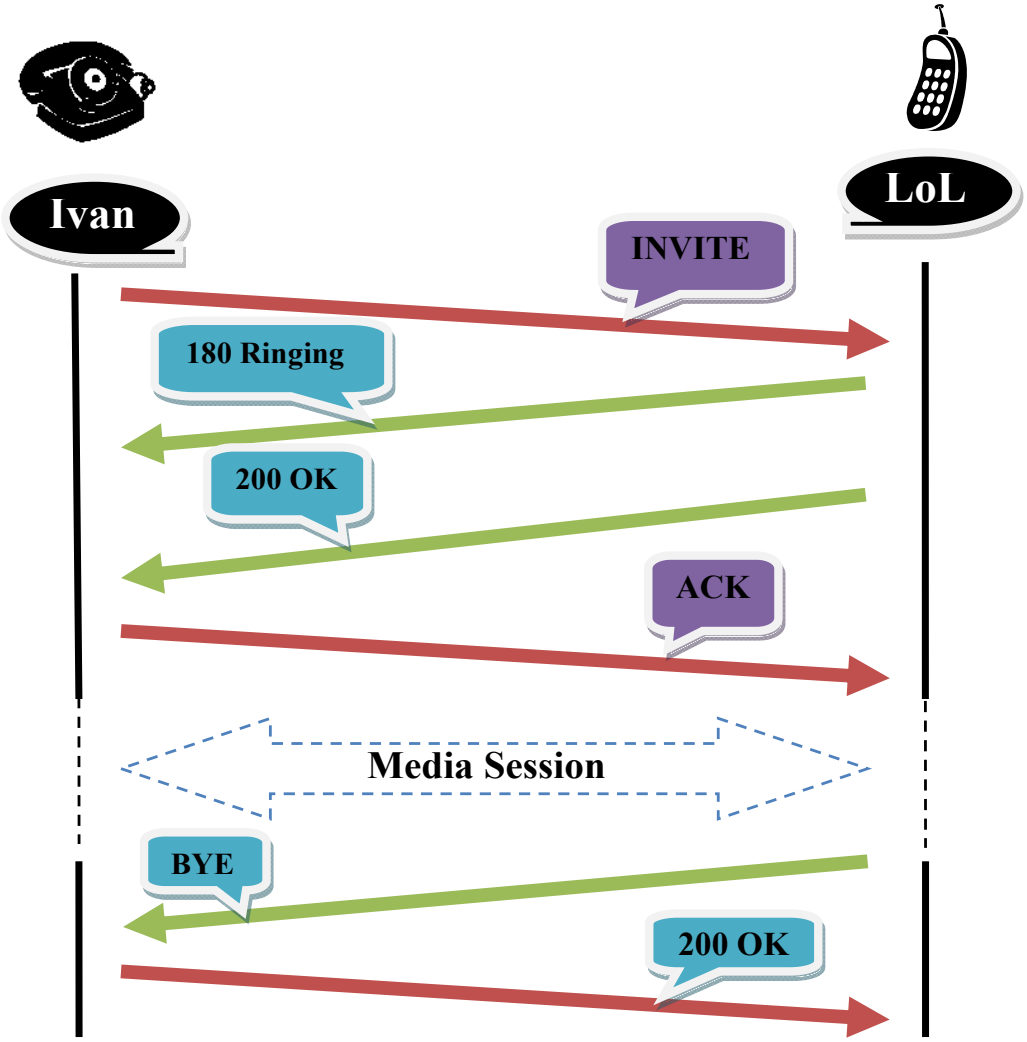

Figure 3: A Simple SIP Session Establishment [20]

Besides SIP Proxy Servers, there are other important entities that may be part of a SIP communications infrastructure, such as SIP Registrar Servers, Location Servers, and SIP Redirect Servers. They interact with each other to provide a complete SIP infrastructure. It is important to note that these are logical entities, therefore, they could be implemented on separate servers or on a single physical server. When a user logs on to his or her device, the SIP user agent sends a SIP REGISTER message to the user's SIP Registrar Server which is located in this user's SIP domain. When a registrar receives this REGISTER message it associates the user's SIP URI with the (IP) address of the current device, and stores this information in a database (in order to later provide a Location Service). Later the Location Service can retrieve this IP address information as it indicates the user's current UA(s), thus a SIP Proxy Server can forward INIVITEs to this user's SIP UA(s). A SIP Redirect Server responds to client requests with the address of the user or redirects the caller to another SIP server

or servers.

There are several classes of Response Codes based upon the HTTP/1.1 response codes. Some of these, such as 180 Ringing and 200OK were shown earlier in Figure 3 These classes are:

Provisional 1xx	The server being contacted is performing some further action and does not have a definitive response.
Successful 2xx	The request was successful.
Redirection 3xx	This response gives information about the user's new location, or about alternative services that might be able to satisfy the call.
Request Failure 4xx	This class of responses represents a failure indication from a particular server.
Server Failure 5xx	The server itself has erred.
Global Failures 6xx	The request failed and should not be tried again.

An ACK message from Ivan to LoLo is a sign of a successful session establishment. The media transmissions between participants based on this session can begin and will continue until the SIP BYE message is sent by LoLo (or it could be sent by Ivan). The BYE message indicates that the participant wishes to terminate the session. Ivan replies with a 200OK message, then the session ends.

2.3 SIP SIMPLE

2.3.1 Introduction

As introduced in the previous section, the SIP protocol is only responsible for session establishment, modification, and termination. Therefore, all media transmission relies on other related protocols. In our context-aware system, Figure 2 indicated several clear requirements for context transmission. The SIP SIMPLE protocol was utilized to convey context information from sensors to the context server and from the context server to the applications. SIP SIMPLE seems to be a perfectly fit to our needs.

SIMPLE (Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions) provides a group of extensions to the SIP protocol, focusing on instant messaging and presence publication functions. It is an open standard protocol with many components, and still being standardized. Instant messaging and the presence mechanism enable multiple participants within a session to exchange real-time messages and to be notified of changes in presence information of each other. Presence information is a kind of user state indicator which conveys ability and willingness for a potential communication partner [19]. Presence information such as

"Away" or *"Busy"* state is provided in numerous instant messaging applications, i.e., Microsoft Windows Messenger (MSN), AIM, Skype, etc. However, simple presence information can be extended by an appropriate event package to offer addtional information.

Specifically, the protocols standardized in RFC 3265 [20], RFC 3856[21], and RFC 4662 [22] provide a Subscribe-Notify mechanism which allows participants to request notification from remote nodes when particular events have occurred. Moreover, it allows user to subscribe to other users' events through their individual SIP URIs. The mechanism for presence information publication is described in RFC 3903 [23].

Figure 4 shows a typical SIP SIMPLE scenario. Here the terms defined in RFC 2778 are used to describe the process. Firstly, on the left side, there is an application or participant known as a watcher, who is interested in some specific presence information. It sends a Subscribe Request to the Presence Agent (PA) to subscribe for such information. A PA could be a SIP agent responsible for Subscribe Requests from watchers for specific presence information. Moreover, when there is a change in presence state for which watchers have subscribed, this will trigger the PA to generate notifications and send these to watchers. Such a presence state change can be based on information provided by a Presence User Agent (PUA). The PUA is aware of the presence information of the presentity, and utilizes the Publish method to transfer the updated presence information from the PUA to the PA. There are two aspects which we must note; one is that the subscription has a finite lifetime, the watcher has to keep renewing it before this subscription expires, and another aspect is that the PU and the PUA are both *logical* entities - hence they can be co-located, distributed, or even integrated into other software.

As shown earlier in Figure 2, this project utilized SIMPLE in two ways. One is to convey information between the sensing part and context server. Thus when a context change matches the condition which we are waiting for, this triggers the Meeting Detector or Location Sensing system to send a Publish message containing the updated presence information to the context server. Another use is based upon a Subscribe-notify interaction, as used in the applications on the right side of Figure 2. Consider the Context Agent, it first subscribes to the presence information about the number of people in each room by sending a Subscribe Request message to the context server. After receiving an OK message from the context server, the Context Agent continues listening for Notify messages from this server. When the context server receives a Publish message from a Meeting Detector, it generates a Notify message to subscribed Context Agent(s).

In our context-aware system, the PUA sends Publish messages to the PA containing presence information about a presentity. Watchers send Subscribe Request messages to learn about updates to the presence information which they are interested in. Moreover, after receiving Notify messages from the PA, the watchers parse and

process these messages. The following sections provide details about these three message and their formats.

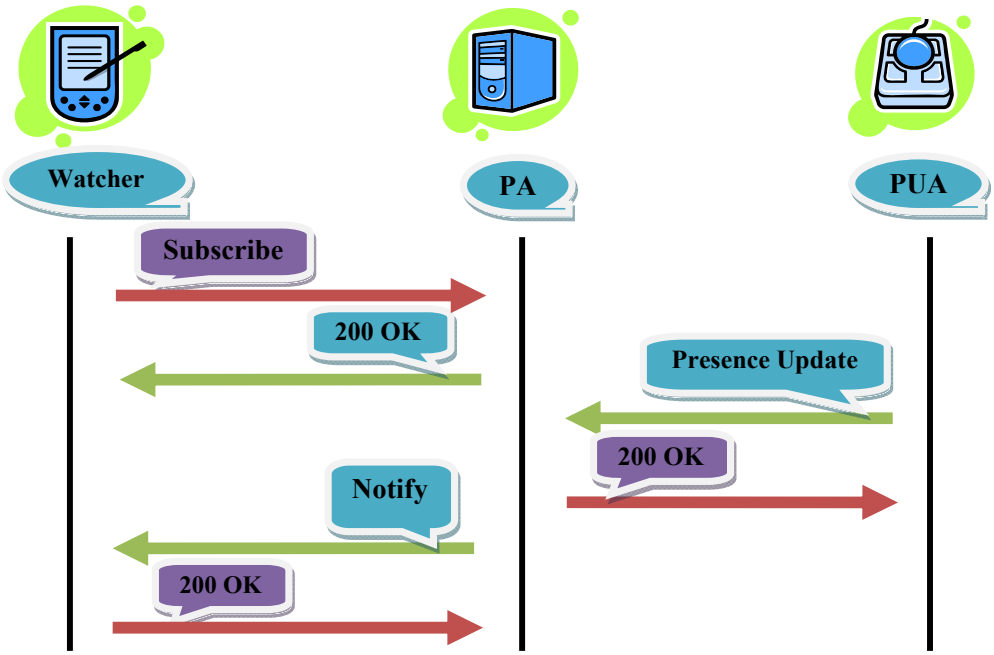

Figure 4: SIP-SIMPLE Working Scenario

2.3.2 Publish message-PUA's work

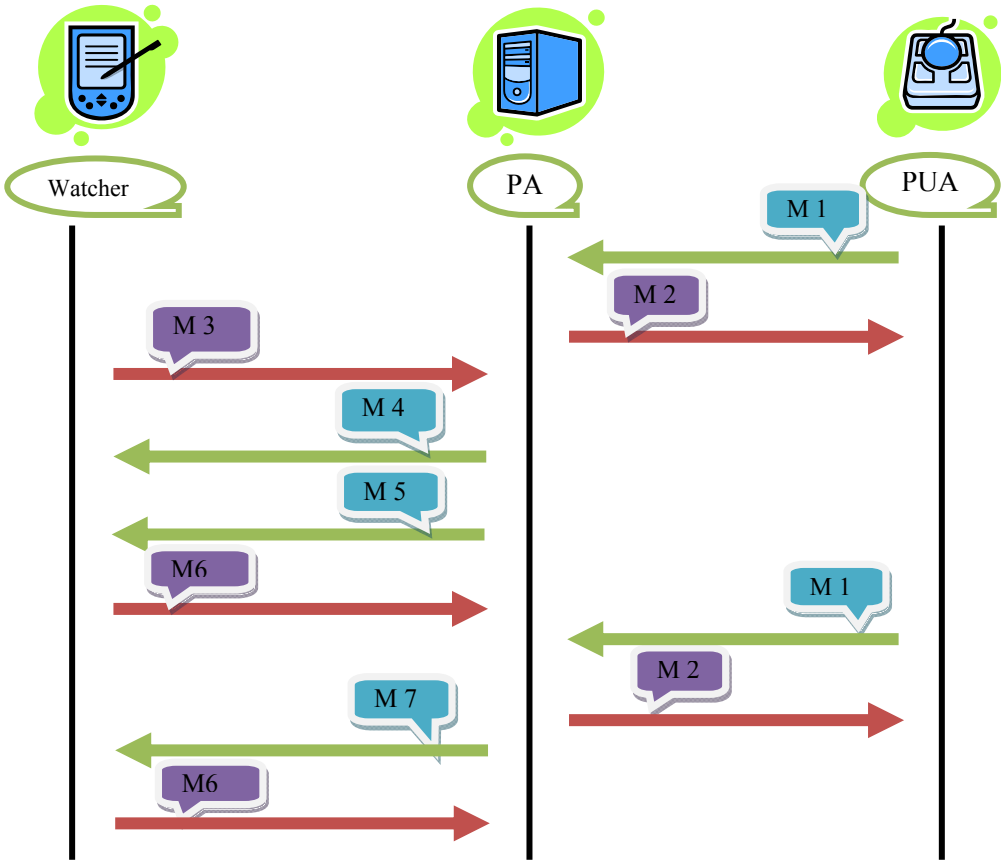

Figure 5: Messages Transmission among Watcher, PA, and PUA

Figure 5 shows several case of simple message transmission between the Watcher, PA, and PUA. Firstly, we take a closer look at message M1, the Publish Request Message.

Publish is a SIP method defined in RFC 3903 [67]. As with other SIP methods, there are Request and Response messages. The Publish Request message implements the Publish method (which is similar to the Register method). The request is composed of a header and a body. The header includes the indicator, the URI addresses of the PA and PUA, some variables, etc. Multiple Publish Requests may have the same PA or PUA addresses in headers, which means that there could be more than one PUA which provides presence information to a PA. For example, in the Context Agent application, user's locations and room occupancy information are published to a single SER server. Alternatively, a PUA could provide the same presence information to multiple PAs. The body of a Publish Request message contains information about a specific context encoded in the Presence Information Data Format (PIDF, see section 2.5.2). Such presence information could be created, modified, and removed; much as a SIP Register method creates/modifies/removes a UA's registration. Moreover, according to the expire heading included in the message,

another Publish Request message has to be sent in order to refresh interest in the event before the interest expires. Thus both subscriptions to get presence information and the presence information itself both have limited lifetimes.

When the PA receives a request from a PUA, it will send a response message back according to the URI's address, this is message M2. This Publish Response message is the same as the SIP Response message described in section 8.1.2 of RFC 3261. Along with sending a Response message to every successful Publish Request, a unique entity-tag is generated and assigned by the PA to identify this publication event. This tag is used by the PUA in any subsequent Publish Requests to modify, refresh, or remove the associated event state. However, when the publication event expires or is removed, then the PA has to refresh it by sending a new Publish Request. In this case, the previous entity-tag was valid and contained in the new request message. Otherwise, a new tag will be assigned by the PA in its response message.

Note that some different terms are used in RFC 3903. In this RFC the Event Publication Agent (EPA) and Event State Compositor (ESC) are respectively the PUA and PA defined in RFC 3261.

Figure 6 shows the format of the Publish Request message, M1.

PUBLISH sip:ccsleft@130.237.15.238 SIP/2.0
Via: SIP/2.0/UDP pua.example.com:5060;branch=z9hG4bK652hsge
To: <sip: ccsleft@130.237.15.238>
From: <sip: ccsleft@130.237.15.238>;tag=1234wxyz
Call-ID: 9090920@130.237.15.238
CSeq: 1 PUBLISH
Max-Forwards: 70
Expires: 3600
Event: presence
Content-Type: application/pidf+xml
Content-Length: ...
[Published PIDF document]

Figure 6: An Example of a Publish Request Message

2.3.3 Subscribe message-Watcher's work

As introduced in section 2.3.1, the Subscribe-notify mechanism provides SIP with the ability to enable participants to request asynchronous notification of context information changes. Subscribe messages, such as M3 in Figure 5, are actually SIP Subscribe Request messages, initiated by Watchers (a subscriber) which is interested in certain presence information from a presentity (including watchers themselves). In one Subscribe Request message, the Subscribe Request URL is a very important component for both routing the request to the appropriate server, and identifying the

desired presentity. The Event Type defines which presence information is to be subscribed for. This message eventually arrives at a PA (or even a presence server)

PA generates a Subscribe Response message to the Watcher indicating the success or failure of this Subscribe Request, M4 in Figure 5. If it is a 200-class response, then it indicates this request was successfully registered and the Watcher is authorized to subscribe to the requested notification.

However, this subscription has its lifetime defined in an Expire header field in the Subscribe Request message. Hence, before it expires, the subscriber may refresh the timer on such a subscription by sending another Subscribe request with the same "Event" header "id" parameter (a Subscribe message with a different "id" value in "Event" header would be considered as a new subscription). This refreshing Subscribe request will trigger a final Notify message sent by server about the current presence information of the presentity. Therefore, one way to fetch the latest presence information is to send a Subscribe request to the PA with an immediate expiration (that is "0" expiration time). However, if no refresh request is received before the expiration time, a subscription will be removed from the server and a notify request generated and sent to the subscriber. Within this message, the "subscription-state" is set with the value of "terminated". Note that this header should also contain a "reason=timeout" parameter.

On the other hand, the subscriber can terminate a subscription by sending a refreshing request with the value of the Expires header field set to "0". Note that a successful unsubscription will also trigger a final Notify request from the server.

Figure 7 shows a format of Subscribe Request message, M3.

SUBSCRIBE sip:lolo@130.237.15.238 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:5060;branch=z9hG4lfOiPzxGo
To: < sip:ccsleft@130.237.15.238>
From: <sip:hlllab4@130.237.15.238>;tag=naVc
Call-ID: 238@130.237.238.165
CSeq: 1275 SUBSCRIBE
Max-Forwards: 70
Event: location
Accept: application/pidf+xml
Contact: <sip:hlllab4@130.237.238.165>
Expires: 800
Content-Length: 0

Figure 7: An Example of a Subscribe Request Message

In the first line, the "SUBSCRIBE" indicates the message is a Subscribe message. The following two variables are the user name "lolo" and the IP address (or Domain name) of the server "130.237.15.238".

In the second line, the "UDP" indicates the protocol being used for the transport

layer. Because in this example, the Watcher sends a Subscribe message directly to the server, the following two variables contain the IP address of the server “130.237.15.238”, the port number being used “5060”. The random number “z9hG4lfOiPzxGo” in the “branch” field has to be different in each subscribe message.

In the third and forth lines, the “To” and “From” headers indicate which server this request message is going to and which watcher sent this request (respectively). Specifically, “ccsleft@130.237.15.238” is the URI of the server, while “hlllab4@130.237.15.238” indicates the hostname of the Watcher and indicate swhich server it registered with. The “Tag” field includes a random number.

The variables in the fifth to eighth lines provide additional information. The “Call-ID” field is composed of a random number “238” and the Watcher’s IP address “130.237.238.165”, the random number in combination with the Watcher's IP address should be different in every Subscribe message. The “CSeq” field contains a number which is incremented by one for every Publish message. The “SUBSCRIBE” value in the Cseq header indicates that this is a Subscribe message. The “Max-Forwards” field is used to limit the hops that a request can be forwarded before it reaches the destination server.

In the ninth and tenth lines, the “Event” header indicates that the Watcher wishes to subscriber to the presence information. The Accept header indicates that this presence information should be returned in as PIDF using the XML format and encapsulated in a Notify request message.

The “Expires” header defines the lifetime of this subscription which is “800” seconds.

In the final line, the value of the “Content-Length” header indicates the body length of this Subscribe Request message. However, as Subscriptions usually do not contain bodies, so for this case, the value is generally zero, as in this example.

After a Notify request is received, a 200 OK message, M6, is sent to the server as a response.

SIP/2.0 200 OK
From: <sip:ccsleft@130.237.15.238>;tag=xlB3;received=130.237.15.227
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: presence
Content-Type: application/pidf+xml
Content-Length: 0

Figure 8: An Example of a 200 OK Message

2.3.4 Notify message generation

A PA typically accepts Subscribe Requests and creates subscriptions accordingly to these requests. When a change in the presence state occurs, a Publish message from the PUA will arrive at the PA to update the specific presence information associated with a specific subscription. Consequently, the server would generate a Notify Request and send them to the Subscriber(s). This process continues until the subscription expires.

As introduced in the previous section, when the PA receives a new Subscribe Request or a subscription refreshing message, it will answer the Subscriber with a 200-class response. Additionally the PA needs to immediately construct and send a Notify Request message to the subscriber containing the current presence state, M5 in Figure 5.

In all, there are four main scenarios related to a Notify Request message. A new Subscription is created, the subscription is refreshed, the subscription expires, and the presence state of the presentity changes.

The Notify request contains the current/updated presence information of the presentity.

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: location
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="0xb58d60e0x4a4b0c39x4715c26e">
<status><basic>open</basic>
<location>
<description>Kista</description>
<room>Grimeton</room>
</location>
</status>
<contact priority="0.80">KeWang</contact>
<note>location</note>
```

```
</tuple>
</presence>
```

Figure 9: An Example of a Notify Message with Context Information

In the application, I mainly need to construct 3 different messages: the Subscribe Request, the Refreshing Request, and 200-class Response. Moreover, after receiving any Notify Request from server, the application needs to parse and process it to extract the presence information contained inside the body of the message.

2.4 XML

The Extensible Markup Language (XML) [24] is a simplified subset of the Standard Generalized Markup Language (SGML) [25]. It was firstly developed in 1996, by an XML Working Group and SGML Working Group. It was sponsored and recommended by W3C (World Wide Web Consortium). The latest version is XML 1.1.

XML enables the definition of a set of annotations for documents that describe how a document is to be structured. These structured XML documents can be parsed and processed according to predefined semantic rules. XML encoding facilitates information being shared through different systems, particularly via Internet. However, XML is sometimes misunderstood to be a programming language. Actually, it is only a specification for creating custom markup languages; which means that it allows users to describe a makeup language. However, by adding semantic constraints, application languages can be implemented using XML, such as XHTML [40], RSS [41], CPL, etc. [26]

Enabling different systems to share structured data is the main purpose of XML, thus such data should has been encoded with sufficient and clear annotations as that it can be parsed correctly by other information systems. Therefore, two levels of correctness have to be fulfilled in constructing structured data to form an XML document. One level of correctness is that the XML document should be well-formed, this means that it has to conform to all of XML's syntax rules. The second level of correctness is that an XML document should conform to some semantic rules. These rules could be defined by users or included as an XML schema (i.e., Document Type Definition). If an XML document disobeys either of these forms of correctness, then, the parser will refuse to process it.

Here is a simple XML Document

```
<?xml version="1.0"?>
<location>
<campus>Kista</campus>
<building>Electrum</building>
```

```
<floor>5</floor>
<room>523</room>
</location>
```

The <?xml …?> is the XML declaration, this identifies the document as an XML document and also indicaties the version of XML used in the following lines. Unlike HTML, XML tags have no predefined meaning, but are simply symbols. The XML document can be processed by parser following pre-defined semantic rules or using an XML schema.

2.5 Context model

2.5.1 Introduction

So far, the architecture for a context-aware system has been introduced. For this system I have adopted the SIP SIMPLE protocol for context information transmission. I also introduced the general scenario illustrating how the context information is collected from a PUA, published to a PA, and finally forwarded to specific Watchers. During this process, we noted that a particular PA, PUA, and Watcher could involve wide variety of devices. For example, several different sensors could be utilized by PUAs to monitor and collect state events. Additionally, a number of different devices and applications can be waiting for context information notifications (these devices could be a PDA, a PC, a mobile phone, etc). It is desirable to have a single context model to make sure that all different kinds of context information can be parsed and processed correctly by all these devices and applicaitons.

A context model is a key element in any context-aware system, as it defines approaches to describe, represent, exchange, and store context information. Strang and Linnhoff-Popien have presented and evaluated some popular models [27]. These models can be categorized as Key-Value models, Graphical Models, Object oriented models, Logic models, Markup scheme models, and ontology based models. They each have their own advantages and disadvantages and the choice of model which should be adopted depends upon the implementation and needs.

In this project, I choose to use a Markup scheme model. A markup scheme model is a hierarchical data structure including markup tags specifying attributes and context values. Profiles are typically used with this kind of model. Some of these profiles are defined as an extension of the Composite Capabilities/Preferences Profile (CC/PP) [36] or the User Agent Profile (UAProf) [37] standards. Additionally, there are some profiles specifically to support instant messaging and presence, such as Common Profiles for Instant Messaging (CPIM) [38] and Common Profiles for Presence (CPP) [39]. These profiles define a group of operations and parameters to allow

communications between different Instant Messaging and Presence protocols, as specified in RFC 2779 [43]. Based on this specification, the Present Information Data Format (PIDF) [42] was developed as a common presence data format for CPP-compliant presence protocols. By following the PIDF standard presence information can be transferred across different CPP-compliant protocols without modification. Because our system implements SIP SIMPLE to distribute context information among different entities, PIDF will be used to encode context information.

2.5.2 PIDF

As stated above, in a context-aware system, presence information transmitted via the SIP SIMPLE protocol among different entities needs to follow a standard data format. Specifically, when presence information is published to a PA and when subscribers receive their desired notification from PA, the data format has to be uniform. PIDF meets my requirements. PIDF, as specified in RFC 3863 [15], defines a base presence format and presence state values. It is also extensible as required by Instant Messaging and Presence Protocol (IMPP). PIDF defines a minimal set of presence state values which are specified in an IMPP Model document [45].

A presence document uses XML to encode presence information. Therefore, a PIDF object should be a well formed XML document. It must have the XML declaration and an encoding declaration in the XML declaration, for example, “<?xml version='1.0' encoding='UTF-8'?> ”. It will also generally have some basic PIDF elements associated with the XML namespace name “urn:ietf:params:xml:ns:pidf”. Here is an example of PIDF using the PIDF XML namespace.

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:someone@example.com">
<tuple id="en24xy">
<state>
<basic>closed</basic>
</state>
<contact >IP address='192.168.1.1'</contact>
</tuple>
</presence>
```

Here the object <presence> is the root of an “application/pidf+xml” object. This tag must contain a namespace declaration “xmlns” to indicate which presence document it is based on, and an “entity” attribute with a URI value representing the presentity which published this presence document. This presence object can include any number of <tuple> elements, followed by any number of <note> elements, followed by any number of *optional* extension elements.

In practice, there is a broad range of context information which can be encoded in a presence document. Therefore, the pre-defined elements might not be sufficient. As extensions, two common solutions are used to support additional kinds of context information. In one approach a context application defines specific elements under the <state> element. This requires that an extra namespace has to be imported into document. Another approach is to extend the basic format of PIDF, such extensions are: RPID, Timed Presence-extensions, and PIDF-diff [10]:

RPID (the Rich Presence Information Data format) [45]**:** It defines additional presence attributes to describe objects, such as location, user input, person, and service. These elements are in addition to <tuple>, i.e., <location>, <device>, <print>, etc.

Timed Presence-extensions (Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals) [46]: Basic PIDF only deals with current presence information. This extension defines a <timed-state> element to describe a wider range of presentity state information covering its past, present, and future time period.

PIDF-diff (PIDF for Partial Presence) [47]: In the original PIDF whenever an update needs to be published it always contains the full presence information of that presentity. This extension enables the updated presence information (which is PIDF document) to contain only the changed part. Therefore, bandwidth can be saved.

2.6 SER

2.6.1 Introduction

In our project, each PA transmits context information using SIP SIMPLE. However, in a large context-aware system, a server can be used to act as an intermediary between sensors and application to process the context information. SER is a high-performance, configurable, and free SIP server. It has integrated redirect, proxy, and registrar servers; and includes a PA module and MySQL database interface [30]. Thus SER provides us with a suitable high performance PA and also provides many of the other elements of a complete context server.

SER was initially developed by a team of developers employed by Fraunhofer Fokus. It is now a part of the iptel.org project. The iptel.org website serves as the entry point to all information about SER. However, on June, 2005, the project forked creating OpenSER. Both SER and OpenSER share the same configuration files (ser.cfg), and are somewhat similar. However, there are some differences emerging with time, thus the configuration files are no longer completely compatible. So the

user should decide which one to utilize. In Mohammad Zerifi's project, he chose to use SER. Therefore, I will continue to use SER in this project.

In SER, the configuration file “ser.cfg” configures the core so that it knows what to do when it receives SIP messages and allows the user to configure optional modules for handling SIP messages. The different modules provide most of SER’s actual functionality. The ser.cfg file specifies which modules needed to be loaded and also sets variables which are passed to these modules. The process of configuring the SER is described in Mohammad Zarifi's thesis in Appendix A [10]. In this project, I extend this by implementing a Presence module to provide a presence Subscribe-Notify mechanism and modify the CPL module to allow the Call Secretary to specify personal call processing (based upon presence information). Details of these modules and our changes will be provided in the following sections.

2.6.2 Presence module

This module implements a server to process Subscribe requests from watchers and to distribute Notify messages to subscribers. When any changes in presence state occur, it publishes requests containing fresh presence information to update the original subscription information, and distributes he updated Notify messages to the relevant subscribers. This module implements the essential part of the presence Subscribe-Notify mechanism [31].

2.6.3 CPL module

The Call Processing Language (CPL) is an XML based language, designed to describe and control Internet telephony services [32]. It provides an easy means to implement call processing programs on a SIP or H.323 signaling server. This language is purposely limited to allow no variables, loops, or the ability to call external programs; thus it is considered safe enough to be executed on the signaling server. Because of these limitations not only can the signaling server administrator utilize CPL to define service policies, but end users can create their own customized CPL scripts to manage their own call functionalities. These CPL scripts are usually associated with particular SIP URIs. On the signaling server (which is typically the incoming and outgoing SIP proxy), each registered user has a CPL script associated with their account. This script can be defined by an administrator or users can upload their own script to overwrite the default script or to replace their previous scripts. When a SIP INVITE is received, the usual priority of processing is to execute the user’s own CPL scripts followed by the CPL script defined by the administrator, and finally the default call processing configuration. Before CPL can be used with SER, the ser.cfg file has to be configured to enable the CPL module on the SER server. The details of this are presented in Appendix A.

2.6.3.1 Creating, uploading, and removing CPL scripts

There are three CPL script operations: creating, uploading, and removing a script. Each of these operations will be described below.

Creating CPL scripts

There are three methods to generate a CPL script. It can be easily created by hand, i.e., using an editor to create the CPL script. A user can use web middleware to create the script, typically by using a web tool which knows CPL's syntax. However, the easiest way is utilizing graphical tools. Graphical User Interface (GUI) tools provide inexperienced users with simple interfaces to provision CPL scripts. For example, one popular and free GUI tool CPLed [64], a java based application. It can be used to create, edit, or upload CPL scripts to a SIP server. In the project, I utilize CPLed to create CPL scripts.

Uploading scripts

After successfully creating a CPL script, the script can be uploaded to SER through a SIP Register message or via SER's FIFO (command line interface) facility.

- SIP Register Message

  Because the CPL script is encoded in XML, a SIP Register message can contain the script in its body. When this message reaches SER, the body is parsed, and the CPL script is extracted. In SER, a SER database stores CPL scripts in a CPL table. In this table each user is associated a specific CPL script. When a incoming call is addressed to certain user, the script (if there is one) will be executed and the call processed as specified in the script. In the project, I use the SIP Register message to upload CPL scripts into SER.

  CPLed can use HTTP or the SIP Register method to download, upload, or remove CLP scripts from SER. Note that utilizing the SIP Register method requires authentication support.

- Serctl FIFO Interface

  SER provides both a command-line interface (CLI) and a web-based interface for administration. Specifically, "serctl" is a command-line utility which enables a user to perform most management tasks needed to operate SER, such as start/stop SER, manage users, or monitor the server [34]. Alternatively, "serweb" is PHP based web application used for creating new SER accounts and managing them.

  SER's FIFO server is built-in facility to program SIP services. It provides a

simple textual interface that enables external applications to communicate with SER. Serctl can be used to send instant messages, manipulate user contacts, monitor the server's health, etc. This command-line utility can manipulate FIFO functionalities.

In order to upload a CPL script, a command similar to the following can be used:

serctl fifo LOAD_CPL user@domain /path/to/cpl/script

(e.g. serctl fifo LOAD_CPL kewang@server /opt/ser/etc/ser/cplscript.cpl.xml)

After receiving a new CPL script, SER will confirm its validity. This is to ensure that when processing an incoming call there will be no failure due to this associated CPL script. Note that this does not mean that the CPL script will do what the user expected, only that it will not harm the SER server.

Removing scripts

To easily remove scripts from SER's database the FIFO facility can be used.

serctl fifo REMOVE_CPL user@domain

The current CPL script for a given user also can be removed via a Register message. Note that in the example below, the removal happens because of the content of the "Accept" header.

REGISTER sip: @130.237.15.238 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:5060 ;branch=z9hG4bKJla2aBfl7
From: <sip:loloandnono@130.237.15.238>;tag=sge1sg1
To: <sip:loloandnono@130.237.15.238 >
Call-ID: 2395@130.237.15.227
CSeq: 18 REGISTER
Accept: application/cpl, application/sdp, text/html
Contact: <sip: hlllab4@130.237.15.227:5029 >
Content-Type: context_type
Content-Length: 0

2.6.3.2 CPL script structure

A CPL script contains two kinds of information, ancillary information about the script and the call processing actions. Ancillary information is information necessary for a server to correctly process a script. This information is not related to any specific call processing operation or decision. However, details of this ancillary information are not currently defined, so this represents an area for future development. Call processing actions are the primary purpose of CPL scripts. Generally, an action is a tree structured collection of nodes. These nodes describe where the predicates are

tested. If the predicate is true, then the associated action is taken by SER to process this call. When a script is executed, it starts from the root node and works its way down to subnodes based on the previous nodes. The script is processed until an applicable action is found (if any exist), then the action is taken and the script terminates [35].

2.6.3.2.1 Actions

Specifically, there are two kinds of call processing actions, top-level actions and subactions.

- **Top-level actions** are grouped into two types of actions: Incoming and Outgoing. The first are triggered when telephony signals arrive (i.e., when the owner of this script receives a call). The later actions are performed when the owner of this CPL script makes a call. The Incoming and Outgoing trees are independent, but are specified in the same CPL script.
- **Subactions** are actions containing predicates and operation to be applied to a call.

Both top-level actions and subactions are presented as a tree of nodes and outputs, as showing in Figure 10.

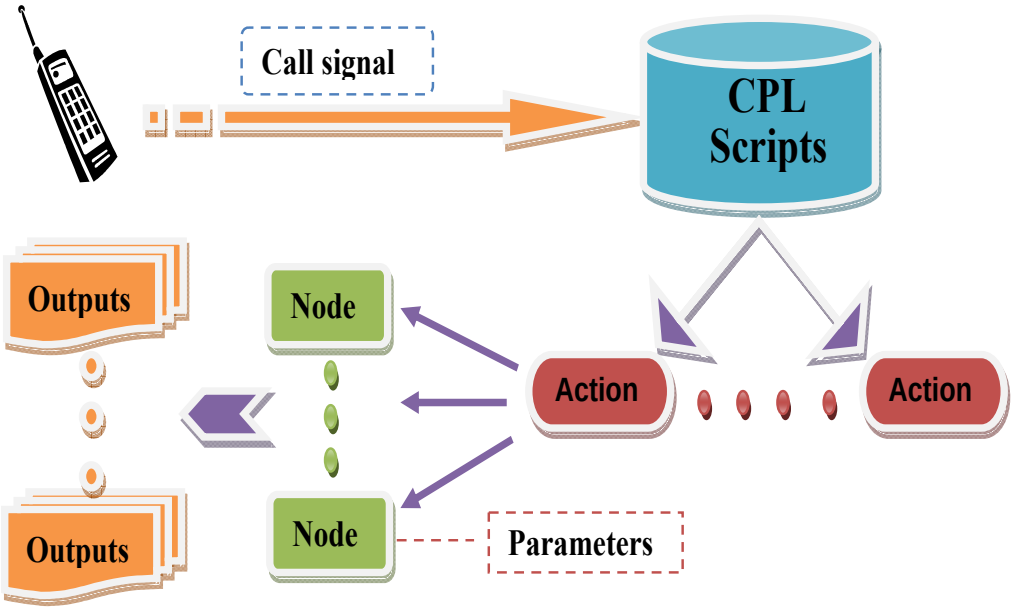

Figure 10: Actions and Nodes

2.6.3.2.2 Nodes categories

CPL scripts are represented as XML documents. In scripts, nodes and output operations are encapsulated in an element with an XML tag and the parameters are represented by tag attributes. There are four categories of nodes: Switches, Location modifiers, Signaling operations, and Non-signaling Operations.

Switches

Switches represent the choices that a CPL script can make**.** All switches are arranged as a list of conditions. Each condition consists of a node and output, the former defines which items are to be matched, while the output points to the next node to be executed. The match criterion is based on either attributes of the call signaling event or items independent of the call. All conditions can have different priorities and are listed in the CPL script in priority order, thus they are tried one by one until the first one is matched. Based upon the output of this match, the next node is selected.

There are five different categories of switches:

- **Address switches** allow a CPL script to make decisions based upon a specific address being present in the original call request message. Specifically, the node defines addresses to be matched. Additionally, there are two parameters in this node: “field” and “subfield”, and three parameters in the output: “is”, “contains”, and “subdomain-of”. The node parameters specify which elements of this address are to be matched; while the output parameter indicates the matching criterion.

  Node parameters

  Field defines which address is to be considered for the switch. They are “origin”, “destination”, and “origin-destination”.

  Subfield an optional parameter specifies which part of the address is to be considered, such as “address-type”, “user”, and “host”, etc.

Node:	*"address-switch"*	
Outputs:	*"address"*	*Specific addresses to match*
Parameters:	*"field"*	*"origin", "destination", or "original-destination"*
	"subfield"	*"address-type", "user", "host", "port", "tel", or "display" (also: "password" and "alias-type")*

  Output parameters

  Is the output of this match operator is the result when there is a exact match of the address-switch condition. The match can be made against any subfield or for the entire address (if there is no subfield parameter was specified).

  Subdomain-of when the subfields “host” and “tel” are match ed, then the result is the output specified.

  Contains this match operator applies only for the subfield “display”.

Output:	*"address"*	
Parameters:	*"is"*	*Exact match*
	"contains"	*Substring match (for "display" only)*
	"subdomain-of"	*Sub-domain match (for "host", "tel")*

- **String Switches** allow a CPL script to make a decision based on strings present in a call request. Specifically, there is one node parameter, "field" and two output parameters: "is" and "contain".

  Node parameter

  Field — specifies which form of string is to be matched. There are four types of fields namely: "subject", "organization", "user-agent", and "display". The value of each of these fields is a free-form Unicode string with no structural limitation.

  Output parameter

  Is — indicates this is whole string match.

  Contain — indicates this is substring match.

- **Language Switches** allow a CPL script to make a decision based on the languages specified in the call. If a specific language is defined by the call originator as their preferred communication language. Then this switch will be matched only if this language falls in the language-range defined in the output parameter. There is no node parameter and only one output parameter: "match".

  Node parameter:

  None

  Output parameter:

  Matches — this output is matched if the given language matches a language-range of the call.

- **Time Switches** allow a CPL script to make decisions based on the time and/or date the script is being executed. It is matched if the call request fulfills the time and/or date restricted in output parameter. Specifically, there are two node parameters, and several output parameters.

- **Priority Switches** allow a CPL script to make decisions based on the priority indicated for the call. A specific priority defined in the node is compared with the priority of the call request. There are three different results of this comparison, "less", "greater", or "equal" generate three different outputs.

  Node parameters:

  None

  Output parameters:

  Less — matches if the priority of the call request is lower than the specified priority.

  Greater — matches if the priority of the call request is higher than the specified priority.

  Equal — matches if the priority of the call request is the same as the specified priority.

Location modifier nodes

Location modifier nodes are a set of locations indicating where a call is to be directed. This information is not given as node parameter, but rather is stored in an implicit global variable. To manage the location set, such as adding or removing locations from it, three available locations nodes are defined.

- **Explicit location node** adds specific (literal) locations to the current location set. There are three parameters: “url” presents the address in URL format to be added to the location set; “priority” indicates the priority of this location which has a value from 0.0 to 1.0; and “clear” indicates whether the location set should be cleared before adding the new location value. Note that the output is the next node.

Node:	*"location"*	
Outputs:	*None*	*(Next node follows directly)*
Next node:	*Any node*	
Parameters:	*"url"*	*URL of address to add to location set*
	"priority"	*Priority of this location (0.0-1.0)*
	"clear"	*Whether to clear the location set before adding the new value*

- **Location lookup node** fetches a location value from some outside source and adds it to the current location set. There is one mandatory parameter and two optional parameters in such a node. There are three possible outputs, “success”, “notfound”, and “failure”.

  Node parameters:

Source	the source of the lookup which could be a URI, or a non-URI value.
Timeout	a positive integer indicating number of seconds how long the script can wait for the lookup operation to be performed. The default value is 30 seconds.
Clear	specifies whether the location set should be cleared before the new locations are added.

- **Location remove node** removes locations from the location set. It specifies a URI in the node parameter. If this URI matches, then the correspondent location entry is deleted from the location set. If there is no specific URI given in the node parameter, then all locations are removed from the set. This node has no explicit output. In XML syntax, the XML “remove-location" tag directly encloses the next node's tag.

Signaling operation nodes

There are three types of signaling operations that can be performed via three different nodes: “proxy”, “redirect”, and “reject”. Each of these operations causes specific signaling events in the underlying signaling protocol.

- **Proxy node** causes the ongoing call to be forwarded to the locations listed in the location set. Three specified parameters produce output to the current call attempt. If the call attempt is successfully processed, then the CPL script execution terminates and the server forwards this call to the specified location. However, if this call attempt fails, then one of the five outputs specifies the next node. The “busy” output is followed if the callee is busy. However, if the call generate an 180 Ringing response *and* the callee did not answer it, then the “noanswer” output is followed. If the call was redirected, then the “redirection” output is followed. If the call setup failed, then the “failure” output is followed. If any of the former outputs is not specified, then the “”default” output will be followed instead. If an output is followed, then control is passed to next node, and when a proxy operation is finally completed all locations which have been used are deleted from the location set.

  Node parameters

Timeout	a value defining how long to try to establish the call session.
Recurse	indicate whether to recursively look up redirections.
Ordering	specifies an order in which each of the locations of the location set is going to be tried. There are three values, "parallel", "sequential", and "first-only".

- **Redirect node** causes the server to direct the callee to try calling the currently specified set of locations. This node has no output and no next node.

- **Reject node** cause the server to reject the call setup attempt. This node has no output and no next node.

Non-signalling operation nodes

This group of nodes provides operations which do not affect and are not based on a telephony signaling protocol. The Mail node uses Email to notify a user of the state of the associated CPL script. The Log node causes the server to create and store log information about the call to non-volatile storage.

3. Goals

3.1 Context Agent

A Context Agent was developed to provide "real-time" context information of meeting room occupancy to the Meeting Room Booking System (MRBS) [69]. The room occupancy context information is initially collected by Xueliang Ren's "A Meeting Detector" [8]. This meeting detector is a sensing system, used to monitor a meeting room's occupancy state. When a person enters or leaves a meeting room this triggers the sensing system to publish an SIP message to a context server. The context server parses and processes the received publish message, stores the updated room occupancy context associated with the indicated room in its database. The Context Agent subscribes to room occupancy information from the context server by sending Subscribe Request message to this server. Meanwhile, it listens for incoming Notify message from the context server with updated room occupancy information. When receiving a Notify message, the Context Agent extracts room occupancy context and update the associated database entry within the MRBS database. Figure 11 shows the overall architecture.

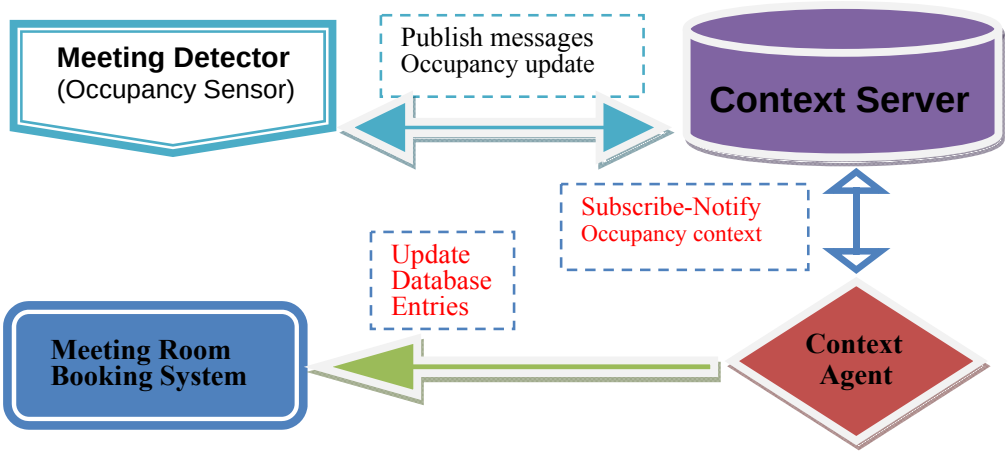

Figure 11: Context Agent Architecture

The Context Agents is responsible for:

1. Compiling Subscribe messages.
2. Sending Subscribe messages to a context server. Listening for incoming Notify messages from this context server.
3. Upon receiving a SIP message from the context server, the Context Agent examines the validation and message type of the message. Different SIP messages will be processed differently. For each valid Notify message containing context information, the Context Agent replies to the context server with a 200 OK message, then extracts the relevant information from the message, e.g. meeting room name and the number of people.
4. Connecting to database within MRBS and updating the associated entry.

3.2 Call Secretary

A Call Secretary is used to automatically redirect a subscriber's incoming call to his or her voice mail when he or she is in a meeting. The application detects if a subscriber is in a meeting or not based on matching results current context information against the pre-defined criterion. These criterions are associated with meeting information specified by the subscriber, for example, the starting and ending time of a meeting, the planned meeting room name, meeting size (we define that a "Small" meeting requires at least 2 participants, while a "Big" meeting requires 5 or more participants). The Call Secretary acquires context information associated with its subscribers: the current time, the location of the subscriber, and the room occupancy state. The current time can be simply retrieved by reading a local time of day clock (generally this will be synchronized to an Internet time server using Network Time Protocol (NTP) [70]. The user's physical location and room occupancy status context information can be collected and published to a context server by separate sensing applications. The room occupancy context information is initially collected by Xueliang Ren's "A Meeting Detector" [8]. This meeting detector is a sensing system, used to monitor a meeting room's occupancy state. The location context information is collected by Haruumi Shiode's "Location Sensing" [9]. The Call Secretary subscribes to these two context information sources via the context server. When all three types of context information match the user's pre-specified meeting criterion, then the Call Secretary will redirect incoming calls for this subscriber to his or her voice mail. Figure 12 shows the overall architecture of the system.

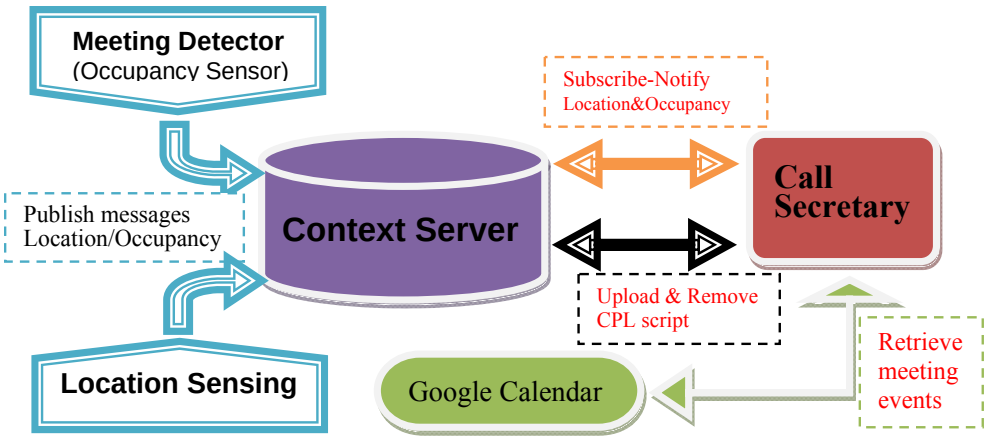

Figure 12: Call Secretary Architecture

A Call Secretary is responsible for:

1. The subscribers schedule their meeting events using their favorite calendar application (e.g. iCal of Mac OS or Google Calendar). Because Google Calendar is widely used and accessible via Google's Calendar APIs [48], I choose this calendar for the subscribers' meeting events. Some meeting information needs to be specified to create meeting events in the user's Google Calendar, such as the

starting and ending time of the meeting, the meeting room's name, and the size of the meeting. (Note that I leave interfacing to other calendaring systems as future work.)

2. When subscribing to the call secretary service, subscribers provides their Gmail/Google Calendar account name and password (Gmail and Google Calendar are accessed via one account), and the address of their voice mail server. There are some possible platforms that can be used to complete their subscription to this service, for example, subscribers can log on via a web-based interface to register their information or they can send a well formed SMS/MMS message to a server address. After registration, the subscriber's information is stored in a database. I leave the details of automatic registration to future work and for the purposes of the prototype make database entries manually.
3. The Call Secretary retrieves subscribers' meeting events from Google Calendar via Google Calendar APIs using the registered Google account information.
4. The Call Secretary sends Subscribe Request messages to a context server to subscribe to both location context information for each of the subscribers and for meeting room occupancy context (for the set of meeting rooms which these subscribers plan to use in the near future). Then it listens for incoming Notify message from the context server.
5. When context information contained in the Notify message matches the meeting criterion, this means the current time is between the starting time and ending time of a meeting, a subscriber appears inside of the specified meeting room, and enough participants are in this meeting room (with respect to the planned meeting size), then this subscriber is considered to be in a meeting.
6. When a subscriber is detected to be in a meeting, the Call Secretary compiles a CPL script according to the subscriber's voice mail address and uploads the resulting CPL script to the context server (acting as the subscriber's incoming SIP proxy) which will trigger this server to redirect incoming calls to this subscriber to his or her voice mail. When the subscriber is detected to no longer be in a meeting, then his or her CPL script will be removed from the context server.

As shown above, the Call Secretary utilizes four types of context information: the current time, the planned meeting room, this meeting room occupancy state, and the subscriber's current location. In my prototype, the current time is retrieved through calling Java method. The planned meeting room is extracted from the subscriber's calendar information. The meeting room occupancy state can be directly extracted from Notify message of room occupancy which contains the name of the meeting room and the number of participants. However, Notify message indicating the user's current location generally contains geo-coordinates as longitude and latitude values, thus this information can not be used directly to decide if the subscriber is in a particular meeting room or not. I need to be able to translate the coordinate system of the meeting rooms (or of the users) to a common coordinate system in order to map

both subscribers and meeting rooms to a common set of geo-coordinates in order to decide if the subscriber's location is in the meeting room. This coordinate transformation approach will be described in section 4.3.1.1.

4. Implementation

4.1 SER server

As described in chapter 2, Mohammad Zarifi Eslami (an earlier thesis student) selected SER (SIP Express Router) to be as a context server [10]. He modified the PA module of SER to support both Presence and Location events. Based on his work, I was supposed to add Occupancy event support to this PA module and to configure the CPL module for SER. However, his SER server had crashed before my implementation effort started. Unfortunately, Mohammad Zarifi had downloaded an unstable version of SER (version ser-0.10.99) and made all of his modifications on this version of source code. As I choose the latest stable version of SER (ser-2.0) to based my implement on, his modified source code could not be directly applied to this new version of SER. Because of a full implementation of SER with presence support was out of this thesis project's scope, I implemented the required basic SER presence event support and simulate location and room occupancy events by developing Java applications which communicate with clients via SIP messages. I use this approach to evaluate the two applications in next chapter. An advantage of using this approach was that testing could be very well controlled; the disadvantage is that a complete working system could not be deployed and real user measurements performed.

Install and configure a SER

I utilized the latest SER version 2.0 (by downloading ser-2.0.0_src.tar.gz) published on August 6, 2008 on the iptel.org website [49]. There are instructions for installing SER in the file "INSTALL" of the root directory of this SER source code.

To unpack the source package, one executes the command (in this case on the host "ccsleft" when connected to the "/usr/src" directory:

ccsleft: /usr/src/ # tar xzf ser-2.0.0_src.tar.gz

To build SER from its sources, some pre-requisites are needed:

- gcc or icc : gcc >= 2.9x; version 3.1 or higher recommended (older versions will work, but they may require some options tweaking for best performance)
- bison or yacc (Berkley yacc)
- flex
- GNU make, version 3.79 or newer (on Linux this is the standard "make", on *BSD and Solaris it is called "gmake")
- sed and tr (used in the makefiles)
- GNU tar ("gtar" on Solaris) and gzip are only necessary if you want "make tar" to work

- GNU install, BSD install, or Solaris install if you want "make install", "make bin", and "make sunpkg" to work

Compiling the core and a set of standard modules is easy with the following command:

ccsleft: /usr/src/ser-2.0.0 # make all

However, this project needs some specific modules that are not included as standard modules by default, one of these modules is the presence module. I can specify additional modules or groups of modules to include:
ccsleft: /usr/src/ser-2.0.0 # make group_include="standard presence" all (this command includes the entire presence module group)

It is also possible to compile all modules with the following command:

ccsleft: /usr/src/ser-2.0.0 # make group_include="standard standard-dep stable experimental" all

Note that when building some of the modules, there are additional requirements, specifically:

- libmysqlclient & libz (zlib) for MySQL support (the mysql module)
- libexpat for jabber gateway support (the jabber module)
- libxml2 for the cpl-c (CPL support), pa (presence) and xmlrpc modules
- libradiusclient-ng (> 5.0) for radius support (the acc_radius,auth_radius, avp_radius, and uri_radius modules)
- libpq for PostgreSQL support (the postgres module)
- libssl for SSL/TLS support (tls module)

Once you have build all the modules that you wish to use, then it is time to install SER in directory */usr/local*:

ccsleft: /usr/src/ser-2.0.0 # make prefix=/usr/local install

After you have configured SER using the ser.cfg file, it is not time to start SER:

ccsleft: # /usr/local/sbin/ser –E

After the install completed, a basic database for SER needs to be created. There is a script in directory */SER/scripts/* of the source packet. I used the MySQL database, thus the script “ser_mysql.sh” is executed to create SER database.

ccsleft: # /usr/src/ser-2.0.0/scripts/mysql/ser_mysql.sh create

SER uses “Serctl” tool to manage this database. Serctl is a set of command line utilities which has to be downloaded from iptel.org website [50] (most of the previous SER versions include serctl). The commands below can be used to specify

the domain name for SER and to create a new user named “loloandnono” with password “heslo” (the default SER password).

```
$ ser_user add loloandnono
$ ser_uri add loloandnono loloandnono @ser
$ ser_cred add loloandnono loloandnono 130.237.15.238 ser helso
```

As introduced previously in section 2.6, the configuration file “ser.cfg” is the core of the SER server. After installation, there will be an initial ser.cfg file available in the directory */usr/local/etc/ser* with basic functions. However, this configuration file needs some modification to support the database functions, presence module, and cpl-c module. The ser.cfg file used in this project is presented in Appendix B.

4.2 Context Agent

For the developing environment I have used:

Linux (OpenSUSE 10.3), Java JDK 1.6.0_06, JRE 1.6.0_07, MySQL 5.0.25, SER 2.0.0, and Netbeans IDE 6.1 running on a DELL “OptiPlex GX620” computer equipped with a 2.80 GHz “Intel Pentium D” processor and 2.0 GB of memory.

A Context Agent communicates with the SER server via SIP messages. It sends two types of messages to SER: Subscribe Request messages for meeting room occupancy information and OK messages to acknowledge a successful transaction. On the other hand, SER sends two types of SIP messages to the Context Agent as response: Notify messages and OK messages. We will explain how the Context Agent handles subscriptions and processes these SIP messages in the following sections. The source code of the Context Agent is presented in Appendix D.

Compiling SIP messages

Subscribe Message

There are 6 meeting rooms available in the local Meeting Room Booking System: “Mint”, ”Grimeton”, ”Open Area”, ”Motala”, and “Hörby”. The Context Agent subscribes to each meeting room’s occupancy information by building a specific Subscribe message. Thus, after starting up, the Context Agent needs to create 6 different Subscribe messages associated with these 6 meeting rooms. Figure 13 shows a Subscribe message to subscribe for the occupancy information of meeting room “Mint”. The explanation of each line of this Subscribe message can be found in section 2.3.3.

```
SUBSCRIBE sip:mint@130.237.15.238:1028 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:1028;branch=z9hG4bKSUpgPNO
From: <sip:ccsleft@130.237.15.238>;tag=1E6d
```

To: <sip:mint@130.237.15.238>
Call-ID: 821127@130.237.15.227
CSeq: 19025 SUBSCRIBE
Max-Forwards: 70
Event: occupancy
Accept: application/pidf+xml
Contact: <sip:ccsleft@130.237.15.227:1028>
Expires: 600
Content-Length: 0

Figure 13: A Subscribe Message for Occupancy Context of "Mint"

200 OK message

Upon receiving a valid Notify message from the SER server, the Context Agent should reply with a 200 OK message. Such an 200 OK message is created according to the content of the Notify message received. Figure 14 shows a 200 OK message when received a valid Notify message from SER containing occupancy information for the meeting room "Mint".

SIP/2.0 200 OK
From:
<sip:ccsleft@130.237.15.238>;tag=xlB3;received=@130.237.15.227
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Content-Length: 0

Figure 14: Example of a 200 OK Message

4.2.2 Sending SIP messages

In our scenario, SER server uses IP address 130.237.15.238 and the port number 5060. To submit a subscription, the Context Agent simply packs the Subscribe message into a UDP (user datagram packet) packet and sends the packet to this IP address and port number. In the same way, the 200 OK message is sent to SER server as a response to a valid Notify message.

4.2.3 Processing incoming SIP messages

The SIP message sent by the Context Agent includes its IP address and port number, thus the SER server knows where to sends response. After sending a Subscribe message to SER, the Context Agent listens on a port (i.e., a socket is

listening on the same IP address and UDP which were used to send the SIP message) for any incoming messages. Generally, there are 3 types of SIP message sent by a SER server: 202 Accepted message, a Notify message, and its sending Exception message. The Context Agent processes each type of message in a different manner. The processing of these three messages are described below.

4.2.3.1 202 Accepted message

Upon receiving a valid Subscribe message from the Context Agent, SER replies with a 202 Accepted message. Note that some papers confused 200 OK and 202 Accepted message and state that a 200 OK message was used as the response to a Subscribe message. While the 2xx messages represent successful final status responses if a "INVITE" message is accepted, then SER replies with a 200 OK message. Whereas, a 202 Accepted message indicates that SER accepted this subscribe request. However, upon received a 202 Accepted message from SER, the Context Agent has to wait for an associated Notify message to complete this subscription. This message is described next.

4.2.3.2 Notify messages

The Notify message sent by SER server is divided into 3 different types: a message with context information, a message without context information, and a message informing the recipient that its subscription has expired. Depending upon which type of Notify message the Context Agent receives different processing will be applied. Note that the Context Agent currently simply ignores errors and irrelevant Notify message from SER.

4.2.3.2.1 Notify message contains room occupancy context information

This type of Notify message will be sent to by SER in two scenarios: (1) when SER receives a Subscribe message and the context information being requested is available, SER sends a 202 Accepted message and a Notify message with occupancy information. (2) when SER receives a publish message containing an update of meeting room occupancy during a subscription's lifetime, SER sends a Notify message containing the updated context information. Figure 15 shows a Notify message containing occupancy information for the meeting room "Mint" which currently has 11 persons in it.

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
```

```
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="0xb58d60e0x4a4b0c39x4715c26e">
<status><basic>open</basic>
<occupancy>
<description>Electrum</description>
<room>Mint</room>
<value>11</value>
</occupancy>
</status>
<contact priority="0.80">KeWang</contact>
<note>occupancy</note>
</tuple>
</presence>
```

Figure 15: A Notify message contains room occupancy context information

This type of Notify message contains two parts: a SIP header and a body. The body is encoded in PIDF XML format. The Context Agent extracts the relevant context information from the body. In my case, I am concern about the **room name** and **occupancy value** that are tagged <room> and <occupancy> (respectively) in the body. The context information is used to update a database within MRBS, which we will describe in the section 4.2.4. At the same time, a 200OK message needs to be sent to the SER server indicating that the Notify message was successfully received.

4.2.3.2.2 Notify message without context information

This type of Notify message is usually sent immediately after a 202 Accepted message when there is currently no context information available to SER corresponding to the Subscribe messages request. This type of Notify message is sent not to update context information, but simply to complete the subscription process. However, if there is context information available to SER, then the first type of Notify message will be sent to convey this context information to the subscriber. Figure 16 shows a Notify message without context information. Upon receiving this type of Notify message, the Context Agent sends a 200 OK message to the SER server indicating that the Notify message was successfully received.

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="none">
<status><basic>closed</basic></status>
</tuple>
</presence>
```

Figure 16: A Notify Message without Context Information

4.2.3.2.3 Notify message indicating an expired subscription

When a subscription is expired SER sends this type of Notify message to inform subscribers. It has “terminated” as the value of Subscription-State attribute. If subscriber wants to continue this subscription, a Subscribe Request message has to be sent to SER.

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: terminated
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="none">
<status><basic>closed</basic></status>
</tuple>
```

</presence>

Figure 17: Notify message indicating an expired subscription

4.2.4 Updating database

I created a new table “mrbs_occupancy” within the MRBS database. This table has three columns: id, room_name, and occupancy. The Context Agent extracts the room name and occupancy values from a received Notify message containing context information, then connects to the database and updates the associated entry according to these values. After the entry is updated, the database connection is to be closed. Note that the database connection will be set to keep open when this application is applied to practical meeting room occupancy detecting (which means there would be much database operations).

4.3 Call Secretary

The developing environment used when developing and testing the Call Secretary application consisted of a Dell Optiplex model 755 PC with zz gigabytes of memory and qqq Gigabytes of available disk space. This computer was running: Linux (OpenSUSE 10.3), Java JDK 1.6.0_06, JRE 1.6.0_07, MySQL 5.0.25, SER 2.0.0, Netbeans IDE 6.1, Franson GPS Tool SDK version 2.3, Google SketchUp 6 [60] . In addition a GlobalSat company BT-338 model Bluetooth GPS receiver [68] and a Hewlett-Packard Company (HP) iPAQ Pocket PC h5550 [58] were used for the mobile client.

For each subscriber to the Call Secretary service, the incoming call redirection function is triggered based on four types of context information: the subscriber’s current location, the room which is planned for the meeting, the meeting room’s occupancy, and the current time. Only if all of these four elements of context information match pre-defined criterion will the Call Secretary redirect the incoming call into the voice mail of this subscriber. In the following sections, I will first explain how to collect each element of context information, then I will describe the approach used to implement this application in Java.

4.3.1 Collecting contexts

4.3.1.1 Location context

The subscriber’s location state can be monitored by one or more sensing applications. A location change will be published to the SER server via a SIP message.

SER processes this message and stores the updated location context information in a database entry associated with the subscriber's identity. Thus the Call Secretary can subscribe to learn this location context from SER.

However, there is currently no operational sensing application available to provide location context information to SER. A previous thesis student (Haruumi Shiode) developed a location sensing application based on WLAN signal strength [9]. His application provides location context updates to SER. However, this application currently was not interfaced to SER. While there are quite a number of location context sensing approaches these are the topics of other thesis projects and not part of this thesis project. Therefore in the evaluations, I simulate the Subscribe-Notify scenario by using a Java application "Notify Sender" to directly send Notify messages to the Call Secretary containing specific location context information.

In our scenario, location context is represented as geographic coordinate values, i.e., Longitude and Latitude in the WGS82 world coordinate system. Because geo-coordinates are widely used by location sensing applications, this choice gives the Call Secretary extensibility to accept input for additional location sensing applications that output geo-coordinates. However, detecting a subscriber's geo-coordinates is not sufficient. The Call Secretary needs to know if the subscriber is currently in a specified meeting room. Thus, all meeting rooms' bounding geo-coordinates have to be measured. When received a Notify message containing the subscriber's geo-coordinates, the Call Secretary will compare these coordinates with each meeting room's coordinates to determine if this user is within the bounds of a given meeting room.

The project was conducted at the Center for Wireless Systems at KTH (Wireless@KTH) and the Department of Communication Systems. Together these facilities have 5 meeting rooms. These meeting rooms are located on two different floors of a building. To measure each meeting room's geo-coordinates, several steps needs to be done: firstly, a set of reference points outside of the building needs to be selected, then I measure their geo-coordinates (I will later explain the reason to select reference points outdoors rather than indoors). Secondly, relative to these reference points' geo-coordinates and a scaled map of the building, I can calculate the bounding geo-coordinates of each indoor meeting room. As all of these meeting rooms are all rectangular, only the geo-coordinates of the four corners of each room are necessary to determine a geo-coordinate bounding box for each meeting room. Given this data, upon receiving a Notify message containing a subscriber's current geo-coordinates, the subscriber's location in terms of a meeting room can be determined by comparing the subscriber's geo-coordinates with each meeting room's bounding geo-coordinates. Note that in a practical implementation it would be desirable to also use the history of where the user has been recently to help tell if they are approaching a meeting room or departing from a meeting room.

4.3.1.1.1 Measuring geo-coordinates of a reference point

In this section, I introduced how to measure the geo-coordinates of a outdoor point via a Bluetooth enabled GPS receiver along with an iPAQ Pocket PC. Following this I will utilize this point as a reference in order calculation of the indoor meeting rooms' geo-coordinates.

Choosing a reference point

First of all, I started that the reference point has to be located outside of the building. This is necessary because the type of the GPS receiver I am using does not have good GPS reception inside of a building. Thus it can not provide accurate value (there are some types of GPS receiver that support indoor positioning [51]). Additionally, this reference point has to be some distance away from the building itself in order to reduce the *multipath effect* [65]. Such multipath corrupts the direct GPS signal by one or more signals reflected from the local surroundings (the three Electrum buildings in our scenario). The reflected signal might also cause interference with the signal from the direct path. Figure 18 is a photo taken from the outside of the "Electrum" building (actually both buildings in the picture are part of the set of Electrum buildings, but the building I have marked as Electrum is Electrum building one - and I will refer to it simply as the Electrum building in the rest of this thesis). Figure 19 is a snapshot of the building as display by Google Earth. Inside of the red zone is the entire building, whereas inside of the green zone is the portion of the building where the meeting rooms that I am concerned with are located. The marker "V" indicates the reference point which was 10 meters perpendicular distance away from the building's wall.

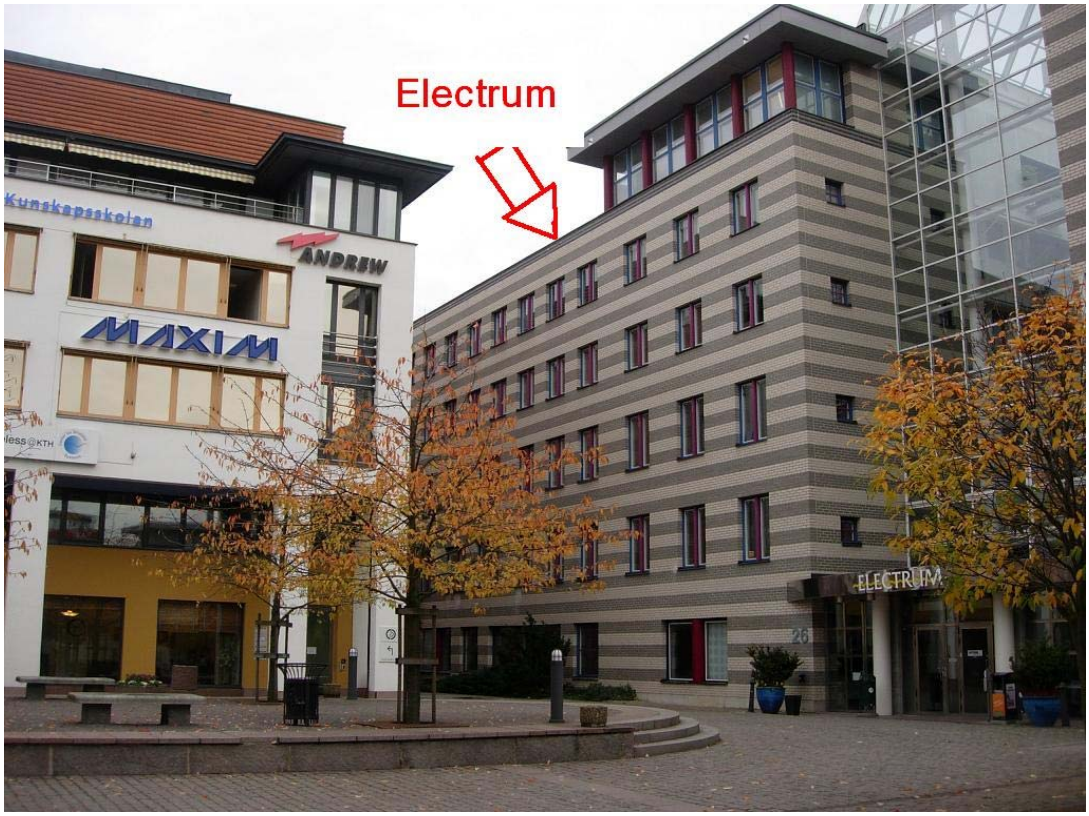

Figure 18: Photo of the Meeting Room Building “Electrum”

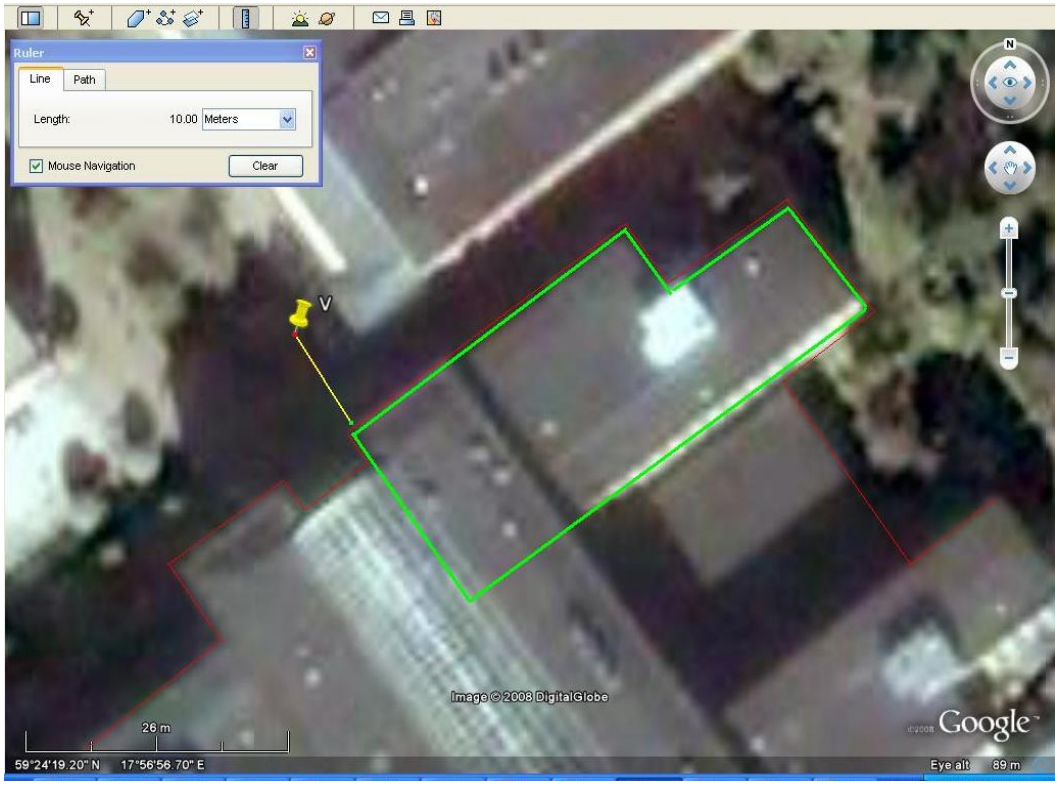

Figure 19: A snapshot of the building “Electrum” on Google Earth

Developing a GPS application

In order to measure the geo-coordinates of the reference point V, I utilized the

method presented in Alisa Devlic's earlier paper [53]. She utilized the Franson GpsTools [52] to develop an application for a HP iPAQ Pocket PC to collect GPS information from a Bluetooth equipped GPS receiver. Franson has provided a set of tools which are integrated with the Visual Studio .NET environment.

I downloaded the Franson GPS Tool SDK version 2.3 from its website. After installation, there were several sample projects of providing simple GPS functions. For example, a C#.NET project named *SerialPortNoEventsCS* reads data from the logical serial port and parses NMEA 0813 data (the standard GPS serial protocol). This application presents a position in the form of latitude & longitude or as grid coordinates. I modified the source code of this project to communicate with a GPS receiver to collect the geo-coordinates of the reference point. In Appendix C, I described how to modify the SerialPortNoEventsCS application and implement it on a Pocket PC.

Measuring the geo-coordinates of the reference point

In this section, I describe how the geo-coordinates of the reference point V is collected.

a) It started by placing the GPS receiver at the point V. The Pocket PC must be within Bluetooth range of the GPS receiver.
b) Turning on the GPS receiver and give it time to acquire a number of satellites, then running the modified SerialPortNoEventsCS application on the pocket PC.
c) The application receives data from the GPS receiver via the Bluetooth connection, which acts as a logical serial link.

Figure 20 shows a snapshot of the interface of the SerialPortNoEventsCS application running in Microsoft Visual Studio.Net.

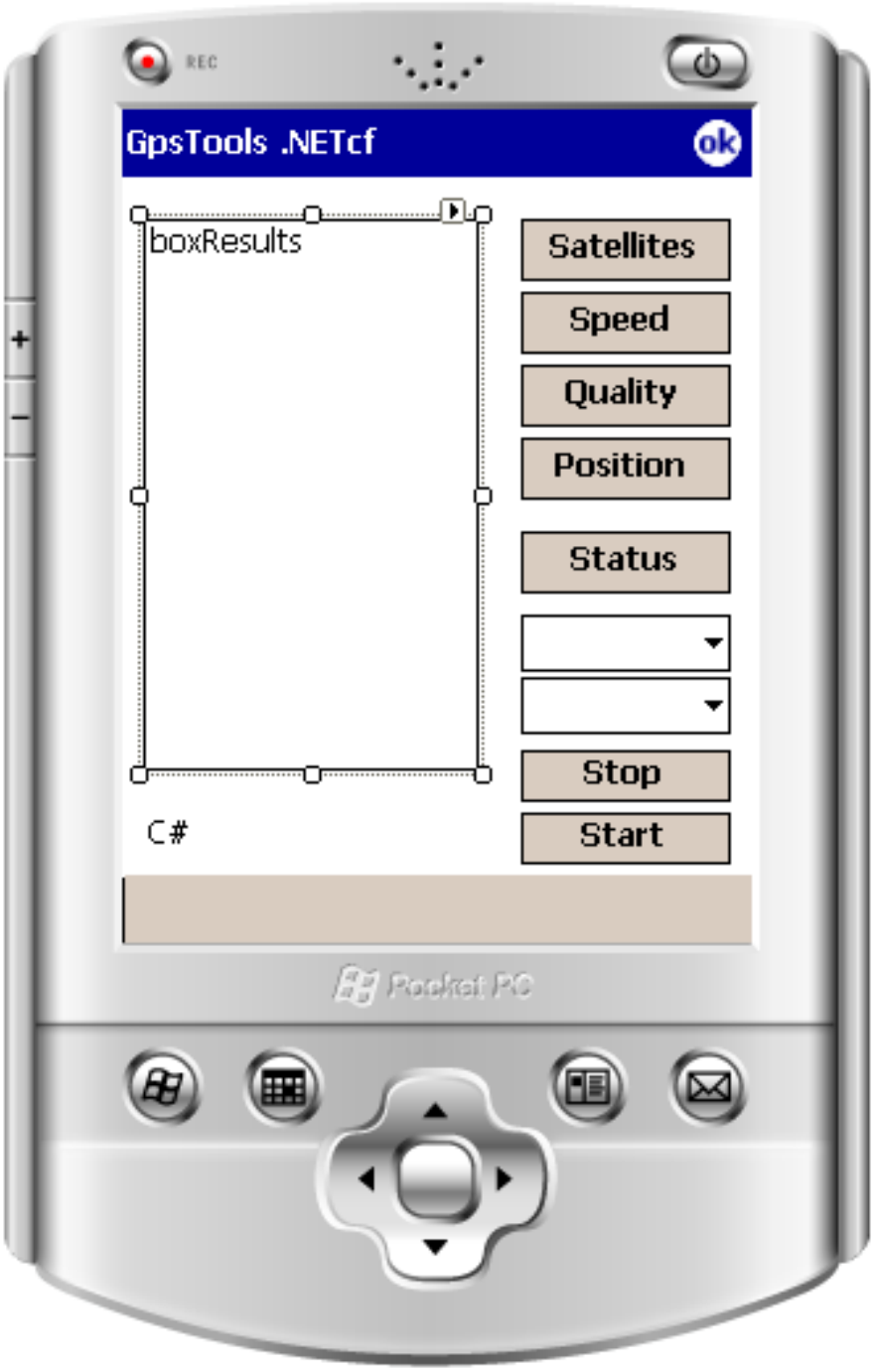

Figure 20: Interface of the SerialPortNoEventsCS Applicatio

I collected a set of geo-coordinates and took an average of these measured values to determine the final geo-coordinates of the point V. This measurement established the reference point's latitude is 59°24'19.47"N and longitude is 17°56'56.34"E.

Using the same method, I established another reference point Z. This point located on the same perpendicular line between the point V and the wall of the building. In this case it is 3 meters away from the building wall and 7 meters away from the point V. Figure 21 shows the point V and Z as display by Google Earth. I measured

geo-coordinates of the point Z: latitude value is 59°24'19.28"N and longitude value is 17°56'56.59"E. Note that the distance between these two reference points is calculated by Google Earth to be 7 meters (see the inset in the upper lefthand corder of Figure 21).

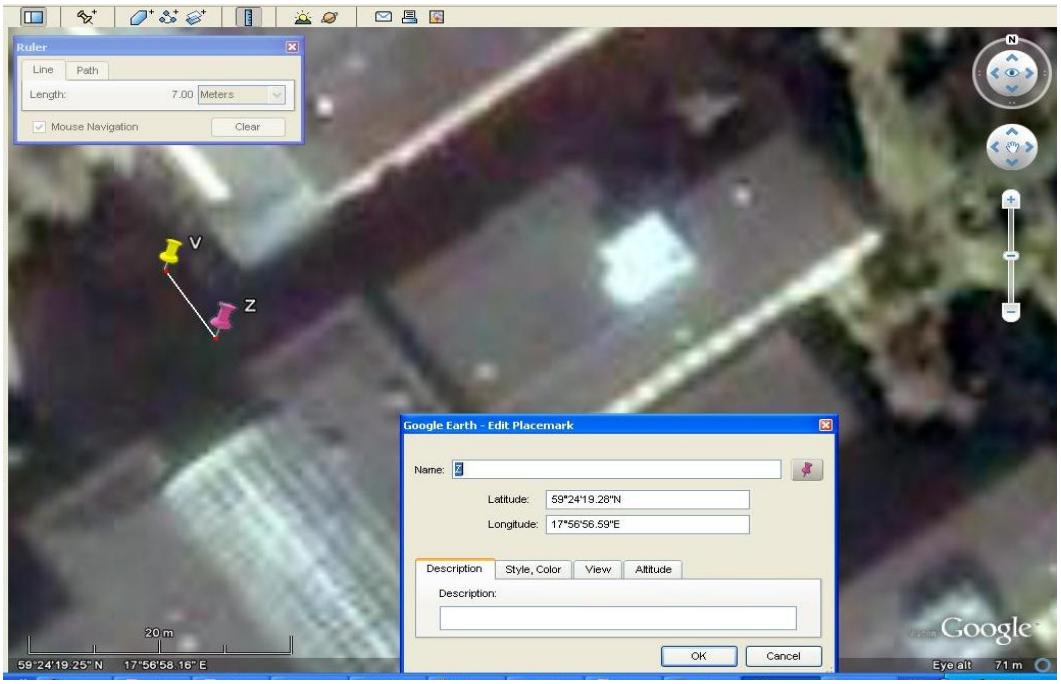

Figure 21: Snapshot of the Reference Points V and Z

Calculating each Meeting Room's Geo-coordinates

There are 5 meeting rooms on two floors in the building Electrum that are relevant to us. Figure 22 is a part of the scaled map of the lower (third) floor which has 3 meeting rooms: Mint, Grimeton, and Open Area. The forth floor has 2 additional meeting rooms: Motala and Hörby.

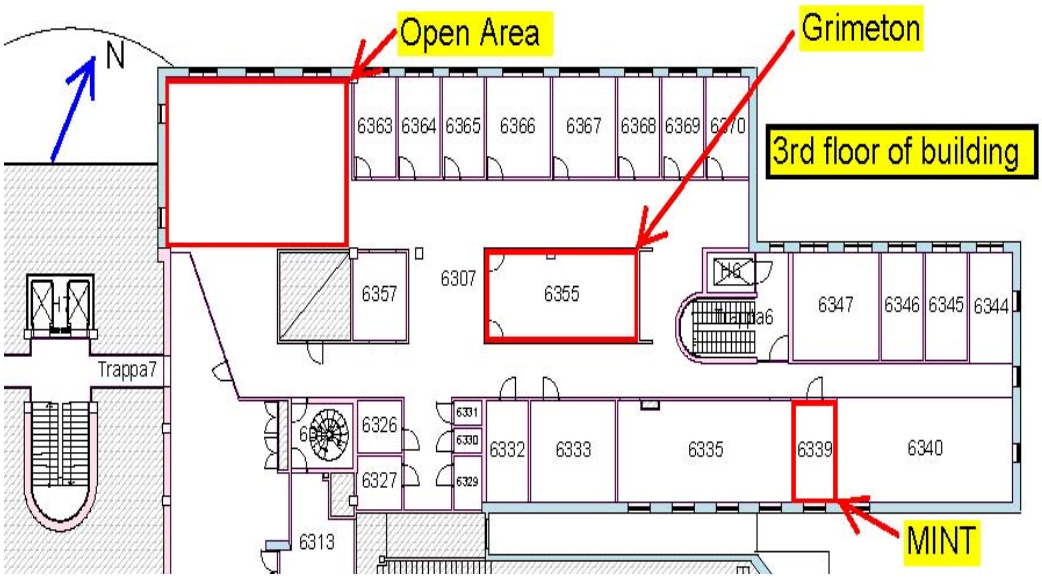

Figure 22: Floorplan of part of the 3rd floor of Electrum

I start by calculating the meeting rooms' geo-coordinates for the third floor. The general principle is that given the distance of a second in latitude and longitude at the longitude & latitude of the reference point; by measuring the distance to a coordinate on this map relative to reference point in terms of the distance parallel and perpendicular to a north-south line through the reference point, I can just directly compute the latitude and longitude values.

Google SketchUp utilized to do distance measurements [60]. This software was developed by Google to create, modify, and share 2/3D models. I will import a scaled map of the third floor into Google SketchUp to develop a 3D model and measure relative distances between the reference point(s) and meeting rooms. Figure 23 is a snapshot taken from Google SketchUp which superimposes the scaled map of third floor of the Eletrum building on a Google Earth image. The red-colored zone is the portion of the building shown in Figure 22; the orange-colored spot is the reference point V; the green-colored arrow is the north-south line.

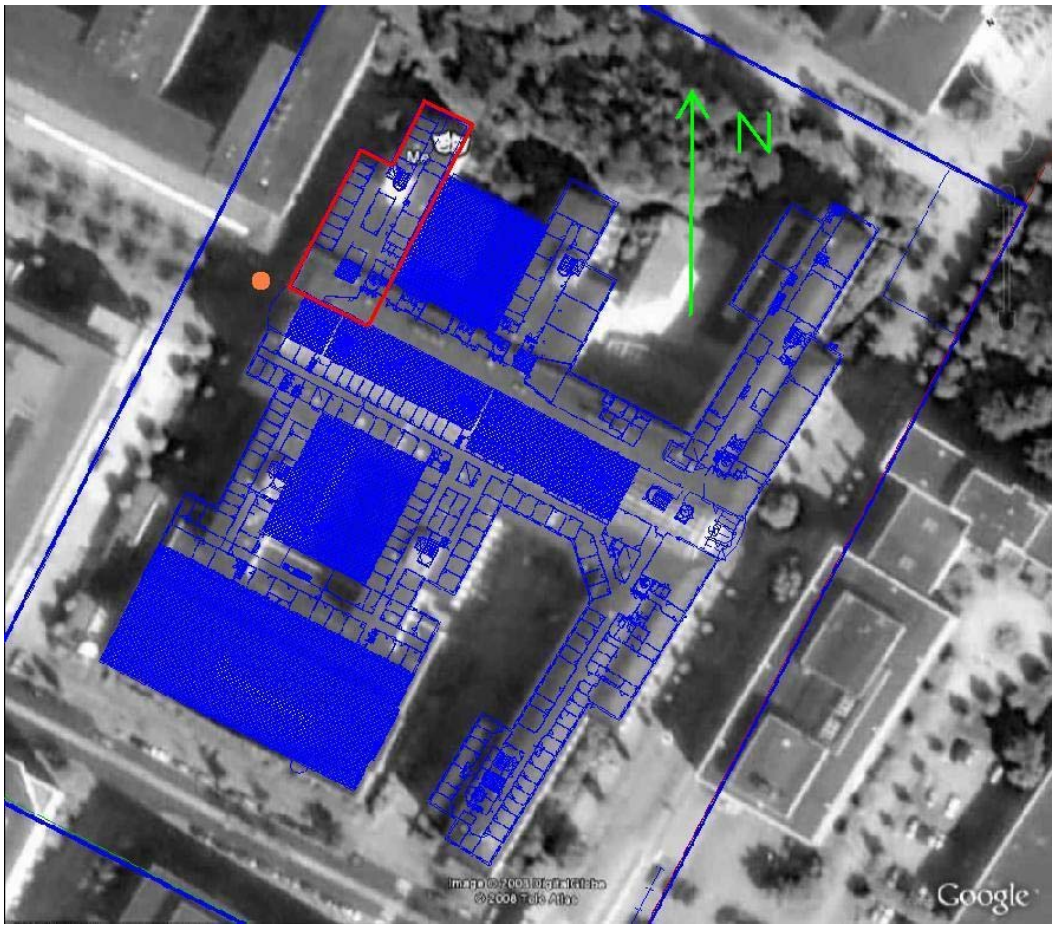

Figure 23: Scaled map of the third Floor of Electrum superimposed on a Google Earth image of the Electrum building

Thaddeus Vincenty devised formulae for calculating very precise geodesic distances between a pair of latitude/longitude points on the earth's surface, using a

very accurate ellipsoidal model of the earth [54]. Hence, I can use his formula to calculate the distance of one second in latitude and longitude within the building area. I believe that within small area the latitude is uniformly distributed on the line parallel to a north-south line and the longitude is uniformly distributed on the line perpendicular to the north-south line. Hence, given the distance differences of latitude and longitude between two points, I can calculate the degree difference of latitude and longitude between these two points - to derive the latitude and longitude of the points.

I imported the third floor's scaled map into Google SketchUp and built a 3D model to calculate the meeting rooms' geo-coordinates. Figure 24 is the initial 3D model of the third floor in Google SketchUp.

Figure 24: Initial 3D Model of the third Floor in Google SketchUp

First of all, to measure the distance differences (in terms of latitude and longitude) of two points, I draw a north-south line. Then using this north-south line and the reference point V, I can measure the relative distances between a point and the reference point V. Given this measurement I can calculate the latitude and longitude of the point. The general principle is that through two given points I can draw lines which are parallel to a north-south line. The vertical distance between these two lines is the longitude distance (I use the term longitude distance to represent this kind of distance). Then I draw two lines through these two points which are perpendicular to a north-south line. The vertical distance between these two lines is the latitude distance (I use the term latitude distance to represent this kind of distance).

Here are the steps necessary to calculate the geo-coordinates of the corners of the meeting rooms.

a) Draw a north-south line through the reference point V.

1. Setting the scale of the model through the Tools-Scale function of Google SketchUp.
2. Marking reference point V (Latitude 59°24'19.47"N, Longitude 17°56'56.34"E) on the model which is 10 meters away from the front wall of the building. Marking reference point Z (Latitude 59°24'19.28"N, Longitude 17°56'56.59"E) which is 3 meters away from the front wall.
3. Drawing a north-south line through reference point V.
   First, calculate the distances of a second of latitude and longitude via the Vincenty formula.
   1 second of longitude (at latitude 59 24 19 N) is 15.77m
   1 second of latitude (at longitude 17 56 58 E) is 30.945m
   Second, calculate the latitude and longitude in seconds as the differences between points V and Z.
   Latitude second is 59°24'19.47" - 59°24'19.28"= 0.19
   Longitude second is 17°56'56.59" - 17°56'56.34" = 0.25
   Third, calculate the angle α between the line VZ and the north-south line.
   $TAN\ \alpha = \sqrt{[(0.25*15.77)^2 / (0.19*30.945)^2]} = 0.6705$
   $\alpha = -33.8436°$
   Finally, known the reference point V and the angle α, a north-south line can be drawn through point V via Protractor tool in Google SketchUp. Then using the tool Axes I create a coordinate system as showing in Figure 25. The point V is set to be the original point (0, 0); the red-axis is parallel to the north-south line; the green axis is perpendicular to the north-south line (because the scale map is a 2D map, the blue axis is not used).

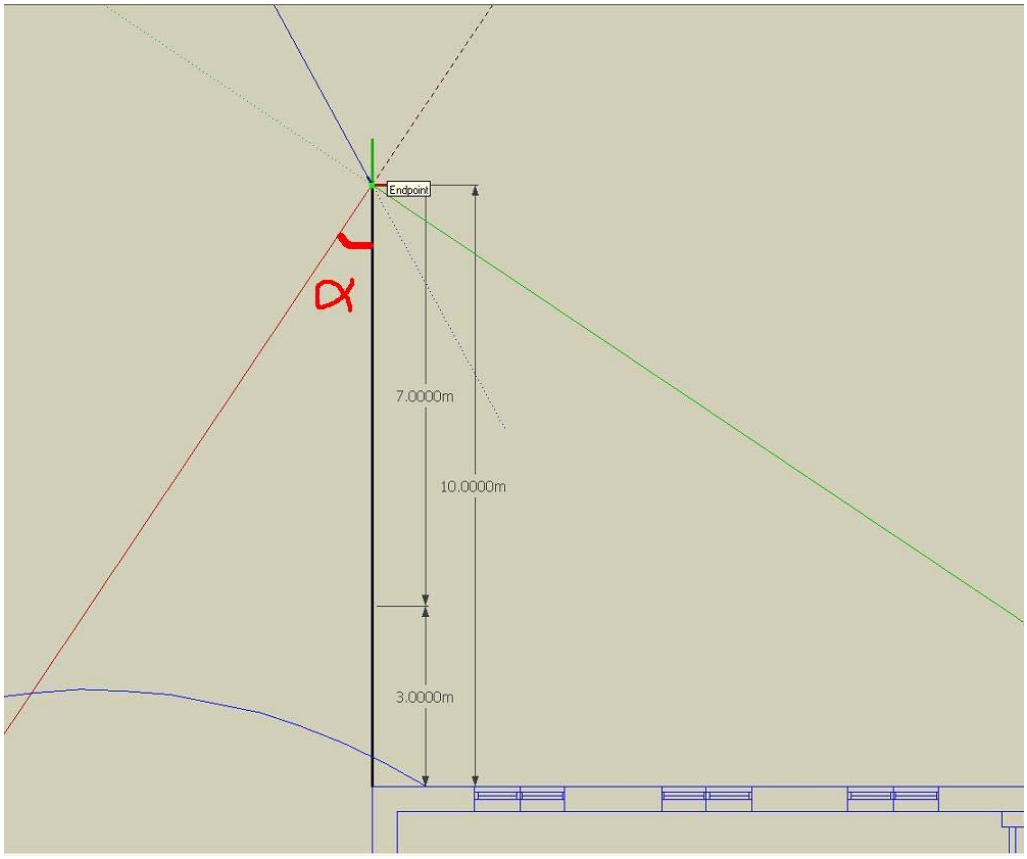

Figure 25: Initial Coordinates System

b) Measure the latitude and longitude distances of each meeting room's corner points to the point V via the Dimension tool.

In the Google SketchUp model, given a point, its latitude distance is the vertical distance from itself to the red axis and its longitude distance is the vertical distance from itself to green axis. Figure 26 shows the measurements of latitude and longitude distances of 4 corner points of the meeting room Grimeton. Figure 27 shows such measurements performed on all 3 meeting rooms of the third floor.

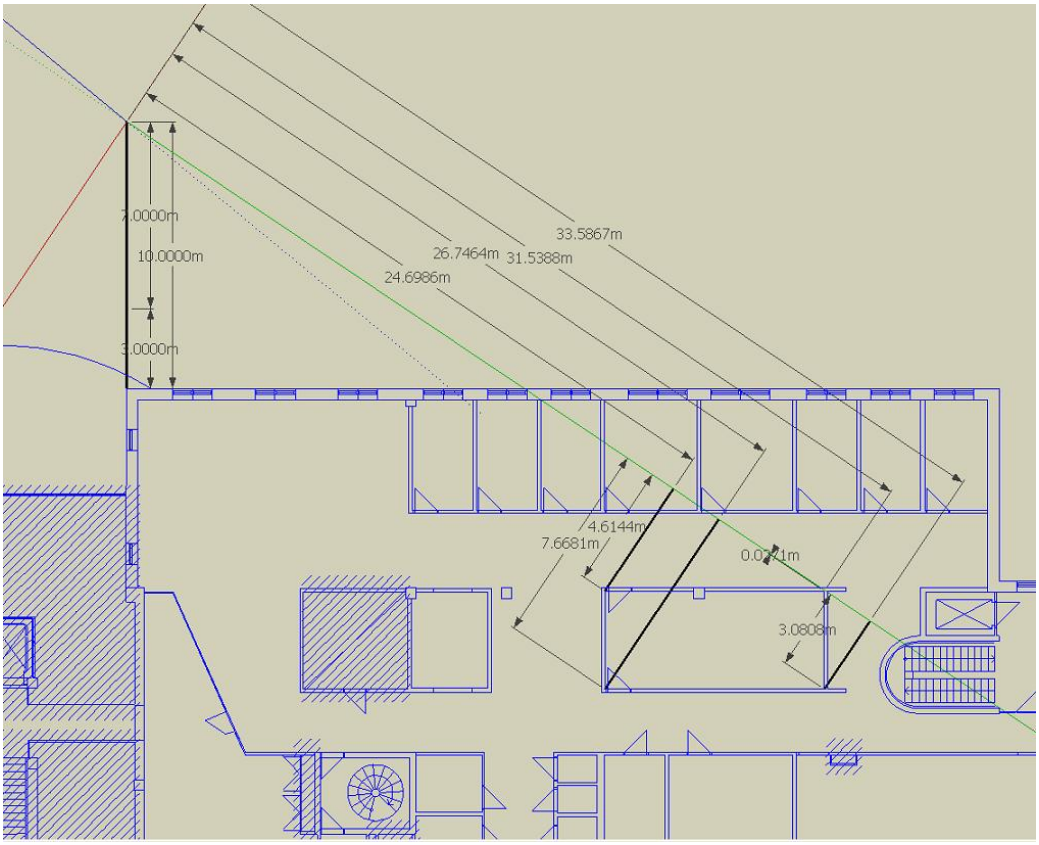

Figure 26: Latitude and Longitude Distances of Grimeton

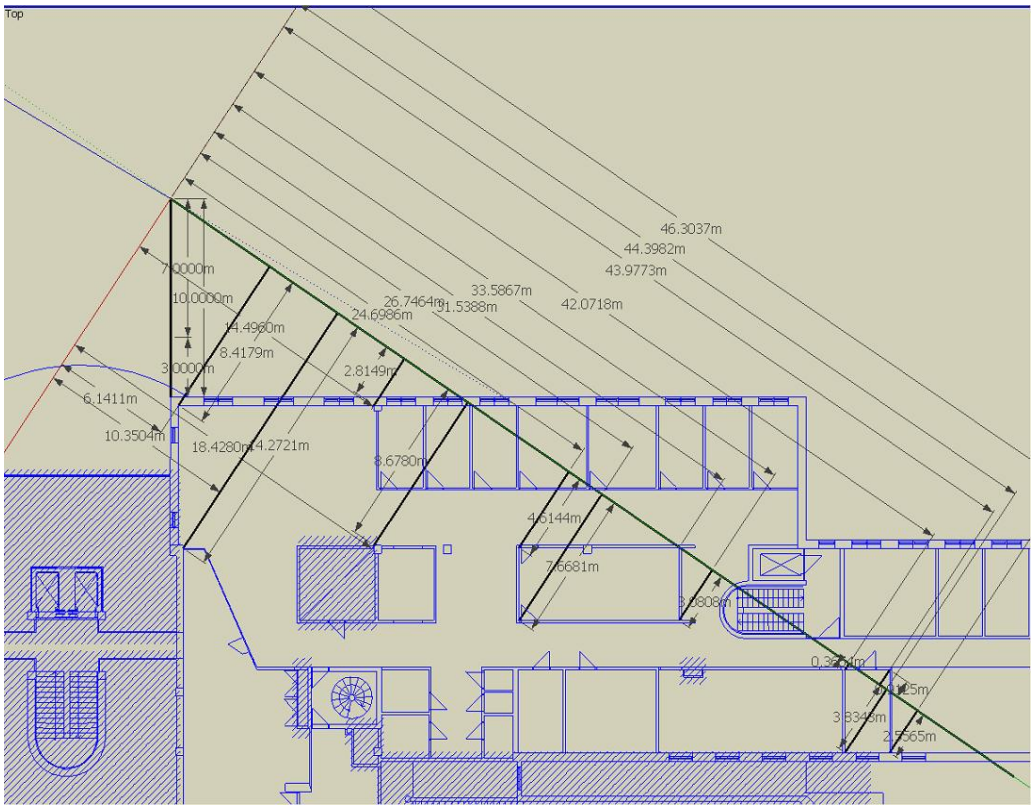

Figure 27: Latitude and Longitude Distances of 3 meeting rooms of the third floor

Note that I set the point V to be the original point with coordinates (0, 0), so the distance value from a point below the green axis to the point V is negative, the distance value from point above green axis to the point V is positive. The corner points of all meeting rooms are presented in Table 2 and their latitude and longitude distance values are respectively shown in the columns “Lat Dis” and “Long Dis” of Table 3.

Table 2: Corner Points of All Meeting Rooms

3rd Floor	Corner Points (upper left, upper right, lower right, lower left)
Grimeton	A, B, C, D
Mint	E, F, G, H
Open Area	J, K, M, L
4th Floor	
Hörby	N,O, P, Q
Motala	R, S, T, U

Table 3: Meeting Rooms’ Latitude and Longitude Values

3rd Floor	Long Dis(m)	Lat Dis(m)	Long Seconds	Lat Seconds	Longitude	Latitude
A	24.6986	-4.6144	1.57	-0.15	57.91	19.32
D	26.7464	-7.6681	1.7	-0.25	58.04	19.22
C	33.5867	-3.0808	2.13	-0.1	58.47	19.37
B	31.5388	-0.0271	2	0	58.34	19.47
E	42.0718	-0.3654	2.67	-0.01	59.01	19.46
F	43.9773	0.9125	2.79	0.03	59.13	19.5
G	46.3037	-2.5565	2.94	-0.08	59.28	19.39
H	44.3982	-3.8343	2.82	-0.12	59.16	19.35
J	6.1411	-8.4179	0.39	-0.27	56.73	19.2
K	14.496	-2.8149	0.92	-0.09	57.26	19.38
L	10.3504	-14.2721	0.66	-0.46	57	19.01
M	18.428	-8.678	1.17	-0.28	57.51	19.19
4th Floor						
N	6.1409	-8.418	0.39	-0.27	56.73	19.2
O	11.6121	-4.749	0.74	-0.15	57.08	19.32
P	13.9544	-8.2418	0.89	-0.27	57.23	19.2
Q	8.4832	-11.9108	0.54	-0.38	56.88	19.09

R	21.7884	-6.5647	1.38	-0.21	57.72	19.26
S	27.6552	-2.6807	1.76	-0.09	58.1	19.38
T	29.7223	-5.6703	1.89	-0.18	58.23	19.29
U	23.8603	-9.6014	1.52	-0.31	57.86	19.16

c) Developing a formula to calculate each point's latitude and longitude.

First, compute the difference between the point V and each corner point in terms of latitude/longitude in seconds.

Formula:

Latitude second=latitude_distance / 1_second_of_latitude

Longitude second=longitude_distance / 1_second_of_longitude

Results:

1 second of longitude (at latitude 59°24'19 N) is 15.77m

1 second of latitude (at longitude 17°56'58 E) is 30.945m

Second, compute the latitude and longitude of each corner point.

Formula:

Latitude value = 59°24'00.00 + Latitude second

Longitude value = 17°56'00.00 + Longitude second

Results are shown in Table 4.2.

The column "Long Sec" represents the second difference between the reference point V and a corner point and the column "Lat Sec" represents the second difference between the reference point V and a corner point. The columns "Longitude" and "Latitude" represent only the seconds part of longitude and latitude values. Thus the complete longitude value of each point is (17°56' + Longitude column) and the complete latitude value of each point is (59°24' + Latitude column). For example, the point A has latitude value 59°24'19.32 and longitude value 17°56'57.91.

4.3.1.1.2 Determining the subscriber's location in terms of a meeting room

SER server sends a Notify message to the Call Secretary containing updated context information about a subscriber's location. Such location context information is represented as latitude and longitude values. Hence, to determine if a given location point P with latitude Pla and longitude Plo is within a specific meeting room, I should compare Pla and Plo with each meeting room's bounding box in terms of latitude and longitude.

For example, Grimeton has four corner points: A (Alo,Ala), B (Blo, Bla), C(Clo, Cla), and D(Dlo, Dla); and the subscriber's location is P (Plo, Pla). Only if the following conditions are fulfilled: AP>0, AP<|AB|, DP>0, and DP<|AD|, can this subscriber P be in the meeting room Grimeton. Using the same approach, the subscriber's coordinates will be compared with each meeting room's geo-coordinates

until either it matches a meeting room, or all meeting rooms have been check and the subscriber P is determined to be outside of any meeting room.

I could write this as a decision function:

if (0 <(Plo-Alo)(Blo-Alo)+(Pla-Ala)(Bla-Ala)<(Blo-Alo)² +(Bla-Ala)² &&

0 <(Plo-Alo)(Dlo-Alo)+(Pla-Ala)(Dla-Ala)<(Dlo-Alo)²+(Dla-Ala)²)

{
System.out.println("Subscriber P is in this Grimeton!");
}

4.3.1.1.3 A better means to determine a subscriber's location

As introduced in previous section, I calculated latitudes and longitudes of all meeting rooms' corners. By performing a comparison between a subscriber's coordinates and each room's geo-coordinates, this subscriber's location in terms of each of the meeting rooms can be determined. For the worst situation, this comparison has to be performed five times (i.e., once for all 5 meeting rooms). I developed a simpler and smarter mechanism to do this.

First of all, instead of utilizing a geo-coordinates system, I define a building's coordinate system. This building coordinates system is a kind of Cartesian coordinate system [61]. In the building's coordinates system the rectangle meeting room's walls are generally parallel or perpendicular to the X axis and Y axis. Hence, consider the meeting room Grimeton with four corner points: A (Ax, Ay), B (Bx, By), C (Cx, Cy), and D (Dx, Dy) where A is the upper lefthand corner of the room, B is the upper righthand corner, C is the lower righthand corner, and D is the lower lefthand corner, Ax equals Dx and Ay equals By. Given a location point P (Px, Py), I can determine if P is inside of Grimeton by performing two comparisons: if Ax<P'x<Bx and Cy <P'y< Ay. If they are both *true*, then P is inside of the meeting room Grimeton. This solution makes computing whether the user is within a room more efficient.

To define a building's coordinates system, I use the reference point V as the original point (0, 0), the Vincenty formula, and all known latitude and longitude distances of meeting rooms' corner points in the model within Google SketchUp.

The Point V Latitude 59°24'19.47"N, Longitude 17°56'56.34"E
1 second of longitude (at latitude 59°24'19 N) is 15.77m
1 second of latitude (at longitude 17°56'58 E) is 30.945m
Angle α is -33.8436

a) It starts by converting the geo-coordinates of A point of Grimeton into building coordinates system as shown in Figure 28. The green axis represents

geo-coordinate system, and the red axis represents building coordinates system.

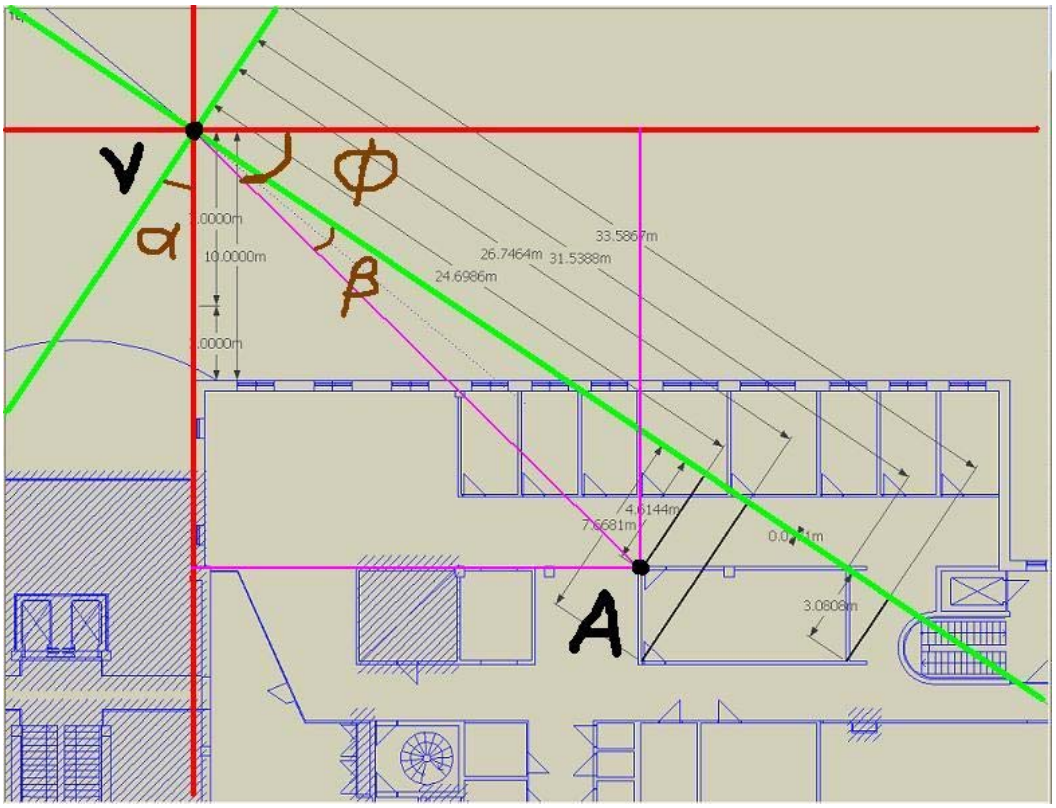

Figure 28: Building's coordinates system

Distance VA = √(A_latitude_distance² + A_longitude_distance²) (it shows in the figure with red lines)

β = ASIN [√ (A_latitude_distance² / VA²)]

Φ = α –β
*Ax = VA * COS Φ*
*Ay =VA * SIN Φ*

Note that the point V is the original reference point, hence all meeting rooms are located on the positive X axis and the negative Y axis, the values on X axis are positive and on Y axis are negative (for this meeting room).

In the same way, all meeting room's corners can be mapped into a building's coordinate system as showing in Table 4.

Table 4: Building's Coordinates System

3rd Floor	Long Dis (m)	Lat Dis (m)	Dis to V (m)	Φ	α	β	X axis	Y axis
A	24.6986	-4.6144	25.126	-10.582	-56.153	-45.571	17.94	-17.59
B	26.7464	-7.6681	27.8239	-15.997	-56.153	-40.156	17.94	-21.27
C	33.5867	-3.0808	33.7277	-5.241	-56.153	-50.912	26.18	-21.27
D	31.5388	-0.0271	31.5388	-0.049	-56.153	-56.104	26.18	-17.59
E	42.0718	-0.3654	42.0734	-0.498	-56.153	-55.655	34.74	-23.74
F	43.9773	0.9125	43.9868	1.189	-56.153	-57.342	37.03	-23.74
G	46.3037	-2.5565	46.3742	-3.16	-56.153	-52.993	37.03	-27.91
H	44.3982	-3.8343	44.5635	-4.936	-56.153	-51.217	34.74	-27.91
J	6.1411	-8.4179	10.4199	-53.888	-56.153	-2.265	0.41	-10.41
K	14.496	-2.8149	14.7668	-10.989	-56.153	-45.164	10.47	-10.41
L	10.3504	-14.272	17.6302	-54.05	-56.153	-2.103	0.65	-17.62
M	18.428	-8.678	20.3691	-25.216	-56.153	-30.937	10.47	-17.47
4th Floor								
N	6.1409	-8.418	10.4199	-53.889	-56.153	-2.264	0.41	-10.41
O	11.6121	-4.749	12.5457	-22.243	-56.153	-33.91	7	-10.41
P	13.9544	-8.2418	16.2066	-30.567	-56.153	-25.586	7	-14.62
Q	8.4832	-11.911	14.623	-54.54	-56.153	-1.613	0.41	-14.62
R	21.7884	-6.5647	22.7559	-16.767	-56.153	-39.386	14.44	-17.59
S	27.6552	-2.6807	27.7848	-5.537	-56.153	-50.616	21.48	-17.63
T	29.7223	-5.6703	30.2583	-10.801	-56.153	-45.352	21.53	-21.26
U	23.8603	-9.6014	25.7197	-21.92	-56.153	-34.233	14.47	-21.26

b) Given a location P with latitude Pla and longitude Plo, I can use the formulae below to calculate its Px, Py value in building's coordinates system.

$$P\_longitude\_distance = (Plo - Vlo) * 0.01 * 15.738$$
$$P\_latitude\_distance = (Pla - Vla) * 0.01 * 30.945$$

$$Distance\ VP = \sqrt{(P\_latitude\_distance^2 + P\_longitude\_distance^2)}$$

$$\Phi = ATAN(P\_latitude\_distance / P\_longitude\_distance)$$
$$Px = VP * COS\ \Phi$$
$$Py = VP * SIN\ \Phi$$

c) Finally, performing a set of simple comparison I can easily determine if P is within a meeting room. Appendix H "Location Indicator" is the Java source code to determine a point's location in terms of a meeting room according to its latitude and longitude.

Note that a value 30 (meters) is added to all coordinates values of meeting room to offset negative values and store the resulting positive value in a table "coordinates" within SER's database.

4.3.1.2 Current Time Context

The current time context is used to determine if a scheduled meeting is currently taking place. This context information can be easily collected by calling the DateTime.now() method from the Java package "com.google.gdata.data.DateTime" which is a Java package of Google Calendar APIs [48] (this APIs will later be introduced in section 4.4.2).

As described previously in chapter 2, when a subscriber creates a new meeting event in his or her Google Calendar, he or she provides the starting and ending time of the meeting. Thus the Call Secretary needs to access the subscriber's calendar to retrieve the starting and ending time of each meeting via Google's calendar APIs. In the Call Secretary, there is a loop calling the DateTime.now() method regularly (every minute) , this value will be compared to a meeting's starting and ending time. Determing a meeting's state is simple: if and only if the current time is equal or greater than the staring time of the meeting and is equal or less than the ending time of the meeting, is this meeting considered to be taking place.

4.3.1.3 Meeting Room Occupancy

The meeting room occupancy context is the last context information I need to collect. The approach is based on the Subscribe-Notify mechanism, thus it is very similar to the Context Agent. The Call Secretary initially sends a Subscribe Request messages to the SER server to subscribe to each meeting room's occupancy context information. It listens for Notify messages containing updated room occupancy context information. Upon receiving a valid Notify message, the body of the message is parsed and the meeting room's name and the updated occupancy value are extracted. Then the occupancy state of the associated meeting room is updated.

4.4 Retrieving Information about a Meeting Event

A subscriber to the Call Secretary service needs to use Google's Calendar to create meeting events. Such a meeting event will be read and parsed by the Call Secretary to extract all the necessary information about the meeting, i.e., the meeting room's name, the starting and ending time of the meeting, and the meeting size. Meeting information is used to create specific making criterion that will serve as a trigger for a pre-planned action. When the current condition of a subscriber matches the meeting

criterion, then the call redirecting function will be triggered. Hence, subscribers need to follow some rules when they post a new meeting event on Google Calendar to make sure that necessary meeting information is provided. Figure 29 shows the interface used for creating a meeting event in Google Calendar. There are 4 parameters relevant to the Call Secretary: “What”, “When”, “Where”, and “Description”.

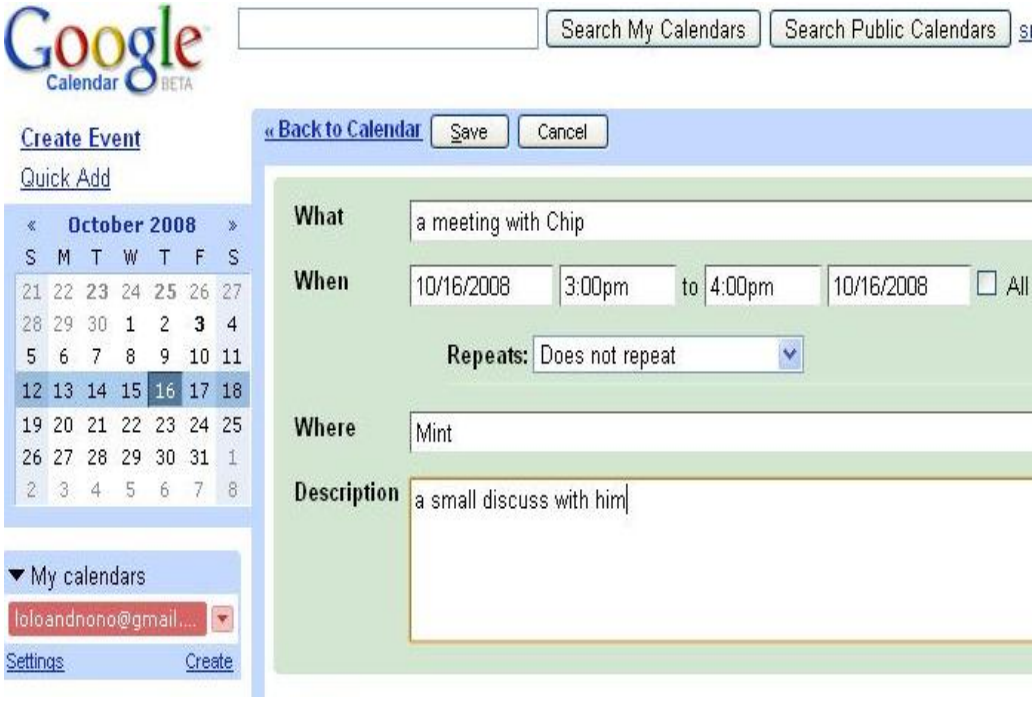

Figure 29: Google Calendar Interface

- The value of “What” needs to contain the key word “meeting” which indicates a meeting event. The Call Secretary will only retrieve calendar events which include this key word. Google Calendar enables users to use either a single calendar to store all events, e.g. meetings, dinners, dates, or multiple calendars that each one used for a specific type of event, e.g. the meeting calendar stores only the user’s meeting schedule, while the dinner calendar stores only dinner engagements. By requiring the use of the key word “meeting”, I provide a great deal of flexibility to subscribers. As these meeting events can be added to either a mixed calendar or a specific meeting calendar. Therefore, subscribers do not have to use a special meeting calendar.
- The format of the value of “When” is predefined by Google Calendar, no extra information is needed.
- The value of “Where” must be one of meeting rooms’ name: “Mint”, “Grimeton”, “OpenArea”, “Hörby”, and “Motala”. Here I can convert the meeting room name to a single case, before comparing it with the list of the names of the meeting rooms.
- The value of “Description” may contain a key word, such as “small” or “big”,

which indicates the size of the meeting (small or big respectively). If neither key word is included, this meeting is considered to be a small meeting. The size of the meeting is one of essential parameters to determine if a meeting is taking place. A “Small” meeting needs at least 2 participants and a “Big” meeting requires at least 5 participants inside the meeting room, otherwise the meeting is considered to have not yet started (even if the time for the start of the meeting has come and there are several people in the meeting room).

The Call Secretary retrieves the meeting events from a subscriber’s Google Calendar in order to get the relevant meeting information. However, a question is how to retrieve these calendar entries? Fortunately, Google Calendar enables users to request calendar events using the HTTP Get Method. In Google Calendar account interface, there is a “Settings” option in the left hand menu bar that is used to configure and personalize each calendar. By clicking in a calendar, you can see the “Calendar Address” close to the bottom of the page with 3 icons: “XML”, “ICAL”, and “HTML”. Figure 30 shows this interface.

Figure 30 : Calendar Setting Interface

Each icon of the Calendar Address attribute relates to a specific URL. The URL of the “HTML” icon is used to access Google Calendar via any web browser; the URL of “ICAL” icon is used to access Google Calendar from other applications that supports the iCal format, e.g. the popular iCal application of the Apple Macintosh can subscribe to the “ICAL” URL, and the URL of the “XML” icon is used to access Google Calendar applications that can read and parse XML formatted file. These different formats of the subscriber’s calendar contents can be provided to the

subscribers via an HTTP Get request. Simply put one of these URLs in a web browser, then a response from the Google Calendar server will present a associated calendar content. However, before performing this test an initial calendar has to be created that would allow "Public", otherwise, the request will be denied by the Google Calendar server. There is even an application, "alarm it"developed by Alex King [56] to request calendar events.

I developed two approaches to retrieve the contents from Google Calendar using a Java program: (1) using HttpClient components and (2) using Google's Calendar APIs. I adopted the latter approach in the Call Secretary, but first I will introduce both approaches.

4.4.1 HttpClient components of HttpComponents project

The HttpComponents project is developing low-level libraries for several aspects of the HTTP protocol [57]. Users can use different components of this project to build custom HTTP services, i.e., based upon HttpCore components and HttpClient components. HttpClient components are used for client-side authentication, HTTP state management, connection management, and an HTTP/1.1 compliant HTTP agent implementation [55]. I have utilized the HttpClient components to develop an application to retrieve the subscriber's calendar content. The following steps were needed to develop this application:

a) Create an instance of the HttpClient.
b) Create an instance of the method GetMethod() which contains the Google Calendar URL (e.g. XML or ICAL address).
c) Create an HttpClient instance to execute the GetMethod in order to send HTTP Get request to the Google Calendar server.
d) Parse the response from the server which contains a Google calendar entry.
e) Release the TCP connection.
f) Process the content of Google Calendar entry.

Appendix E contains the Java source code of this application using HttpClient components to retrieve Google Calendar's events through an "ICAL" address.

However, this solution has a number of drawbacks. First, it does not provide any authentication. It can only be applied to "Public" calendars, this means that the calendar subscriber has to exhibit all his or her calendar contents worldwide (i.e., everyone can visit (and view) this calendar). Second, an HTTP Get request retrieves all the events contained in the calendar. It does not provide direct method to request only a specific category or range of events. For example, if we only want to retrieve today's calendar events, we have to request all events of the calendar even including history events, this will consume an unnecessary amount of system capacity (on the server, the network, and the client). Third, the response is in XML format data. Hence,

we have to parse each response according to the XML tags to extract the necessary meeting information. Obviously, this solution is not very efficient or scalable.

4.4.2 Google Calendar APIs

Fortunately, Google Calendar offers many ways to create and share calendar contents other than via a web interface [48]. Google enables client applications to view and update calendar events via the Google Calendar Data API. Client applications can use the Google Calendar Data API to create new events, edit or delete existing events, and query for events that match a particular criteria. Hence, the Call Secretary can retrieve and process a small amount as it need only deal with the events that are within a specific period of time. Most importantly, it supports authentication, which means applications can retrieve events from a private calendar by providing the subscriber's account information. Below I explain how to utilize the Google Calendar API to retrieve calendar events.

- First, I create a table "call_secretary" within SER's database which has 6 columns: id, name, email, email_psw, active, and voice mail. As explained in chapter 3, the subscribers must subscribe to the Call Secretary service by registering their user name, Gmail address and password, and voice mail address. The *user name* represents the subscriber (it is also used in the SER system to identify the subscriber, this means this name will be same in other tables of the SER database), the *Gmail address* is the subscriber's Gmail account (this represents their account name for various Google services), the *password* is the subscriber's Gmail password (as the subscriber's Gmail account and Google Calendar share the same account information), and the *voice mail address* is where the subscriber wants incoming calls to be redirected to if they are in a meeting. After completing their registration, the value of the *active* attribute is modified (i.e., the value is changed from '0' to '1') indicating that the Call Secretary service was successfully subscribed to and enabled. As an example, I created a Google mail/Calendar account "loloandnono", then I subscribed to the Call Secretary service for this account. The user name is "loloandnono"; the Gmail address is "loloandnono@gmail.com"; the voice mail address is "sip:loloandnono@kth.se". Developing an interface for registration and cancellation is left for future work, in this project, I manually add the subscriber's registeration information to the database.

- Within the Java source code, an ICalTask class extended from the TimerTask class runs every two hours (an explanation of the reason to choose the value two hour is given in the next section). This class establishes a database connection to fetch the registration information of subscribers who have successfully

subscribed to the Call Secretary service. The Call Secretary utilizes the subscriber's Gmail account and password to retrieve the next two hours' worth of calendar events via the calendar API. Figure 31 part of the Java source code containing the functions necessary to retrieve meeting events from a Google Calendar (Appendix F contains the full version of the source code for the Call Secretary).

```
CalendarService myService = new CalendarService("ccslab-CallSecretary");
myService.setUserCredentials(email, iCalPSW);
URL feedURL = new URL("http://www.google.com/calendar/feeds/" +email + "/private/full");
CalendarQuery myQuery = new CalendarQuery(feedURL);

//startDate and StartTime are extracted from current time in the previous codes
myQuery.setMinimumStartTime(DateTime.parseDateTime(startDate + "T" + startTime));
myQuery.setMaximumStartTime(DateTime.parseDateTime(endDate + "T" + endTime));

// Send the request and receive the response:
CalendarEventFeed resultFeed = myService.query(myQuery, CalendarEventFeed.class);
```

Figure 31: Jave Code for Retrieving Calendar Events

In the first line, an instance of the class CalendarService named myService is created. In the second line, a subscriber's Gmail address and password are set as authentication arguments. The Gmail address is also used to generate the subscriber's calendar address (feedURL). The format of this calendar address is: "*http://www.google.com/calendar/feeds/" + email + "/private/full" (this URL is the same as used for the "XML" for Calendar address)*

This address is set to be the argument of the instance feedURL. This feedURL instance is used by the CalendarQuery instance along with a two hour time range. Finally the request is sent to the Google Calendar server retrieving the next two hours' calendar events for this subscriber.

- A method getTitle() of the calendar API can return a calendar entry's name (the value of "What" when creating a event). When the Google Calendar server replies with calendar entries, this method will be called in order to select only the entries that are labeled as meetings (i.e., the value returned by the method getTitle() contains the key word "meeting"). Moreover, the instance resultFeed of class CalendarEventFeed of the calendar API provides many methods to extract the necessary meeting information from meeting events, such as the meeting room name, the meeting size, and the starting and ending time of the meeting. These values are stored locally along with the associated the subscriber's name.

- For subscribers who have meeting events scheduled in next two hours, their location context is subscribed to by sending a Subscribe Request messages to SER server.

4.5 Meeting state estimation

Utilizing the retrieved meeting information along with context information about the current time, the location of a subscriber, and the meeting room occupancy state, the Call Secretary can estimate this subscriber's condition in terms of meeting state. Here a brief explanation of the Java source code to eventually produce the subscriber's meeting state is given:

- The Call Secretary sends Subscribe messages to SER server to subscribe to each meeting room's occupancy context information.

- A loop "ICalTask" runs every 2 hours. It retrieves the subscribers' calendar account information from the SER database. This information is used to retrieve the subscriber's meeting events via the Google Calendar API. All of the retrieved meeting events are processed and information about these meetings is stored locally associated with the subscriber's name. Finally, those subscribers who have a meeting event within the next 2 hours, cause the Call Agent to subscriber to subscriber to their location context information by sending the SER server a Subscribe Request message.

  Choosing the value of 2 hours as timer for the loop calling "ICalTask" is for two reasons: (1) too low a frequency rate increase the possibility of missing meeting events as a subscriber could post new meeting events during wating period of the loop. These events new events would not be read and parsed. Furthermore, a low frequency polling rate means more capacity will be consumed to store more meeting events retrieved. (2) the ICalTask class involves many external Internet connections to retrieve information, database querying, and Google Calendar events fetching. Hence a higher frequency of polling would consume more system resource, although it reduces the chance of missing a newly posted meeting event. As a compromise, 2 hours is set as was selected as the polling interval, because people normally do not post meeting event in their calendar less than two hours before a meeting startes and retrieving 2 hours' worth of meeting information from all subscribers requires that only a modest amount of information will be retrieved, stored, and processed.

- A MeetingTask loop runs once every minute. It simply extracts the necessary information from each subscriber's meeting events to determine the next meeting

using the results from the ICalTask loop (e.g. the meeting room's name, the meeting size, and the starting and ending time). After determining which are the upcoming meetings, this tasks processes them based upon the current time relative to these meetings' starting and ending times. There are three different matching results: the meeting has not started (i.e., the current time is earlier than the meeting's starting time), the meeting has begun (i.e., the current time is later than the meeting's starting time and earlier than the meeting's ending time), or the meeting has ended (i.e., the current time is later than the meeting's ending time). An indicator is set to indicate the meeting's state with respect to time, thus if a meeting has begun, then the meeting time state indicator of the associate subscriber will be set to be *true* otherwise this indicator will be set to *false*.

- The main method specifies how to process the Notify messages sent by SER. These messages are read and parsed much as the Call Agent does. However, there are three different elements of context information utilized by the Call Secretary: the subscriber's location, the planned meeting room's location, and the meeting room's occupancy context. Upon receiving a Notify message containing a meeting room's occupancy context, the occupancy value will be stored locally in a Vector associated with the meeting room's name. If the Notify message contains a subscriber's location context, then the latitude and longitude pair is processed to match against each meeting room to determine if this subscriber is located in a given meeting room.

  As the MeetingTask extracted the meeting room's name and the meeting size from each subscriber's meeting event, this meeting room's name (indicating which room the subscriber is scheduled to be in) is compared with his current meeting room name or location. If the subscriber is in the planned meeting room, then the meeting room state indicator of this subscriber will be set to *true*. Next, the meeting size will be compared with this meeting room's current occupancy value. If the required number of participants' number are present (a small meeting needs at least 2 participants and a big meeting needs at least 5 participants), then the meeting size state indicator of this subscriber will be is set *true*.

- A subscriber's meeting state can be determined by these three indictors: the meeting room state indicator, the meeting time state indicator, and the meeting size state indicator. As soon as all three indicators are "true", then this subscriber is considered to be in a meeting. It should be noted that more complex methods of computing whether a user is in a meeting or not could be used, but for the purposes of this prototype I believe that these three indicators are sufficient to show the power of the Call Secretary.

4.6 CPL script processing and call redirection

As introduced in section 2.6.3, the Call Processing Language (CPL) is a XML based language, designed to describe and control Internet telephony services. This CPL script is applied by the signaling server to process Internet calls, most often resulting in a decision to redirect or reject calls. Each subscriber can generate their own customized CPL scripts to manage their own call handling preferences. In our scenario, the desired call processing is that all incoming calls of subscriber are redirected to this subscriber's voice mail during the meeting. Thus it involves creating a specific CPL script for each subscriber, uploading this CPL script to activate the call redirecting function, and removing the CPL script after the meeting to deactivate this redirect function.

4.6.1 Redirecting an incoming call

As noted earlier there are three methods to generate a CPL script: directly writing the script by hand, using web middleware, and utilizing graphical tools. I chose a popular GUI tool called CPLed to create the CPL script. CPLed is a Java-based CPL editor with a graphical interface that has been designed to be used for creating, editing, or uploading CPL scripts to a SIP server. Figure 32 shows a snapshot of the CPLed interface.

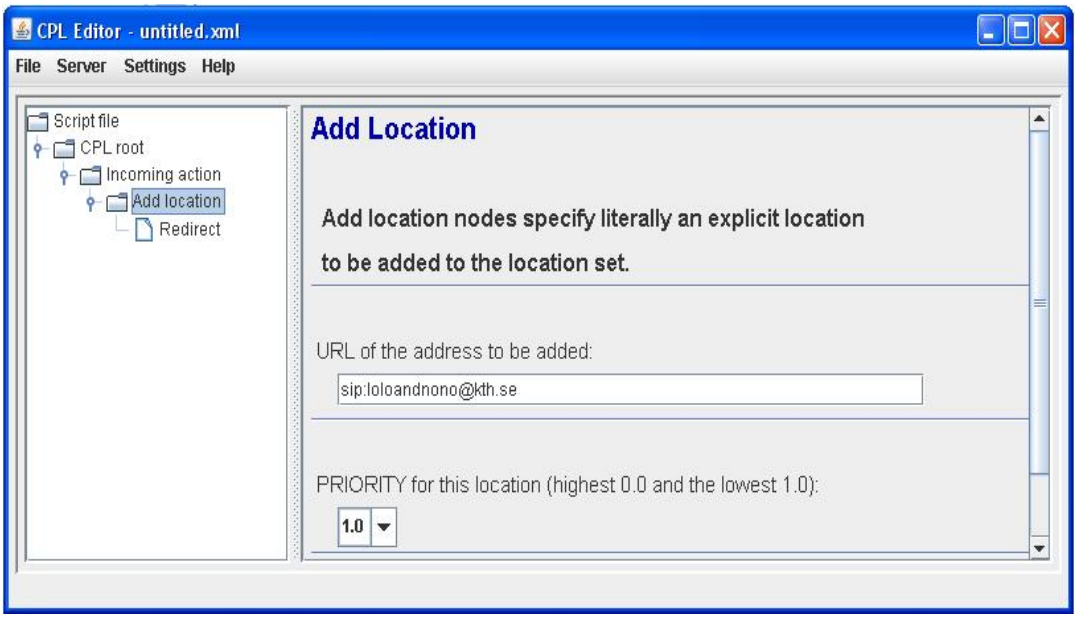

Figure 32: The Interface of CPLed

As introduced in section 3.2, when subscribing to the Call Secretary service, the subscriber provides their voice mail address to which he or she wants the incoming calls to be redirected during the meeting. This voice mail address is used when generating a CPL script specific for each subscriber. Figure 33 shows the subscriber *loloandnono*'s CPL script when she registered her voice mail address as

"sip:loloandnono@kth.se".

```
<?xml version="1.0" encoding="UTF-8"?>
<cpl xmlns="urn:ietf:params:xml:ns:cpl"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:ietf:params:xml:ns:cpl cpl.xsd" >
  <incoming >
    <location url="sip:loloandnono@kth.se" >
      <redirect permanent="yes" />
    </location>
  </incoming>
</cpl>
```

Figure 33: Example CPL Script for a Subscriber

4.6.2 Uploading CPL script

A CPL script has to be uploaded to the SER server in order to activate the call redirecting function. There are two ways to upload it, through a SIP Register message, or via SER's FIFO facility. An introduction to both methods was presented in section 2.6.3.1. In this application, I use the former approach. Figure 34 shows a Register message containing a CPL script.

```
REGISTER sip: @130.237.15.238 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:5060 ;branch=z9hG4bKJlsg3l7
From: <sip:loloandnono@130.237.15.238>;tag=12314
To: <sip:loloandnono@130.237.15.238 >
Call-ID: 2395@130.237.15.227
CSeq: 18 REGISTER
Accept: application/cpl, application/sdp, text/html
Contact: <sip: hlllab4@130.237.15.227:5029 >
Content-Type: context_type
Content-Length: 163
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cpl PUBLIC '-//IETF//DTD RFCxxxx CPL 1.0//EN' 'cpl.dtd'>
<cpl>
<incoming>
<location url="sip:loandno@kthvoicemail.se">
<redirect permanent="yes">
</location>
</incoming>
</cpl>
```

Figure 34: A Register message containing CPL script

A SER server that has been configured with the CPL module will recognize this type of Register message because of the value of the "Content-Type" attribute in the header which is "application/cpl". When this message reaches SER, the CPL script is extracted and stored in a CPL table within SER's database and is associated with the subscriber.

Java source code was used to implement an ICalTask class to check the subscribers' meeting state and to create the appropriate CPL script. The ICalTask class is extended from the TimerTask class and runs every minute. It simply checks each subscriber's meeting state by examining the 4 indicators (the meeting room state indicator, the meeting time state indicator, the meeting size state indicator, and the CPL script indicator). The first 3 indicators are used for determining a subscriber's meeting state, the last one indicates if this subscriber's meeting CPL script has already been uploaded. If a subscriber's meeting state is *true* and the CPL script has not yet been uploaded, a method will be called to generate the appropriate CPL script for this subscriber and to upload it immediately. Once the CPL script has been successfully uploaded, then the CPL script indicator of this subscriber will be set to *true*.

If a valid CPL script is successfully uploaded to SER server, the all incoming calls to this subscriber will be redirected to his or her voice mail.

4.6.3 Removing CPL script

A subscriber's meeting state will be changed for many reasons, for example, when the subscriber walks out of the meeting room the location sensing application will publish a SIP message which triggers SER server to generate a Notify message containing his or her current location context to the Call Secretary. Then the meeting room state indicator will be modified to *false* which makes the meeting state of the subscriber *false*.

When a subscriber is no longer in a meeting, then his or her CPL script will be removed in order to stop redirecting the user's incoming calls. A CPL script can be deleted by sending a Register message which has *application/cpl* as the value of the Content-Type attribute and an empty body. Figure 35 is an example of Register message used for removing a subscriber's CPL script.

REGISTER sip: @130.237.15.238 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:5060 ;branch=z9hG4bKJla2aBfl7
From: <sip:loloandnono@130.237.15.238>;tag=sge1sg1
To: <sip:loloandnono@130.237.15.238 >
Call-ID: 2395@130.237.15.227
CSeq: 18 REGISTER
Accept: application/cpl, application/sdp, text/html
Contact: <sip: hlllab4@130.237.15.227:5029 >

Content-Type: context_type

Content-Length: 0

Figure 35: A Register message used for removing a subscriber's CPL script

5. Evaluation

In this chapter, I will evaluate the performance of two applications through testing. These two applications should correctly operate messages and produce the expected correct results. For example, communication with the SER server via SIP messages, updating entries within a database, estimation of a subscriber's meeting state, uploading and removing a CPL script, etc. However, the SER server does not provide fully support for subscription to location and room occupancy events. Hence, I developed a Java application "Notify Sender" to simulate the specific behavior of such a fully configured SER server. This test application generates several different types of Notify messages to be used for testing the two applications. Once I have satisfied myself that the software is functionally correct, then I will focus on measuring the system's efficiency in terms of the time delay associated with different settings and events.

5.1 Context Agent evaluation

The Context Agent must be able to successfully communicate with SER via SIP messages, process Notify messages, and update the appropriate database entries when necessary. I designed three testing scenarios. Based on the performance of the application in each test, an analyses and evaluation will be given in the sections below.

5.1.1 Methodology

The Context Agent application runs on a DELL "OptiPlex GX620" computer equipped with a 2.80 GHz "Intel Pentium D" processor and 2.0 GB of memory. The operation system is OpenSUSE 10.3, Java Runtime Environment version is 1.6.0_07. The Notify Sender runs on a similarly configured DELL "OptiPlex GX620" computer.

The three testing scenarios are:

Subscription scenario

This test involves sending Subscribe messages to the SER server to request meeting room occupancy context and receiving response messages from the SER server. Because currently SER can only process a presence event request, rather than a meeting room occupancy event, I change the context event from "occupancy" to "presence" in order to verify if the format of the Subscribe message the Call Agent sent was correct. This tests works successfully, thus verifying that the application could correctly subscribe -- if SER were extended to handle *occupancy* events.

Notification scenario

This test contains 2 steps: reading & processing Notify message from SER server and generating a correct reply to the SER server. Because the SER server can not generate a Notify message for the room occupancy context, I use the "Notify Sender" application to send 4 types of Notify message to the Context Agent: (1) a message with context information, (2) a message without context information, (3) an invalid/irrelevant message, and (4) a subscription expired notification message. Using test I can verify that the Context Agent can process each type of Notify message correctly.

Database updating scenario

The database updating is performed when the Context Agent receives a Notify message with valid room occupancy context information. This test will verify that the database update is done correctly.

5.1.2 Notify Sender application

The Notify Sender application is a small Java application developed to send different types of Notify message to a client with a specified port number and IP address. Because the Context Agent utilizes meeting room occupancy context, to text the functionality of the Context Agent the Notify Sender must be able to produce and send all four types of Notify messages regarding room occupancy. When the application starts running, the Notify Sender prompts the user to input all the necessary information, such as the type of Notify messages, a destination IP address, a destination UDP port number, a call-ID value, a meeting room name, an occupancy context (a number), and the number of messages the user wants to send. Based upon the user's input, the Notify Sender generates suitable Notify message(s) and sends them to the specified destination (which for my testing is the Context Agent).

In next section, I present samples of the four different types of Notify messages generated by the Notify Sender. The Java source code for this applicaiton is included as Appendix G.

5.1.3 Analyzing the results of the functional tests

A Context Agent runs on computer with the IP address 130.237.15.227 and listens on a random (unused) port. I also run the Wireshark software [62] on this computer to capture the relevant data traffic.

Subscription scenario:

Figure 36 shows a Subscribe Request message sent by the Context Agent requesting

the occupancy context data for the meeting room Grimeton:

SUBSCRIBE sip:Grimeton@130.237.15.238:33091 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:33091;branch=z9hG4bKk76dz5g
From: <sip:hlllab4@130.237.15.238>;tag=41In
To: <sip:Grimeton@130.237.15.238>
Call-ID: 821127@130.237.15.227
CSeq: 16477 SUBSCRIBE
Max-Forwards: 70
Event: occupancy
Accept: application/pidf+xml
Contact: <sip:hlllab4@130.237.15.227:33091>
Expires: 600
Content-Length: 0

Figure 36: A Subscribe Request message for Occupancy Context

Figure 37 shows a Subscribe message by the Context Agent for a subscriber Tere's presence context. Figure 38 shows SIP message transaction as captured by Wireshark.

SUBSCRIBE sip:tere@130.237.15.238:33091 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238:33091;branch=z9hG4bKk7erzE68
From: <sip:hlllab4@130.237.15.238>;tag=Lj54n
To: <sip:tere@130.237.15.238>
Call-ID: 08237@130.237.15.227
CSeq: 12487 SUBSCRIBE
Max-Forwards: 70
Event: presence
Accept: application/pidf+xml
Contact: <sip:hlllab4@130.237.15.227:33091>
Expires: 600
Content-Length: 0

Figure 37: A Subscribe Request message for Presence Context

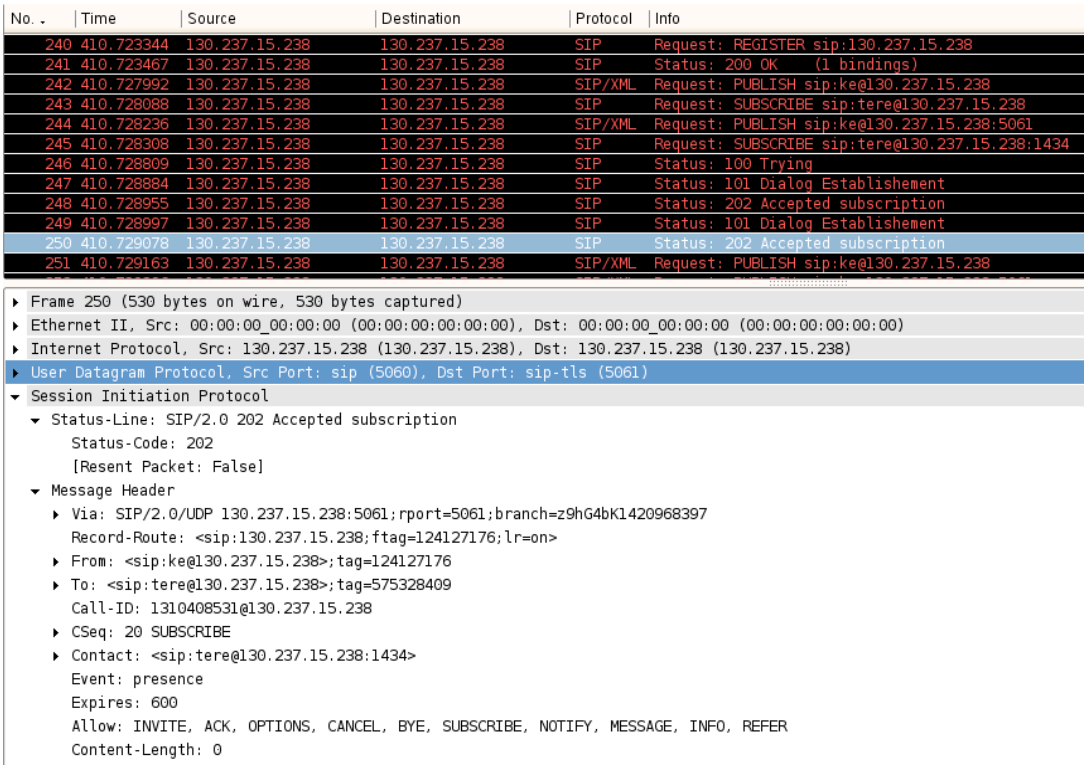

Figure 38: Subscription for a Presence Context

In this scenario, I observed that SER could not accept room occupancy event subscriptions, but could accept subscriptions for a presence event. When the Context Agent subscribes to a presence event, SER correctly replies with a 202 Accepted subscription message indicating that this subscription is accepted. As the only difference between a room occupancy event and a presence event is the value of the parameter *even,* I believe that this test shows that the basic functionality works as intended. Given that SER can accept a presence event request, I can believe that the format of the Subscribe message that the Context Agent generates is correct. However, it remains for future work to fully implement the occupancy event handling in SER.

Notification scenario

Figure 39 shows a Notify message sent by the Notify Sender containing room occupancy context information which indicates 11 participants are currently inside of the meeting room named “Grimeton”.

NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>

Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="0xb58d60e0x4a4b0c39x4715c26e">
<status><basic>open</basic>
<occupancy>
<description>Electrum</description>
<room>grimeton</room>
<value>11</value>
</occupancy>
</status>
<contact priority="0.80">KeWang</contact>
<note>occupancy</note>
</tuple>
</presence>

Figure 39: A Notify Message Containing Room Occupancy Context

Figure 40 shows a 200 OK message is sent in reply by the Context Agent:

SIP/2.0 200 OK
From: <sip:ccsleft@130.237.15.238>;tag=xlB3;received=130.237.15.227
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Content-Length: 0

Figure 40: A 200 OK Message Replied by the Context Agent

Figure 41 shows a Notify message sent by the Notify Sender without context information:

NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">

```
<tuple id="none"
<status><basic>closed</basic></status>
</tuple>
</presence>
```

Figure 41: A Notify Message without Context Information

A 200 OK message is sent by the Context Agent. This is the same message shown in Figure 40 (but of course in reality it would have a different Call-ID header value and perhaps a different CSeq header field value - but since these values are based upon those entered by the user when the test is run - for testing the values are the same)..

Figure 42 shows a Notify message sent by the Notify Sender containing room occupancy context information but with an incorrect *call-ID* value:

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 007@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="0xb58d60e0x4a4b0c39x4715c26e">
<status><basic>open</basic>
<occupancy>
<description>Electrum</description>
<room>grimeton</room>
<value>11</value>
</occupancy>
</status>
<contact priority="0.80">KeWang</contact>
<note>occupancy</note>
</tuple>
</presence>
```

Figure 42: A Notify Message with an Incorrect CallID

No message is generated by the Context Agent, as this Notify message was ignored.

Figure 43 shows a sample of a Notify message sent by the Notify Sender

containing an expired Subscription-State:

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 821127@130.237.15.238
CSeq: 1 NOTIFY
Event: occupancy
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: terminated
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="none"
<status><basic>closed</basic></status>
</tuple>
</presence>
```

Figure 43: A Notify Message due to Expired Subscription

A 200 OK message is sent by the Context Agent. This is the same message shown in Figure 40. A Subscribe Request message is sent by the Context Agent to SER server (the same message as shown in Figure 36). Figure 44 shows SIP message transaction captured by Wireshark.

In this testing scenario, the Notify Sender sent all four types of Notify message to the Context Agent. The Context Agent processed each of them correctly. Table 5 shows the responses of the Context Agent to each type of Notify message.

Table 5: Responses of the Context Agent to each type of Notify message

Notify message received	200 OK message	Subscribe message	Ignore	Extract context	Update database
contains context	√			√	√
no context	√				
irrelevant message			√		
subscription expired	√	√			

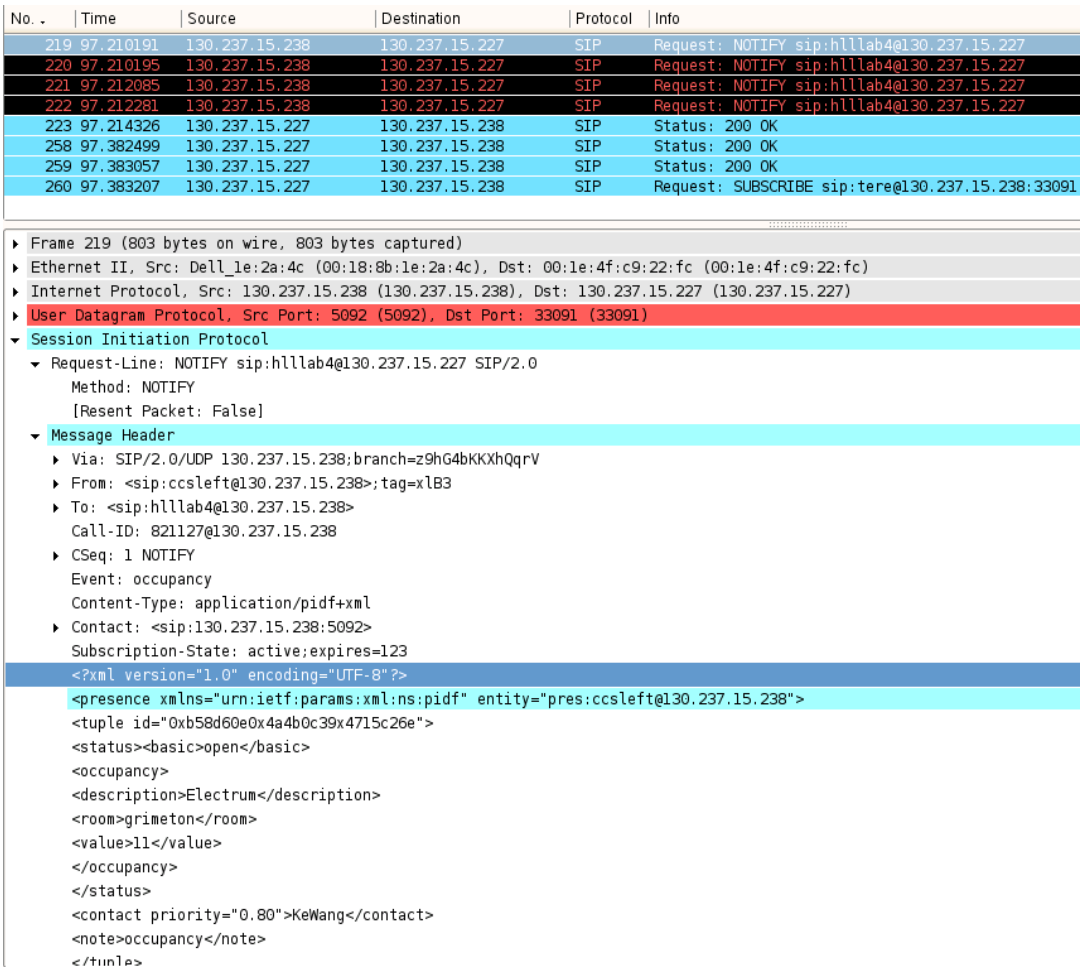

Figure 44: SIP Message Transaction between Context Agent and Notify Sender

Database updating scenario

After receiving a valid Notify message with room occupancy context information, the Context Agent extracts information about the meeting room name and the occupancy value. Then it sets up a connection to the MRBS database and executes a SQL query to update the associated entry. The database connection is to be released after the updating has been successfully completed. The column “Update database” in Table 5.1 shows that the database operation was performed correctly.

5.2 Call Secretary evaluation

To verify the correct functioning of the Call Secretary, I designed two test scenarios: (1) a subscriber is in a meeting state and (2) a subscriber is not in a meeting state. These two scenarios involve SIP message communications with SER and the Notify Sender application, database updating, meeting events fetching from Google Calendar, etc. In both scenarios, the performance of the Call Secretary was monitored and analyzed.

Before performing an evaluation of the Call Secretary, the building’s coordinates

system was evaluated in advance as this information will be used by the Call Seretary to determining the value of a subscriber's meeting room indicator. Moreover, the accuracy of geo-coordinates measurements in Google Earth will be examined in order to evaluate the building's coordinate system.

5.2.1 Accuracy of geo-coordinate measurements in Google Earth

The purpose of evaluating the accuracy of Geo-coordinate measurements in Google Earth is that a number of applications need to have accurate geo-coordinate for some set of reference points. Actually, the geo-coordinated of most of the outdoor points can be measured accurately using the GPS receiver and HP iPAQ with the specific measurement application installed. However, the geo-coordinates of points inside of the building can only be calculated through my formulae. Therefore I decided to evaluate if Google Earth is sufficiently accurate enough to be used for establishing the geo-coordinates of points located indoors. This evaluation can also be used to verify the accuracy of the formulae I have used.

- In section 4.3.1.1.1, I measured the geo-coordinates of two reference points V (Latitude value 59°24'19.47"N, Longitude value 17°56'56.34"E) and Z (Latitude value 59°24'19.28"N, Longitude value 17°56'56.59"E) using GPS. The Point V was measured to be 10 meters away from the building wall using a measuring tape and protractor (the laster was used to ensure that the distance to V was perpendicular to the wall). Reference point Z is located between point V and building wall and was 7 meters from reference point V as shown in Figure 21.
- Putting the geo-coordinates values of points V and Z into the Vincenty formula enables use to calculate a precise distance value between these two points. The result was exactly 7 meters (which means my distance measurements are quite accurate).
- Entering the points V and Z into Google Earth using the measured geo-coordinates was shown in Figure 21.
- Using the ruler function in Google Earth to measure the distance between the point V and Z, also results in a distance measurement of 7 meters (as also shown in Figure 21).

Therefore, I believe that geo-coordinates values collected in Google Earth are accurate enough to be used as reference. This is useful as it means that I can use Google Earth to determine the coordinates of points and do not have to make subsequent GPS measurements to confirm them. Of course my measurements only showed that V and Z had the correct separation, thus I can not make a general claim for the accuracy of the Google Earth coordinates. However, some calculations of coordinates that did not use the more exact Vincenty formula showed that it was very important to use this more exact formula to get values that were at all correct - given the very high latitude of Kista, Sweden.

5.2.2 Accuracy of the Building's coordinate system

As described in the chapter 4, I developed a building coordinate system and then computed a bi-directional mapping between this coordinate system and geo-coordinates. A geo-coordinate can be mapped using this new system via a formula to the building's coordinate system and the reverse. The location of each of the meeting rooms were computed in units of meters in the building's coordinate system instead of using geo-coordinate unit in a geo-coordinate system. Using the building coordinate system enabled us to quickly perform a comparison against each meeting room's coordinates to determine if this subscriber is in a specific meeting room. Because the walls of the building are mainly aligned with the building's coordinate system the computation is simpler in this coordinate system than in geo-coordinates. Additionally, distance measurements in the building coordinate system are easily made in units of meter, where as a very complex set of equations has to be solved to compute an accurate distance between two points in geo-coordinates.

I created a "coordinates" table within the SER database in order to store information about each meeting room's bounding box using building coordinates. The meeting rooms are rectangular shaped and the walls are aligned along the building coordinate axes. Each corner is associated with a column of the table: top left, top right, bottom left, and bottom right. In the Java source code of the Call Secretary, a class "Location Indicator" provides methods to compute a subscriber's meeting room state according to his or her location geo-coordinates and the table of meeting room's building's coordinate system. In order to evaluate this class, I developed a main method to be able to invoke this class directly (for testing purposes). To evaluate the accuracy and correctness of this code, I designated some reference points' geo-coordinates in Google Earth and used them as input to the Location Indicator.

The reference points I collected in Google Earth are shown in Figure 45 and their geo-coordinates are presented in Table 6.

Figure 45: Meeting Room's Reference Points

Table 6: Geo-coordinates of Meeting Room's Reference Points

Reference point	Latitude	Longitude	Meeting room	Result from Location Indicator
1	59°24'19.16"N	17°56'56.84"E	Open Area	Open Area
2	59°24'19.20"N	17°56'57.26"E	Open Area	Open Area
3	59°24'19.19"N	17°56'58.40"E	Open Area	Open Area
4	59°24'19.30"N	17°56'58.06"E	Grimeton	Grimeton
5	59°24'19.40"N	17°56'59.26"E	Mint	Mint
6	59°24'19.30"N	17°56'59.90"E	None	None

Using the geo-coordinates of each reference points as input to the Location Indictor produces a specific meeting room indication given where this point is located. The results are listed in the column "Result from Location Indicator" in Table 5.2.

The reference points I collected refer to different meeting rooms and different positions within a meeting room (and even close to the border of a meeting room). The results produced by the Location Indicator are all correct. Hence, I believe that the building's coordinates system and the formulae are accurate enough for my requirements.

5.2.3 Notify Sender application

The Call Secretary utilizes two kinds of context information: location and room

occupancy context. A Subscription-Notification mechanism which uses SIP message to communicate with SER server is used to obtain context information. Because the SER server does not currently support location and occupancy events, I utilize another Notify Sender Application in my evaluation to generate Notify messages containing these two types of context information. Note that this is the same application used earlier for testing the Call Agent.

The Notify Sender application prompts the user for the necessary input data, specifically: a destination IP address, a destination UDP port number, a call-ID value, and a type of Notify message (location or room occupancy). For the Notify message containing a location some additional information also needs to be entered, specifically: a subscriber's name, a latitude value, a longitude value, and a floor number. Note that in the implementation I have not considered how the user would be located vertically within the building; this clearly remains necessary in future work.

Figure 46 shows an example of a Notify message sent by the Notify Sender containing location context.

```
NOTIFY sip:hlllab4@130.237.15.227 SIP/2.0
Via: SIP/2.0/UDP 130.237.15.238;branch=z9hG4bKKXhQqrV
From: <sip:ccsleft@130.237.15.238>;tag=xlB3
To: <sip:hlllab4@130.237.15.238>
Call-ID: 3461@130.237.15.238
CSeq: 1 NOTIFY
Event: location
Content-Type: application/pidf+xml
Contact: <sip:130.237.15.238:5092>
Subscription-State: active;expires=123
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
entity="pres:ccsleft@130.237.15.238">
<tuple id="0xb65d60e0x4a4b0c39x4715c26e">
<status><basic>open</basic>
<location>
<description>Electrum</description>
<subscriber>loloandnono</subscriber>
<floor>3</floor>
<coordinates>
<latitude>59241920</latitude>
<longitude>17565709</longitude>
</coordinates>
</location>
</status>
<contact priority="0.80">KeWang</contact>
<note>location</note>
</tuple>
</presence>
```

Figure 46: A Notify Message Containing Location Context

5.2.4 Methodology

A Call Secretary was run on a DELL "OptiPlex GX620" computer equipped with a 2.80 GHz "Intel Pentium D" processor and 2.0 GB of memory. The operation system is OpenSUSE 10.3 and the Java Runtime Environment version is 1.6.0_07. The SER server was run on a computer with the IP address 130.237.15.238. The SER server listens on default SIP UDP port 5060. A Notify Sender was also run on this computer. (i.e., the Notify Sender application was run on the same computer used for running SER)

The testing steps are described below. The performance of the Call Secretary will be analyzed and evaluated according to the test results.

a) I began by creating three subscribers in the SER server, subscribers A, B, and C. The first two subscribers were registered as "service activated" with their complete registration information manually entered and stored in the database, i.e., the subscriber's name, a Gmail account, a password, and a voice mail address.
b) The Call Secretary was started.
c) Next I created both meeting and non-meeting calendar events in A's and B's Google Calendar. When creating a meeting event, the requirements listed in section 3.2 were followed.
d) The Notify Sender is used to send different location and room occupancy Notify messages to the Call Secretary and recorded the respones.

5.2.5 Analyzing results

In this section, I will evaluate the Call Secretary's performance when different test events occur. I will begin by performing a set of functional tests to see that the system works correctly, then proceed to examine the time delay when the system is to perform different operations.

5.2.5.1 Starting up a Call Secretary

After the Call Secretary started it connects to the SER's database to retrieve registration information regarding the subscribers (A and B). Then it requests their meeting events during the next two hours from the Google Calendar server and locally stores information about the meeting events associated with subscriber's names. For these subscribers who have meeting events during the next two hours, a Subscribe Request message for location context is sent to SER server. Meanwhile, the Call Secretary requests meeting room information from the MRBS database and sends Subscribe Request messages for each meeting room to its associated occupancy context. Finally it begins listening on a specific port for incoming SIP messages from

SER. These tasks are repeated every two hours. Each of these steps operated correctly during the functional testing.

As the meeting events are fetched and stored locally, two other loops which run every minute start working. One loop examines each subscriber's latest meeting event and tries to detect if this meeting has started according to the current time and a meeting's pre-defined starting and ending times. The other loop examines each subscriber's meeting state. Both of these loops were shown to operate properly. Note that these two loops operate independently and set two different (and independent) state indicators.

5.2.5.2 Sending Notify messages

A Notify Sender sends Notify message containing location context or room occupancy context. The Call Secretary was shown to process both types of context information properly. When utilizing the proper Notify messages a subscriber's meeting room state indicator and meeting size state indicator could be updated to *true* (indicating that he or she is in a meeting room as pre-defined by the meeting event and that this meeting room was occupied by a suffient number of participants).

5.2.5.3 Uploading or removing a CPL script

As soon as a subscriber appears to be in a meeting state, a CPL script r to redirect this subscriber's incoming calls to their voice mail is created and uploaded to the SER server via a Register message. Conversely, when a subscriber appears to be not in a meeting state (and his or her meeting CPL script has been uploaded), then his or her meeting CPL script will be immediately removed from SER via a Register message with an empty body.

However, there is time delay between a subscriber's physical meeting state changing and his or her incoming call redirect service starting to work. This delay is composed by several factors: the time delay due to sensing, the time delay in SER to process the relevant messages and perform the relevant operations, and the time delay in the Call Secretary itself.

5.2.5.4 Time delay in the sensing system

In any context-aware system, the sensing systems detect and publish updated context about the subscriber's context changing, in this case the subscriber's physical location changing or a meeting room's occupancy changing. Each of these processes require some period of time, i.e., sensing, detecting, data processing, sending a

Publish message integrating, and SIP message processing. A detail discussion of the sensing system's time delay can be found in Xueliang Ren's thesis [8].

5.2.5.5 Time delay in SER

SER system consumes time to operate on SIP messages and to update the database. A detailed discussion of the time delays in SER can be found in Mohammad Zarifi Eslami's thesis [10].

5.2.5.6 Time delay in the Call Secretary

The primary delay within the Call Secretary is due to the timer values selected for each of the loops. In particular, the most sensitive of these is the timing for the loop which examines each subscriber's meeting state - as not action can be taken before this loop checks the subscriber's meeting state. As one minute was chosen to be the value of this timer, this means that each subscriber's meeting state is examined once every minute, thus there is on average a 30 second delay before a change in a subscriber's meeting state is reacted to. Clearly a faster rate of checking for state changes could directly reduce this delay. However, if the Call Secretary must handle a very large number of active subscribers, then a higher rate will dramatically increase the consumption of system resources. The one minute value leads to a maximum 1 minute delay depending on when the Notify message arrived. Actually, there is another time delay caused by examining each subscriber's meeting state. However, this value is quite small comparing with one minute, so I simply ignore it at this point. Note that the delay due to this checking (polling) loop will be near zero if a Notify message has arrived and been processed right before the loop starts running. However, for Notify messages that arrive just after the loop has finished running, there will be quite a long delay before they have an effect. Furthermore, there are some other delays caused by processing of SIP messages and communicating with the SER server and Google Calendar. However, these subsecond delays are small compared with the one minute delay of checking loop. Thus, I believe that the Call Secretary causes 30 seconds delay.

This 30 seconds delay could be reduced to zero by introducing a new class to the Call Secretary application to replace the checking (polling) loop. This loop examines four elements of context information once every minute (the subscriber's current location, the room which is planned for the meeting, the meeting room's occupancy, and the current time) to determine if this subscriber is in meeting. However, the new class could be designated to examine all four elements when any of these elements being updated. For example, when a meeting room's occupancy context information contained in a Notify message matches the meeting criterion, the meeting room's occupancy element is set *true* and the other three elements will be examined (if these

three elements are all *true* a CPL script to this subscriber will be compiled and uploaded to SER immediately, however, if any of these elements is *false* no further action will be take). Hence, the Call Secretary starts or ends redirecting the incoming calls of a subscriber as soon as his or her starts or ends a meeting. The time delay of this system will only be those subsecond delays caused by processing SIP messages, communicating with the SER server & Google Calendar, and examining four elements of meeting state.

Figure 47 shows a SIP message transaction between the Call Secretary and SER (404 User not found error messages are due to the lack of location and room occupancy support within SER).

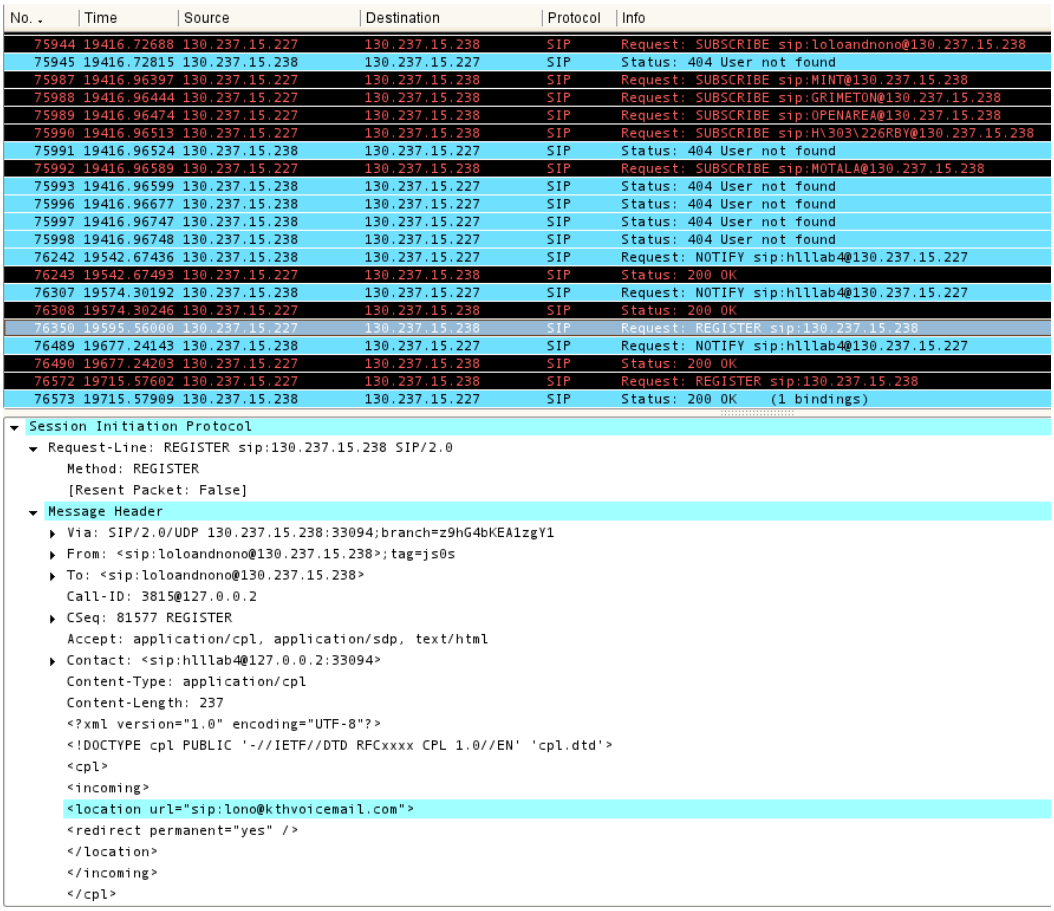

Figure 47: SIP Message Transactions between the Call Secretary and SER

6. Conclusions

In this thesis, I examined two applications (a Context Agent and a Call Secretary) that make use of context information (a subscriber's location information and room occupancy information) via a Presence Agent. Based upon my study of context aware architectures, I implemented a middleware infrastructure for a context aware system and deployed a SIP proxy server (SER with a presence module) as a Presence Agent. The SIMPLE protocol is utilized to communicate among different entities within the system. The context applications are able to retrieve specific context information via using SIP messages to subscribe to a Presence Agent. The Context Agent subscribes to room occupancy information via the SER server. Whereas the Call Secretary retrieves both room occupancy and location context information from the SER server. The prototypes of these two context aware applications have been successfully designed, implemented, and evaluated.

A Context Agent is able to extract useful information from a Notify message which was sent by the SER server containing room occupancy context and provides the Meeting Room Booking System with "real-time" updates of each meeting room's occupancy state. The evaluation of this application clearly shows that context information can easily be acquired from a presence source (i.e., a sensing application which monitors room occupancy state) and this context information be used to extend the capabilities of a client system.

The Call Secretary service utilizes four types of context information (the room which is planned for the meeting, the subscriber's current location, the meeting room's occupancy, and the current time) to estimate a subscriber' s condition in terms of meeting state and redirect the incoming calls to a specified voice mail box when the subscriber is in meeting. Specifically, it starts by acquiring a subscriber's pre-defined meeting event from his or her calendar application in order to compare with criterion for a meeting. Then the Call Secretary subscribes to the subscriber's location and room occupancy context information via a SER server. Once all four context elements match the criterion of a meeting, this subscriber is considered to be in a meeting. This triggers the upload of a specific CPL script for this subscriber to the SER server that will redirect incoming calls to the subscriber's voice mail box. The prototype of the Call Secretary application illustrates that different types of context information can be processed and integrated to provide new services to the subscribers.

Two additional tasks are involved in the Call Secretary to determine a subscriber's meeting state: retrieving the meeting events of the subscriber (in order to extract meeting information to form a meeting state criterion) and estimating a subscriber's location in terms of meeting room based on the latitude & longitude of this subscriber. In the former case, I chose Google Calendar application for the subscribers to create meeting events. These events are accessible via Google's calendar APIs along with

the subscriber's account information. A subscriber's location information is represented in the form of latitude & longitude. I developed a building's coordinate system and transform a subscriber's latitude and longitude values into this building coordinate system to determine if the subscriber is in a specific meeting room. This process is much easier to do if the subscriber's location is represented in the building's coordinate system, rather than in terms of latitude and longitude values.

The Call Secretary utilizes a SIP Register message to upload or remove a CPL script from a SER server. When SER is configured with its CPL module, then SER is able to process the CPL script of a subscriber and can redirect incoming calls to the subscriber's voice mail box or perform other call signaling processing which the subscriber specifies should be done when they are in a meeting.

These two prototypes were developed in Java which makes them independent of the underlying platform. With the support of a Java virtual machine they execute on a variety of different devices (i.e., computer and PDA) even though these devices utilize different operation systems (i.e., Windows and Linux). However, in practice I expect that in the near term the processing of context information will primarily take place in computers attached to the fixed network, for both power and availability reasons.

7. Future work

7.1 Modifying SER presence module

I installed the latest stable version of SER (ser-2.0.0) to implement the required basic SER. The applications receive the room occupancy and location context information via this SER server. However, the default presence module of this basic SER server does not provide full support for subscriptions to location and room occupancy events (as it only supports the presence event). Therefore, I developed a Java application "Notify Sender" to simulate the expected behavior of a SER server which could handle these additional types of events. For further development, the source code of the presence module and the configuration file (ser.cfg) of SER need to be modified to provide location and room occupancy event support.

7.2 Supporting multiple calendar applications

In order to subscribe to the Call Secretary service, the subscribers need to create their meeting events in calendar applications and register their calendar account. Using this calendar account information the Call Secretary will retrieve each subscriber's meeting schedule as events in order to provide input to the meeting state detection. In the prototype of this application, I chose Google's Calendar for subscribers to schedule their meetings as this information is subsequently accessibly via Google's calendar APIs. For further development, the Call Secretary should provide the subscribers with more choices of calendar applications to schedule their meetings (i.e., in addition to Google Calendar). Thus in the future the Call Secretary would retrieve a subscriber's meeting events using his or her calendar account information along with the name of the subscriber's calendar application (selected from an "available calendar applications" list of supported calendars - based upon these calendar applications making available to appropriate information by an API).

7.3 Developing an management interface for the Context Agent

The prototype of the Context Agent application is able to subscribe to five meeting room's occupancy information and provides the Meeting Room System with "real-time" context information. With minor modification of the source code (i.e., the format of Subscribe Request message and the destination address) this prototype could provide additional functions: it could subscribe to more meeting rooms' occupancy information, it could subscribe to different context events (i.e., location

and temperature context), or it could provide context information to different systems (i.e., in addition to the Meeting Room System). In this way, a management interface could be developed to integrate all these functions together. Then an administrator could easily subscribe to the desired context information and/or provide context information to a specific system by changing the settings of the Context Agent. For example, an obvious new application would utilize room occupancy information for improved control of heating, ventilation, and air conditioning - in order to reduce costs and improve the comfort of people in the building.

7.4 Developing an interface for the Call Secretary

In order to subscribe to the Call Secretary service, the subscribers need to register their calendar account information. In this scenario, I manually entered the account information of the subscribers in a table within SER's database. To facilitate users doing these themselves, an interface could be developed to enable the subscribers to perform this registration.

7.5 Time delay

Based upon the tests, I believe that the prototype Call Secretary has an average delay of approximately 30 seconds to respond to changes in context. Additional testing should be done to estimate more accurately this delay (i.e., these tests should examine scenarios with one subscriber and many subscribers). As described in section 5.2.5, a new solution could greatly reduce such delay. In this solution, the Call Secretary would evaluate the subscribers' meeting state when an element of meeting state was updated.

7.6 Security Mechanism

Security features were not included in this project. The Java language provides basic security mechanisms to reduce the risk of security compromise, loss of data and program integrity, and damage to system users. However, the two prototype applications send the subscribers' registration and other information (i.e., a subscriber's calendar account information) via unencrypted UDP packets. For further development, an encryption algorithm needs be applied to secure the authentication and the data transmission. Note that the basic methods to do so are known and in some cases standardized, but not widely used in the context of SIP, SIMPLE, and Calendar communication.

7.7 User Experience

The prototype of Call Secretary was designed upon the basis of limited study and estimation in terms of the subscriber's behavior. Thus the evaluation of the Call Secretary application from the subscriber's perspective is very important. The subscriber's experience and satisfaction need to be collected in order to adjust the design of Call Secretary to provide better assist.

References

[1] B. Schilit and M. Theimer, “Disseminating Active Map Information to Mobile Hosts”. Network, IEEE Volume 8, Issue 5, Sep/Oct 1994 Page(s):22-32. Available at: http://ieeexplore.ieee.org/iel3/65/7582/00313011.pdf?tp=&isnumber=&arnumber=313011

[2] Anind K. Dey, “Context-Aware Computing: The CyberDesk Project”, March 1998. Available at: http://www.cc.gatech.edu/fce/cyberdesk/pubs/AAAI98/AAAI98.html

[3] Anind K. Dey and Gregory D. Abowd, “Towards a Better Understanding of Context and Context-Awareness”, Graphics, Visualization and Usability Center and College of Computing, Georgia Institute of Technology. Available at: http://www.it.usyd.edu.au/%7Ebob/IE/99-22.pdf

[4] Yu Sun, “Context-aware applications for a Pocket PC”, Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), December 2007.

[5] M. Baldauf, S. Dustdar, and F. Rosenberg, “A survey on context-aware systems”, Int. J. Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, pp.263–277, 2007. Available at: https://berlin.vitalab.tuwien.ac.at/%7Eflorian/papers/ijahuc2007.pdf

[6] Anind K. Dey, Gregory D. Abowd, and Daniel Salber, “A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications”, Human-Computer Interaction, 2001, Volume 16, pp. 97–166. Available at: http://www.cc.gatech.edu/fce/ctk/pubs/HCIJ16.pdf

[7] Harry Chen, Tim Finin, and Anupam Joshi, “An Intelligent Broker for ContextAware Systems”, University of Maryland Baltimore County, 2004. Available at: http://www.cs.umbc.edu/%7Efinin//papers/ubicomp03-poster.pdf

[8] Xueliang Ren, “A Meeting Detector to Provide Context to a SIP Proxy”, Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), October 2008.

[9] Haruumi_Shiode, “In-building Location Sensing Based on WLAN Signal Strength”, Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), April 2008.

[10] Mohammad Zarifi Eslami, “A Presence Server for Context-aware Applications”,

Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), December 2007.

[11] Lidan Hu, “An Intelligent Presentation System”, Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), August 2008.

[12] Wikipedia, “Signalling (telecommunications)”, last modified on 27 May 2008. Available at: http://en.wikipedia.org/wiki/Signalling_%28telecommunications%29

[13] Wikipedia, “H.323”, last modified on 23 May 2008. Available at: http://en.wikipedia.org/wiki/H.323

[14] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, “SIP: Session Initiation Protocol”, RFC 3261, June 2002. Available at: http://www.ietf.org/rfc/rfc3261.txt

[15] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, J. Peterson, “Presence Information Data Format (PIDF)”, RFC 3863, August 2004. Available at: http://www.ietf.org/rfc/rfc3863.txt

[16] Wikipedia. “Codec”, last modified on 29 May 2008. Available at: http://en.wikipedia.org/wiki/CODEC

[17] M. Handleym and V. Jacobson. “SDP: Session Description Protocol”, RFC 2327, IETF, April 1998. Available at: http://www.ietf.org/rfc/rfc2327.txt

[18] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. “RTP: A Transport Protocol for Real-Time Applications”, RFC 1889, IETF, January 1996. Available at: http://www.ietf.org/rfc/rfc1889.txt

[19] Wikipedia. “Presence information”, Last modified on 6 June 2007. Available at: http://en.wikipedia.org/wiki/Presence_information

[20] A. B. Roach. “Session Initiation Protocol (SIP)-Specific Event Notification”, RFC 3265, IETF, June 2002. Available at: http://www.ietf.org/rfc/rfc3265

[21] 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

[22] A. B. Roach,B. Campbell, and J. Rosenberg. “A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists”, RFC 4662, August 2006. Available at: http://www.ietf.org/rfc/rfc4662

[23] A. Niemi, Ed. “Session Initiation Protocol (SIP) Extension for Event State Publication”, RFC 3903, IETF, October 2004. Available at: http://www.ietf.org/rfc/rfc3903

[24] “Extensible Markup Language (XML) 1.1 (Second Edition)”, W3C Recommendation 16 August 2006, edited in place 29 September 2006. Available at: http://www.w3.org/TR/xml11/#dt-xml-doc

[25] "Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML)", ISO 8879, 1986. Available at: http://www.iso.ch/cate/d16387.html

[26] Wikipedia. “XML”, last modified on 25 May 2008. Available at: http://en.wikipedia.org/wiki/Xml

[27] Thomas Strang and Claudia LinnhoffPopien, “A Context Modeling Survey”, in First International Workshop on Advanced Context Modelling, Reasoning and Management (UBICOMP), September 2004. Available at: http://www.itee.uq.edu.au/%7Epace/cw2004/Paper15.pdf

[30] Fraunhofer FOKUS, ”SIP Express Router”. Available at: http://www.iptel.org/ser

[31] Fraunhofer FOKUS, ”Presence Agent”. Available at: http://www.iptel.org/ser/doc/modules/pa

[32] Wikipedia. “Call-Processing Language”, Last modified on 9 October 2007. Available at: http://en.wikipedia.org/wiki/Call-Processing_Language

[33] Paul Hazlett, Simon Miles, and Greger V.Teigre, “SER - Getting Started”. Available at: http://siprouter.teigre.com/doc/SER-GettingStarted.pdf

[34] Jiri Kuthan, Jan Janak, and Yacine Rebahi, “iptel.org SIP Express Router v0.11.0 - Admin’s Guide”, 2001. Available at: http://old.iptel.org/ser/doc/seruser/seruser.pdf

[35] J. Lennox, X. Wu, and H. Schulzrinne, “Call Processing Language (CPL): A Language for User Control of Internet Telephony Services”, RFC 3880, IETF, October 2004. Available at: http://www.ietf.org/rfc/rfc3880.txt)

[36] Composite Capability/Preference Profiles (CC/PP), W3C Recommendation 15 January 2004. Available at: http://www.w3.org/TR/CCPP-struct-vocab/

[37] Using UAProf (User Agent Profile) to Detect User Agent Types and Device Capabilities, Learn about UAProf by Developers 'Home webpage, Available at: http://www.developershome.com/wap/detection/detection.asp?page=uaprof

[38] J. Peterson, “Common Profiles for Instant Messaging (CPIM)”, RFC 3860, IETF, August 2004. Available at: http://tools.ietf.org/html/rfc3860

[39] J. Peterson, and NeuStar, “Common Profiles for Presence (CPP)”, RFC 3859, IETF, August 2004. Available at: http://www.ietf.org/rfc/rfc3859.txt

[40] “XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)”, A Reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 January 2000, revised 1 August 2002. Available at: http://www.w3.org/TR/xhtml1/

[41] RSS Advisory Board, “RSS 2.0 specification”, Oct 2007. Available at: htttp://www.rssboard.org/rss-specification

[42] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, and J. Peterson, “Presence Information Data Format (PIDF),” RFC 3863 (Proposed Standard), RFC 3863, Aug 2004. Available at: http://www.ietf.org/rfc/rfc3863.txt

[43] M. Day, S. Aggarwal, G. Mohr, and J. Vincent, “Instant Messaging / Presence Protocol Requirements”, RFC 2779, IETF, Feb. 2000. Available at: http://www.ietf.org/rfc/rfc2779.txt

[44] M. Day, J. Rosenberg, and H. Sugano, “A Model for Presence and Instant

Messaging", RFC 2778, IETF, Feb. 2000. Available at: http://www.ietf.org/rfc/rfc2778.txt

[45] H. Schulzrinne, V. Gurbani, P. Kyzivat, and J. Rosenberg, "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

[46] H. Schulzrinne, "Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals", RFC 4481, IETF, July 2006. Available at: http://www.ietf.org/rfc/rfc4481.txt

[47] M. Lonnfors, E. Leppanen, H. Khartabil, and J. Urpalainen, Presence Information Data format (PIDF) Extension for Partial Presence, Internet-Draft, IETF, November 2006. Available at: http://www.ietf.org/internet-drafts/draft-ietf-simple-partial-pidf-format-08.txt

[48] Google, "Google Calendar APIs and Tools". Available at: http://code.google.com/apis/calendar/

[49] iptel.org, SIP Express Router Source Code version 2.0, last visited:ed September 2008. Available at: http://ftp.iptel.org/pub/ser/2.0.0/src/ser-2.0.0_src.tar.gz,

[50] iptel.org, Serctl tool, last visited: November 13, 2008. Available at: http://ftp.iptel.org/pub/serctl/daily-snapshots/

[51] Jean-Marie Zogg, "Essentials of Satellite Navigation ", 26 April 2007. Available at:http://www.u-blox.com/customersupport/docs/GPS_Compendium(GPS-X-02007).pdf

[52] Franson Technology AB, "GPS Tools". Available at: http://franson.com/gpstools/guide.asp?platform=net

[53] Alisa Devlic, "Creating GPS applications with .NET Compact Framework", September 30, 2005. Available at: http://web.it.kth.se/~devlic/instructions.html

[54] Thaddeus Vincenty, Vincenty formula for distance between two Latitude/Longitude points. Available at: http://www.movable-type.co.uk/scripts/latlong-vincenty.html

[55] the Apache software foundation, HttpClient components of HttpComponents project, 08 February 2008. Available at: http://hc.apache.org/httpclient-3.x/tutorial.html

[56] Alex King, Google calendar alarm it, December 20th, 2006. Available at: http://alexking.org/blog/2006/12/20/google-calendar-alarm-it

[57] the Apache software foundation, "HttpComponents project", last visited:ed on September 21, 2008. Available at: http://hc.apache.org/

[58] Gerald Q. Maguire Jr., Notes on using the HP iPAQ h5550, last visited: 18 September 2008. Available at: http://web.it.kth.se/~maguire/ipaq-notes.html

[59] Athanasios Karapantelakis, "A mobile SIP client: From the user interface design to evaluation of synchronised playout from multiple SIP user agents",

Master thesis, COS/CCS, Royal Institute of Technology (KTH), July 2007

[60] Google, Google SketchUp, last visited: 20th October 2008. Available at: http://sketchup.google.com/

[61] Wikipedia, “Cartesian coordinate system”, November 4 2008. Available at: http://en.wikipedia.org/wiki/Cartesian_coordinate_system

[62] Gerald Combs, Wireshark software, last visited: November 11 2008. Available at: http://www.wireshark.org/

[63] Luan Dang, Cullen Jennings, and David Kelly, Practical VoIP: Using VOCAL, O'Reilly, 2002, ISBN 0-596-00078-2

[64] CPL Editor, last visited: November 12 2008. Available at: http://sourceforge.net/projects/cpled/

[65] Qiang Fu, “Building models of Wireless Local Area Network coverage”, Master thesis, School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH), January 2007.

[66] S. Donovan, “The SIP INFO Method”, RFC 2976, IETF, October 2000. Available at:http://www.ietf.org/rfc/rfc2976.txt

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

[68] GlobalSat, “BT-338 model Bluetooth GPS Receiver”, last visited: November 18, 2008. Available at: http://www.globalsat.com.tw/eng/product_detail_00000039.htm

[69] Meeting room booking system, last visited: November 20, 2008. Available at: http://mrbs.sourceforge.net/

[70] Wikipedia, “Network Time Protocol”, last modified: November 14, 2008. Available at: http://en.wikipedia.org/wiki/Network_Time_Protocol

Appendix A: Modification of the SER configuration

Before CPL scripts can become operational on the SER server, the ser.cfg must be modified to interpret the scripts correctly.

- The cpl-c module must first be loaded by ser:
  loadmodule "/opt/ser/lib/ser/modules/cpl-c.so"

- Next the modules must be configured:
  modparam("cpl-c", "cpl_db", "mysql://root:password@localhost/ser")
  In this case "username" and "password" represent the username and password for the mysql database named ser. The entry at "localhost" should match with the server name on which the mysql database is running. ser and heslo are the default username and password.

  modparam("cpl-c", "cpl_table", "cpl")
  This refers to the "cpl" table which is the default table for the cpl-scripts in the database.

  modparam("cpl-c", "cpl_dtd_file", "/tmp/ser-0.9.0/modules/cpl-c/cpl-06.dtd")
  Pointers to the location of the CPL XML DTD file must be given (the XML DTD is necessary for parsing CPL scripts and is described in further detail later in this section).

  modparam("cpl-c", "lookup_domain", "location")
  This parameter should be set to "location" to let the lookup-node work correctly.

  All the above parameters are mandatory. Two more parameters exist which are optional: A debugging parameter pointing to the existence of a log file and a parameter that specifies the maximum of recursive executions in CPL.

- Call Type Processing Section
  # 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==myself) {
if (method == "INVITE"){
if(!cpl_run_script("incoming", "is_stateless"))
{
# script execution failed
t_reply("500", "CPL script execution failed");
};
route(3);
break;
} else if (method == "REGISTER"){
#handle REGISTER messages with CPL script
cpl_process_register();
route(2);
break;
};
```

- Once the ser.cfg has been modified and SER has been restarted.

Appendix B: the ser.cfg used in this project

```
# This is the ser.cfg file for SER configuration with presence module.

debug=3 # debug level (cmd line: -ddddddddddd)
check_via=no      # (cmd. line: -v)
dns=no                    # (cmd. line: -r)
rev_dns=no           # (cmd. line: -R)
port=5060
children=2
#alias="wireless.kth.se"
mhomed=yes    # usefull for multihomed hosts, small performance penalty
#tcp_accept_aliases=yes # accepts the tcp alias via option (see NEWS)
#tcp_poll_method="sigio_rt"

# ------------------ module loading ----------------------------------

loadmodule "/usr/local/lib/ser/modules/sl.so"
loadmodule "/usr/local/lib/ser/modules/avp.so"
loadmodule "/usr/local/lib/ser/modules/avpops.so"
loadmodule "/usr/local/lib/ser/modules/tm.so"
loadmodule "/usr/local/lib/ser/modules/rr.so"
loadmodule "/usr/local/lib/ser/modules/maxfwd.so"
loadmodule "/usr/local/lib/ser/modules/usrloc.so"
loadmodule "/usr/local/lib/ser/modules/registrar.so"
loadmodule "/usr/local/lib/ser/modules/textops.so"
loadmodule "/usr/local/lib/ser/modules/mysql.so"
loadmodule "/usr/local/lib/ser/modules/dialog.so"
loadmodule "/usr/local/lib/ser/modules/rls.so"
loadmodule "/usr/local/lib/ser/modules/pa.so"
loadmodule "/usr/local/lib/ser/modules/presence_b2b.so"
loadmodule "/usr/local/lib/ser/modules/uri.so"
loadmodule "/usr/local/lib/ser/modules/uri_db.so"
loadmodule "/usr/local/lib/ser/modules/domain.so"
loadmodule "/usr/local/lib/ser/modules/fifo.so"
```

```
loadmodule "/usr/local/lib/ser/modules/xmlrpc.so"
loadmodule "/usr/local/lib/ser/modules/xlog.so"
loadmodule "/usr/local/lib/ser/modules/msilo.so"
loadmodule "/usr/local/lib/ser/modules/xcap.so"
loadmodule "/usr/local/lib/ser/modules/cpl-c.so"
#loadmodule "/usr/lib/ser/modules/unixsock.so"

# Uncomment this if you want digest authentication
# mysql.so must be loaded !
loadmodule "/usr/local/lib/ser/modules/auth.so"
loadmodule "/usr/local/lib/ser/modules/auth_db.so"

# ----------------- setting module-specific parameters ---------------

# modparam("msilo","registrar","sip:registrar@test-domain.com")
modparam("msilo","use_contact",0)
modparam("msilo","expire_time",7200)

# -- usrloc params --

# -- 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", 0)
modparam("rls", "db_url", "mysql://root:ccslab1@localhost:3306/ser")
```

```
modparam("pa", "use_db", 0)
# 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://root:ccslab1@localhost:3306/ser")
# mode of PA authorization: none, implicit or xcap
#modparam("pa", "auth", "xcap")
#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)
# don't 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", 10)

# 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://root:ccslab1@localhost:3306/ser")
#cpl-c
modparam("cpl-c","cpl_db","mysql://root:ccslab1@localhost:3306/ser")

modparam("cpl-c","cpl_table","cpl")

modparam("cpl-c","cpl_dtd_file","/usr/src/ser-2.0.0/modules/cpl-c/cpl-06.dtd")

modparam("cpl-c","lookup_domain","location")


modparam("fifo", "fifo_file", "/tmp/ser_fifo")
modparam("xcap", "xcap_root", "http://localhost/xcap")


# -------------------------    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=~"130.237.15.238") {

        #if (!lookup_domain("To")) {
                    if (lookup_domain("$fd","@from.uri.host")) {
            xlog("L_ERR", "Unknown domain to: %tu from: %fu\n");
            route(1);
            break;
        }

        if (method=="REGISTER") {

            # Uncomment this if you want to use digest authentication
            #if (!www_authorize("130.237.15.238", "subscriber")) {
            #    www_challenge("130.237.15.238", "0");
            #break;
            #};
                                cpl_process_register();
            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;
        };


        if (method=="SUBSCRIBE") {
            if (!t_newtran()) {
                sl_reply_error();
                break;
            };

            if (@to.tag=="") {
                # only for new subscriptions (with empty to tag)

                if (lookup_user("To")) {
                    # existing user -> it is subscription to PA
                    if (handle_subscription("registrar")) {
                        if ((@msg.event=~"presence\.winfo")) {
                            # new watcher info subscription
                            # sends one watcher info NOTIFY message with all
saved authorization requests
                            xlog("L_ERR", "dumping stored winfo to %fu\n");
                            dump_stored_winfo("registrar", "presence");
                        }
                        else {
                            # new presence subscription
                            if ((@msg.event=~"presence") &&
(%subscription_status=="pending")) {
                                # if offline user and new pending subscription
                                if (!target_online("registrar")) {
                                    #%subscription_status="waiting"; # store it as
waiting subscription
                                    xlog("L_ERR", "storing 'pending' winfo      to:
%tu, from: %fu\n");
                                    store_winfo("registrar");
                                }
                            }
                        }
                    }
                    break;
                }
```

```
                if (((@msg.supported=~"eventlist")) {
                    # such user doesn't exist and Supported header field
                    #       -> probably RLS subscription

                    if (lookup_domain("$td","@ruri.host")) {
                        if (lookup_user("From")) {
                            if (is_simple_rls_target("$uid-list")) {
                                # log(1, "it is simple subscription!\n");
                                # takes From UID and makes XCAP query for
user's
                                # list named "default"
                                if (!query_resource_list("default")) {
                                    t_reply("404", "No such user list");
                                    break;
                                }
                            }
                        }

                    if (!have_flat_list()) {
                        # query_resource_list failed or was not called
                        # do standard RLS query acording to To/AOR
                        if (!query_rls_services()) {
                            log(1, "XCAP query failed\n");
                            t_reply("404", "No such list URI");
                            break;
                        }
                    }

                    handle_rls_subscription("1");
                }
                else {
                    # not resource list subscription -> invalid user
                    xlog("L_ERR", "subscription to invalid user %tu\n");
                    t_reply("404", "User not found");
                }

                break;
            }
            else {
                # renewal subscriptions - try to handle it as RLS and if failed, handle
it as PA subscription
```

```
                # FIXME: better will be test like existing_rls_subscription()
                #               and existing_subscription("registrar")
                if (!handle_rls_subscription("0")) {
                    handle_subscription("registrar");
                }
                break;
            }
        };

        if (method=="PUBLISH") {
            if (!t_newtran()) {
#                   log(1, "newtran error\n");
                sl_reply_error();
                break;
            };
            handle_publish("registrar");

            # deliver messages to online user
            # TODO: only if user goes from offline to online?
            if (target_online("registrar")) {
                # log(1, "Dumping stored messages\n");
                # 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;
        };

        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;
        }

        lookup("aliases");
        if (!uri==myself) {
        append_hf("P-hint: outbound alias\r\n");
            route(1);
            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");
    };
}
```

Appendix C: How to acquire data from the GPS receiver

This introduction was originally created by Alisa Devlic [59]. We have done necessary modifications on this introduction.

Environment requirements:
A trial version of GpsTools SDK, Visual Studio .NET (2003 or 2005 version), the latest version of ActiveSync (If using .NET 2005 and want to deploy application on IPAQ device)

1) Download and unzip GpsTools SDK.
2) Run Setup.exe
3) Then take a look at the sample project we applied in our application. The directory: ..*Franson\GpsTools SDK v2.3\dotNetcf\C#\SerialPortNoEvents.*
4) Apply for a 30 Days Trial licence key at the Franson web site. You will receive it by e-mail.
5) In our scenario, we use physical device, HP pocket PC. You can skip this step is you also use physical device.
6) If you use the Pocket PC Emulator and have a GPS receiver connected to your PC, you should set up the emultator's serial port. Go to menu: select *Tools - Options* and then choose the device (Pocket PC 2003 Emulator). Then click on *Properties* and/or *Configure* (depending on the version of .NET). Select the tab *Peripherals* or *Hardware*, then select the serial port on the PC you want to be the virtual device "COM1:" on the Pocket PC emulator.
7) Install a native DLL GpsLibCE.dll.
   Note: this is done differently for the emulator and a physical device.
   For physical device:
   a) Connect computer with device using ActiveSync.
   b) Transfer the appropriate CAB file to the device. The name of the CAB file that you need for Pocket PC 2003 (ARM, Xscale CPU) is located in directory ..\Franson\GpsTools SDK v2.3\dotNetcf\wince300\arm\GpsLibCE.arm.CAB.
   c) Run GpsLibCE.arm.CAB on the device. GpsLibCE.dll will be installed under "\Windows".

   For emulator:

Right-click on the sample project in Visual Studio and select "Add existing item...". Change "Files of type:" to "*.*" and then select "wince300\x86em\ppc2002\GpsLibCE.dll".

8) Open the project in Visual Studio.
   If you are using .NET 2005, Visual Studio will need to convert it to the current version. Open project file *(SerialPortNoEvents.proj)* in a text editor, where you can edit it. Find the tab *<TargetFrameworkVersion>* and modify its value to v2.0 in order to find the right compiler version.
9) In the main class (Form1.cs), find the line in the code that contains the license.LicenseKey and modify it to take value of the 30 Days Trial license key that you have been received by the e-mail.
10) Add a reference to GpsToolsNET.dll, GpsViewNET.dll, and GpsShapeNET.dll. Right click on "References" and select "Add Reference". Click "Browse...". Go to the root directory of the unzipped downloaded file. Select "GpsToolsNET.dll", "GpsViewNET.dll", and "GpsShapeNET.dll" and click "OK" on the "Add Reference" dialog. And then build the project.
11) Deploy the project to physical device or emulator, and then enjoy it.

Appendix D: Context Agent source code

The Java source code of the Context Agent

```
package contextagent;
/**
 *
 * @author Ke Wang June,2008
 */
import java.io.*;
import java.net.*;
import java.lang.System.*;
import java.util.Random;
import java.sql.*;

public class ContextAgent {

    private byte[] buf = new byte[1024];
    private static DatagramSocket ds = null;
    private static String stringServer = "130.237.15.238";
    InetAddress serverAddr = InetAddress.getByName(stringServer);
    private static int port = 5092;
    //private static int listenPort = 5072;
    private InetSocketAddress socketAddress = null;
    private InetAddress i = InetAddress.getLocalHost();
    private String hostName = i.getHostName();
    private String hostAddr = i.getHostAddress();
    private static java.sql.Connection conn = null;

    /**
     * UDP Socket
     * @throws Exception
     */
    public ContextAgent() throws Exception {
        //socketAddress = new InetSocketAddress(host, port);
```

```
        //auto use a unused port to send and receive datagram
        ds = new DatagramSocket();
        System.out.println("Call Agent is up and running...");
}

/**
  * bind time
  * @param timeout
  * @throws Exception
  */
public final void setSoTimeout(final int timeout) throws Exception {
        ds.setSoTimeout(timeout);
}

/**
  * get bind time
  * @return timeout
  * @throws Exception
  */
public final int getSoTimeout() throws Exception {
        return ds.getSoTimeout();
}

public final DatagramSocket getSocket() {
        return ds;
}

/**
  * Method for generating random cSeq value used in SIP Subscribe header
  * the value can be arbitrary less than 2**31
  * We use current time as random seed and limit the value from 1 to 99999
  */
private static String getcSeq() {
        Random random = new Random(System.currentTimeMillis());
        int r = random.nextInt();
        int cSeqSeed = Math.abs(r % 99999);
        String cSeq = Integer.toString(cSeqSeed);
        return cSeq;
}

/** Generating random call-ID for SIP Subscribe header
  * We simply use the time based random number
```

```
  * plus another random number use Math.random()
  */
private static String getCallID() {
        Random rd = new Random(System.currentTimeMillis());
        int r = rd.nextInt();
        int c = Math.abs(r % 88888);
        double j = Math.random() * 100;
        int call-IDSeed = (int) j + c;
        String call-ID = Integer.toString(call-IDSeed);
        return call-ID;
}

/**
  * Method for generating random value for CallID,branch,and Tag
  * used in SIP header
  */
private static String getRandomNum(int randomLen) {
        //62 elements: 0~9, A~Z, a~z.    the length of Arry is 62-1
        final int maxNum = 61;
        int i;    //random number

        int count = 0; //random number length

        char[] str = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
                'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
        };

        StringBuffer random = new StringBuffer("");
        Random r = new Random();
        while (count < randomLen) {
                //collect only positive number

                i = Math.abs(r.nextInt(maxNum));

                if (i >= 0 && i < str.length) {
                        random.append(str[i]);
                        count++;
                }
        }
```

```
        return random.toString();
    }

    /**
     *Method of sending message to sepicific SER server
     * @param data needs to be sent
     * @return DatagramPacket
     * @throws IOException
     */
    public final DatagramPacket send(String sendOut)
            throws IOException {
        byte[] bytes = sendOut.getBytes();
        DatagramPacket dp = new DatagramPacket(bytes, bytes.length, serverAddr,
port);
        ds.send(dp);
        return dp;
    }

    /**
     * Method of receiving message from specific SER server
     * @return datagram received
     * @throws Exception
     */
    private final String receive()
            throws Exception {
        DatagramPacket dp = new DatagramPacket(buf, buf.length);
        ds.receive(dp);
        String info = new String(dp.getData(), 0, dp.getLength());
        return info;
    }

    /**
     * Close UDP socket.
     */
    private final void close() {
        try {
            ds.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

```
        private String getREG(String cSeq, String call-ID, String branch, String tag,
int lport) {

        String context_type = "occupancy";
        String subHeader = "";
        subHeader = "REGISTER " + "sip:" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + stringServer +
                ":" + lport + ";rport;branch=z9hG4bK" + branch + "\r\n" +
                "From: <sip:" + hostName + "@" + stringServer +
                ">;tag=" + tag + "\r\n" +
                "To: <sip:"+ hostName +"@" + stringServer + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + "130.237.15.227" + "\r\n" +
                "CSeq: 1" + "REGISTER" + "\r\n" +
                "Max-Forwards: 70" + "\r\n" +
                "Contact: <sip:" + hostName + "@" + "130.237.15.227" + ":" +
lport + ">" + "\r\n" +
                "Expires: 600" + "\r\n" +
                "Content-Length: 0" + "\r\n\r\n";
        return subHeader;
    }
    /**
     * Method of sending Subscribe message
     */
    private String getSUB(String cSeq, String call-ID, String branch, String tag, int
lport) {

        String context_type = "occupancy";
        String subHeader = "";
        subHeader = "SUBSCRIBE " + "sip:tere@" + stringServer + ":"+lport +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + stringServer +
                ":" + lport + ";branch=z9hG4bK" + branch + "\r\n" +
                "From: <sip:" + hostName + "@" + stringServer +
                ">;tag=" + tag + "\r\n" +
                "To: <sip:tere@" + stringServer + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + "130.237.15.227" + "\r\n" +
                "CSeq: " + cSeq + " SUBSCRIBE" + "\r\n" +
                "Max-Forwards: 70" + "\r\n" +
                "Event: " + context_type + "\r\n" +
                "Accept: application/pidf+xml" + "\r\n" +
```

```
                    "Contact: <sip:" + hostName + "@" + "130.237.15.227" + ":" +
lport + ">" + "\r\n" +
                    "Expires: 600" + "\r\n" +
                    "Content-Length: 0" + "\r\n\r\n";
            return subHeader;
        }

        /**
         * Method of make up 200 OK message according to SIP message received from
SER
         */
        private String getOK(String packet) {
            String okHeader = "";
            String str[] = new String[16];
            str = packet.split("\r\n");
            String receivedAddr=
packet.substring(packet.indexOf("@"),packet.indexOf(" ", packet.indexOf("@")));
            okHeader = "SIP/2.0 200 OK"+ "\r\n" +
                    str[2] + ";received="+ receivedAddr + "\r\n" +
                    str[3] + "\r\n" +
                    str[4] + "\r\n" +
                    str[5] + "\r\n" +
                    str[6] + "\r\n" +
                    str[7] + "\r\n" +
                    "Content-Length: 0" + "\r\n\r\n";
            return okHeader;
        }

        //load database driver and make a connection
        private static void makeConnection() throws SQLException,
ClassNotFoundException {
            try {
                String databaseName = "mrbs";
                String usrName = "context";
                String password = "adios";
                String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
                Class.forName("org.gjt.mm.mysql.Driver").newInstance();
                //Class.forName("com.mysql.jdbc.Driver").newInstance();
                conn = DriverManager.getConnection(url, usrName, password);
            // } catch (java.lang.ClassNotFoundException e) {
            } catch (Exception e) {
                System.err.print("ClassNotFoundException: ");
```

```
                System.err.println(e.getMessage());
            }
        }

        //Close database connection
        private static void closeConnection() {
            try {
                if (conn != null) {
                    conn.close();
                }
                conn = null;
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        /**
         * Main method
         * @param args
         * @throws Exception
         * Creation date: 22/06/2008
         */
        public static void main(String[] args) throws Exception {
            InetAddress i = InetAddress.getLocalHost();
            String hostName = i.getHostName();
            String hostAddr = i.getHostAddress();
            String packet = null;

            //generate SIP Header random parameters
            String cSeq = "";
            cSeq = ContextAgent.getcSeq();
            String call-ID = "821127";
            //call-ID = ContextAgent.getCallID();
            String branch = "";
            branch = ContextAgent.getRandomNum(7);
            String tag = "";
            tag = ContextAgent.getRandomNum(4);

            // construct subscribe and ok message, and sending subscribe message to
server
            ContextAgent watcher = new ContextAgent();
            int portNo = ds.getLocalPort();
```

```
            System.out.println("call-ID=" + call-ID);
            System.out.println("portNumber=" + portNo);
            String sub = watcher.getSUB(cSeq, call-ID, branch, tag, portNo);
            watcher.send(sub);

            System.out.println(sub);
            // read incoming message from server
            while (true) {
                  packet = watcher.receive();
                  System.out.println("\r\n"+ "Received one message: ");
                  System.out.println(packet);
                  if (packet != null) {
                        // parse SIP message and process it according to its category
                        int ifSIP = packet.indexOf("SIP/2.0");
                        if (ifSIP < 0) {
                              System.out.println("This is not a SIP message.");
                        } else {
                              String msgType = null;

                              //extract call-ID of incoming message
                              String tagCallID = "Call-ID: ";    //check if there is a space

                              int startCallID = packet.indexOf(tagCallID) +
tagCallID.length();
                              int endCallID = packet.indexOf("@", startCallID);
                              String recCallID = packet.substring(startCallID, endCallID);
                              System.out.println("recCallID=" + recCallID);

                              if (recCallID.equals(call-ID)) {
                                    System.out.println("Same call-ID value, available SIP
message");
                                    if (ifSIP == 0) { // this is a 200/202 OK message or an
error messge

                                          String msg = packet.substring(packet.indexOf(" ")
+ 1, packet.indexOf("\r\n"));
                                          System.out.println("This is a " + msg + "
message!");

                                    } else {
                                          //this is a Notify message
                                          msgType = "Notify";
```

```
                                String ok = watcher.getOK(packet);
                                watcher.send(ok);
                                System.out.println("This is a Notify message");
                                System.out.println("200 OK msg is sent out.. ");
                                System.out.println(ok);

                                // extract Subscription state of incoming message
                                String subState = "Subscription-State: ";
                                int startState = packet.indexOf(subState) +
subState.length();
                                int endState = packet.indexOf("\r\n", startState);
                                String recState = packet.substring(startState,
endState);
                                System.out.println(recState);

                                if (recState.equals("terminated")) {
                                        //expired Notify message,send OK, and
subscripe message
                                        System.out.println("recState=" + recState);
                                        msgType = "expiredNotify";
                                        watcher.send(sub);
                                        System.out.println("Subscription is expired,
resubscribed"+"\r\n"+ sub);

                                } else {
                                        // a Notify msg with context information

                                        //parse packet for xml context
                                        // read room name and occupation number
                                        String occupContext = null;
                                        String occupStart = "<value>";
                                        String occupEnd = "</value>";
                                        String roomStart = "<room>";
                                        String roomEnd = "</room>";
                                        String roomContext = null;
                                        int occupIndex = packet.indexOf(occupStart);
                                        if (occupIndex <= 0) {
                                                //notify message without xml context
information
                                        } else {
                                                occupContext =
packet.substring((packet.indexOf(occupStart) + occupStart.length()),
```

```
                                                    (packet.indexOf(occupEnd)));
                                            roomContext =
packet.substring((packet.indexOf(roomStart) + roomStart.length()),
                                                    (packet.indexOf(roomEnd)));
                                             System.out.println("Room " +
roomContext + " has " + occupContext + " people inside.");
                                            //update mrbs database according to
context of incoming notify message
                                            makeConnection();
                                            String sqlQuery = "update
mrbs_occupation set occupation = '" +
                                                    occupContext + "' where
room_name = '" + roomContext +
                                                    "'";
                                            Statement stmt;
                                            stmt = conn.createStatement();
                                            try {
                                                stmt.executeUpdate(sqlQuery);

                                            } catch (Exception e) {
                                                System.out.print(e);
                                                System.out.println(
                                                        "No exsiting table
found");
                                            }
                                            stmt.close();
                                            closeConnection();
                                        }

                                    }
                                }
                            } else {
                                System.out.println("Wrong CallID, discard this
message");
                            }

                        }
                    }
                    packet = null;
                }
            }
        }
```

Appendix E: Source code for retrieving scheduled events from Google's Calendar

The Java source code of a small application using HttpClient components used for retrieving Google Calendar's content

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package httpclientcalendar;

/**
 *
 * @author Ke Wang
 */
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;
import org.apache.commons.httpclient.params.HttpMethodParams;

import java.io.*;

public class HttpClientCalendar {
   //Retrieve calendar events through "ICAL" address
   private static String url =
"http://www.google.com/calendar/ical/mrv6umfmmq7107anoqagcflhro%40group.cale
ndar.google.com/public/basic.ics";

   public static void main(String[] args) {
      // Create an instance of HttpClient.
      HttpClient client = new HttpClient();
      // Create a method instance.
      GetMethod method = new GetMethod(url);
```

```
        // Provide custom retry handler is necessary
        method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
                new DefaultHttpMethodRetryHandler(3, false));

        try {
            // Execute the method.
            int stateCode = client.executeMethod(method);

            if (stateCode != HttpStatus.SC_OK) {
                System.err.println("Method failed: " + method.getStatusLine());
            }

            // Read the response body.
            byte[] responseBody = method.getResponseBody();

            // Deal with the response.
            // Use caution: ensure correct character encoding and is not binary data
            String responseString = new String(responseBody);
            System.out.println(responseString);

        } catch (HttpException e) {
            System.err.println("Fatal protocol violation: " + e.getMessage());
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("Fatal transport error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Release the connection.
            method.releaseConnection();
        }
    }
}
```

Appendix F: Call Secretary source code

The Call Secretary package contains 5 class file: Call Secretary, Context Agent, Location Indicator, Meeting Info, and Meeting List.

CallSecretary.java

```
package callsecretary;
/**
  * @author Ke Wang
  */
import com.google.gdata.client.calendar.*;
import com.google.gdata.data.calendar.*;
import com.google.gdata.data.DateTime;

import java.util.Date;
import java.text.SimpleDateFormat;

import java.net.*;
import java.io.*;

import java.sql.*;


import java.util.ArrayList;
import java.util.Calendar;
import java.util.Timer;
import java.util.TimerTask;

import java.util.Iterator;
import java.util.TimeZone;
import java.util.Vector;

import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class CallSecretary {

	static ArrayList hourList = new ArrayList();
	static ArrayList minList = new ArrayList();
	//format of matchVector is
username,cpl,loc,currentloc,meetingsize(small/big),occupancy,timetag(meeting start
or not),cpltag
	static Vector matchVector = new Vector();
	//Database connection method
	private static java.sql.Connection conn = null;
	//SER subscription variables
	static DatagramSocket ds = null;
	static ContextAgent watcher = null;
	static String contextLocation = "location";
	static String contextoccupancy = "occupancy";
	static String contextCPL = "application/cpl";
	static String cSeq = null;
	static String call-ID = null;
	static String branch = null;
	static String tag = null;
	static String ok = null;
	//room names
	static String mint = "MINT";
	static String grimeton = "GRIMETON";
	static String openarea = "OPENAREA";

	static String h 枚 rby = "H 脰 RBY";

	static String motala = "MOTALA";
	//used for meeting room geo coordinates
	static LocationIndicator roomCoordinates = new LocationIndicator();
	static Vector roomCoordinateVector = new Vector();
	static Boolean meetingStart = true;
	//meeting size definition
	int bigPeople = 5;
	int smallPeople = 2;

	public CallSecretary() {
		// Schedule a task that executes once every 2 hrs
		// to fetch users's next 2 hrs meeting entries
		Timer iCalTimer;
		iCalTimer = new Timer();
```

```
            iCalTimer.schedule(new ICalTask(), 0, //initial delay
                        3600 * 1000); //subsequent rate 2 hours

            // Schedule a task that executes once a minute
            // to match the user's incoming meeting time to current time
            Timer meetingTimer;
            meetingTimer = new Timer();
            meetingTimer.schedule(new MeetingTask(), 0, //initial delay
                        60 * 1000); //subsequent rate one minute

            //execute once a minute to match 3 indictors and decide if uploading CPL
script
            Timer cplTimer;
            cplTimer = new Timer();
            cplTimer.schedule(new CPLTask(), 20, //initial delay
                        60 * 1000); //subsequent rate one minute
                  }

      class CPLTask extends TimerTask { // runs every minute, so there will be one
minute delay to start up service for subscriber

            public void run() {
                  int port = watcher.getPort();
                  while (matchVector.size() >= 1) {
                        System.out.println("CPLTask is examining subscriber's meeting
status");
                        for (int i = 0; i < (matchVector.size() / 8); i++) {
                              //from index 0 to index 7, username,cplvoicemail,meeting
room,currentlocation,meetingsize,occupancy,time,cplTag
                              String userName = (String) matchVector.get(i * 8);
                              String cplVoicemail = (String) matchVector.get(i * 8 + 1);
                              String room = (String) matchVector.get(i * 8 + 2);
                              String currentRoom = (String) matchVector.get(i * 8 + 3);
                              String sizeStr = (String) matchVector.get(i * 8 + 4);
                              Boolean start = (Boolean) matchVector.get(i * 8 + 6);
                              Boolean cplTag = (Boolean) matchVector.get(i * 8 + 7); //
initial status is false

                              Boolean sizeBoolean = false;
                              Boolean locBoolean = false;

                              if (room.equalsIgnoreCase(currentRoom)) { //subscriber's
```

```
location
                                locBoolean = true;
                                System.out.println(userName + " is currently at the
meeting room " + matchVector.get(i * 8 + 3) + "\r\n");

                            if (sizeStr.equalsIgnoreCase("BIG")) {      // big meeting
needs 5 or more than 5 participants

                                if ((Integer) matchVector.get(i * 8 + 5) >= bigPeople) {
                                    sizeBoolean = true;
                                    System.out.println("This is a big meeting,
participates are ready for it" + "\r\n");
                                //    break;
                                } else {
                                }
                            } else if (sizeStr.equalsIgnoreCase("SMALL")) {    // big
meeting needs 2 or more than 2 participants

                                if ((Integer) matchVector.get(i * 8 + 5) >= smallPeople)
{
                                    sizeBoolean = true;
                                    System.out.println("This is a small meeting,
participates are ready for it" + "\r\n");
                                } else {
                                }
                            } else {
                                // break;
                            }
                                 } else {}

                            if (sizeBoolean && locBoolean && start) {

                                if (cplTag) {
                                    // break;
                                } else {
                                    try {
                                        //upload CPL script by sending out SIP msg
                                        Boolean trueCPLTag = true;
                                        String regCPL =
watcher.getRegCPL(userName, contextCPL, port, cplVoicemail);
                                        watcher.send(regCPL);
```

```
                                        for (int x = 0; x < matchVector.size() / 8; x++)
{
                                            String name = (String)
matchVector.get(x);
                                            if (name.equals(userName)) {
                                                matchVector.set((x * 8 + 7),
trueCPLTag);
                                            } else {
                                            }
                                        }
                                        System.out.println(userName + "'s CPL script
has been uploaded" + "\r\n");
                                        System.out.println("after CPL uploaded,
meeting status is " + matchVector + "\r\n");
                                        break;
                                    } catch (IOException ex) {

Logger.getLogger(CallSecretary.class.getName()).log(Level.SEVERE, null, ex);
                                    }
                                }
                            } else {
                                if (cplTag) { // when mis matching meeting status,
meanwhile, CPL script is already uploaded, remove CPL.

                                    try {
                                        //delete CPL script by sending register
message with zero content length
                                        String regCPL =
watcher.getRemoveCPL(userName, contextCPL, port);
                                        watcher.send(regCPL);
                                        System.out.println(userName + "'s CPL script
has been deleted");
                                         break;
                                    } catch (IOException ex) {

Logger.getLogger(CallSecretary.class.getName()).log(Level.SEVERE, null, ex);
                                    }
                                }
                            }
                        }
                        break;
                }
```

```
        }
    }

    static class MeetingTask extends TimerTask {

        public void run() {

            while (hourList.size() >= 1) {
                //System.out.println("MeetingTask starts working!");
                DateTime startTime = new DateTime();
                DateTime endTime = new DateTime();

                System.out.println("Service is working on " + hourList.size() + "
users" + "\r\n");

                //take out each user's first meeting info, and then packet them into
a ArrayList
                for (int i = 0; i < hourList.size(); i++) {     //if call i<=1,
IndexOutofBounds Error

                    ArrayList userEntry = new ArrayList();
                    userEntry = (ArrayList) hourList.get(i);

                    //because each meeting entry would be deleted after its
endTime
                    //so it is possible in the two hours loop one user on hourList
                    //might have no meeting entry to be uploaded into the
minList
                    MeetingList meetingList = new MeetingList();
                    meetingList = (MeetingList) userEntry.get(new Integer(2));
                    //System.out.println(meetingList.getSize());

                    while (meetingList.getSize() >= 1) {
                        //read data from meetingList
                        //MeetingInfo meetingInfo =
meetingList.getMeeting(meetingList.getSize()-1);
                        MeetingInfo meetingInfo =
meetingList.getMeeting(new Integer(0));
                        //System.out.println(meetingInfo.getRoomName());
                        String userName = (String) userEntry.get(new
Integer(0));
```

```
                        String cpl_xml = (String) userEntry.get(new
Integer(1));
                        startTime = meetingInfo.getStartTime();
                        endTime = meetingInfo.getEndTime();

                        Long start = startTime.getValue();
                        Long end = endTime.getValue();
                        Long nowTime = DateTime.now().getValue();

                        // System.out.println(nowTime);

                        if (start.compareTo(nowTime) > 0) {    //meeting not
started
                            // break;

                        } else {
                            if (end.compareTo(nowTime) > 0) {    // meeting
already started

                                matchVector.set(new
Integer(matchVector.indexOf(userName) + 6), meetingStart);
                                //System.out.println(matchVector.get(new
Integer(matchVector.indexOf(userName) + 6)));
                                System.out.println(userName + " starts
meeting from " + startTime + "\r\n");
                            } else if (end.compareTo(nowTime) < 0)
{ //meeting ended, delete this entry from hourList

                                if (meetingList.getSize() - 1 >= 1) {
                                    meetingList.removeMeeting(new
Integer(0));
                                    //update location, meetingsize and time
elements with next meeting information
                                    MeetingInfo nextMeetingInfo =
meetingList.getMeeting(new Integer(0));
                                    String nextMeetingLocation =
nextMeetingInfo.getRoomName();
                                    String nextMeetingSize =
nextMeetingInfo.getMeetingSize();

matchVector.set(matchVector.indexOf(userName) + 2, nextMeetingLocation);
```

```
matchVector.set(matchVector.indexOf(userName) + 4, nextMeetingSize);

matchVector.set(matchVector.indexOf(userName) + 6, new Integer(0));
                                        System.out.println(userName + "'s
meeting started from " + startTime + "\r\n");
                                    } else {
                                        int elementRemove =
matchVector.indexOf(userName);
                                        for (int er = 0; er <= 7; er++) {

matchVector.remove(elementRemove); // always remove the first index,repeat 7 times
can remove all elements of this entry

                                        }
                                        hourList.remove(i);
                                        System.out.println(userName + "'s
meeting schedule in this two hour period is finished..." + "\r\n");

                                    }
                                }
                            //      break;
                            }
                            break;
                        }
                    }
                    break;
                }
            }
        }

        public static class ICalTask extends TimerTask {

            public ICalTask() {
            }
            int j = 0;
            Integer integer = new Integer(j);

            public ArrayList getList() {
                return hourList;
            }

            public void run() {
```

```
                //empty the hourList and matchVector everytime call run() method.
                hourList.clear();
                matchVector.clear();
                int port = watcher.getPort();

                while (true) {
                    System.out.println("Starting fetching iCal info every 2 hours...");

                    //Database connection getting users' iCal url
                    //public static void main(String[] args) throws IOException,
ServiceException {
                    String sqlQuery = "SELECT name,cpl_voicemail,email,ical_psw
from call_secretary where active = '1'";

                    try {
                        makeConnection();
                        ResultSet rs;
                        Statement stmt;
                        stmt = conn.createStatement();
                        rs = stmt.executeQuery(sqlQuery);
                        ResultSetMetaData rsMetaData = rs.getMetaData();
                        int columnNumber = rsMetaData.getColumnCount();
                        rs.last();
                        int numRows = rs.getRow();
                        rs.beforeFirst();

                        System.out.println("There are " + numRows + " users
registered this service");
                        System.out.println();

                        String[][] str = new String[new Integer(numRows +
1)][7];//from zero

                        int y = 0;

                        //get current time by java.util.Date format and converted into
GMT timezone
                        //finally modify them to stupid calendar query accepted
format
                        java.util.Date currentTime = new Date();

                        Calendar cal =
```

```
Calendar.getInstance(TimeZone.getTimeZone("GMT"));
                        SimpleDateFormat myDate = new
SimpleDateFormat("yyyy-MM-dd");
                        SimpleDateFormat myTime = new
SimpleDateFormat("HH:mm:ss");
                        myDate.setTimeZone(TimeZone.getTimeZone("GMT"));
                        myTime.setTimeZone(TimeZone.getTimeZone("GMT"));

                        String startDate = myDate.format(cal.getTime()).toString();
                        String startTime = myTime.format(cal.getTime()).toString();

                        java.util.Date twoHours = new Date(currentTime.getTime()
+ 7200000);
                        String endDate = myDate.format(twoHours).toString();
                        String endTime = myTime.format(twoHours).toString();

                        System.out.println("start Time: " + startTime);
                        System.out.println("end Time: " + endTime);


                        while (rs.next()) {
                              y++;

                              for (int i = 1; i <= columnNumber; i++) {
                                    str[y][i] = rs.getString(i);
                              }

                              String usrName = str[y][1];
                              String cpl_voicemail = str[y][2];
                              String email = str[y][3];
                              String iCalPSW = str[y][4];


                              CalendarService myService = new
CalendarService("ccslab-CallSecretary-1.0");
                              myService.setUserCredentials(email, iCalPSW);
                              URL feedURL = new
URL("http://www.google.com/calendar/feeds/" +
                                          email + "/private/full");
                              CalendarQuery myQuery = new
CalendarQuery(feedURL);
```

```
//myQuery.setMinimumStartTime(DateTime.parseDateTime("2008-09-15T17:00:00"
));

//myQuery.setMaximumStartTime(DateTime.parseDateTime("2008-09-15T18:00:59"
));


myQuery.setMinimumStartTime(DateTime.parseDateTime(startDate + "T" +
startTime));

myQuery.setMaximumStartTime(DateTime.parseDateTime(endDate + "T" +
endTime));

                                // Send the request and receive the response:
                                CalendarEventFeed resultFeed =
myService.query(myQuery, CalendarEventFeed.class);

                                while (resultFeed.getTotalResults() >= 1) {
                                    ArrayList usrEntry = new ArrayList();
                                    // String timeZone = resultFeed.getTimeZone();

                                    //write user,location,occupancy,time,cpl indicators
into matchVector
                                    try {
                                        // Constructing location and occupancy
subscribe message, and sending them to SER
                                        String subLocation =
watcher.getSUB(usrName, contextLocation, port);
                                        watcher.send(subLocation);
                                        System.out.println(usrName + "'s location
subscription has been sent out");

                                    } catch (IOException ex) {

Logger.getLogger(CallSecretary.class.getName()).log(Level.SEVERE, null, ex);
                                    }


                                    //Store each meeting information into MeetingInfo
class instance
                                    //Store one user's meetings into MeetingList class
instance
```

```
                                MeetingList meetingList = new MeetingList();
                                String meetingTitle = null;
                                String meetingDescription = null;
                                Boolean start = false;
                                Boolean cplTag = false;
                                for (int i = 0; i < resultFeed.getEntries().size(); i++)
{

                                        CalendarEventEntry eventEntry =
resultFeed.getEntries().get(i);

                                        // while (eventEntry != null) {
                                        //System.out.println(eventEntry.getContent());
                                        meetingTitle =
eventEntry.getTitle().getPlainText();

                                        //judge this event entry is a meeting, searching
the key word "meeting"
                                        if
(meetingTitle.toUpperCase().indexOf("MEETING") >= 0) {
                                                System.out.println(usrName + " has
meeting schedule ..." + "\r\n");
                                                System.out.println("Meeting title: " +
meetingTitle);
                                                System.out.println("Meeting room name:
" + eventEntry.getLocations().get(0).getValueString());
                                                System.out.println("Meetting starting
time: " + eventEntry.getTimes().get(0).getStartTime() + "\r\n");
                                                usrEntry.add(usrName);
                                                usrEntry.add(cpl_voicemail);
                                                MeetingInfo meetingInfo = new
MeetingInfo();
                                                // meetingSize value taken from
TextContent, big or small,
                                                //if user hasnt define any meeting size
info, make it as small meeting
                                                String meetingSize = null;
                                                int bigSize =
eventEntry.getTextContent().toString().toUpperCase().indexOf("BIG");
                                                int smallSize =
eventEntry.getTextContent().toString().toUpperCase().indexOf("SMALL");
                                                if (bigSize >= 0) {
```

```
                                        meetingSize = "BIG";
                                    } else {
                                        meetingSize = "SMALL";
                                    }

meetingInfo.setMeetingValues(eventEntry.getTimes().get(0).getStartTime(),

eventEntry.getTimes().get(0).getEndTime(),

eventEntry.getLocations().get(0).getValueString(),
                                                meetingSize);

                                    meetingList.addMeeting(0,  meetingInfo);
//put the closest meeting the first place

                                    usrEntry.add(meetingList);

                                    //from index 0 to index 7,
username,cplvoicemail,meeting
room,currentlocation,meetingsize,occupancy,time,cplTag
                                    String MeetingRoom =
meetingList.getMeeting(new Integer(0)).getRoomName();
                                    String MeetingSize =
meetingList.getMeeting(new Integer(0)).getMeetingSize();
                                    int zeroOccupancy = 0;

                                    matchVector.add(0, usrName);
                                    matchVector.add(1, cpl_voicemail);
                                    matchVector.add(2, MeetingRoom);
                                    matchVector.add(3, null);
                                    matchVector.add(4, MeetingSize);
                                    matchVector.add(5,
zeroOccupancy);//initial occupancy is int 0

                                    matchVector.add(6, start); //false stands
for meeting has not started

                                    matchVector.add(7, cplTag);
                                    System.out.println("matchVector is " +
matchVector);
                                    System.out.println("first lolo: " +
matchVector.get(matchVector.indexOf("loloandnono") + 2));
```

```
                                    }

                                }
                                if (usrEntry.size() > 0) {
                                    hourList.add(usrEntry);
                                }
                                break;
                            }
                        }

                        rs.close();
                        stmt.close();
                        System.out.println("In total, " + hourList.size() + " users
have meetings");
                        System.out.println();

                    } catch (Exception e) {
                        System.out.print(e);
                        System.out.println(" No exsiting table found");
                    }
                    //import room coordinates information
                    roomCoordinateVector =
roomCoordinates.getRoomCoordinates();
                    //System.out.println("room coordinates
"+roomCoordinateVector);

                    //Vector contains all room names used for sending subscription of
each room occupancy context
                    Vector roomNameVector = new Vector(7);
                    roomNameVector.add(mint);
                    roomNameVector.add(grimeton);
                    roomNameVector.add(openarea);
                    roomNameVector.add(h 枚 rby);
                    roomNameVector.add(motala);

                    //int portNo = ds.getLocalPort();
                    int portNo = watcher.getPort();

                    //subscribe occupancy context
                    Iterator itr = roomNameVector.iterator();
```

```
                    while (itr.hasNext()) {

                        //String room = (String) roomNameVector.get(i);
                        String room = (String) itr.next();
                        String suboccupancy = watcher.getSUB(room,
contextoccupancy, portNo);
                        try {
                            watcher.send(suboccupancy);
                            System.out.println(room + " occupancy subscription has
been sent out");
                        } catch (IOException ex) {

Logger.getLogger(CallSecretary.class.getName()).log(Level.SEVERE, null, ex);
                        }
                    }
                    System.out.println("OK");
                    closeConnection();
                    break;
                }
            }
        }

    private static void makeConnection() throws SQLException,
ClassNotFoundException {
        try {

            String databaseName = "ser";
            String usrName = "context";
            String password = "adios";
            String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
            Class.forName("org.gjt.mm.mysql.Driver");
            //Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = DriverManager.getConnection(url, usrName, password);
        // } catch (java.lang.ClassNotFoundException e) {
        } catch (Exception e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }

    //Close database connection
    private static void closeConnection() {
```

```
            try {
                if (conn != null) {
                    conn.close();
                }
                conn = null;
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        public static void main(String args[]) throws Exception {
            //ds = new DatagramSocket();
            watcher = new ContextAgent();
            int portNo = watcher.getPort();
            System.out.println(portNo);

            System.out.println("About to schedule task.");
            //ICalTask iCalTask = new ICalTask();
            //CPLTask cplTask = new CPLTask();
            CallSecretary callSecretary = new CallSecretary();

            // Reading incoming message from server
            String packet = null;

            while (true) {
                packet = watcher.receive();
                System.out.println("\r\n" + "Received one message: ");
                // System.out.println(packet);
                if (packet != null) {
                    // parse SIP message and process it according to its category
                    int ifSIP = packet.indexOf("SIP/2.0");
                    if (ifSIP < 0) {
                        //this is not a SIP message.
                        System.out.println("This is not a SIP message.");
                    } else if (ifSIP == 0) {
                        // this is a 200/202 message or error message, just ignore it
                        String msg = packet.substring(packet.indexOf(" ") + 1,
packet.indexOf("\r\n"));
                        System.out.println("This is a " + msg + " message!" +
"\r\n");
                    } else {
                        //this is a notify message
```

```
                    String userName = null;
                    String nameTag = "sip:";
                    userName = packet.substring(packet.indexOf(nameTag) +
nameTag.length(), packet.indexOf("@"));

                    String event = null;
                    event = packet.substring(packet.indexOf("Event") +
"Event".length(),
                            packet.indexOf("\r\n", packet.indexOf("Event")));
                    String msgType = "Notify";

                    // examine Sbuscription State of incoming message

                    String subState = "Subscription-State: ";
                    int startState = packet.indexOf(subState) + subState.length();
                    int endState = packet.indexOf("\r\n", startState);
                    String recState = packet.substring(startState, endState);

                    //send OK message
                    String okMessage = watcher.getOK(packet);
                    watcher.send(okMessage);

                    if (recState.equals("terminated")) {
                        //Expired Notify message, and subscripe message
                        msgType = "expiredNotify";
                        String sub = watcher.getSUB(userName, event,
portNo);
                        watcher.send(sub);
                        System.out.println("\r\n" + msgType + " send Subscribe
message");
                    } else {
                        //This is an    Notify message with context information
                        msgType = "Notify";


                        //parsing packet for xml context
                        // Reading room name and occupancy number OR user's
location context
                        String occupContext = null;
                        String occupStart = "<value>";
                        String occupEnd = "</value>";
                        String roomStart = "<room>";
```

```
                        String roomEnd = "</room>";
                        String roomContext = null;
                        int occupIndex = packet.indexOf(occupStart);
                        String subscriberStart = "<subscriber>";
                        String subscriberEnd = "</subscriber>";
                        String floorStart = "<floor>";
                        String floorEnd = "</floor>";
                        String latitudeStart = "<latitude>";
                        String latitudeEnd = "</latitude>";
                        String longitudeStart = "<longitude>";
                        String longitudeEnd = "</longitude>";
                        int locIndex = packet.indexOf(latitudeStart);

                        if (occupIndex < 0 && locIndex < 0) {
                            //Notify message without xml context information
                        } else if (occupIndex > 0 && locIndex < 0) { // room
occupancy notification

                            occupContext =
packet.substring((packet.indexOf(occupStart) + occupStart.length()),
                                    (packet.indexOf(occupEnd)));
                            int occupContextInteger =
Integer.parseInt(occupContext); // convert string to int

                            roomContext =
packet.substring((packet.indexOf(roomStart) + roomStart.length()),
                                    (packet.indexOf(roomEnd)));
                            System.out.println(roomContext + " currently has "
+ occupContext);

                            // update the occupancy elements of matchVector
with the same meetingroom name
                            for (int x = 0; x < matchVector.size() / 8; x++) {
                                String locationRoomName = (String)
matchVector.get(x * 8 + 2);
                                System.out.println(locationRoomName);
                                System.out.println("examining room
occupancy status");
                                if
(locationRoomName.equalsIgnoreCase(roomContext)) {
                                    matchVector.set((x * 8 + 5),
occupContextInteger);
```

```
                                        System.out.println("meeting room's
occupancy updated " + matchVector);
                                    } else {
                                    }
                                }

                            } else if (occupIndex < 0 && locIndex > 0) {    //user
location notification

                                String subscriberValue =
packet.substring((packet.indexOf(subscriberStart) + subscriberStart.length()),
                                        (packet.indexOf(subscriberEnd)));
                                String floorValue =
packet.substring((packet.indexOf(floorStart) + floorStart.length()),
                                        (packet.indexOf(floorEnd)));
                                int floorInt = Integer.parseInt(floorValue);
                                String latitudeValue =
packet.substring((packet.indexOf(latitudeStart) + latitudeStart.length()),
                                        (packet.indexOf(latitudeEnd)));
                                String longitudeValue =
packet.substring((packet.indexOf(longitudeStart) + longitudeStart.length()),
                                        (packet.indexOf(longitudeEnd)));

                                //call LocationIndicator to calculate subscriber's
location

                                String roomNm =
roomCoordinates.getLocation(longitudeValue, latitudeValue, floorInt);
                                System.out.println("He is in " + roomNm +
"\r\n");

                                // update the occupancy elements of matchVector
with the same meetingroom name
                                for (int x = 0; x < matchVector.size() / 8; x++) {
                                    String name = (String) matchVector.get(x *
8);
                                    if (name.equals(subscriberValue)) {
                                        matchVector.set((x * 8 + 3), roomNm);
                                        //System.out.println(name + "'s location
updated " + matchVector);
                                    } else {
                                    }
```

```
            }

        } else {
            // break; // no location nor occupation context xml
        } } } } }}}
```

ContextAgent.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package callsecretary;

/**
 *
 * @author Ke Wang
 */
import java.io.*;
import java.net.*;
import java.lang.System.*;
import java.util.Random;
import java.sql.*;



public class ContextAgent {

    protected byte[] buf = new byte[1024];
    protected static DatagramSocket ds;
    protected static String stringServer = "130.237.15.238";
    InetAddress serverAddr = InetAddress.getByName(stringServer);
    protected static int port = 5060;
    //protected static int listenPort = 5072;
    protected InetSocketAddress socketAddress = null;
    protected InetAddress i = InetAddress.getLocalHost();
    protected String hostName = i.getHostName();
    protected String hostAddr = i.getHostAddress();
    protected static java.sql.Connection conn = null;


    public ContextAgent() throws Exception {
        //socketAddress = new InetSocketAddress(host, port);
        ds = new DatagramSocket();
        System.out.println("port number is "+ ds.getLocalPort());
        System.out.println("Call Agent is up and running...");
    }
```

```
public int getPort () {
        int lport = ds.getLocalPort();
        return lport;
}

public final void setSoTimeout(final int timeout) throws Exception {
        ds.setSoTimeout(timeout);
}


public final int getSoTimeout() throws Exception {
        return ds.getSoTimeout();
}

public final DatagramSocket getSocket() {
        return ds;
}

/**
  * Method for generating random cSeq value used in SIP Subscribe header
  * the value can be arbitrary less than 2**31
  * We use current time as random seed and limit the value from 1 to 99999
  */
protected static String getcSeq() {
        Random random = new Random(System.currentTimeMillis());
        int r = random.nextInt();
        int cSeqSeed = Math.abs(r % 99999);
        String cSeq = Integer.toString(cSeqSeed);
        return cSeq;
}

/** Generating random call-ID for SIP Subscribe header
  * We simply use the time based random number
  * plus another random number use Math.random()
  */
protected static String getCallID() {
        Random rd = new Random(System.currentTimeMillis());
        int r = rd.nextInt();
        int c = Math.abs(r % 88888);
        double j = Math.random() * 100;
        int call-IDSeed = (int) j + c;
        String call-ID = Integer.toString(call-IDSeed);
```

```
        return call-ID;
}

/**
  * Method for generating random value for CallID,branch,and Tag
  * used in SIP header
  */
protected static String getRandomNum(int randomLen) {

        final int maxNum = 61;
        int i;

        int count = 0;

        char[] str = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
                'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
        };

        StringBuffer random = new StringBuffer("");
        Random r = new Random();
        while (count < randomLen) {


                i = Math.abs(r.nextInt(maxNum));

                if (i >= 0 && i < str.length) {
                        random.append(str[i]);
                        count++;
                }
        }

        return random.toString();
}


/**
  *Method of sending message to sepicific server
  */
public final DatagramPacket send(String sendOut)
```

```
                throws IOException {
        byte[] bytes = sendOut.getBytes();
        DatagramPacket dp = new DatagramPacket(bytes, bytes.length, serverAddr,
port);
        ds.send(dp);
        return dp;
    }

    /**
     * Method of receiving message from server
     */
    protected final String receive()
                throws Exception {
        DatagramPacket dp = new DatagramPacket(buf, buf.length);
        ds.receive(dp);
        String info = new String(dp.getData(), 0, dp.getLength());
        return info;
    }

    /**
     * Close UDP socket.
     */
    protected final void close() {
        try {
            ds.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Method of sending Subscribe message
     */
    protected String getSUB(String userName, String context, int lport) {
        String cSeq = ContextAgent.getcSeq();
        String call-ID = ContextAgent.getCallID();
        String branch = ContextAgent.getRandomNum(7);
        String tag = ContextAgent.getRandomNum(4);

        String context_type = context;
        String subHeader = "";
        subHeader = "SUBSCRIBE " + "sip:" + userName + "@" + stringServer +
```

```
                    " SIP/2.0" + "\r\n" +
                    "Via: " + "SIP/2.0/UDP " + stringServer +
                    ":" + lport + ";branch=z9hG4bK" + branch + "\r\n" +
                    "From: <sip:" + hostName + "@" + stringServer +
                    ">;tag=" + tag + "\r\n" +
                    "To: <sip:" + userName + "@" + stringServer + ">" + "\r\n" +
                    "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                    "CSeq: " + cSeq + " SUBSCRIBE" + "\r\n" +
                    "Max-Forwards: 70" + "\r\n" +
                    "Event: " + context_type + "\r\n" +
                    "Accept: application/pidf+xml" + "\r\n" +
                    "Contact: <sip:" + hostName + "@" + hostAddr + ":" + lport +
">" + "\r\n" +
                    "Expires: 600" + "\r\n" +
                    "Content-Length: 0" + "\r\n\r\n";
        return subHeader;
    }


    /**
     * Method of sending Register message to upload CPL script
     */
    protected String getRegCPL(String userName, String context,int lport, String
cplVoicemail) {
        String context_type = context;
        String cSeq = ContextAgent.getcSeq();
        String call-ID = ContextAgent.getCallID();
        String branch = ContextAgent.getRandomNum(7);
        String tag = ContextAgent.getRandomNum(4);
        String cplScript =
                    "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                    "<!DOCTYPE cpl PUBLIC '-//IETF//DTD RFCxxxx CPL
1.0//EN' 'cpl.dtd'>" + "\r\n" +
                    "<cpl>" + "\r\n"    +
                   "<incoming>" + "\r\n"    +
                   "<location url=\"sip:"+ cplVoicemail+"\">"    + "\r\n"    +
                   "<redirect permanent=\"yes\" />" + "\r\n"    +
                   "</location>" + "\r\n"    +
                   "</incoming>" + "\r\n"    +
                   "</cpl>" + "\r\n\r\n" ;

        String regHeader = "";
```

```
        regHeader = "REGISTER " + "sip:" + stringServer +
                    " SIP/2.0" + "\r\n" +
                    "Via: " + "SIP/2.0/UDP " + stringServer +
                    ":" + lport + ";branch=z9hG4bK" + branch + "\r\n" +
                    "From: <sip:" + userName + "@" + stringServer +
                    ">;tag=" + tag + "\r\n" +
                    "To: <sip:" + userName + "@" + stringServer + ">" + "\r\n" +
                    "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                    "CSeq: " + cSeq + " REGISTER" + "\r\n" +
                    "Accept: application/cpl, application/sdp, text/html" + "\r\n" +
                    "Contact: <sip:" + hostName + "@" + hostAddr + ":" + lport +
">" + "\r\n" +
                    "Content-Type: "+ context_type + "\r\n" +
                    "Content-Length: " + cplScript.length() + "\r\n" +
                    cplScript;
        return regHeader;
    }

    /**
     * Method of sending Register message without body to remove CPL script
     */
    protected String getRemoveCPL(String userName, String context,int lport) {
        String context_type = context;
        String cSeq = ContextAgent.getcSeq();
        String call-ID = ContextAgent.getCallID();
        String branch = ContextAgent.getRandomNum(7);
        String tag = ContextAgent.getRandomNum(4);

        String regHeader = "";

        regHeader = "REGISTER " + "sip:" + stringServer +
                    " SIP/2.0" + "\r\n" +
                    "Via: " + "SIP/2.0/UDP " + stringServer +
                    ":" + lport + ";branch=z9hG4bK" + branch + "\r\n" +
                    "From: <sip:" + userName + "@" + stringServer +
                    ">;tag=" + tag + "\r\n" +
                    "To: <sip:" + userName + "@" + stringServer + ">" + "\r\n" +
                    "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                    "CSeq: " + cSeq + " REGISTER" + "\r\n" +
                    "Accept: application/cpl, application/sdp, text/html" + "\r\n" +
                    "Contact: <sip:" + hostName + "@" + hostAddr + ":" + lport +
```

```
">" + "\r\n" +
                        "Content-Type: "+ context_type + "\r\n" +
                        "Content-Length: 0" + "\r\n\r\n";

            return regHeader;
        }


        /**
        * Method of make up 200 OK message according to SIP message received from
SER
        */
        protected String getOK(String packet) {
            String okHeader = "";
            String str[] = new String[16];
            str = packet.split("\n");
            String receivedAddr=
packet.substring(packet.indexOf("@"),packet.indexOf(" ", packet.indexOf("@")));
            okHeader = "SIP/2.0 200 OK"+ "\r\n" +
                        str[2] + ";received="+ receivedAddr + "\r\n" +
                        str[3] + "\r\n" +
                        str[4] + "\r\n" +
                        str[5] + "\r\n" +
                        str[6] + "\r\n" +
                        str[7] + "\r\n" +
                        "Content-Length: 0";
            return okHeader;
        }


        //load database driver and make a connection
        private static void makeConnection() throws SQLException,
ClassNotFoundException {
            try {
                String databaseName = "ser";
                String usrName = "context";
                String password = "adios";
                String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
                Class.forName("org.gjt.mm.mysql.Driver").newInstance();
                //Class.forName("com.mysql.jdbc.Driver").newInstance();
                conn = DriverManager.getConnection(url, usrName, password);
            // } catch (java.lang.ClassNotFoundException e) {
```

```
            } catch (Exception e) {
                System.err.print("ClassNotFoundException: ");
                System.err.println(e.getMessage());
            }
        }
        //Close database connection
        protected static void closeConnection() {
            try {
                if (conn != null) {
                    conn.close();
                }
                conn = null;
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
}
```

LocationIndicator.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package callsecretary;

import java.sql.*;
import java.util.Vector;
import javax.swing.JOptionPane;

/**
 *
 * @author Ke Wang
 */
public class LocationIndicator {
    //Database connection method

    private static java.sql.Connection conn = null;
    static String roomName = null;
    Vector vc = null;

    public void LocationIndicator() {
    }

    // Database connection
    protected static void makeConnection() throws SQLException,
ClassNotFoundException {
        try {
            String databaseName = "ser";
            String usrName = "context";
            String password = "adios";
            String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
            //Class.forName("org.gjt.mm.mysql.Driver");
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = DriverManager.getConnection(url, usrName, password);
        // } catch (java.lang.ClassNotFoundException e) {
        } catch (Exception e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }
```

```
	//Close database connection
	protected static void closeConnection() {
		try {
			if (conn != null) {
				conn.close();
			}
			conn = null;
		} catch (Exception ex) {
			ex.printStackTrace();
		}
	}


	// B(longitude 17.56.56.34 , latitude 59.24.19.47)

	protected Float getXProjection(String longit) {
		 String longitude = longit;
		float fLongt = (float) ((Float.valueOf(longitude) - 17565634)*0.01*
15.738);
		return fLongt;
	}

	protected Float getYProjection(String lat) {
		String latitude = lat;
		float fLat = (float)(((Float.valueOf(latitude)) - 59241947) *0.01* 30.945);
		return fLat;
	}

	protected Vector getRoomCoordinates () {
		vc =new Vector();
		String meetingRoom = null;
		String sqlQuery = "select roomname, leftside, rightside, top, bottom,floor
from coordinates";
		try {
			makeConnection();
			ResultSet rs;
			Statement stmt;
			stmt = conn.createStatement();
			rs = stmt.executeQuery(sqlQuery);
			ResultSetMetaData rsMetaData = rs.getMetaData();
			int columnNumber = rsMetaData.getColumnCount();
```

```
                while (rs.next()) {
                    meetingRoom = rs.getString(1);
                    float left = rs.getFloat(2);
                    float right = rs.getFloat(3);
                    float top = rs.getFloat(4);
                    float bottom = rs.getFloat(5);
                    int floor = rs.getInt(6);
                    vc.add(meetingRoom);
                    vc.add(left);
                    vc.add(right);
                    vc.add(top);
                    vc.add(bottom);
                    vc.add(floor);
                }

                rs.close();
                stmt.close();

            } catch (Exception e) {
                System.out.print(e);
                System.out.println(" No exsiting table found");
            }
            return vc;
    }

    protected String getLocation (String lo, String la, int floorValue) {
        float lon=this.getXProjection(lo);
        float lat=this.getYProjection(la);

        //System.out.println ("longitude = " + lon);
        //System.out.println ("latitude = " + lat);
       // System.out.println ("angel = "+ Math.atan2(lat, lon)*(180/Math.PI));
        //System.out.println ("cos = "+ Math.cos((-33.847+(Math.atan2(lat,
lon)*(180/Math.PI)))*Math.PI/180));
        //System.out.println ("sin = "+ Math.sin((-33.847+(Math.atan2(lat,
lon)*(180/Math.PI)))*Math.PI/180));


        float buildingX = (float) (Math.sqrt((lat*lat)+(lon*lon)) *
Math.cos((-33.847+(Math.atan2(lat, lon)*(180/Math.PI)))*Math.PI/180) + 30);
        float buildingY= (float) (Math.sqrt((lat*lat)+(lon*lon)) *
Math.sin((-33.847+(Math.atan2(lat, lon)*(180/Math.PI)))*Math.PI/180) + 30);
```

```
            //System.out.println ("X value = "+buildingX);
            //System.out.println ("Y value = " + buildingY);
            String nm=null;
            for (int i=0;i<vc.size()/6;i++){
                  nm= (String) vc.get(i*6);
                  float lt=(Float)vc.get(i*6+1);
                  float rt=(Float)vc.get(i*6+2);
                  float tp=(Float)vc.get(i*6+3);
                  float bm=(Float)vc.get(i*6+4);
                  int fl=(Integer)vc.get(i*6+5);

             while
(Float.compare(buildingX,lt)>=0&&Float.compare(rt,buildingX)>=0&&Float.compa
re(buildingY,bm)>=0&&Float.compare(tp,buildingY)>=0 && floorValue==fl)

             {
             roomName = nm;
             //System.out.println ("subscriber is in " + roomName);
             break;
             }

            }
                  return roomName;
      }

}
```

MeetingInfo.java

```
package callsecretary;

import com.google.gdata.data.DateTime;

/**
* @author Ke Wang
  */
public class MeetingInfo {

	DateTime startTime = null;
	DateTime endTime = null;
	String roomName = null;
	String meetingSize = null;
	public MeetingInfo() {
	}

	public void setMeetingValues(DateTime start, DateTime end, String room,
String size) {
		this.startTime = start;
		this.endTime = end;
		this.roomName = room;
		this.meetingSize = size;
	}

	public String getRoomName() {
		return this.roomName;
	}

	public DateTime getStartTime() {
		return this.startTime;
	}

	public DateTime getEndTime() {
		return this.endTime;
	}

	public String getMeetingSize(){
		return this.meetingSize;
	}
}
```

MeetingList.java

```
package callsecretary;

import java.util.ArrayList;

/**
 *
 * @author Ke Wang
 */
public class MeetingList {

      ArrayList meetingList;

      protected MeetingList () {
            meetingList = new ArrayList();
      }

      protected void addMeeting(int index, MeetingInfo meetingInfo) {
         meetingList.add(index, meetingInfo);
      }

      protected int getSize() {
        int size = meetingList.size();
            return size;
      }

        protected MeetingInfo getMeeting(int i) {
            MeetingInfo meeting = new MeetingInfo();
            Object object = meetingList.get(i);
            meeting = (MeetingInfo) object;
            return meeting;
      }
        protected void removeMeeting (int i){
          meetingList.remove(i);

        }


}
```

```
/**String ss;
String ee;
String rr;
String meetings;
ArrayList ll = new ArrayList();

public void setValues(String start, String end, String room) {
ss = start;
ee= end;
rr= room;
}

public ArrayList getVaules() {
ll.add(ss);
ll.add(ee);
ll.add(rr);

return ll;
}
public String getRoomName() {
return rr;
}
public String getStartTime() {
return ss;
}
public static void main(String args[]) {
String sT = "1 ";
String eT = "2 ";
String rN = "MINI";
ArrayList meetingList = new ArrayList ();

MeetingList list = new MeetingList();
list.setValues(sT, eT, rN);

meetingList = list.getVaules();
System.out.println(meetingList.get(new Integer (1)));
}}
 **/
```

Appendix G: Notify Sender source code

The Java source code of Notify Sender

```
package notifysender;

/**
 *
 * @author Ke Wang
 */
import java.io.*;
import java.net.*;
import java.lang.System.*;
import java.util.Random;
import java.sql.*;
import javax.swing.JOptionPane;

public class NotifySender {

    private byte[] buf = new byte[1024];
    private static DatagramSocket ds = null;
    //private static String stringServer = null;
    private static String stringServer = "130.237.15.227";
    InetAddress serverAddr = InetAddress.getByName(stringServer);
    private static String serverName = "hlllab4";
    private static int listenPort = 5092;
    private InetSocketAddress socketAddress = null;
    private InetAddress i = InetAddress.getLocalHost();
    private String hostName = i.getHostName();
    //private String hostAddr = i.getHostAddress();
    private String hostAddr = "130.237.15.238";
    private static java.sql.Connection conn = null;

    /**
     * UDP socket
     * @throws Exception
```

```
 */
public NotifySender(String lhost, int lport) throws Exception {
   //socketAddress = new InetSocketAddress(lhost, lport);
   ds = new DatagramSocket(lport);
   System.out.println("Call Agent is up and running...");
}

public final void setSoTimeout(final int timeout) throws Exception {
   ds.setSoTimeout(timeout);
}

public final int getSoTimeout() throws Exception {
   return ds.getSoTimeout();
}

public final DatagramSocket getSocket() {
   return ds;
}

/**
 * Method for generating random cSeq value used in SIP Subscribe header
 * the value can be arbitrary less than 2**31
 * We use current time as random seed and limit the value from 1 to 99999
 */
private static String getcSeq() {
   Random random = new Random(System.currentTimeMillis());
   int r = random.nextInt();
   int cSeqSeed = Math.abs(r % 99999);
   String cSeq = Integer.toString(cSeqSeed);
   return cSeq;
}

/** Generating random call-ID for SIP Subscribe header
 * We simply use the time based random number
 * plus another random number use Math.random()
 */
private static String getCallID() {
   Random rd = new Random(System.currentTimeMillis());
   int r = rd.nextInt();
   int c = Math.abs(r % 88888);
   double j = Math.random() * 100;
   int call-IDSeed = (int) j + c;
```

```
    String call-ID = Integer.toString(call-IDSeed);
    return call-ID;
}

/**
 * Method for generating random value for CallID,branch,and Tag
 * used in SIP header
 */
private static String getRandomNum(int randomLen) {

    final int maxNum = 61;
    int i;

    int count = 0;

    char[] str = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
        'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
        'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
    };

    StringBuffer random = new StringBuffer("");
    Random r = new Random();
    while (count < randomLen) {

        i = Math.abs(r.nextInt(maxNum));

        if (i >= 0 && i < str.length) {
            random.append(str[i]);
            count++;
        }
    }

    return random.toString();
}

/**
 *Method of sending message to sepicific server
 * @throws IOException
 */
public final DatagramPacket send(String sendOut, int lport)
```

```
            throws IOException {
        byte[] bytes = sendOut.getBytes();
        DatagramPacket dp = new DatagramPacket(bytes, bytes.length, serverAddr,
lport);
        ds.send(dp);
        return dp;
    }

    /**
     * Method of receiving message from server
     * @throws Exception
     */
    private final String receive()
            throws Exception {
        DatagramPacket dp = new DatagramPacket(buf, buf.length);
        ds.receive(dp);
        String info = new String(dp.getData(), 0, dp.getLength());
        return info;
    }

    /**
     * close UDP socket
     */
    private final void close() {
        try {
            ds.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Method of sending Subscribe message
     */
    private String getSUB(String cSeq, String call-ID, String branch, String tag) {
        /**
         * Generating parameters for SIP Subscribe message
         * of Room Occupation information
         */
        String context_type = "presence";
        String subHeader = "";
```

```
        subHeader = "SUBSCRIBE " + "sip:ke@" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + hostAddr +
                ":" + listenPort + ";branch=z9hG4bK" + branch + "\r\n" +
                "From: <sip:" + hostName + "@" + stringServer +
                ">;tag=" + tag + "\r\n" +
                "To: <sip:ke@" + stringServer + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                "CSeq: " + cSeq + " SUBSCRIBE" + "\r\n" +
                "Max-Forwards: 70" + "\r\n" +
                "Event: " + context_type + "\r\n" +
                "Accept: application/pidf+xml" + "\r\n" +
                "Contact: <sip:" + hostName + "@" + hostAddr + ":" + listenPort + ">" +
"\r\n" +
                "Expires: 600" + "\r\n" +
                "Content-Length: 0" + "\r\n\r\n";
        return subHeader;
    }

    /**
     * Method of sending out fake Notify message
     */
    private String getNotifyContext(String call-ID, String roomNM, String occup) {
        String context_type = "occupancy";
        String notifyHeader = "";
        String available = "active;expires=123";
        String expire = "";
        notifyHeader = "NOTIFY " + "sip:" + serverName + "@" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + hostAddr +
                ";branch=z9hG4bKKXhQqrV" + "\r\n" +
                "From: <sip:" + hostName + "@" + hostAddr +
                ">;tag=xlB3" + "\r\n" +
                "To: <sip:" + serverName + "@" + hostAddr + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                "CSeq: " + "1" + " NOTIFY" + "\r\n" +
                "Event: " + context_type + "\r\n" +
                "Content-Type: application/pidf+xml" + "\r\n" +
                "Contact: <sip:" + hostAddr + ":" + listenPort + ">" + "\r\n" +
                "Subscription-State: " + available + "\r\n" +
                "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                        "<presence                xmlns=\"urn:ietf:params:xml:ns:pidf\"
```

```
entity=\"pres:ccsleft@130.237.15.238\">" + "\r\n" +
            "<tuple id=\"0xb58d60e0x4a4b0c39x4715c26e\">" + "\r\n" +
            "<status><basic>open</basic>" + "\r\n" +
            "<occupancy>" + "\r\n" +
            "<description>Electrum</description>" + "\r\n" +
            "<room>" + roomNM + "</room>" + "\r\n" +
            "<value>" + occup + "</value>" + "\r\n" +
            "</occupancy>" + "\r\n" +
            "</status>" + "\r\n" +
            "<contact priority=\"0.80\">KeWang</contact>" + "\r\n" +
            "<note>occupancy</note>" + "\r\n" +
            "</tuple>" + "\r\n" +
            "</presence>" + "\r\n\r\n";
        return notifyHeader;
    }

    private String getNotifyNoContext(String call-ID) {
        String context_type = "occupancy";
        String notifyHeader = "";
        String available = "active;expires=123";
        String expire = "";
        notifyHeader = "NOTIFY " + "sip:" + serverName + "@" + stringServer +
            " SIP/2.0" + "\r\n" +
            "Via: " + "SIP/2.0/UDP " + hostAddr +
            ";branch=z9hG4bKKXhQqrV" + "\r\n" +
            "From: <sip:" + hostName + "@" + hostAddr +
            ">;tag=xlB3" + "\r\n" +
            "To: <sip:" + serverName + "@" + hostAddr + ">" + "\r\n" +
            "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
            "CSeq: " + "1" + " NOTIFY" + "\r\n" +
            "Event: " + context_type + "\r\n" +
            "Content-Type: application/pidf+xml" + "\r\n" +
            "Contact: <sip:" + hostAddr + ":" + listenPort + ">" + "\r\n" +
            "Subscription-State: " + available + "\r\n" +
            "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                        "<presence                xmlns=\"urn:ietf:params:xml:ns:pidf\"
entity=\"pres:ccsleft@130.237.15.238\">" + "\r\n" +
            "<tuple id=\"none\"" + "\r\n" +
            "<status><basic>closed</basic></status>" + "\r\n" +
            "</tuple>" + "\r\n" +
            "</presence>" + "\r\n\r\n";
```

```
        return notifyHeader;
    }

    private String getNotifyIrre(String roomNM, String occup) {
        String context_type = "occupancy";
        String notifyHeader = "";
        String available = "active;expires=123";
        String expire = "";
        String call-ID = "007";
        notifyHeader = "NOTIFY " + "sip:" + serverName + "@" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + hostAddr +
                ";branch=z9hG4bKKXhQqrV" + "\r\n" +
                "From: <sip:" + hostName + "@" + hostAddr +
                ">;tag=xlB3" + "\r\n" +
                "To: <sip:" + serverName + "@" + hostAddr + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                "CSeq: " + "1" + " NOTIFY" + "\r\n" +
                "Event: " + context_type + "\r\n" +
                "Content-Type: application/pidf+xml" + "\r\n" +
                "Contact: <sip:" + hostAddr + ":" + listenPort + ">" + "\r\n" +
                "Subscription-State: " + available + "\r\n" +
                "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                        "<presence xmlns=\"urn:ietf:params:xml:ns:pidf\"
entity=\"pres:ccsleft@130.237.15.238\">" + "\r\n" +
                "<tuple id=\"0xb58d60e0x4a4b0c39x4715c26e\">" + "\r\n" +
                "<status><basic>open</basic>" + "\r\n" +
                "<occupancy>" + "\r\n" +
                "<description>Electrum</description>" + "\r\n" +
                "<room>" + roomNM + "</room>" + "\r\n" +
                "<value>" + occup + "</value>" + "\r\n" +
                "</occupancy>" + "\r\n" +
                "</status>" + "\r\n" +
                "<contact priority=\"0.80\">KeWang</contact>" + "\r\n" +
                "<note>occupancy</note>" + "\r\n" +
                "</tuple>" + "\r\n" +
                "</presence>" + "\r\n\r\n";
        return notifyHeader;
    }

    private String getNotifyExpired(String call-ID) {
        String context_type = "occupancy";
```

```
        String notifyHeader = "";
        String available = "terminated";
        String expire = "";
        notifyHeader = "NOTIFY " + "sip:" + serverName + "@" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + hostAddr +
                ";branch=z9hG4bKKXhQqrV" + "\r\n" +
                "From: <sip:" + hostName + "@" + hostAddr +
                ">;tag=xlB3" + "\r\n" +
                "To: <sip:" + serverName + "@" + hostAddr + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                "CSeq: " + "1" + " NOTIFY" + "\r\n" +
                "Event: " + context_type + "\r\n" +
                "Content-Type: application/pidf+xml" + "\r\n" +
                "Contact: <sip:" + hostAddr + ":" + listenPort + ">" + "\r\n" +
                "Subscription-State: " + available + "\r\n" +
                "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                        "<presence xmlns=\"urn:ietf:params:xml:ns:pidf\"
entity=\"pres:ccsleft@130.237.15.238\">" + "\r\n" +
                "<tuple id=\"none\"" + "\r\n" +
                "<status><basic>closed</basic></status>" + "\r\n" +
                "</tuple>" + "\r\n" +
                "</presence>" + "\r\n\r\n";
        return notifyHeader;
    }

    private String getLoc(String call-ID, String subscriber, String floor, String lat,
String lon) {
        String context_type = "location";
        String notifyHeader = "";
        String available = "active;expires=123";
        String expire = "";
        notifyHeader = "NOTIFY " + "sip:" + serverName + "@" + stringServer +
                " SIP/2.0" + "\r\n" +
                "Via: " + "SIP/2.0/UDP " + hostAddr +
                ";branch=z9hG4bKKXhQqrV" + "\r\n" +
                "From: <sip:" + hostName + "@" + hostAddr +
                ">;tag=xlB3" + "\r\n" +
                "To: <sip:" + serverName + "@" + hostAddr + ">" + "\r\n" +
                "Call-ID: " + call-ID + "@" + hostAddr + "\r\n" +
                "CSeq: " + "1" + " NOTIFY" + "\r\n" +
                "Event: " + context_type + "\r\n" +
```

```
            "Content-Type: application/pidf+xml" + "\r\n" +
            "Contact: <sip:" + hostAddr + ":" + listenPort + ">" + "\r\n" +
            "Subscription-State: " + available + "\r\n" +
            "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + "\r\n" +
                        "<presence xmlns=\"urn:ietf:params:xml:ns:pidf\"
entity=\"pres:ccsleft@130.237.15.238\">" + "\r\n" +
            "<tuple id=\"0xb65d60e0x4a4b0c39x4715c26e\">" + "\r\n" +
            "<status><basic>open</basic>" + "\r\n" +
            "<location>" + "\r\n" +
            "<description>Electrum</description>" + "\r\n" +
            "<subscriber>" + subscriber + "</subscriber>" + "\r\n" +
            "<floor>" +floor+"</floor>" + "\r\n" +
            "<coordinates>" + "\r\n" +
            "<latitude>" + lat + "</latitude>" + "\r\n" +
            "<longitude>" + lon + "</longitude>" + "\r\n" +
            "</coordinates>" + "\r\n" +
            "</location>" + "\r\n" +
            "</status>" + "\r\n" +
            "<contact priority=\"0.80\">KeWang</contact>" + "\r\n" +
            "<note>location</note>" + "\r\n" +
            "</tuple>" + "\r\n" +
            "</presence>" + "\r\n\r\n";
        return notifyHeader;
    }

    //load Database driver and make a connection
        private static void makeConnection() throws SQLException,
ClassNotFoundException {
        try {
            String databaseName = "mrbs";
            String usrName = "context";
            String password = "adios";
            String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(url, usrName, password);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }

    private static void closeConnection() {
```

```
        try {
            if (conn != null) {
                conn.close();
            }
            conn = null;
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Main
     * @param args
     * @throws Exception
     * Created date: 22/06/2008
     */
    public static void main(String[] args) throws Exception {
        InetAddress i = InetAddress.getLocalHost();
        String hostName = i.getHostName();
        String hostAddr = i.getHostAddress();
        String packet = null;

        String strPort = JOptionPane.showInputDialog("Enter port number:");
        int port = new Integer(strPort);
        String call-ID = JOptionPane.showInputDialog("Enter call-ID value:");

        // Constructing subscribe and ok message, and sending subscribe message to
server
        NotifySender watcher = new NotifySender(hostAddr, listenPort);

        String type = JOptionPane.showInputDialog("Choose a type of Notify message
(o/l):");
        if (type.equals("o")) {
            // user chose to send occupancy notify msg
            String roomName = JOptionPane.showInputDialog("Enter a meeting room
name:");
            String occup = JOptionPane.showInputDialog("Enter participants number:");

            String number = JOptionPane.showInputDialog("How many message to send
(1/4):");
            if (number.equals("4")) {
                String notify = watcher.getNotifyContext(call-ID, roomName, occup);
```

```
            watcher.send(notify, port);
            System.out.println("Notify message with context " + "\r\n" + notify +
"\r\n");

            notify = watcher.getNotifyNoContext(call-ID);
            watcher.send(notify, port);
            System.out.println("Notify message no context " + "\r\n" + notify + "\r\n");

            notify = watcher.getNotifyIrre(roomName, occup);
            watcher.send(notify, port);
            System.out.println("Notify message with wrong call-ID   " + "\r\n" + notify
+ "\r\n");

            notify = watcher.getNotifyExpired(call-ID);
            watcher.send(notify, port);
            System.out.println("Notify message with expired notification" + "\r\n" +
notify + "\r\n");

          } else {

            String notify = watcher.getNotifyContext(call-ID, roomName, occup);
            watcher.send(notify, port);
            System.out.println("Notify message with context " + "\r\n" + notify +
"\r\n");
          }

       } else if (type.equals("l")) {
          //user chose to send location notify msg
           String subscriber = JOptionPane.showInputDialog("Enter subscriber's
name:");
          String floor =   JOptionPane.showInputDialog("Enter floor value 3/4:");
          String latitude = JOptionPane.showInputDialog("Enter latitude value:");
          String longitude = JOptionPane.showInputDialog("Enter longitude value:");
          String notify = watcher.getLoc(call-ID, subscriber, floor, latitude,longitude);
           watcher.send(notify, port);
           System.out.println("Location Notify message is sent"+ "\r\n" + notify +
"\r\n");
          }



       // Reading incoming message from server
```

```
    while (true) {
        packet = watcher.receive();
        System.out.println("\r\n" + "Received one message: ");
        System.out.println(packet);
    }
}}
```

Appendix H: Location Indicator source code

The Java source code of Location Indicator

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package locaitonindicator;

import java.sql.*;
import java.util.Vector;
import javax.swing.JOptionPane;

/**
 *
 * @author Ke Wang
 */
public class LocationIndicator {
    //Database connection method

    private static java.sql.Connection conn = null;
    static String roomName = null;
    Vector vc = null;

    public void LocationIndicator() {
    }

    // Database connection
    protected static void makeConnection() throws SQLException,
ClassNotFoundException {
        try {
            String databaseName = "ser";
            String usrName = "context";
            String password = "adios";
            String url = "jdbc:mysql://130.237.15.238:3306/" + databaseName;
            //Class.forName("org.gjt.mm.mysql.Driver");
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            conn = DriverManager.getConnection(url, usrName, password);
        // } catch (java.lang.ClassNotFoundException e) {
        } catch (Exception e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
```

```
        }
    }

    //Close database connection
    protected static void closeConnection() {
        try {
            if (conn != null) {
                conn.close();
            }
            conn = null;
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    // B(longitude 17.56.56.34 , latitude 59.24.19.47)

    protected Float getXProjection(String longit) {
         String longitude = longit;
        float fLongt = (float) ((Float.valueOf(longitude) - 17565634)*0.01*
15.738);
        return fLongt;
    }

    protected Float getYProjection(String lat) {
        String latitude = lat;
        float fLat = (float)(((Float.valueOf(latitude)) - 59241947) *0.01* 30.945);
        return fLat;
    }

    protected Vector getRoomCoordinates () {
        vc =new Vector();
        String meetingRoom = null;
        String sqlQuery = "select roomname, leftside, rightside, top, bottom,floor
from coordinates";
        try {
            makeConnection();
            ResultSet rs;
            Statement stmt;
            stmt = conn.createStatement();
            rs = stmt.executeQuery(sqlQuery);
```

```
                ResultSetMetaData rsMetaData = rs.getMetaData();
                int columnNumber = rsMetaData.getColumnCount();
                while (rs.next()) {
                    meetingRoom = rs.getString(1);
                    float left = rs.getFloat(2);
                    float right = rs.getFloat(3);
                    float top = rs.getFloat(4);
                    float bottom = rs.getFloat(5);
                    int floor = rs.getInt(6);
                    vc.add(meetingRoom);
                    vc.add(left);
                    vc.add(right);
                    vc.add(top);
                    vc.add(bottom);
                    vc.add(floor);
                }

                rs.close();
                stmt.close();

            } catch (Exception e) {
                System.out.print(e);
                System.out.println(" No exsiting table found");
            }
            return vc;
    }

    protected String getLocation (String lo, String la, int floorValue) {
        float lon=this.getXProjection(lo);
        float lat=this.getYProjection(la);

        //System.out.println ("longitude = " + lon);
        //System.out.println ("latitude = " + lat);
       // System.out.println ("angel = "+ Math.atan2(lat, lon)*(180/Math.PI));
        //System.out.println ("cos = "+ Math.cos((-33.847+(Math.atan2(lat,
lon)*(180/Math.PI)))*Math.PI/180));
        //System.out.println ("sin = "+ Math.sin((-33.847+(Math.atan2(lat,
lon)*(180/Math.PI)))*Math.PI/180));


        float buildingX = (float) (Math.sqrt((lat*lat)+(lon*lon)) *
Math.cos((-33.847+(Math.atan2(lat, lon)*(180/Math.PI)))*Math.PI/180) + 30);
```

```
        float buildingY= (float) (Math.sqrt((lat*lat)+(lon*lon)) *
Math.sin((-33.847+(Math.atan2(lat, lon)*(180/Math.PI)))*Math.PI/180) + 30);

        //System.out.println ("X value = "+buildingX);
        //System.out.println ("Y value = " + buildingY);
        String nm=null;
        for (int i=0;i<vc.size()/6;i++){
                nm= (String) vc.get(i*6);
                float lt=(Float)vc.get(i*6+1);
                float rt=(Float)vc.get(i*6+2);
                float tp=(Float)vc.get(i*6+3);
                float bm=(Float)vc.get(i*6+4);
                int fl=(Integer)vc.get(i*6+5);

         while
(Float.compare(buildingX,lt)>=0&&Float.compare(rt,buildingX)>=0&&Float.compa
re(buildingY,bm)>=0&&Float.compare(tp,buildingY)>=0 && floorValue==fl)

         {
         roomName = nm;
         //System.out.println ("subscriber is in " + roomName);
         break;
         }

        }
              return roomName;
    }


public static void main(String[] args) throws Exception {

LocationIndicator li = new LocationIndicator();
String lo = JOptionPane.showInputDialog("Enter longitude (e.g.17565684): ");
String la = JOptionPane.showInputDialog("Enter longitude (e.g.59241916): ");
String n = JOptionPane.showInputDialog("Enter floor: ");
int floor = Integer.parseInt(n);

Vector vv= new Vector();
vv=li.getRoomCoordinates();
System.out.println(vv);
String roomNm= li.getLocation(lo,la,floor);
System.out.println("he is in " + roomNm);
```

```
roomNm = null;
}
}
```

Thanks for reading ☺

COS/CCS 2008-27

www.kth.se