NTNU – Trondheim
Norwegian University of
Science and Technology

Treasure Hunt Components

Charles Mawutor Adrah

Master of Telematics - Communication Networks and Networked
Submission date: June 2012
Supervisor: Rolv Bræk, ITEM
Co-supervisor: Fatima Urooj, ITEM
Frank Alexander Kraemer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Name of student: Charles Mawutor Adrah

The Treasure Hunt is a collaborative game for Android developed here at the institute. In this thesis, additional components and services for the game should be developed and integrated.

Assignment given: 23.01.2012

Supervisor: Professor Rolv Bræk, ITEM
Co-supervisor: Frank Alexander Kraemer, Ph.D., ITEM
Co-supervisor: Urooj Fatima, ITEM

Abstract

The development of distributed, reactive and collaborative services is quite challenging. Rapidly composing services for collaborative learning activities require some development methods and tools. This thesis presents an extension of the City Guide application, a platform that supports situated collaborative learning services developed by Surya Bahudar Kathayat in his PhD thesis: *On the Development of Situated Collaborative Services.*

The application was developed using the engineering method SPACE and its development tool Arctis. In the extension made, two new services, instant messaging and group chat, have be composed and integrated into the application. The two services have been identified as basic support services that are required for a true collaborative learning experience. By using the architecture employed in the City Guide application, the instant messaging and group chat services were developed as components that could be integrated with other components within the application. The results show the instant messaging and group chat service as standalone functionalities that handle their own message routing within the application and hence did not require the use of any other messaging protocols. The results also show that by using this architecture and with the necessary Arctis modifications, the City Guide application opens up for unexplored possibilities where new services can be rapidly developed and integrated.

Preface

This report documents the results of my work done in the course TTM4905 - Network and Services, Master Thesis, during the spring semester of 2012 as the final part of the Masters degree program in Telematics, from the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Professor Rolv Bræk for providing invaluable comments and suggestions during the report writing process. My sincere gratitude also goes to my co-supervisors Frank Alexander Kraemer and Urooj Fatima for their support, guidance and encouragement throughout my thesis work. Many thanks goes to Surya Bahudar Kathayatin for his assistance during the initial stages of my work.

I wish to thank my parents, Mr. and Mrs. Adrah for their continuous support and funding of my education. To my siblings, friends and love, I appreciate your support and encouragement throughout this period, thank you.

Charles Mawutor Adrah,
Trondheim, June 2012.

Contents

List of Figures

Acronyms

ADT Android Development Tools

API Application Programming Interface

DA Directory Agent

DHCP Dynamic Host Configuration Protocol

ESM External State Machine

FU Functional Unit

GUI Graphical User Interface

GPS Global Positioning System

IETF Internet Engineering Task Force

IM Instant Messaging

J2SE Java 2 Platform, Standard Edition

MDD Model Driven Development

OSGi Open Services Gateway initiative

PC Personal Computer

PDA Personal Digital Assistant

P2P Peer-to-Peer

RMI Remote Method Invocation

SA Service Agent

SCLS Situated Collaborative Learning Services

SDK Software Development Kit

SDP Service Discovery Protocol

SLM Salutation Manager

SLP Service Location Protocol

SMS Short Message Service

TCP/IP Transmission Control Protocol/ Internet Protocol

TM Transport Manager

UA User Agent

UI User Interface

UML Unified Modeling Language

UPnP Universal Plug and Play

URL Uniform Resource Locator

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Wi-Fi Wireless Fidelity

Chapter 1

Introduction

The influx of new and emerging technologies in mobile and ubiquitous computing has greatly influenced our everyday activities in different ways. This is also true with respect to learning within the student-teacher context. Traditionally, learning by students is characterized by sitting in a classroom setting or by reading books privately. In addition, class or group centered activities often focused on a specific task at a time with the students limited in their efforts to collaborate. With the advent of mobile and wireless technologies, learners take advantage of the resources and people that are both globally and locally available to them, allowing a more situated and contextualized learning experience [IC09]. Learning services provided in this manner are called *Situated Collaborative Learning Services* (SCLS).

The *City Guide* application is a platform that seeks to illustrate the SCLS domain. It was developed by Surya Bahadur Kathaya, PhD. With this application, students as members of a group go around the city and collaborate to learn. The students work together on collaborative tasks outside the classroom thereby contributing a fair share to their learning activities. The activities performed are based on learning objects which are situated and are accessed depending on a player's location and state. An application as this that incorporates collaborative learning has the potential of supporting both formal and informal learning since there is interaction among different actors (teachers and students), inside and outside the performance of a task. [JJH98] states key elements of situated and collaborative learning which are: *positive interdependence among the learners, interaction, individual accountability, interpersonal and social skills, situatedness, and group processing.*

Understanding and identifying stable domain concepts will enable us to know the kind of services that should be provided. [Kat12] identifies core domain entities of a city-wide collaborative learning as *users, social config-*

urations, learning objects, enablers and platforms. The services support the learning activities and typically crosscut several domain entities. A service may involve several domain objects and a domain object may participate in several services. The cross-cutting nature of collaborative services is illustrated in Fig. 1.1. [KB09] categorizes the domain services provided into two groups namely, basic and application specific services. Some basic services that have been identified include *location awareness, instant messaging (IM), group chat, document sharing and SMS.* Some application specific services also identified are *group positioning, interactive quiz, scoring, clues and configuration.* The services are distributed and reactive in nature hence their development is complex. To flexibly and rapidly develop these services presents a big and interesting challenge.

SPACE [Kra08], a method for the development of reactive systems and its supporting tool *Arctis* was used in composing some components and services for the City Guide application. Currently, services that have been developed for the platform are the *location service, interactive quiz service and group positioning service.*

In this thesis, we have identified two basic services namely *IM and Group Chat* and have sought out to design building blocks which will be integrated into the already existing platform. In composing these services, we have employed technologies such as the *Service Discovery Protocol* (SDP), which enables collaborating roles to seek out their complementary collaborating roles to perform services as well as *ActorRouter* which enables the distributed deployment of services and systems.

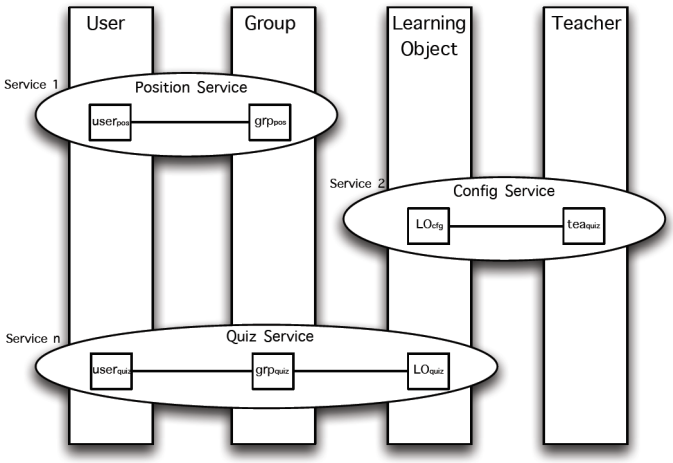


Figure 1.1: Cross-cutting nature of services [Kat12].

1.1 History of the City Guide Application

The City Guide application was originally developed as a Treasure Hunt game in TTM4115 as a term assignment in 2008. A reference to the assignment can found in [TT08]. The students used the RAMSES tool and ActorFrame and were able to simulate the game on Java clients using stationary terminals of desktop PCs and laptops.

The game was extended and modularized in the Fabula Project, still using RAMSES and ActorFrame by Surya Bahadur Kathaya. At this stage the game was demonstrable on Android clients. The need to reduce dependency on RAMSES led to the move of the Treasure Hunt game onto the Arctis platform. This was still carried out during the Fabula project. Challenges encountered during this period included handling sessions and distribution of the system which made this effort infeasible.

During the lab work of TTM3, a course on self-adapting systems [TT11] in 2011, a new platform developed using Arctis could handle sessions and distribution in systems with the introduction of new concepts on *proxies, proxy host and registry.*

A new attempt of the Treasure Hunt using Arctis and the TTM3 platform by Surya resulted in a working system called City Guide application and provides the basis of my work with details presented here.

1.2 Application Overview

City Guide Application is a collaborative game developed for students. Basically the students go around the city looking for "treasures", which could be a historical place, museum or a location in the city. When the game starts, each player moves around with a handheld terminal. On getting close to a treasure the player is interrupted with an interactive quiz task. Questions are asked about the treasure and it is the responsibility of the student to use information surrounding the treasure to answer these questions. On successfully answering all questions, the user then continues playing the game by exploring other parts of the city to discover more treasures.

To ensure the collaborative learning among the students, we have extended the application with new services:

- *Instant Messaging* enables players to contact and interact with other players on a peer-to-peer level.
- *Group Chat* enables all the players to have a common platform to interact at the same time.

1.2.1 Use Case Scenario

The following presents scenarios where the new services implemented are useful to the City Guide application:

Instant Messaging Urooj and Surya are both members of a group who are in different locations and are playing the game (City Guide application). Urooj is engaged in a question-answer task on her game after locating a treasure and is faced with a difficult question. She remembers a previous conversation with Surya pertaining to this question and so she decides to send a private message to Surya. She opens an IM window for Surya and sends him a message via texting. Surya who might be busy doing something else receives a notification on his device. He opens it and receives the message from Urooj. Both of them can then continue their conversation.

Group Chat Ephraim, Victor, Urooj and Surya are participants in the same group using the City Guide application for a group task. For them to work faster and more effectively, they agreed that they would cooperate in solving all questions together when any of them finds a treasure. Ephraim is the first to discover a treasure so he immediately initiates a group chat with the rest. All the group members receive his message and they start a group discussion and suggest ideas to one another about the questions and its solutions. They are able to work faster and learn from each other.

1.3 Design and Development Methodology

The set of methods deployed in developing the new services was based on the principles of model-driven development (MDD). MDD supports the software development process by creating models on different levels of abstraction and platform independence. Our aim was to develop abstract models which specified the pure functionality of services while hiding details of realization. The following models were used:

- UML sequence diagrams
- UML collaboration diagrams

1.3.1 Implementation Procedure

For the transformation into implementation specific models, the SPACE engineering method was used. This creates building blocks that are expressed as UML models combined with Java code, wrapping the details of operations. Development of the system considered reusable building blocks from the Arctis library and also building blocks for new functionality. The composition structure is depicted in Fig. 1.2. The development method involved an iterated process of design and development, testing and evaluation. Based on this sound and successful results were achieved.

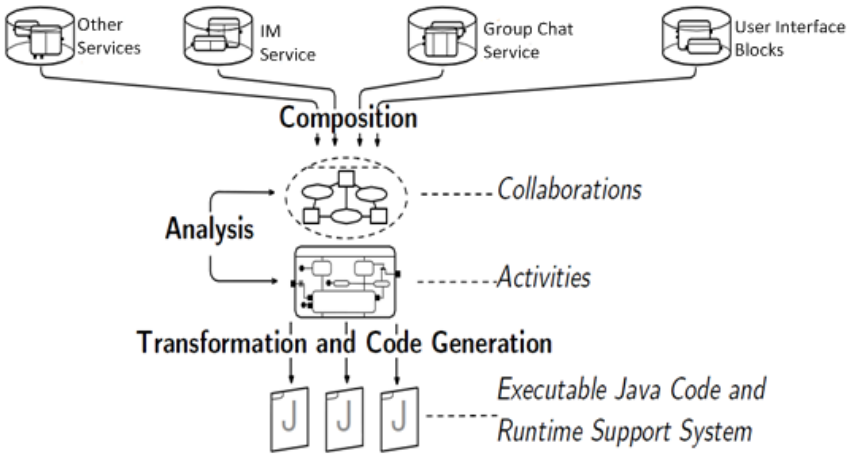


Figure 1.2: The SPACE engineering method.

1.3.2 Development Environment

Development was done in Eclipse Classic 3.7.0 (Helios) on Windows, using the regularly updated Arctis plug-in (latest version 1.0.0.M0714) and Android Developer Tools plugin for Eclipse, version 15.0.0. Testing of the application was performed with an HTC desire, Asus Transformer and Samsung Galaxy Nexus running Android 2.2, 4.1 and 4.2 respectively.

1.4 Contribution

The contribution of this thesis includes a comprehensive study of the City Guide application and documenting the system architecture. This includes

explaining how the system components are distributed and the communication principles used in the distributed architecture. In addition new components for IM service and Group Chat service have been developed. In developing these services we chose to implement a standalone architecture where message handling and routing for the services are done internally by our system without using open messaging protocols such as XMPP.

1.5 Outline of Thesis

The thesis is structured as follows:

Chapter 1 presents the domain as motivation behind this thesis. An overview of the City Guide application is given. In addition an outline of the methodology used is shown.

Chapter 2 provides theoretical background information of the technologies used in achieving the goals of the thesis.

Chapter 3 presents a background to the design and architecture of the City Guide application. The principles of the service discovery technology used are illustrated and communication principles explained as well. The building blocks used for service discovery are shown and explained

Chapter 4 presents the components of the City Guide application that has already been developed. The client system, server system and Registry are introduced and the services that have implemented so far are explained

Chapter 5 presents the architecture and components of the new services. Group Chat service components are used to illustrate the overall principles for the two services. Some specific components of the IM service are presented to show the differences between the two services.

Chapter 6 presents an evaluation of the City Guide application with regard to the technologies used in its current implementation. Alternative technologies are also discussed. In addition a critical assessment is done for the new services and a different architectural design is proposed.

Chapter 7 concludes the thesis by summing up the results, concluding remarks and proposing future work.

Chapter 2

Background

In this chapter, we give an overview of the concepts and technologies used in our work. This is intended to provide the reader a basic understanding of these principles. In Sec. 2.1, there is a brief introduction of Android Software Development Kit with emphasis on its support for location services. In Sec. 2.2, the engineering method SPACE and its model driven development tool Arctis are introduced. Section 2.3 presents the ActorFrame platform and its core component, the ActorRouter which supports asynchronous communication in distributed nodes. In Sec. 2.4, UML collaboration and Structural models used to specify functionality are explained. Finally in Sec 2.5, we introduce the principle of service discovery protocol, a technology used in our implementation work. Jini,UPnP, Salutation and Service Location Protocol are some emerging technologies of service discovery which are discussed and compared.

2.1 Android

The reader is assumed to have basic knowledge of Android; otherwise [OHA11] provides a comprehensive introduction to the operating system. The focus in the following section is on how the Android SDK can support *Situated Collaborative Learning Services.*

2.1.1 Android SDK

Developers can build applications using the Android SDK developed by Google. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language [OHA11]. These APIs facilitate access to contents on the

phone such as GPS or Wi-Fi information as well as integrate them with external web services in order to provide real-time services.

2.1.2 Android Location API

According to [KB09], "Mobile devices combined with location technologies enable what we call Situated Collaborative Learning". Current mobile devices are normally equipped with location detection capabilities. The technologies in use include:

- GPS
- Cell tower triangulation; where the position of a device is determined based on signal strength from nearby towers
- Wi-Fi hotspots that have known geographic locations

Android enables applications to obtain location services provided by these devices through a location library. A location point obtained provides information about the device's current location and it might provide other essential information such as elevation over the sea level and current speed. This normally depends on the type of location provider available. The different location providers have different requirements for power consumption, capital cost and accuracy. Location information is useful for map based applications. Acquiring the location information enables the geographical location to be pinned down on a map. The classes and interfaces from the *android.location* package can be found in [OHA14].

2.2 The SPACE Engineering Method

The specification unit for the implementation of the work in this thesis is based on the SPACE method. The SPACE method [Kra08] uses a combination of UML 2.0 collaborations and activities to describe systems and their services by the compositions of building blocks. The tool support for this method, *Arctis* is presented below.

2.2.1 Arctis

Arctis is an SDK for reactive systems [BA12] which is implemented as a set of Eclipse plugins. It consists of the Arctis Editor, Arctis Analyzer and the Arctis Compiler however the major interfaces towards the users are the library of building blocks and the editor for UML collaborations [KBH09]. This is

so because there is a high degree of automation. The building blocks encapsulate solutions to problems in a self-contained form, also securing in which sequence features are used [KSH09]. These reusable building blocks can simply be combined to develop applications. The external view of a building block is provided by a state machine called external state machine (ESM) which defines the allowed sequences of actions executed by the block. Due to this abstract description via external interfaces expressed by the ESMs, the internals of the building blocks do not have to be considered [KSH09].

In order to build applications with Arctis, available public libraries are checked to locate blocks with useful functionality for the application. The Arctis building block Wizard is used to create a new system, which is also a variant of a building block, and the blocks required can be dragged into the system. The blocks are connected with control flows or object flows in order to control the flows of execution among the blocks. If necessary, Java operations that can perform specific tasks may be added as well.

The Arctis components, the analyzer and the compiler enable formal analysis and model checking to be performed on the building blocks of the system. The analyzer verifies the specification of the blocks against theorems based on Temporal Logic. If a theorem is violated, the analyzer tries to identify possible reasons and presents an error trace as animation in the activity [KSH09]. If the specification is correct, the model transformation to implement components needs no further interaction, that is, it is automated.

The model transformation in Arctis maps the behavior implied by the activities to state machines [KBH09], which also serves as input to the code generator. The code generators produce the final executable implementations deployable on several platforms, with current focus of Arctis on Java and the use of ServiceFrame execution platforms.

2.3 ActorFrame

ActorFrame is a Java framework developed by *Tellu AS* for the development and execution of services. The services are modeled using UML 2.0 concepts for concurrent state machines communicating asynchronously through message passing. The ActorFrame protocol supports the *actor* play *roles* concept, and to create and configure actors. In addition there is support for routing of messages between actors deployed on different machines [TA08]. In Sec. 2.3.1, the general architecture of the ActorFrame service platform is presented. Section 2.3.2 describes the ActorFrame protocol and its communication architecture.

2.3.1 ActorFrame Concepts

The core concept of ActorFrame is the *Actor*. Actors are objects with state machines that communicate asynchronously with other actors. The behavior of the state machines will be according to the generic actor behavior well known to all other actors. Actors can have an optional inner structure of actors. Figure 2.1 illustrates an Actor. While the inner actors can be static,that is having the same lifetime as the enclosing actors, other actors can be dynamically created and deleted within the lifetime of the enclosing actor. An actor communicates with its environment using the *Input* and the *Output* ports. Inner actors within an actor can also communicate internally via these ports.

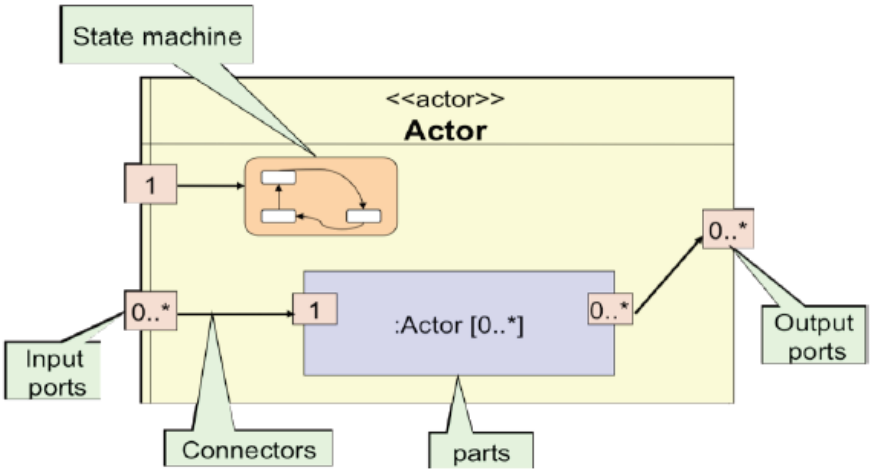


Figure 2.1: Actor[TA08].

A core component in ActorFrame is the *ActorRouter*. This message router carries messages between actors running in different Java containers such as Java J2SE, Midlet and OSGi [TA08]. Each router updates other routers with actors that are running in the same container as the router. When a receiving actor of a message does not run in the same container, the message is then forwarded to the appropriate container. This forwarding is possible through lookup in the forward table maintained by each router. This contains each registered actor's information describing its link to the container that has the actor.

2.3.2 ActorFrame protocol

ActorFrame employs the protocols for role requests and role releases. It allows for the dynamic creation and initiation of new roles on request. The basic feature of the protocol is to allow an actor (requestor) to request another actor to play a specific role. The protocol also allows the actors to interact and perform a service or a play [TA08]. Role release can lead to deleting of actors if their roles do not exist. Figure 2.2 shows a typical pattern of how *RoleRequest* and *RoleRelease* are used to invoke other actors to play services.

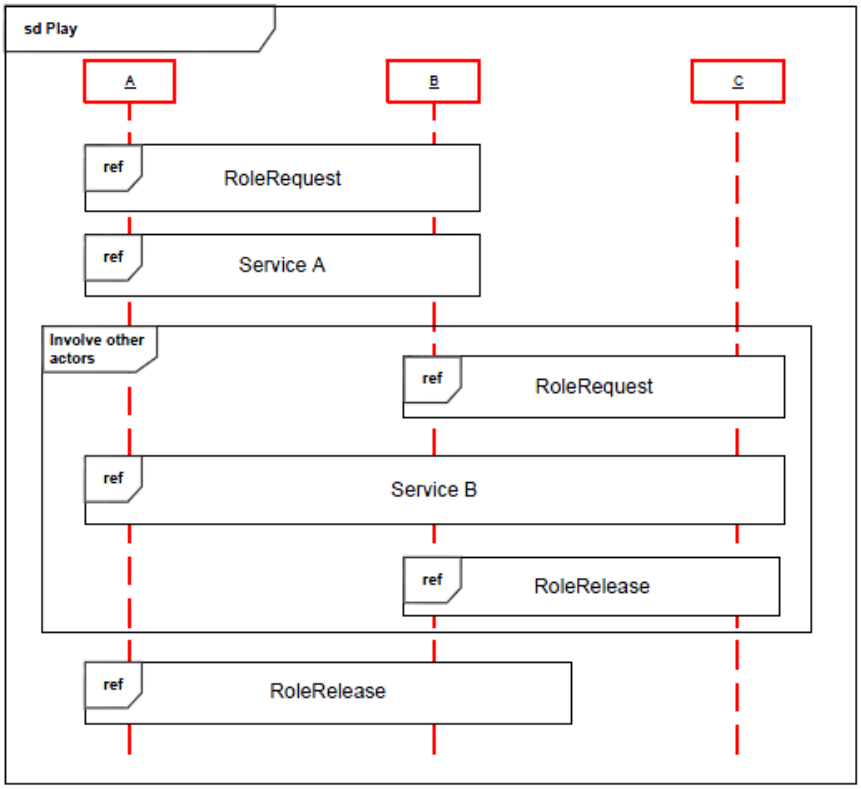


Figure 2.2: A simple service [GM03].

2.4 UML 2.0 Collaborations and Structural Models

According to [OMG09],"collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively

accomplishes some desired functionality." The roles are partial objects that interact with each other to achieve a joint task. UML 2.0 collaborations are structured classifiers and can have any kind of behavioral descriptions associated [OMG09].

Figure 2.3 illustrates the graphical notation of the UML 2.0 collaboration. A Collaboration is shown as a dashed ellipse containing the name of the collaboration. The internal structure is comprised of *roles* and *connectors* among the roles.

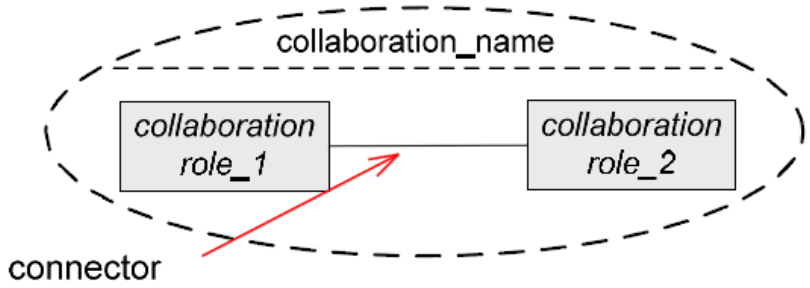


Figure 2.3: Notation for UML 2.0 collaboration.

UML collaborations are well suited to specify structural service models [OMG09, Cas08, Kra08]. The entities collaborating in the service are represented as collaboration roles. Services are classified into elementary and composite services. Elementary services are basic services that cannot be decomposed while composite services are composed from smaller services. When a service is composed from smaller services, the sub-services are specified using the concept of collaboration use where the roles of a collaboration use are bound to the roles of a composite collaboration [Kat12].

Figure 2.4 shows the structural model of a *WalkingTour Service*. This is a composite service specification and comprises of elementary services *Location Service, Quiz Service* and *Information Service*. Taking *Location Service* as an example, the roles of this collaboration *user* and *server* are bound to the roles of the composite collaboration *User* and *Server* respectively.

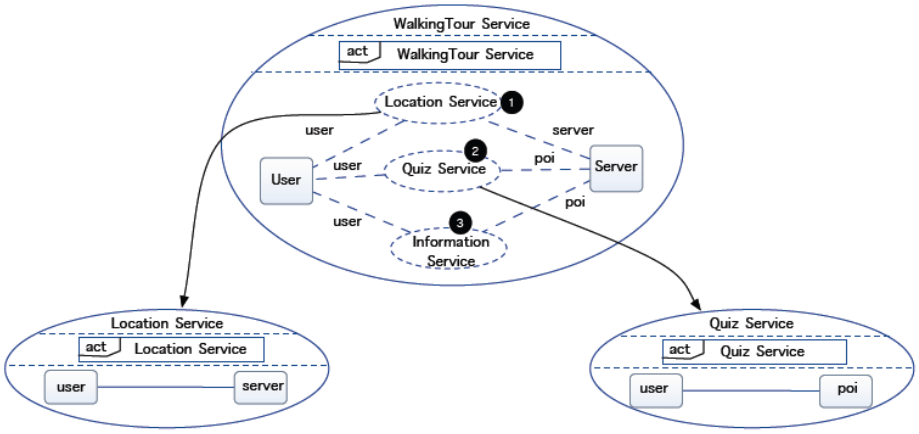


Figure 2.4: Structural model of a walking tour service. [1,2 and3] enumerate the collaboration uses [Kat12].

2.5 Service Discovery Protocols

The advent of mobile technologies such as mobile phones, laptops and PDA's has influenced how current application systems are packaged and deployed. The main requirement for mobile technology is the support of mobility for the users [ONT05]. However there are inherent challenges for the mobile devices such as limited resources in energy, computing capacity and storage capacity. In addition communication networks are faced with a strong dynamicity in connections and disconnections. It is important that the way software and network resources are configured, deployed and advertised be redesigned to meet current challenges of the mobile user such that the access and provision of services by mobile users is independent of time and location.

According to [ONT05], "Service Discovery Protocol enables network devices, applications, and services to seek out and find other complementary network devices, applications, and services needed to properly complete specified tasks". The implementation work done in this thesis is based on the principle of how collaborating roles seek out other complementary collaboration roles in order to perform services through the use of service discovery. Some emerging technologies of service discovery in the context of mobile and wireless computing are *Jini, UPnP, Salution, Service Location Protocol, Trader Service and Rendezvous protocols.* A few of the notable protocols are discussed below.

2.5.1 Jini Connection Technology

Jini is a technology developed by Sun Microsystems and is an extension of the Java programming language. The purpose of the Jini architecture is to federate groups of devices and software components into a single, dynamic and distributed system [LH02]. This enables coordination between different entities, that is the client and server know each other's existence from the other.

The heart of the Jini system is a trio of protocols called *discovery, join and lookup* [ONT05]. A device or an application registers with a Jini network using discovery and join protocols. *Discovery* is the process where the service is looking for a lookup service with which to register and *Join* takes place when the service places itself into the lookup table on the *look up* server. *Lookup* occurs when a client or user needs to locate and invoke a service described by its interface type and other attributes. The look up server is a service of directory which maintains dynamic information on the services. After the look up, a copy of the service object is moved to the client and used by the client to talk to the service [LH02]. The client then interacts directly with the service provider via the service object.

Jini uses Java RMI as the communications protocol between the service and the client which allows the dynamic remote loading of the code and also provides a security mechanism in such a distributed system [ONT05]. The network services run on top of the Jini software architecture. Figure 2.5 shows the architecture of the Jini connection technology.

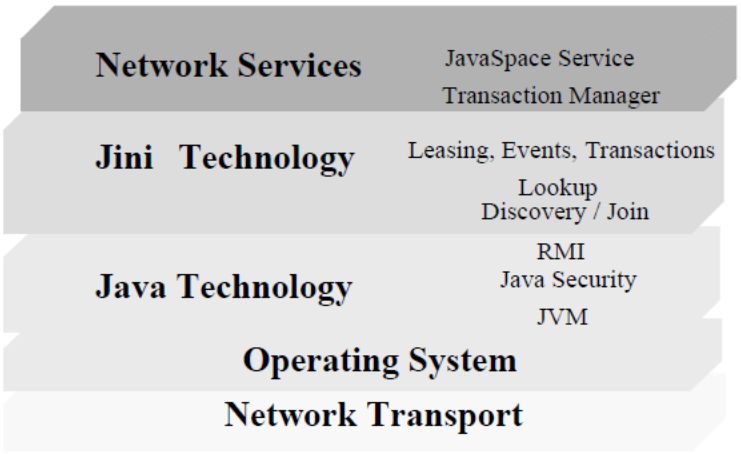


Figure 2.5: Architecture of the Jini connection technology [LH02].

2.5.2 Universal Plug and Play (UPnP)

UPnP is an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs of all forms [LH02]. Developed by Microsoft, it is an extension to Plug and Play (PnP) technology such that devices are reachable through a TCP/IP network. Unlike Jini which depends on mobile code, UPnP aims to standardize the protocols used by devices to communicate using XML [Ri00]. XML description enables complex and powerful description of device capabilities as opposed to Jini's simple service attribute.

Two main entities in UPnP are the *Control Point* and the *Device*. The *device* is an apparatus which offers services while *control point* is the entity which discovers these services and uses them [ONT05]. When devices are introduced into a network, they send multicast advertisements to the control points. These messages are called "alive" messages. When the devices want to terminate the availability of their services, they send "bye-bye" messages. In UPnP's current version (release 0.91) there is no central service register, such as the DA in SLP (see Sec. 2.5.4) or the lookup table in Jini [BR00].

2.5.3 Salutation

Salutation is a service discovery architecture being developed by the open industry consortium known as the Salutation Consortium. The consortium's goal is to build a royalty-free architecture for service advertisement and discovery that is independent of a particular network transport [Ri00]. The architecture provides a standard method for applications, services and devices to describe and to advertise their capabilities to other applications, services and devices [LH02]. According to [SC99], "the architecture enables application, services and devices to search other applications, services or devices for a particular capability, and to request and establish interoperable sessions with them to utilize their capabilities".

The architecture composes of three components namely; *Functional Unit (FU), Salutation Manager (SLM), and Transport Manager (TM).* The FU is the minimal meaningful function to constitute a client or service and can be regarded as the basic building block in the Salutation architecture. FU's register themselves locally or remotely with an SLM.

Figure 2.6 illustrates the model of the salutation manager. The SLM is the central focus of the architecture and this is comparable to the *lookup service* in Jini. It functions as a service broker. According to [LH02], "A service provider registers its capability with a Salutation Manager. When a client asks its local Salutation Manager for a service search, the search is

performed by coordination among Salutation Managers. Then the client can use the returned service." This means that the client's SLM performs the search for the desired service on behalf of the client as well as manages the communication session with the server's SLM.

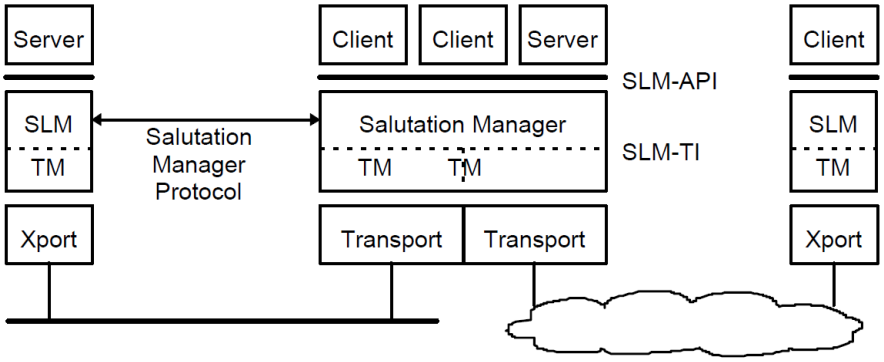


Figure 2.6: Model of the Salutation Manager [SC99].

The TM isolates the implementation of the SLM from a particular transport-layer protocol and thereby gives the Salutation, network transport independence [Ri00]. The TM is dependent upon the network transport it supports. This communication protocol independence is possible in the architecture because of a defined interface SLM-TI which is located between the SLM and TM. In order to support a new network transport, a new TM is needed however the SLM does not require any changes since it sees its underlying transport through the transport-independence interface (SLM-TI).

2.5.4 Service Location Protocols (SLP)

Service location protocol is an IETF standard that provides a scalable framework for automatic resource discovery on IP networks [Gu99]. It has a similar architecture as *Jini* but unlike the other service discovery protocols such as *Jini, UPnP and Salutation* that seek some sort of transport independence; SLP is designed for TCP/IP networks. The SLP architecture consists of three main entities with the following functions [BR00]:

- User Agents (UA) perform service discovery, on behalf of the client (user)
- Service Agents (SA) advertise the location and characteristics of services, on behalf of services

- Directory Agents (DA) collect service addresses and information received from SAs in their database and respond to service requests from UAs.

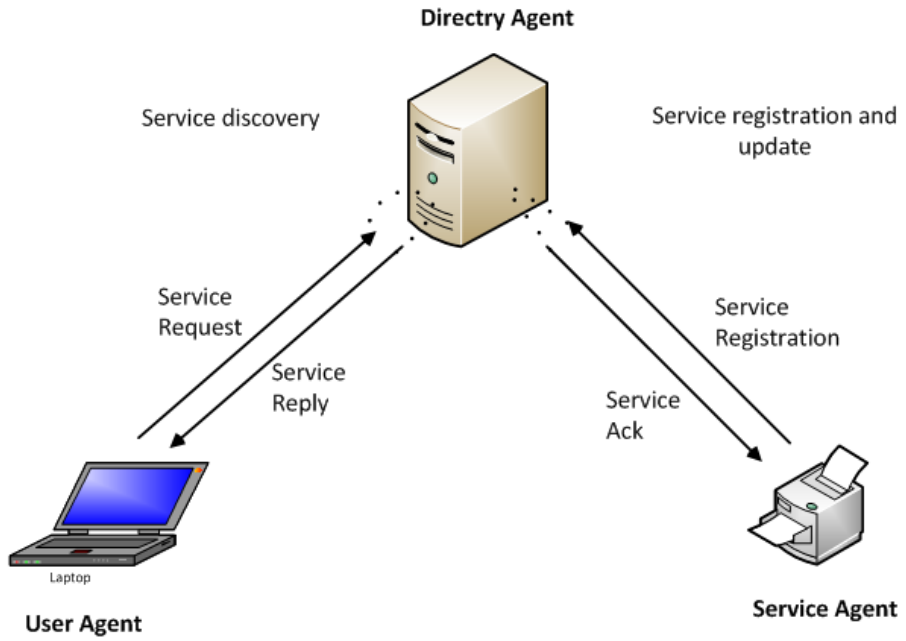


Figure 2.7: SLP agents transactions for service discovery and registration, adapted from [BR00].

The interaction among the three entities is shown in Fig. 2.7. When a new service is available, the SA contacts the DA to advertise its existence (*Service Registration*). A user that needs a certain service will require the UA to query the available service from the DA (*Service Request*). Upon receipt of the address and characteristics of the desired service, the user can then use the service.

DA Discovery

Knowing that a DA exists in a network is important in the service location protocol hence the UA and the SA must find a way to discover its existence. Three different methods for DA discovery are *static, active and passive* [BR00].

For the static discovery of DA, the UAs and SAs learn the locations of DAs by using the DHCP option for Service Location [PG99]. DHCP servers

use the DHCP Option 78 to distribute the address of the DA to hosts that request them [BR00].This means that a fixed address for the DA is known hence the SA and UA use this address to contact the DA.

The active and passive methods of DA discovery are illustrated in Figure 2.8. In active discovery, UAs and SAs send service requests to the SLP multicast group address [BR00]. A DA listening on this address will receive the request and respond with a unicast address to the requesting agent. For passive discovery, DAs multicast advertisements for their services and continue to do this periodically in case any UAs or SAs have failed to receive the initial advertisement [Gu99]. The UAs and SAs then learn the DA address from these advertisements and can contact the DA directly with their unicast address.

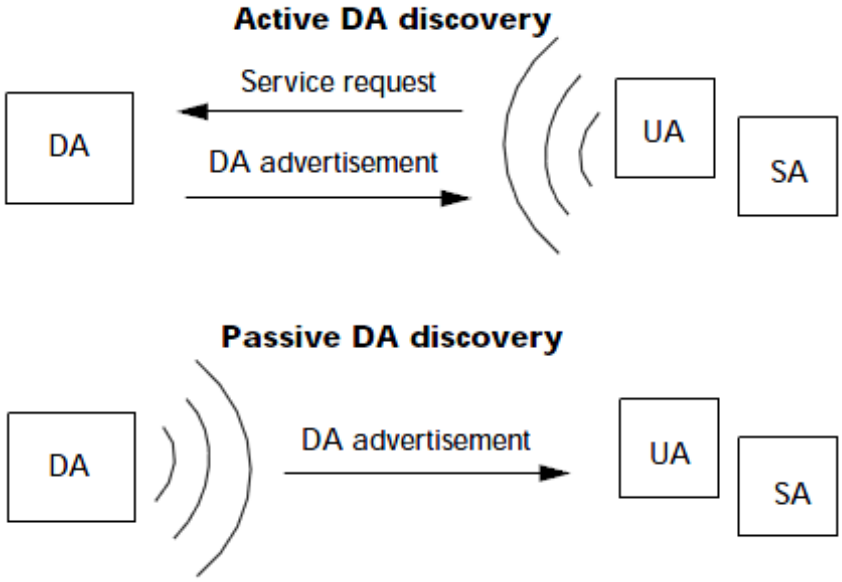


Figure 2.8: Active and Passive methods of DA discovery [Gu99].

Operation Mode

SLP has two modes of operation which are explained below:

- Centralized Service Description: When a DA is present, it collects all service information advertised by SAs as well as UAs and unicast their requests to the DA [Gu99].

- Decentralized Service Description: In the absence of a DA, UAs multicast requests for service and receive unicast responses directly from the SAs that control matching services [Ri00].

The benefits accrued when a DA is present are that; UAs receive faster responses, SLP uses less network bandwidth and fewer multicast messages are issued [Gu99]. Conversely when no DA is present, there is an increase in bandwidth consumption but such a model is simpler and appropriate for smaller networks [Ri00].

Service Advertisement

SLP services are advertised through a service URL [Ri00]. The service URL contains the IP address of the service, the port number, and the path [BR90]. This information contained in the service URL is what clients require to contact the advertised service. In addition Service Templates are used to specify the attributes that characterize the service and their default values [BR90]. The attributes differentiate between services of the same type and communicate configuration information to UAs [Ri00]. SAs advertise services according to attribute definitions in the Service Templates, and UAs issue requests using these same definitions [Gu99]. According to [Gu99], "This ensures interoperability between vendors because every client will request services using the same vocabulary, and every service will advertise itself using well-known attributes."

SLP does not define the protocols for communication between clients and services [Ri00]. This means that the protocol used between the client and the service is independent of the service location protocol.

Chapter 3

Background of the City Guide Application

In this chapter, we present a background to the system design of the City Guide application. The work in this thesis involves developing new components for the application hence it is important to understand the architecture of the application first. Section 3.1 introduces the City Guide application using UML collaborations. In Sec. 3.2 we present an overview of the application using illustrations where we explain the service discovery technology employed in addition to the communication methods used for the distributed architecture. In Sec. 3.3, we present the actual building blocks which are used for the service discovery technology in Arctis. We show how the building blocks of client proxy, server proxy and Proxy Host are used for internal and distributed communication with the system.

3.1 City Guide Application

The City Guide System is a proof of concept application which demonstrates a situated collaborative learning service [KB09]. The system specifies a service for students to learn about different historical places [KKB09]. It is a map-based application which constantly monitors the position of the student in relation to point of interests, which we call *treasures* that are situated nearby. When the student is within the defined proximity of a treasure, a service is initiated towards the student in the form of an interactive quiz. The interactive quiz service is a session between the student and a remote server where the student is asked questions about the treasure found. This service also provides additional information to the student such as a summary of

progress of treasures found and other useful information if necessary.

In the City Guide system, another main service provided is the Group Positioning Service. This is a service that keeps track of the position of the student and makes it available to the other group participants.The behavior of the components in the City Guide system can be specified using collaborations. In Fig. 3.1, UML collaboration is used to illustrate the entire system structure.

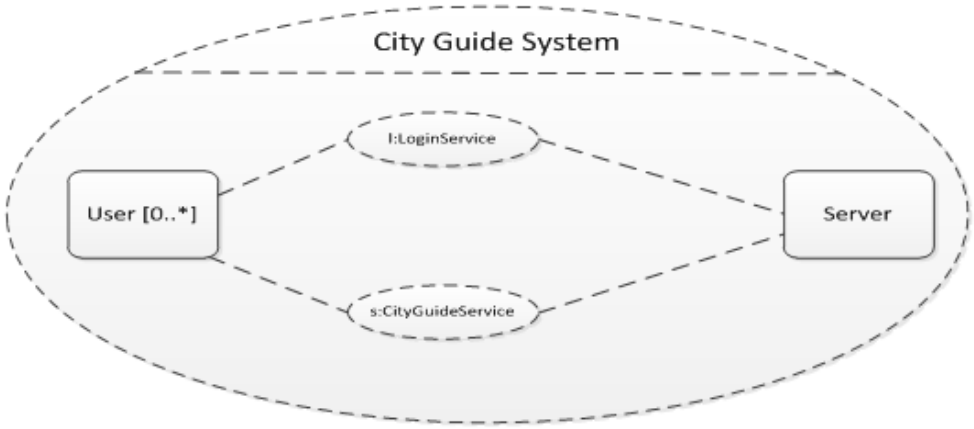


Figure 3.1: Collaboration of the City Guide System.

The system is composed of multiple users each connected to a server and interacting with it. The sub collaboration between the collaboration roles, i.e. l:Login Service and s:City Guide Service are the smaller collaborations into which the system can be decomposed. The structural decomposition of collaborations may result in elementary collaborations. This can be seen in Fig. 3.2 which shows the City Guide Service collaboration.

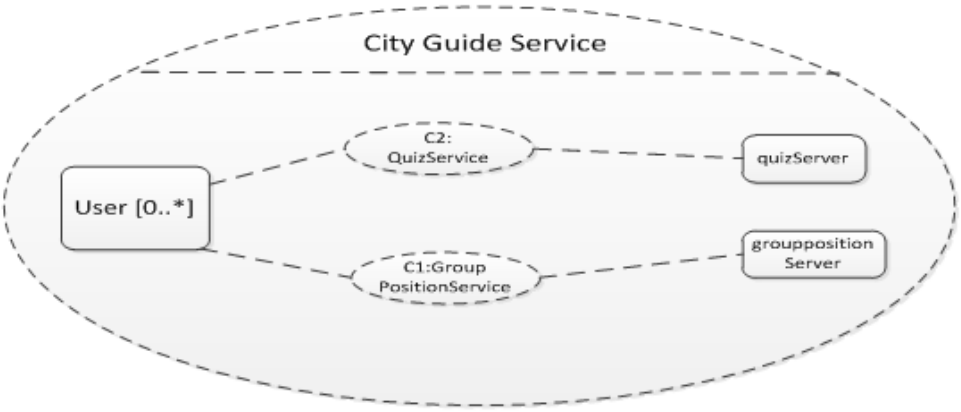


Figure 3.2: Collaboration of the City Guide Service.

3.2 High Level Overview of System

The UML collaborations shown above provide an overview of the overall service structure and the roles provided by the various components but they do not provide any detailed behavior. The engineering approach of SPACE is used to specify the detailed behavior of each component in the system. Figure 3.3 illustrates an overview of the system structure which is used in the implementation of the City Guide application. The three main components are:

- Client(User)-Android
- Server
- Registry

The three components are physically distributed and each has a *Proxy Host* component (shown in Fig. 3.3) which encapsulates communication and is used as an interface to interconnect the components (details to be explained later).

The client is a component in the model that contains the application logic for the client side of the City Guide system. It has an equivalent logical behavior as the corresponding role of the *User* from the UML collaboration in Fig 3.1. The client is run as an Android application deployable on mobile devices and with this distributed and reactive services can be supported.

The server is a component that contains the server side application logic of the City Guide system. It is run as a Java application and may be deployed

together with the Registry on the same machine (PC). The server performs all the corresponding roles of the *Server* from the UML collaboration in Fig. 3.1.

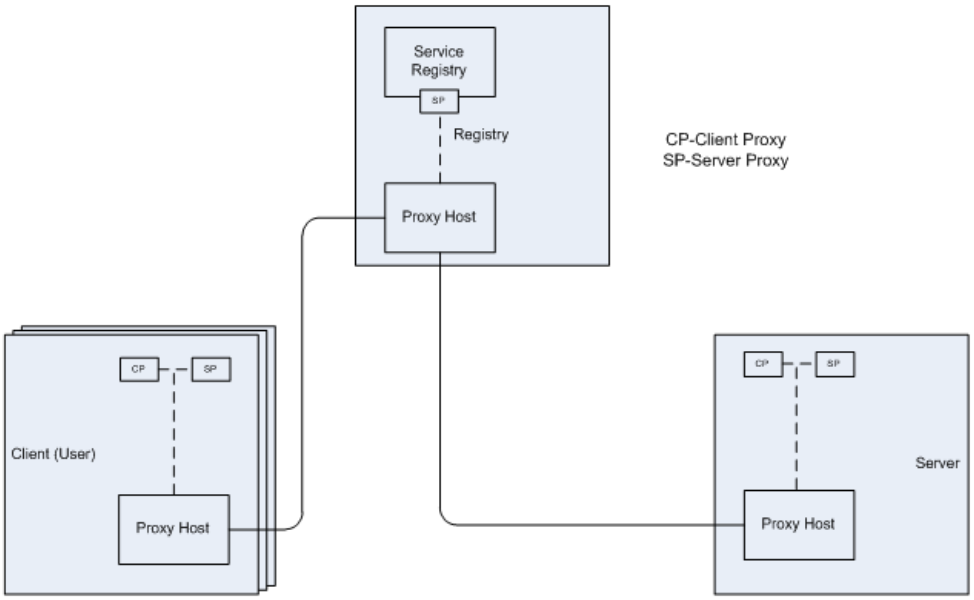


Figure 3.3: High level system representation of City Guide application.

3.2.1 Proxy Host

As seen from the high level model in Fig. 3.3, each of the three main components contains a Proxy Host block. This block is an interface for each component and handles all forms of internal and external communication among the components. It is considered as the router in each of the distributed nodes. To understand the communication principle employed, we consider the illustration in Fig 3.4.

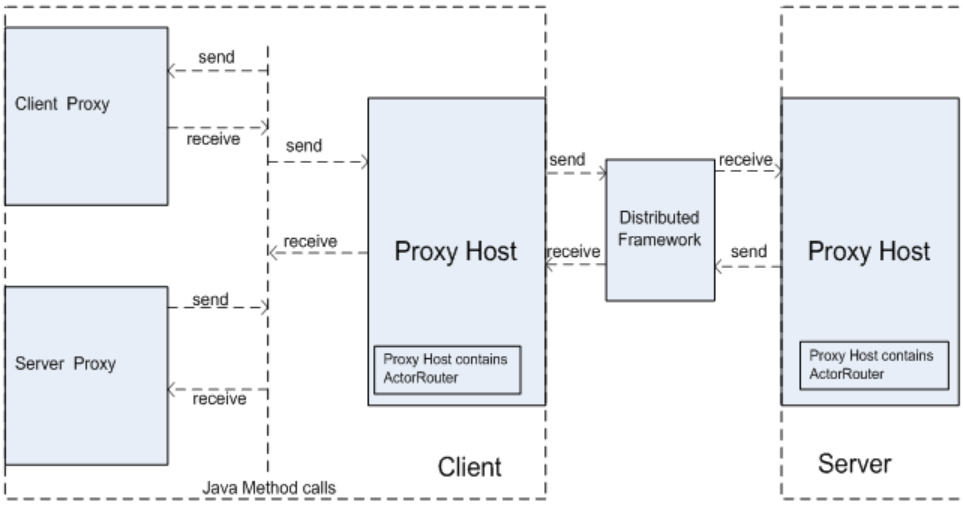


Figure 3.4: Proxy Host routing.

Within each main component, there are several other smaller components. In the Client for example, there are smaller components that are used to perform collaborations such as the *login service, quiz service and group positioning service*. Each of the smaller components contains local roles called either *Client Proxy* or *Server Proxy* which are used to communicate with the Proxy Host. The Client and Server proxies communicate with the Proxy Host by *send* and *receive* operations. The Proxy Host handles internal communication between different components within the Client. For communication with external components in either the Registry or the Server, the Proxy Host encapsulates ActorRouter technology which it uses to route messages across the network. The messages will be received first by the Proxy Host of either Registry or Server.

3.2.2 Service discovery mechanism

The collaboration roles of the users and the server for the various collaboration services are implemented using a service discovery mechanism. The service discovery mechanism employed here has an architecture closely similar to the *Service Location Protocol* explained in Chap 2. We will explain the service discovery mechanism with a service that involves collaboration between two roles, R1 and R2. In this mechanism, one of the collaboration roles (R1) for the service is registered with a central directory (known as Registry in our system). R1 uses an entity containing a set of client proxies and server proxy to do this registration. The entity referred to is called

Generic Service. The Generic Service registers a unique address with the Registry with which the behavior of R1 can be accessed. The Generic Service is therefore not a part of the collaboration role but is used together with the collaboration role to execute its functionality. We refer to the collaboration role together with the Generic Service as a *service provider*. This is because the purpose of the Generic Service is to inform the Registry (through registration) that a particular collaboration role is available and is accessible on this address. We consider the collaboration role as wanting to provide a specialized function or service, and therefore it announces its availability to the other role(s) that need to interact with it. We normally say the Generic Service registers the service behavior of the collaboration role.

For the second collaboration role (R2) to interact with R1, it needs a reference to its location. This is where the discovery happens. R2 uses a set of client proxies to also discover the collaborating roles it needs to interact with. It achieves this by requesting from the Registry the address with which R1 can be reached. R1 must already have been registered at the Registry using the Generic Service. We call the client proxies as *service proxy*. R2 together with the service proxy is then used to interact with R1. We refer to R2 with its service proxy as *service consumer*. This is so because the service proxy discovers the registered service of the R1 collaboration role with which the R2 collaboration role uses and it's now able to interact with R1.

We depict the service discovery mechanism principle in Fig. 3.5 where two collaboration roles CL-R1 and CL-R2 want to interact to perform a collaboration service. The Generic Service registers the behavior of CL-R1 at the Registry using a service name and an address. The behavior of the Generic Service and CL-R1 together is what we call the *service provider*. In the client system, the service proxy contacts the Registry using the same service name in order to find the address of the Generic service. Once it discovers this address, CL-R2 can now collaborate with CL-R1 by sending its messages via the address of its Generic Service. The behavior of CL-R2 together with its service proxy is what we call the *service consumer*. With this service discovery mechanism, there exists a *service provider-service consumer* relationship which has a direct relationship with the actual collaboration roles for each collaboration service.

If you take the collaboration roles of the login service as an example, one of the roles is implemented as a *service provider*. This means that the collaboration role uses a Generic Service to register its behavior with the Registry. The second role is implemented as a *service consumer* which means this role is used together with a service proxy to discover the service behavior of the first role. (See more details under Registry).

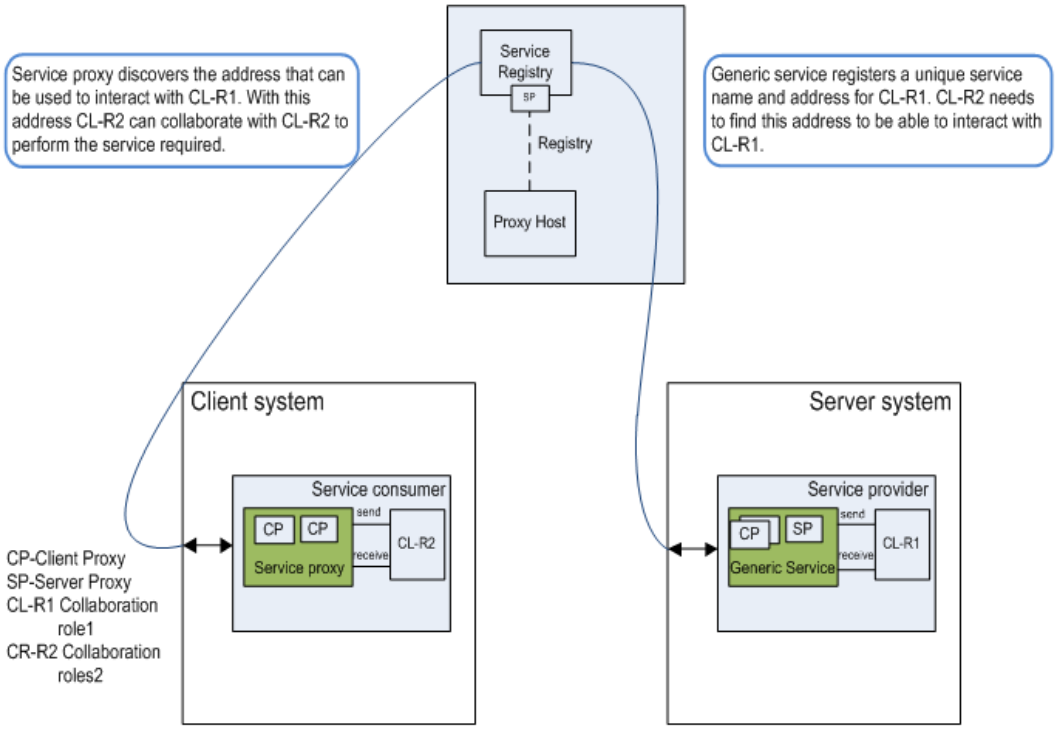


Figure 3.5: Service discovery of two roles collaboratiing to perform a service.

3.2.3 Registry

The Registry component in the model provides the role of registering services from *service providers* and availing these services to the *service consumers*. The Registry maps the address of the *Generic Service* from a service provider with its unique service name and stores it in a table of entry. A *service proxy* from a service consumer will then send a request to the Registry with the service name it wants to discover. The Registry checks to see if its table has a mapping to the requested service name. Once its mappings contain the requested service name, the address of the Generic Service representing the location of the service provider is returned to the service proxy. With this address the service consumer through its service proxy can now interact with the service provider.

A service provider role can be located in either the client or server system. Similarly a service consumer role can be found in either the client or server system. A typical model of the service provider-service consumer role behavior is shown in Fig. 3.6. This model is similar to the client-server architecture where the clients (service consumers) initiate communication sessions while

the servers (service providers) await requests.

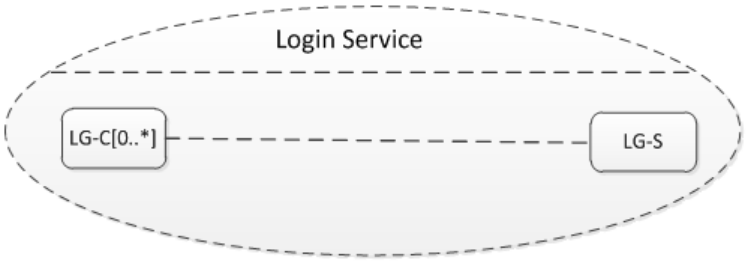


Figure 3.6: Login service collaboration.

Taking the Login Service collaboration as an example, the *server* provides the service provider role while the *user* provides the service consumer role. In this instance the LG-S role located in the *server* will need to be registered in the Registry first with a unique service name. Once this is done, the *user* which has the LG-C role can request (discover) the service from the Registry using the service name of the service provider. Fig. 3.7 shows a UML sequence diagram of the interactions within the Login Service collaboration as seen from the service discovery illustration in Fig. 3.5.

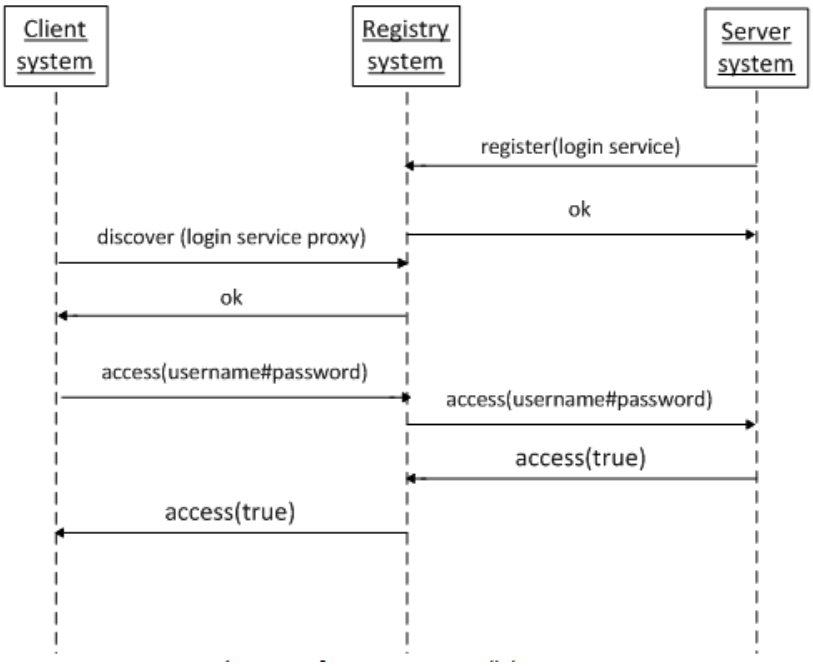


Figure 3.7: UML sequence diagram of Login Service collaboration.

A Location Service collaboration structure where the *user* plays the service provider role and the *server* plays the service consumer role is illustrated in Fig. 3.8. The location service works by using various location technologies such as GPS and WIFI to get the location of a user which is then sent to the server. In this instance since the *server* requests the location from the *user*, the *server* has the service consumer role for this collaboration and the *user* which provides the location service has the service provider role. Therefore, the LS-C in the *user* must be registered first in the Registry also with a unique service name. The *server* with the LS-S role can then discover the service provider through the Registry.

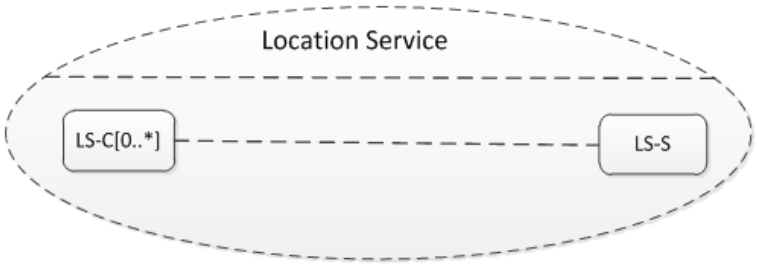


Figure 3.8: Location Service collaboration.

3.3 Relevant Building Blocks for Service Discovery

In this section, the building blocks that are used to implement service registration and service discovery are explained. In addition the Proxy Host block is presented briefly to explain the technologies used to handle internal communication as well as external communication between the distributed components.

The building blocks for service registration and service discovery are developed using parameterization to describe their service behavior. The instantiation of these building blocks by parameterization makes it possible to reuse the same blocks for multiple service registration and service discovery. The building block for service registration called *Generic Service* is shown in Sec. 3.3.1. In Sec. 3.3.2, a set of building blocks for service discovery is illustrated with an example called *Login Service Proxy*, from the login service collaboration.

3.3.1 Generic Service

The Generic Service block is used to register a service behavior with the Registry. It is a parameterized block and so the user instantiates the block with a *service name*. The user is prompted to provide an instance parameter value through a GUI. The value provided is the service name. The service name is then used to register an address of the Generic Service block at the Registry. The internal behavior is as shown below in Fig. 3.9. The two blocks involved here are the *Register and Deregister Service* block and *Server Proxy* block.

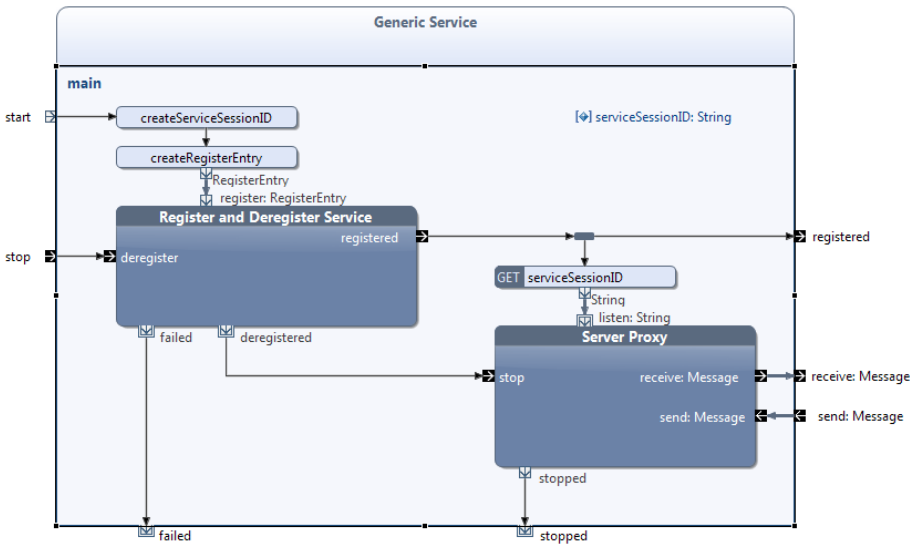


Figure 3.9: Internal behavior of Generic service block.

Server Proxy

The primary purpose of the Server Proxy is to connect to the Proxy Host. The Server Proxy listens to incoming messages that are sent to a particular serviceSessionID. The serviceSessionID is the *service name* used to instantiate the Generic Service block. In the Generic Service, the serviceSessionID is passed to the Server Proxy. Inside the Server Proxy block shown in Fig. 3.10, a *subscribe* operation registers the serviceSessionID with the Proxy Host and can thus monitor and receive incoming messages to this serviceSessionID. The received messages are passed via the *receive* pin.

The Server Proxy also performs *send* operations. This is necessitated when the Generic Service needs to send a reply to a message that has previ-

ously been received. Again for such an operation the Server Proxy collaborates with the Proxy Host to perform this function.

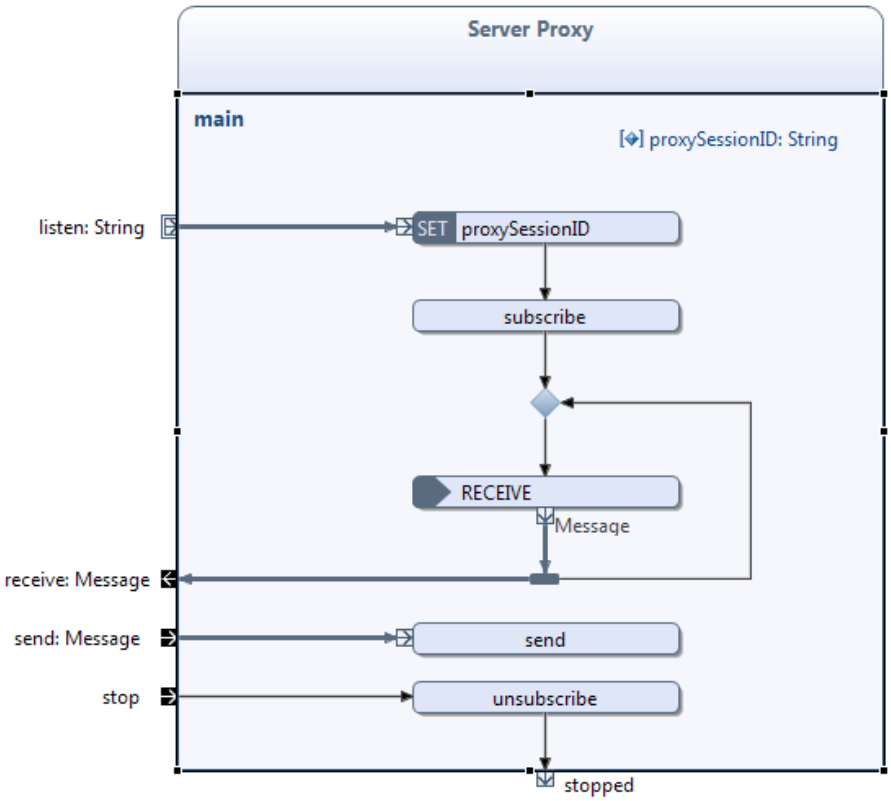


Figure 3.10: Internal behavior of Server Proxy block.

Register and Deregister Service

As the name of the building block suggests, the *Register and Deregister Service* block shown in Fig. 3.11 has the functionality to register and deregister services. To register or deregister a service, the Generic Service creates a registry entry using the address and the service name which is passed to the Register and Deregister service block. In this block a *Register service* block handles registration process separately while the *Deregister service* block handles the deregistration function. The two blocks have the same external behavior. They only differ in the kind of messages they send to the Registry. The internal behavior of the Register Service is shown in Fig. 3.12.

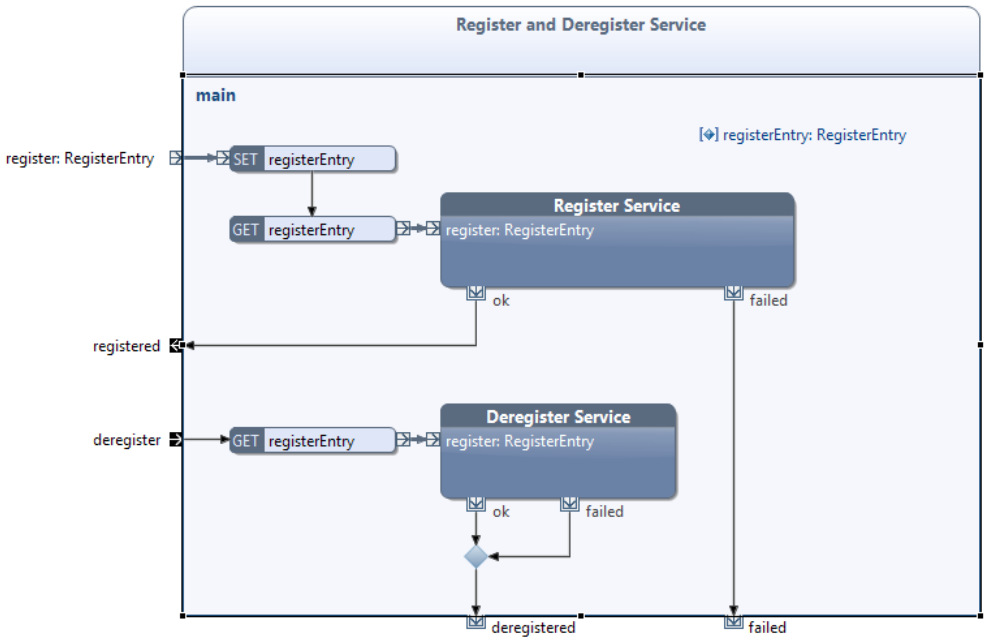


Figure 3.11: Internal behavior of the Register and Deregister Service block.

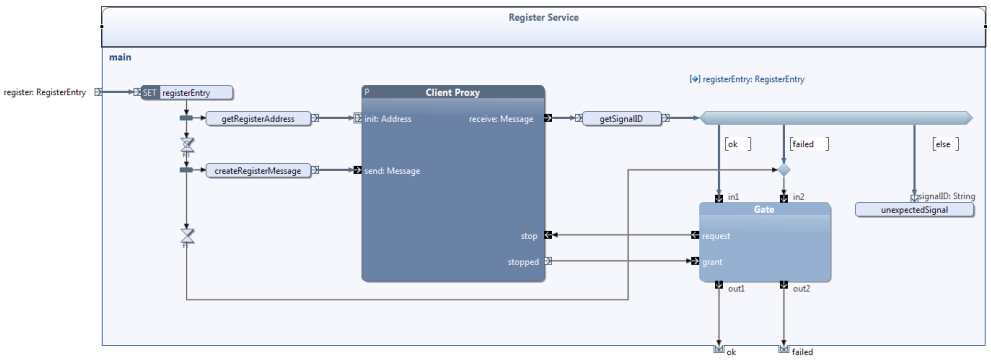


Figure 3.12: Internal behavior of the Register Service block.

The Register Service block contains a *Client Proxy* block shown in Fig. 3.13, which similarly to the *Server Proxy* is a basic building block for communication as explained in Sec. 3.2.1. The Client Proxy has the same external behavior (ESM) as the Server Proxy. It's purpose is also to communicate with the Proxy Host block. The client proxy differs from the server proxy in its internal behavior by passing the Registry address to the Client Proxy as a variable. This address is then used in the *message.setReceiver* method for

every message sent using the Client Proxy. This is done to force all messages sent from the Client Proxy to be received by the Registry.

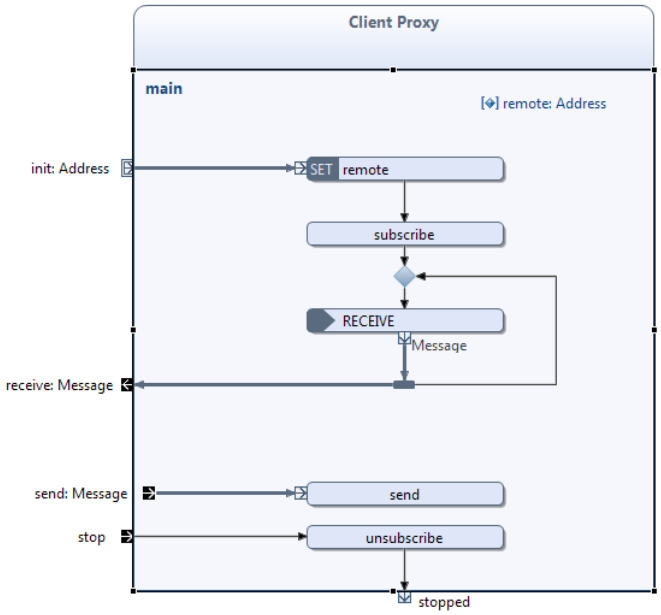


Figure 3.13: Internal behavior of the Client Proxy block.

3.3.2 Login Service Proxy

The *Login Service Proxy* is used to illustrate the set of blocks that are used in service discovery. Shown in Fig. 3.14, the proxy has two blocks namely the *Discover Service* and the *Client Proxy* blocks. The discover service is the block that discovers the service. It is also instantiated through parameterization similar to the configuration of the Generic Service. The service name used to instantiate the *Discover Service* block must be the same as the one used for the service registration. In the *Discover Service* block shown in Fig. 3.15, a remote message is sent to the Registry using the Client Proxy. The message contains the name of the service it wants to discover and it requests the Registry to return an address of that service name contained in the message. The Client Proxy will receive a reply message from the Registry and consequently return the address of the service if available via *found* pin or otherwise terminate via *failed* pin.

The address of the service found by the *Discover Service* block is fed

to another Client Proxy as shown in the Fig. 3.14. The Login Service Proxy having found the address of the Login Service can then send messages using the Client Proxy by setting the address of the Login Service in its *message.setReceiver* method anytime it wants to send a message.

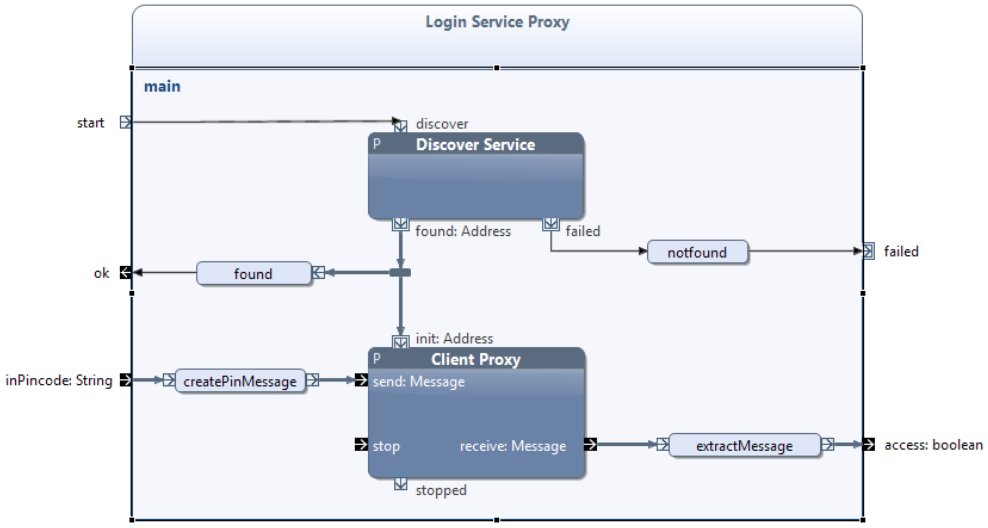


Figure 3.14: Internal behavior of the Login Service Proxy block.

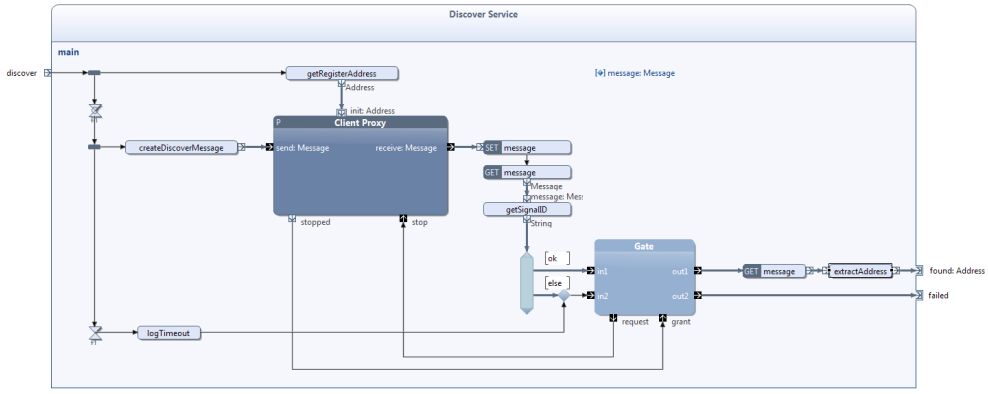


Figure 3.15: Internal behavior of the Discover Service block.

3.3.3 Proxy Host

The Proxy Host block may be considered as an interface for communication. In a particular system, several components may want to communicate

with other components in the same system or different components in other systems. The basic building blocks for communication that are used in the City Guide application; Client Proxy and Server Proxy, collaborate with the Proxy Host to make communication possible in the system.

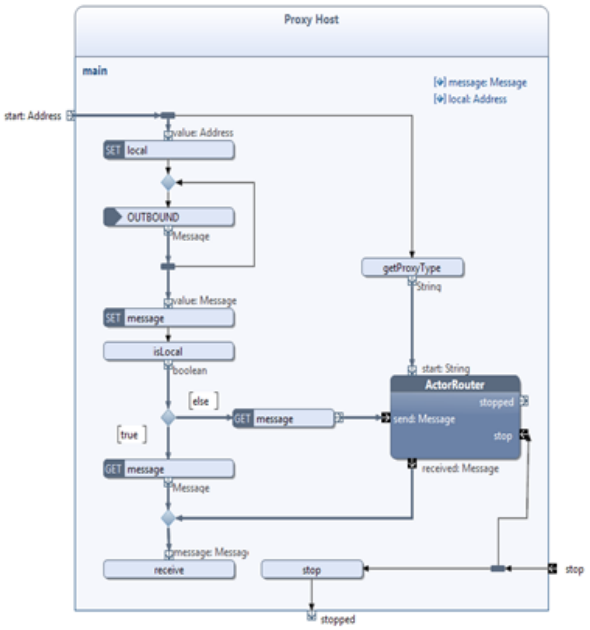
In implementing systems, they could be deployed on the same local host machine or distributed on different platforms. For the city guide application, the Registry and Server systems are deployed on J2SE while the Client system is deployed on an Android platform. The Proxy Host encapsulates ActorFrame technology that supports P2P communication architecture with asynchronous message passing which effectively handles active service and distribution [KB09].

The Proxy Host is parameterized with some system configurations. There are four configuration parameters each separated by "-" and is represented as such; param1-param2-param3-param4.

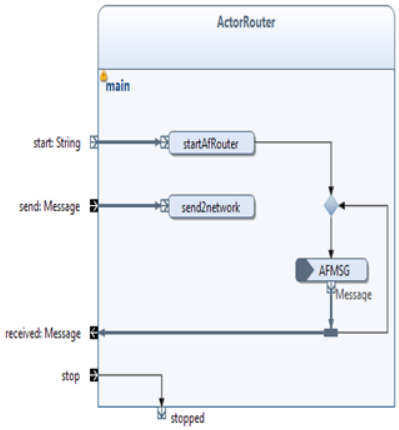
- param1 defines the system type: Server for the server node (Registry), Client for a standard Java client, AndroidClient for an android client.
- param2 is the IP address of the server node (Registry).
- param3 is the listener port of the server node.
- param4 is the listener port of the client node.

The internal behavior of the Proxy Host shown in Fig. 3.16 is started via start pin which sets the local address as a variable. The local address consists of the port number and IP address of the terminals running the client, server and Registry systems. A control flow also performs an operation *getProxyType* which returns the *String* value of the instance parameters (param1-param2-param3-param4) set for the Proxy Host . This is used to start the *ActorRouter* block.

The Proxy Host performs *send* operations on behalf of the Client Proxy and Server Proxy blocks. When a *send* operation is done, a reception signal OUTBOUND is received in the Proxy host together with the message. An operation *isLocal* is performed next which results in a boolean decision. The operation returns *true* if the message sent should be received by the other components in the same system. Other messages will flow via the *else* branch and will be forwarded to the *ActorRouter* block via the *send* pin. This means that these messages are either not addressed to any components within that system or have been intentionally forced out to be received by components in other systems.



(a) Internal behavior of Proxy Host



(b) Internal behavior of ActorRouter

Figure 3.16: Internal behavior Behavior of Proxy Host and ActorRouter.

The *ActorRouter* block encapsulates the ActorFrame protocol and basically supports routing of messages between actors deployed on different machines. This supports the distributed deployment of the City Guide application that has the client system running on Android with the Server and Registry systems running on a J2SE local host machine.

Chapter 4

Components of the City Guide Application

This chapter presents the building block components of the three systems making the City Guide application as has been developed already. We analyze the system components and show how the service discovery mechanism is used to implement the collaborative services between the users and the server. It is important to have this overview of the system design before components for the new services are added. Section 4.1 presents the Registry System. In Sec. 4.2 we present the server part of the application called the *City Guide Server* and in Sec 4.3 we present the client part of the application, called the *City Guide App*. Reference to Chap. 3 is encouraged in order to put the system components in context with regard to the high level overview already explained.

4.1 Registry System

In this section, the various building blocks and call operations that make up the Registry system are presented. The Registry system acts as a registry for all the services in the system. Figure 4.1 gives an overview of the Registry system. The system contains a *Service Registry* block and a *Proxy Host* block. The Proxy Host is a common block that must be contained in each of the three systems as already explained. When the Registry system is started, a token flow starts from the two initial nodes. A timer is used to delay the Service Registry block so as to ensure that the Proxy Host block is started before the Service Registry block. This is necessary since all communication from within the Service Registry block depends on the Proxy Host.

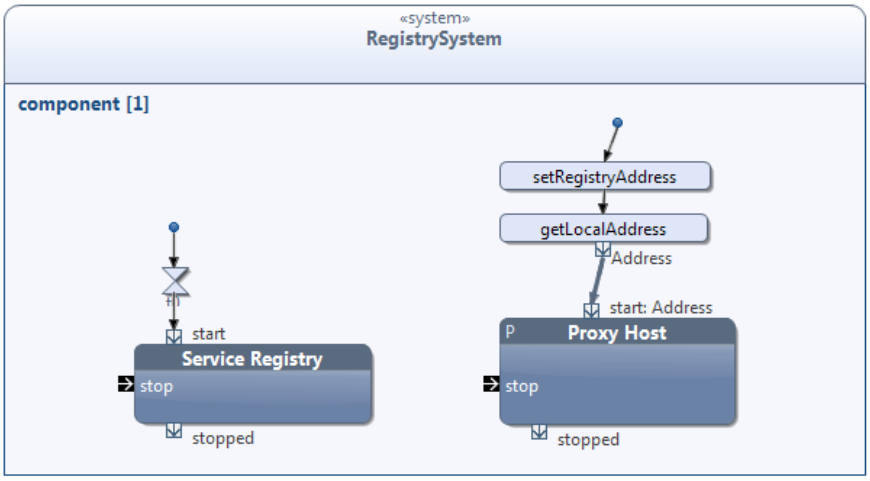


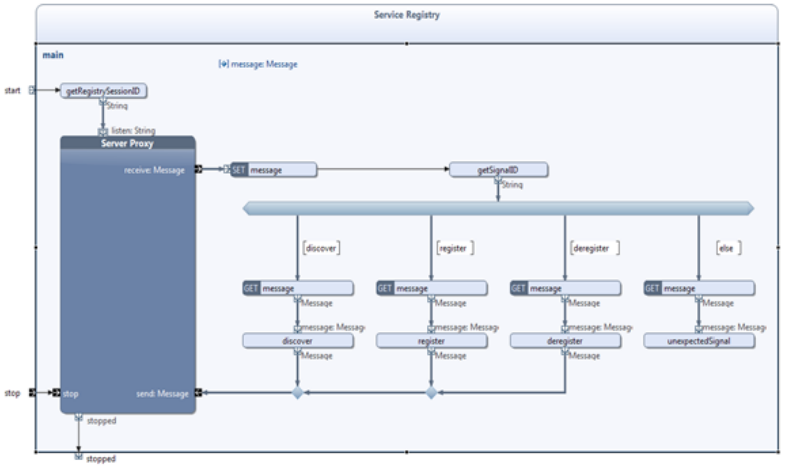
Figure 4.1: Registry System.

4.1.1 Service Registry

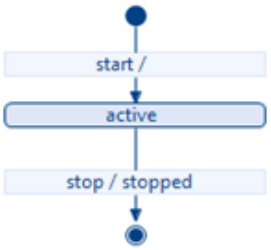
The Service Registry provides the service whereby all services will be registered (service registration) and all service proxies will look up (discover) for the corresponding service. Its behavior and ESM is shown in Fig. 4.2.

The block starts via the initial pin *start* and performs a Java operation to set the sessionID of the Registry which is then passed to the Server Proxy block (explained in Sec. 3.3.1). The block can receive a message of a certain data type *Message* from the *receive* pin of the Server Proxy block. The received message is checked to extract a *signalID*. The *signalID* is a *String* value contained in the message and based on it, a decision is taken to perform either of the following operations: *discover, register, deregister, unexpectedSignal*. As their names suggest, the *register* operation is used to register new services in the Registry. The *discover* operation is where available services are looked up (discovered) and the *deregister* operation is where registered services are removed from the Registry. The *unexpectedSignal* operation handles any other message that is undefined.

The *discover, register and deregister* operations also return values of type *Message* which is sent to the *send* pin of the Server Proxy. The reason for this is that it is important for the Service Registry to communicate back with the sender's of the messages on the status of the operations they want the Registry to execute. The Registry also uses this return value to provide any needed resources such as references to the Registry address and references to other service addresses'.



(a) Internal behavior of ServiceRegistry



(b) ESM of Service Registry

Figure 4.2: Behavior and ESM of Service Registry.

A service that wants to register in the Registry can either be successful or fail in the registration process. When a service is successfully registered, an acknowledgement message of "OK" is sent to the registered service to inform it of the success of registration of the service. Conversely when the registration fails, a "failed" message is sent to that service. Similarly in a *deregister* operation, when the service is successfully deregistered from the Registry, an "OK" message is sent to the service that requested the action and if the operation was unsuccessful, a "failed" message is returned to the service.

In a *discover* operation, similar acknowledgement messages are sent as well, as explained in the register and deregister operations. In addition however, given a successful discovery of a service, a reference to the address of the discovered service is added to the message before it is sent to the requesting component.

Considering the ESM of the Service Registry , after a token has been emitted from the *start* pin the block will remain in active state until it receives a token via the stop pin. The Registry system is supposed to be running throughout the lifetime of the application.

4.2 City Guide Server

The building blocks making up the server system of the City Guide application are presented in Fig. 4.3. As already explained, this part of the application contains the server side application logic of the City Guide application. It consists of the *Group Manager, City Guide Service, Proxy Host and Web Server* blocks. When this system is started, a token is fired simultaneously through all the three initial nodes present. It is important again to ensure that the Proxy Host is started first. This is achieved by using a timer to delay the start of Group Manager block.

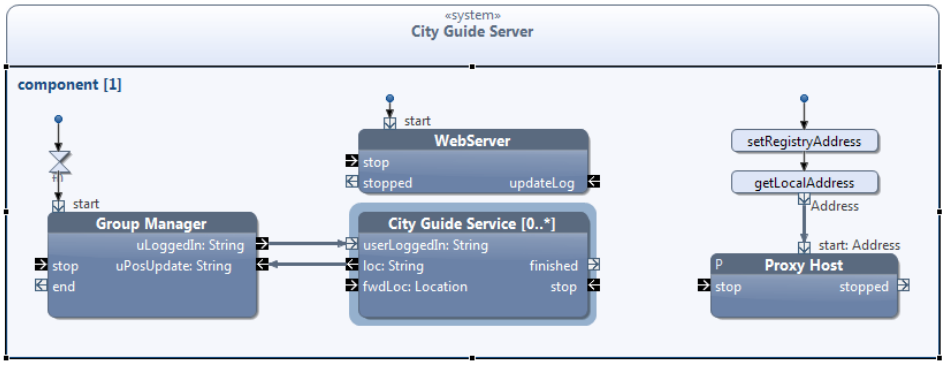


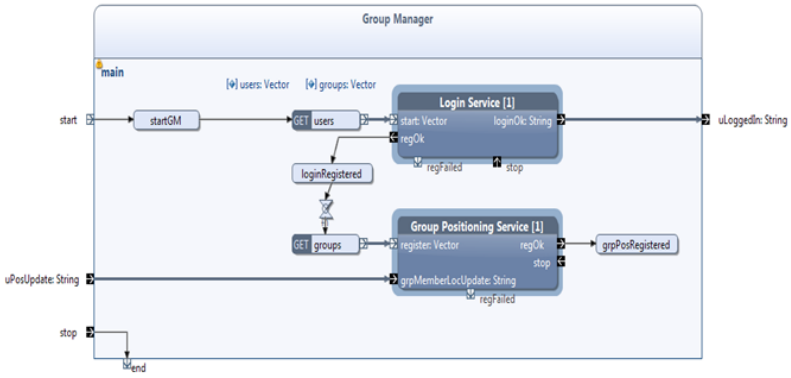
Figure 4.3: City Guide Server system.

After the Group Manager block has been started, it will emit a token via pin *uloggedIn*. This pin is a streaming pin which provides a data flow to start an instance of the City Guide Service. The City Guide Service is a block that can create multiple instances of itself (see Sec. 4.2.2). The City Guide service has a streaming pin *loc* which returns a result that is fed to the Group Manager via its pin *uPosUpdate*.

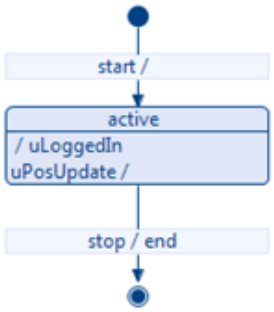
4.2.1 Group Manager

The Group Manager contains the *Login Service* and the *Group Positioning Service* blocks. Figure 4.4 shows the internal behavior and ESM of the Group Manager. The block is started when a token flows via pin *start*. An operation, *startGM* is performed next. This operation loads all configurations for the system including the players of the game and the groups they belong to. Configuring the system by adding, modifying or deleting players and groups is normally done prior to deploying the system. The game owner (teacher) is responsible for doing this configuration and is done using XML. The variables *users* and *groups* of type *Vector* are used to hold the players and the groups.

Considering the ESM of the Group Manager, the block is in active state after a token is received via *start* pin. In the active state, the block can emit tokens via streaming pin *uloggedIn* or receive tokens via streaming pin *uPosUpdate*. The block only terminates when it receives signal via *stop*.



(a) Internal behavior of GroupManager



(b) ESM of GroupManager

Figure 4.4: Behavior and ESM of Group Manager.

Login Service

The Login Service, shown in Fig. 4.5 is the component which executes the service provider role of the Login Service collaboration. It contains a Generic Service which it uses to register the service with the service name "login". The corresponding service consumer role (found in the client) sends login data and is received by the Generic Service via *receive* pin. The login data is fed to a Java operation *extractMessage* which checks whether the client exists and should be authenticated. Based on this operation, a message is sent via *send* pin to the service consumer to either grant or deny the client. An authenticated client will cause the Login Service to emit a data flow via pin *loginOk* while an unauthenticated client causes no further action in this block and is terminated in the *loginFailed* operation.

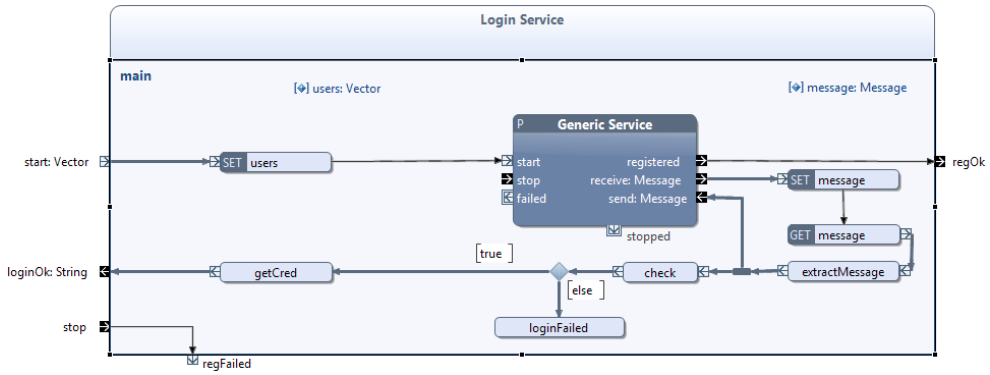


Figure 4.5: Internal behavior of the Login Service block.

Group Positioning Service

For the group position service collaboration (see Fig. 3.2), the Group Positining Service shown in Fig. 4.6 is the component that plays the service provider role. The main function of the Group Positioning Service is to receive position updates from every user and forward them to the other members of the group. The block uses the Generic Service to register in the Registry with service name "grouppositioning". A client sends a message requesting this service and it is received via receive pin. An operation *getGroupMemberLocations* checks to see if the client belongs to any groups. If this is confirmed as true, a location update of the group is returned to the client via *send* pin. Users periodically update their positions to the Group Positining Service via the streaming pin *grpMemberLocUpdate*.

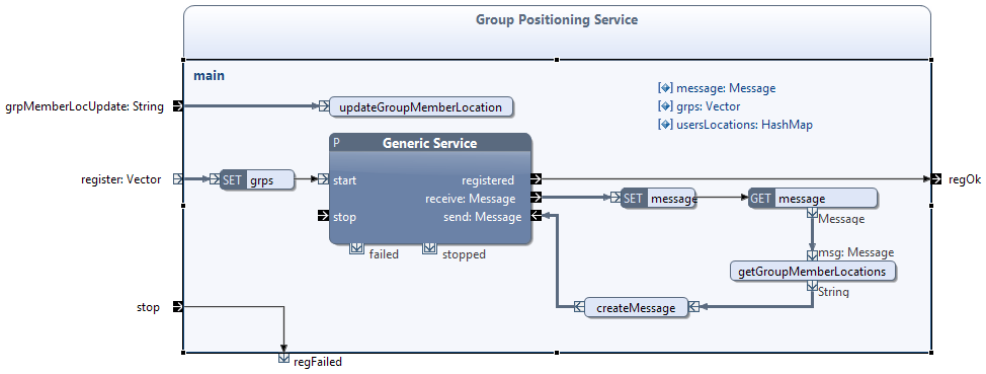


Figure 4.6: Internal behavior of the Group Positioning Service block.

4.2.2 City Guide Service

The City Guide Service shown in Fig. 4.7 manages part of the resources that the server requires to perform the rest of the collaboration with a client after login. Active initiatives that the server makes towards the user are controlled in this block. An instance of the City Guide Service block is created for each user after successful authentication by the Login Service in the Group Manager. This means that a copy of the City Guide Service block with all its internal resources is created for each client. The intended lifetime for each copy of the block is the duration for which the client remains active and is interacting with the server. This is referred to as *multisession* creation. As such, each client using the application will have a corresponding City Guide Service block component in the application which will be managed entirely by the server.

The block is started via the *userLoggedIn* pin which carries a data flow, the value of which is the username of the client for which the City Guide Service instance is created. The operation *setCred* is performed to set the block instance and also to download the system configurations for the treasures (points of interest) associated for the user. The City Guide Service block contains the *Location Aware Quiz Service Proxy and Location Service Proxy* blocks. These two blocks are service discovery blocks and hence execute the service consumer role for their collaboration services. The Location Service Proxy is the component that subscribes to the location service provided by a corresponding component in the client. The Location Service Proxy requests for periodic location updates from the client. The location information received is combined with the client's username and sent to the Group Manager via pin *loc*. In addition the location information is used to check if the client

is near a point of interest. In such a case the *Location Aware Quiz Service Proxy* is started to perform the quiz service collaboration.

The *Location Aware Quiz Service Proxy* is started when the user's location is near a treasure. The *Location Aware Quiz Service Proxy* subscribes to the Quiz Service (found in the client) and takes initiative towards the client by pushing questions to the client via *reqQuiz* pin.

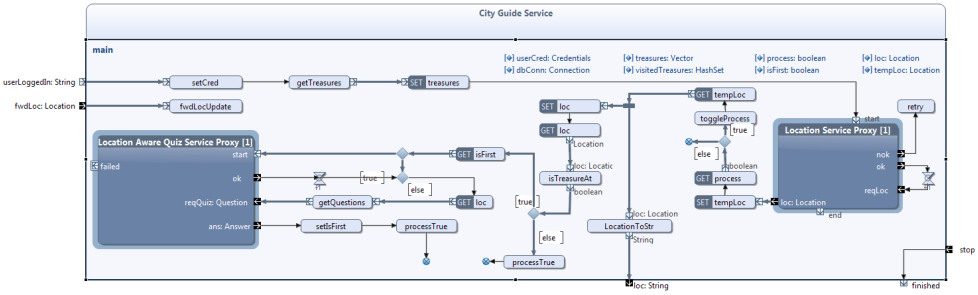


Figure 4.7: Internal behavior of the City Guide Service block.

4.3 City Guide App

The City Guide App shown in Fig. 4.8 is the system deployed on an Android phone and is the final part of the City Guide application. As explained in Sec. 3.2, this is the part that contains the application logic for the client side of the City Guide application. The system consists of *Server Connection Dialog, Login App, City Guide UI and Proxy Host* blocks. Here too, the Proxy Host should be started before the main components are started. A timer delay inserted before the start of the City Guide UI achieves this purpose.

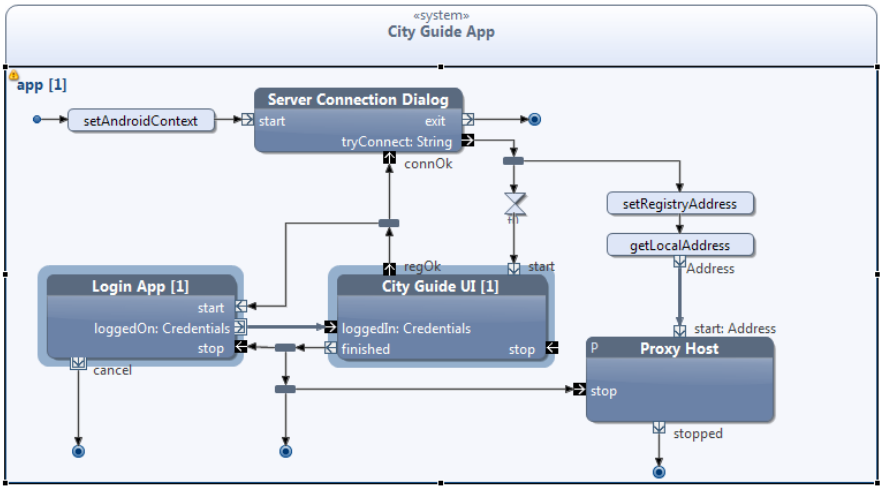
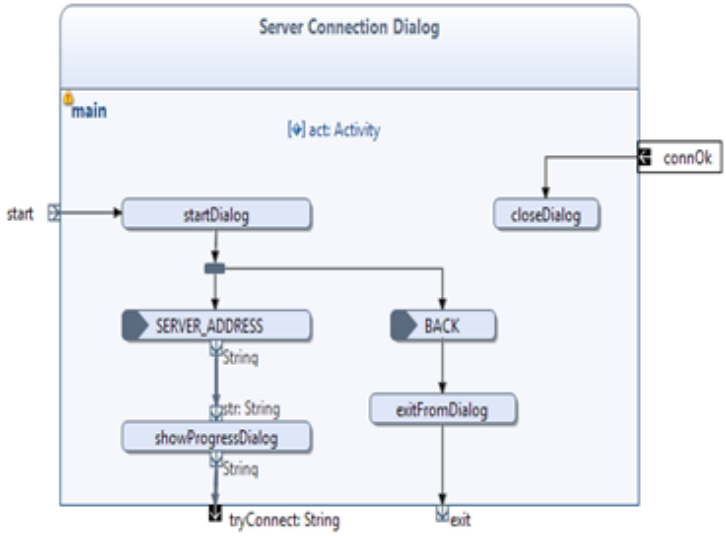


Figure 4.8: Internal behavior of the City Guide App block.

The system starts from an initial node and a token flow to start the *Server Connection Dialog* via *start* pin. An operation *setAndroidContext* is performed prior, in order to set the Android context for the running application. The *Server Connection Dialog* emits a token via *tryConnect* pin which is forked to start the Proxy Host and City Guide UI block. The City Guide UI after it has been started will emit a token via *regOk* pin. The token is then forked to the streaming pin *connOK* and also via *start* pin of Login App. A successful login by the user will cause a data flow via *loggedOn* pin which is then fed back to the City Guide UI via *loggedIn* pin.

4.3.1 Server Connection Dialog

The *Server Connection Dialog* encapsulates an Android activity screen which presents the first interactive screen between the user and the application. The activity shows a dialog window with an editable text field with which the user can enter the "server IP" address it is trying to connect with. The IP address to be entered in this field will be the IP address of the local host machine on which the Registry system is running. A progress bar is shown as well when the connection process is in progress. When the block receives a token via *connOk* pin, the dialog window and the progress dialog are cleared off the screen. Its internal behavior and activity screen are shown in Fig. 4.9.



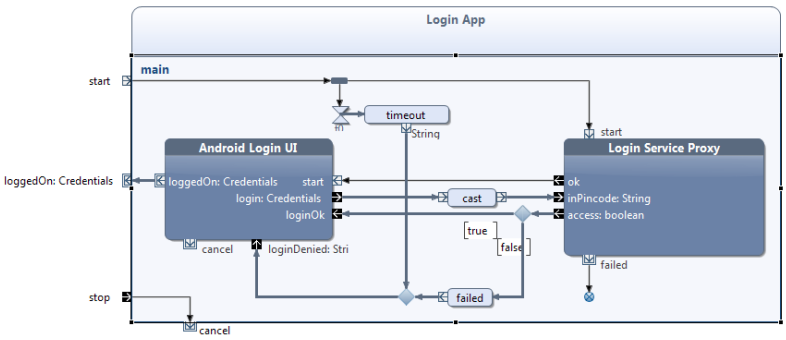
(a) Internal behavior of Server Connection Dialog

(b) Activity screen showing dialog window

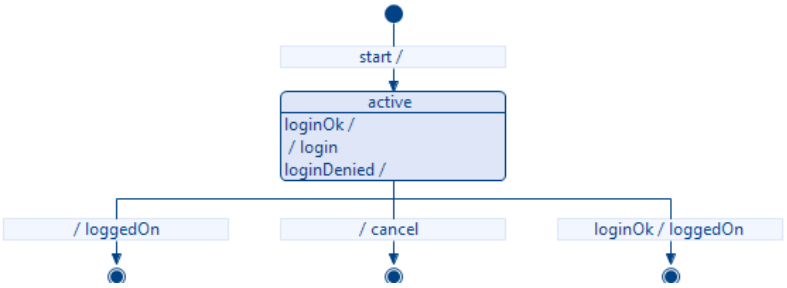
Figure 4.9: Behavior of Server Connection Dialog and Activity screen showing the dialog window.

4.3.2 Login App

The Login App block is the component on the client side that collaborates to perform the Login Service. It executes the service consumer role of the Login Service collaboration. Login App contains two distinct blocks; *Android Login UI* and *Login Service Proxy* block. This decomposition leads to a separation of user interface concerns from service collaboration concerns [KKB09]. Figure 4.10 shows the internal behavior and ESM of the block.



(a) Internal behavior of the Login App block



(b) ESM of Login App block

Figure 4.10: Behavior and ESM of Login App.

The Android Login UI provides a user interface with which the user can log into the system. The behavior of this block is to encapsulate all interactions from and to the user as well as manage operations of interface elements such as windows and buttons. Considering the ESM behavior of the block, after the block is started through the parameter node *start*, it emits a token via *login* pin. This sends the login data to the Login Service Proxy which sends the data to the Login Service located in the server. In this active state, the block waits to receive a token via either its *loginOK* or *loginDenied* pins. Based on this the block will either forward a successful login data via *loggedOn* pin or terminate via *cancel*.

The Login Service Proxy is a service discovery block and its function is to discover the Login Service registered in the Registry. It receives login data via *inPincode* pin which is sent to the Login Service in the server and receives response from the Login Service which it emits via *access* pin. Noteworthy from the Login App block is that the Login Service Proxy is started first and it is only when it emits a token via *ok* pin that the *start* pin of the Android Login UI is triggered. The reason for this design method is to ensure that the Login Service is discovered by the Login Service Proxy first before the user is presented with the user interface.

4.3.3 City Guide UI

After a successful login, a data flow is sent via the streaming pin *loggedIn* to start the main user component which is the City Guide UI block. The block itself is started prior to this via its *start* pin. The reason for this is to make sure all components which are implemented as service providers are registered immediately when the City Guide App is started. It is important to prioritize such components so that they are registered as soon as possible. This ensures that service consumers that intend to find these services can do so immediately they contact the Registry. The internal behavior of the City Guide UI is shown in Fig. 4.11.

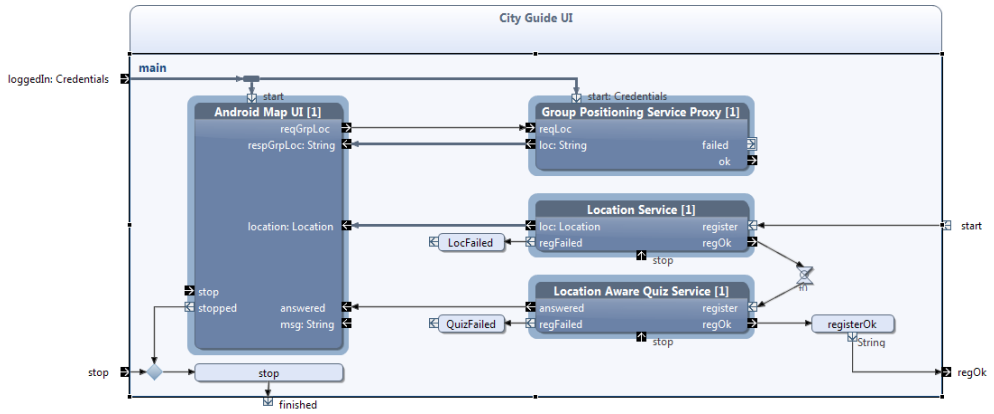


Figure 4.11: Internal behavior of the City Guide UI block.

The *Android Map UI* block encapsulates the main user interface behavior presented to the user. Other component blocks (*Group Positioning Service Proxy, Location Service and Location Aware Quiz Service*) in the City Guide UI interact with the Android Map UI block throughout its lifecycle. The user interface is a map display showing the current location of the user on the map.The Location Service uses the Generic Service block to register its service with the Registry. Its function is to use the location technology capability of the mobile device to periodically get the position of the device. This Location Service block is the user component implemented as a service provider that performs the location service collaboration with the server. In addition to periodically supplying the Location Service Proxy at the server with location data, it emits the location data via its *loc* pin which is fed to the Android Map UI block to update the map.

The Group Positioning Service Proxy block is the user component that performs the group positioning service collaboration. It contains a service

discovery block which it uses to discover the Group Positioning Service at the Registry. Whiles the map is being displayed on the user's screen, the user can request for the position updates of other group participants by going to the options menu and choosing "Show Group Members". With this done a token is sent via *reqGrpLoc* pin to the *reqLoc* pin which then sends a remote message to the server requesting for the updates. The received updates are emitted through parameter node *loc* and fed back to Android Map UI via *respGrpLoc* pin. The Android map UI refreshes itself and the position information of the user and other group participants are displayed on the map, represented as icons.

The Location Aware Quiz Service block is used for the quiz service collaboration. This block contains a Generic Service block with which it registers its service with a service name "quiz". In this collaboration the server component *Location Aware Quiz Service Proxy* forwards questions to the user when the user is near a point of interest. The Location Aware Quiz Service receives the questions and presents the user with an interactive quiz user interface with which the user answers the set of questions. The details of this user interactivity is encapsulated in the Android Quiz UI block which is not further explored here. The answers given by the user are forwarded back to the Location Aware Quiz Service Proxy using the *send* pin of the Generic Service. The Android Quiz UI terminates after the quiz session and the user is taken back to the Android Map UI. The internal behavior of the Location Aware Quiz Service is shown in Fig. 4.12.

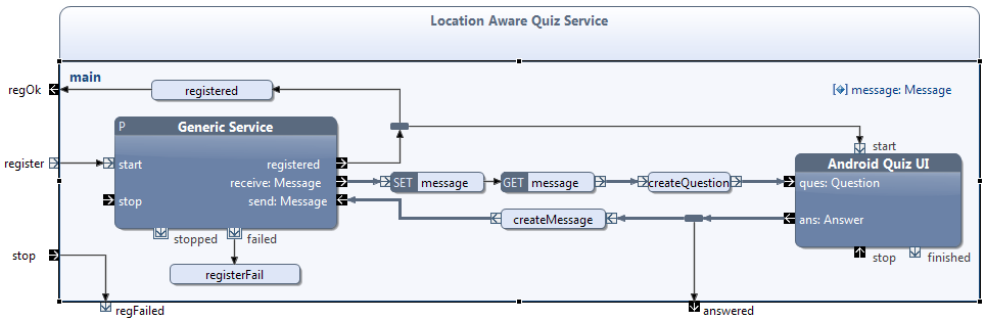


Figure 4.12: Internal behavior of the Location Aware Quiz Service block.

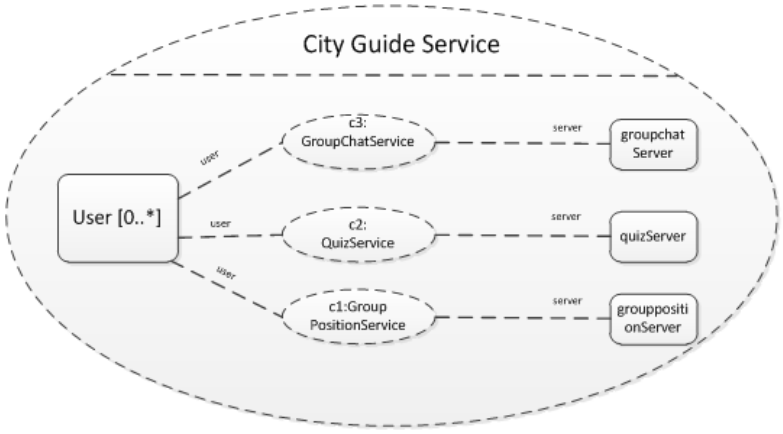
Chapter 5

Introducing New Components

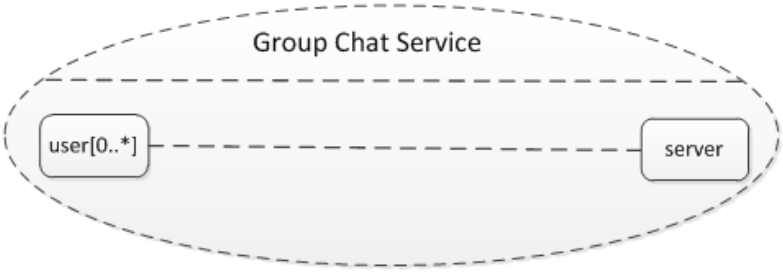
In this chapter we present the components for the two new services implemented, i.e. IM and Group Chat service. In our design, the IM and Group Chat services use a similar architecture with differences only in their functionalities. Whereas the IM enables two group participants to communicate at a time, the Group Chat allows all participants to communicate at the same time. We will illustrate the underlying architecture of our design using the Group Chat platform. In Sec. 5.1 we introduce the Group Chat service collaboration and use examples to illustrate its service behavior. We also explain the communication principle used within the Group Chat service and show how we overcame a limitation imposed on our design by the service discovery mechanism used in deploying the City Guide application. Section 5.2 and 5.3 present the building blocks of the Group Chat service for the server system and client system respectively. Here we show how the building blocks are combined with existing blocks within the City Guide application. In Sec. 5.4, we present an overview of the IM service components and show how they are integrated within the City Guide application. We also highlight the specific building blocks that exhibit differences in behavior from the Group Chat service.

5.1 Group Chat Platform

Group Chat is a basic support service that is important for the city guide application since the concept of this application is to support situated collaborative learning. In SCL there is learning through participation, problem solving and fun [KB09]. With the Group Chat platform, players can collaboratively answer the questions with group members during the question-



(a) City Guide Service with Group Chat service



(b) Group Chat service collaboration

Figure 5.1: City Guide Service and Group Chat Service collaboration.

answer session.

The Group Chat service involves collaborations among many users. The UML collaboration of the City Guide Service is extended to capture the Group Chat service behavior. Figure 5.1 shows the city guide service collaboration and the Group Chat service collaboration. The collaboration role *GroupChat Service* interacts with the users and the server, and is described by the collaboration use c3 for the Group Chat Service.

5.1.1 General Architecture

The purpose of the Group Chat functionality is for users to communicate and be able to send and receive messages among one another simultaneously and in real time. In a typical scenario for the City Guide application, we assume group participants in the game are scattered in the city with each in search of a point of interest. Supposing one member locates a point of interest, he is presented with questions in the quiz service. He could immediately start a Group Chat with the other group members where they could discuss and

solve the questions before he submits a solution.

Considering that the distributed agent based platform is used in deploying the City Guide application, it is important to illustrate how the agent structure and roles, bound to the Group Chat service interact. We will use block diagrams to illustrate this. In Fig. 5.2, a reusable role GCp in the client system participates in Group Chat with a corresponding GCs role found in the server system.

We have chosen to implement the Group Chat platform using the service discovery mechanism explained in Chap. 3. For this, the GCs role is implemented as the service provider and will therefore be registered with the Registry. We name this as *GroupChat Service* building block in our design. The GCp role is implemented as the service consumer and so has a *GroupChat Service Proxy* with which it uses to discover GCs. Since there will be multiple users of the system at a time, the Group Chat service must be made public when registering in the Registry such that each *GroupChat Service Proxy* of the client can discover it.

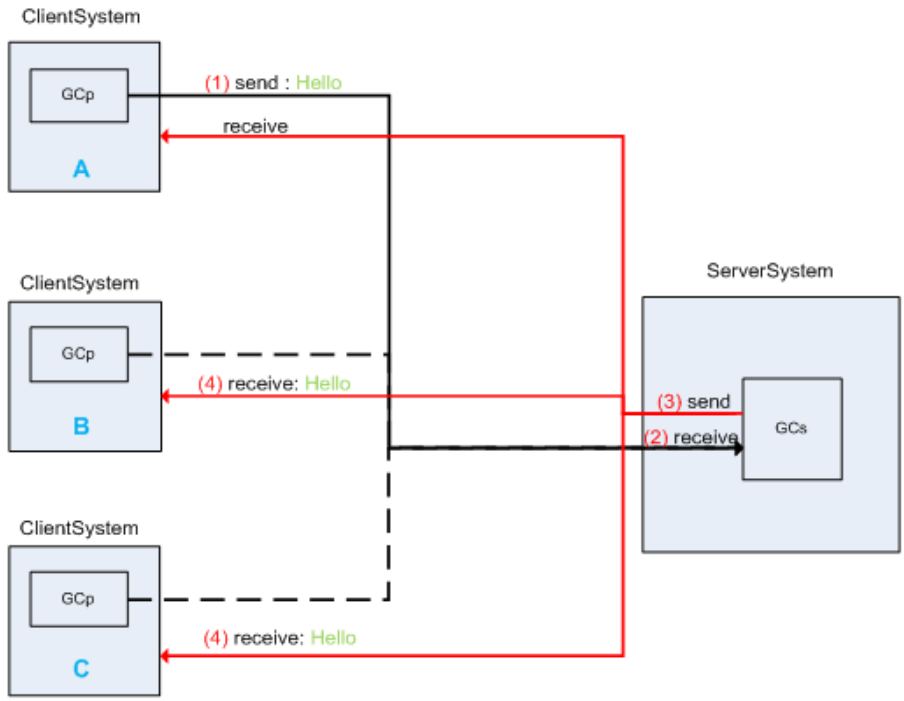


Figure 5.2: Illustration of Group Chat service behavior (user A sends a message).

The main function of the GCs is to route messages among the clients. This

is achieved by receiving messages from the various group members and using its internal mechanism to send the messages to the other group participants. In Fig. 5.2, three users in the application use the Group Chat functionality. User A's GCp role sends a *Hello* message. The message is received by the GCs role in the server which then forwards the message to users' B and C. Similarly user B sends a *Hi* message which will be received by the GCs role in the server system. The message is then forwarded to users' A and C as shown in Fig. 5.3.

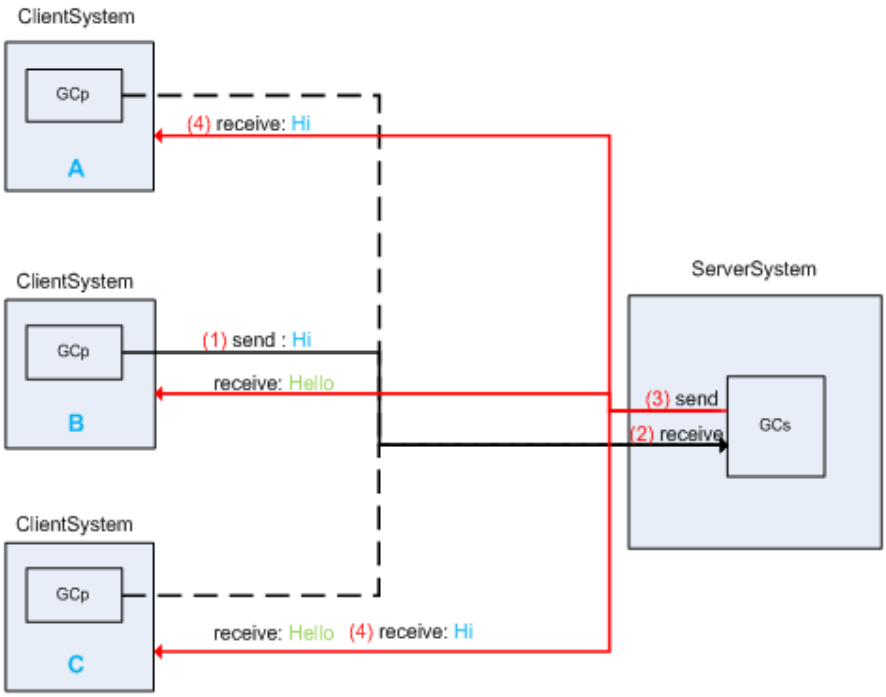


Figure 5.3: Illustration of Group Chat service behavior (user B sends a message).

5.1.2 Communication Principle for the Group Chat Service

From the block diagram illustrating the Group Chat functionality, we can observe that the GCs role in the server has the responsibility of a message router. It can also be regarded as a chat server whereby it routes messages sent from one client to the others. In order for clients to receive Group Chat messages from the server, the GCs role must know of the existence of

the clients in the first place. Clients should have a means of subscribing to the chat server (GCs) such that when a message is received by the GCs, it can look up its routing table and forward the message to clients that are subscribed to receive messages from it.

To illustrate this idea further, we consider a sequence diagram of a simple chat service in Java shown in Fig. 5.4. As shown, each client creates a new listener object. The client then registers its name and listener with the server. When a message is sent from a client to the server, the server will forward the message to the listeners of all clients that have been registered with it (*notifyMessage*). This will be a continuous process for messages sent from all clients.

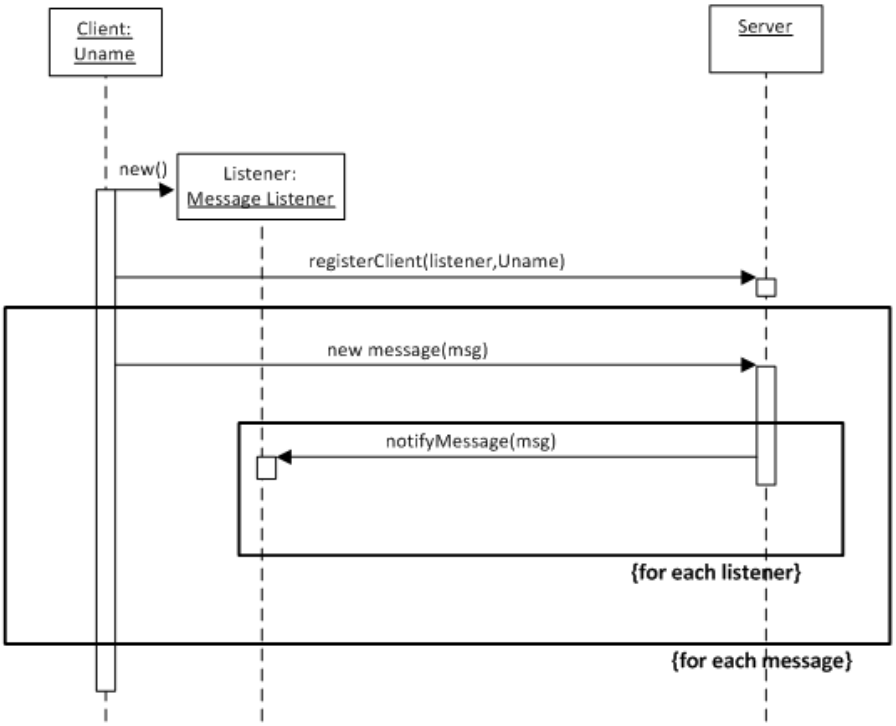


Figure 5.4: Group Chat sequence diagram.

Using this idea for implementation in our architecture proved challenging. This is because, with the service discovery mechanism being used for the Group Chat, the GCs and GCp roles have a service provider-service consumer relationship. This means that the GCp component must first initiate communication session (*send* action) before the GCs can respond by providing its service. Assuming A sends the first message, the GCs in the server

will receive the message and therefore know of the existence of A's GCp role. The function of the GCs is to forward the message to B and C however it cannot achieve this since it will only know of the existence of the GCp roles of B and C if they also send a message to it. With this limitation the Group Chat functionality cannot be achieved since the GCs should be an active service [SB08] that can respond to user initiatives while it takes initiatives towards other users as well.

From the block diagrams in Fig. 5.2 and 5.3, it can be seen that the messages received by the users terminate just on the outer block. This was purposefully done so as to first explain the communication structure and the limitations imposed by the service discovery mechanism.

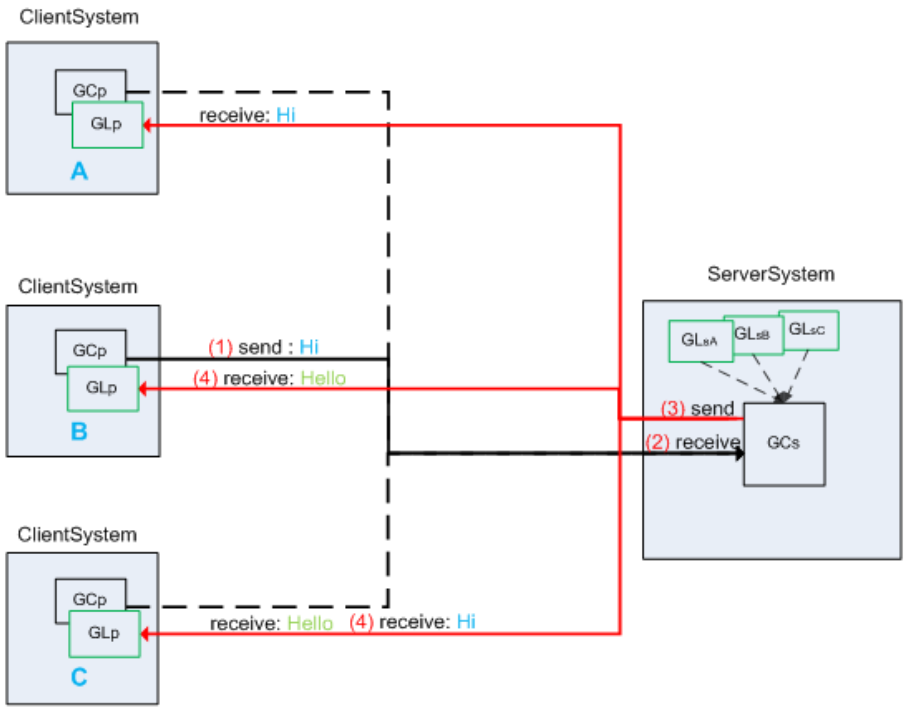


Figure 5.5: Illustration of Group Chat service behavior (with Address collaboration service roles).

To solve this limitation, we have introduced two new reusable roles GLp and GLs in the client and server systems respectively as shown in Fig. 5.5. The function of the GLp role is to inform the server of its existence immediately the client is logged in. It achieves this by collaborating with the GLs role. In the server, each client will be created its own GLs role (GLs_A, GLs_B, *and* GLs_C) as shown in the Fig. 5.5. The GLs components for each client

then establish a relationship with the GCs role by providing it a reference to the GLp of their clients. The GCs keeps a table of the references and will be updated whenever a new client arrives in the system. With this model, the GCs role will now know of the existence of all clients that want to receive Group Chat messages. When it receives a message from any client, it simply checks its table and forwards the message to the GLp of all clients except the GLp of the sender.

The GLp and GLs roles perform a collaboration. This collaboration, we call Address Service collaboration is shown in Fig. 5.6. The collaboration role GLp is implemented as the service provider whiles the GLs role is implemented as the service consumer. This collaboration provides the service of registering a receiver for each client with the GCs in the server. The GLs then has a dynamic role binding with the GCs to perform the Group Chat service collaboration.

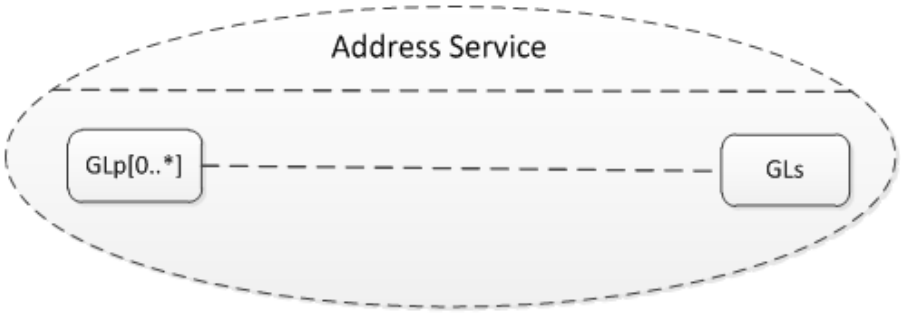


Figure 5.6: Illustration of Group Chat service behavior (with Address collaboration service roles).

5.2 Server Components of Group Chat Service

This section presents the building blocks for the Group Chat service that are located in the server side of the City Guide application. The *City Guide Server* which is the system containing building blocks for the server components has already been explained in Sec. 4.2. The aim now is to integrate the Group Chat service blocks into the City Guide Server. Two building blocks namely *GroupChat Service* and *Block_Address Service Proxy* are introduced. Figure 5.7 shows a pictorial view of the City Guide Server system and the specific blocks in which the *GroupChat Service* and *Block_Address Service Proxy* will be located.

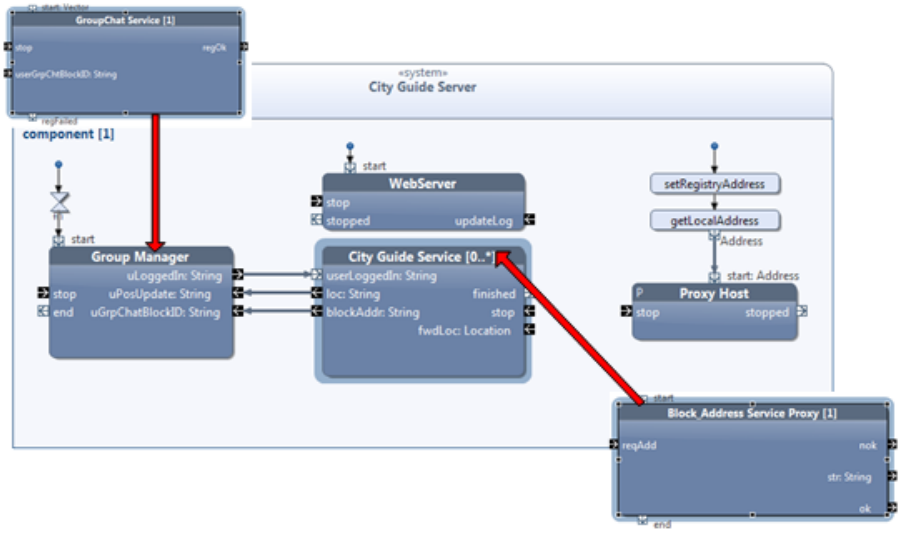


Figure 5.7: City Guide Server showing the location of server components of the Group Chat service.

5.2.1 Block_Address Service Proxy (GLs)

This building block forms part of the components used to perform the *Address_Service* collaboration explained in Sec. 5.1.2. The block is a typical service discovery block and therefore contains a *discover service* block which it uses to discover the *Block_Address Service* from the Registry. The main function of the *Block_Address Service Proxy* is to initiate a request to the *Block_Address Service* located at the client side to provide a reference of itself. The reference provided is a unique address of the *Block_Address Service*.

Placing the *Block_Address Service Proxy* block in the City Guide Service block is required for the server to have a reference to each client for the Group Chat service collaboration as has been explained. Since a new *City Guide Service* instance is created for each user after a successful login procedure, the *Block_Address Service Proxy* component will also be created for each new user with which it can receive the Block_Address Service address from the client.Each client therefore has its own *Block_Address Service Proxy* component in the server with which it performs the Address Service collaboration with. Figure 5.8 shows the new design of the City Guide Service block.

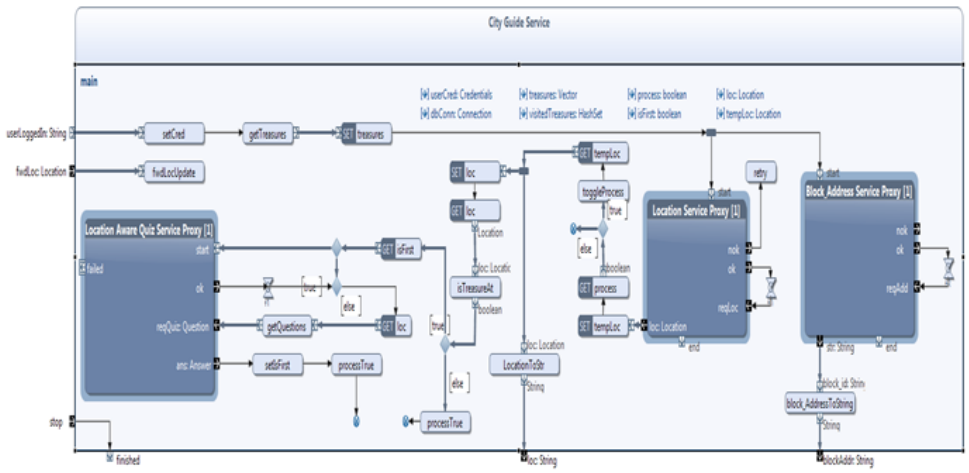
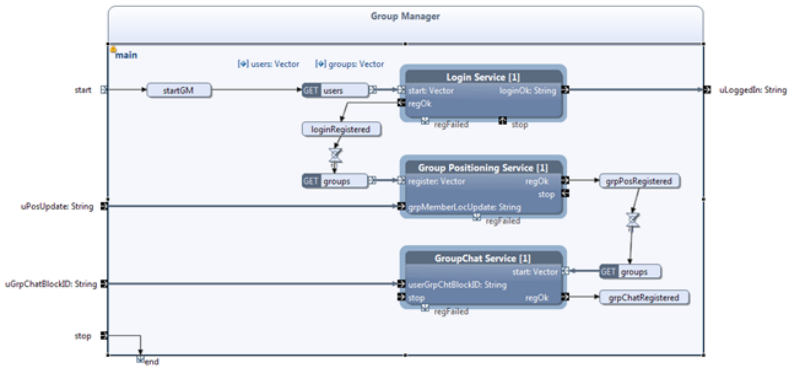


Figure 5.8: Internal behavior of City Guide Service with a new component.

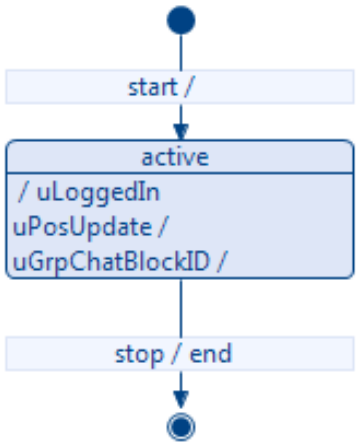
When an instance of the City Guide Service block is created for each user via the *userLoggenIn* pin, a *fork* inserted sends a control flow to start the *Block_Address Service Proxy* block. The *discover service* block uses the service name "Block_Address" to discover the *Block_Address Service* at the Registry and on finding it emits a token via *ok* pin. A control flow through a zero timer is immediately used to emit a token through the streaming pin *reqAdd*. This action then sends a remote message to the client's Block_Address Service prompting it to provide it with an address. The Block_Address Service Proxy emits the received requested address from the client via *str* pin. A special operation *block_AddressToString* is performed to combine the clients username and the address as a *String* value which is then sent out as a data flow via the *blockAddr* pin. From Fig. 5.7, it can be observed that the *blockAddr* pin from the City Guide Service emits a data flow to the *uGrpchatBlockID* pin in the *Group Manager*. The *GroupChat Service* block is shown next and its relationship with the *Block_Address Service Proxy* explained.

5.2.2 GroupChat Service (GCs)

The *GroupChat Service* block constitutes an important component in the Group Chat service collaboration. It has a logical equivalence to the GCs role of the server system illustrated in Fig. 5.2. The block is implemented as a service provider and therefore contains the Generic Service block which is used for the service registration. The function of the service it provides is to receive chat messages from users and forward these messages to other group participants currently signed into the application. The *GroupChat Service*



(a) Group Manager with new block *GroupChat Service*.



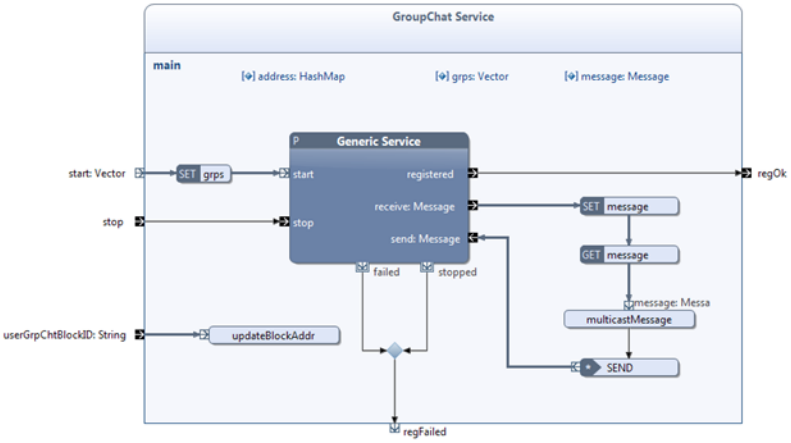
(b) ESM of Group Manager

Figure 5.9: Group Manager behavior and ESM.

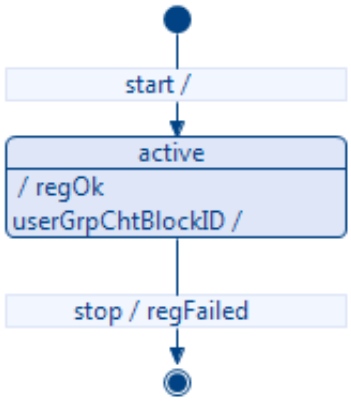
block is placed in the *Group Manager* block so that it can be accessible to all clients who use the service. In addition additional resources such as group and player configurations are contained in the *Group Manager* hence it will be easier for the *GroupChat Service* block to access such information when performing its functionality. Figure 5.9 shows the behavior and ESM of the *Group Manager* block with the *GroupChat Service* block added.

Considering the internal behavior of the Group Manager block, the *GroupChat Service* is started via the *start* pin after a token is emitted via *regOK* pin from the *Group Positioning Service*. The two services are independent of each other and can therefore be started separately. However the *GroupChat Service* is dependent on the *Login Service* and so should be started only after the *Login Service* has registered successfully and emitted a token via its *regOK* pin. The introduction of a new streaming pin *uGrpChatBlockID* to the *Group Manager* passes a data flow to the *GroupChat Service*. The ESM

of the *Group Manager* will change with the addition of this new pin. In the active state the *Group Manager* block will in addition receive tokens via the streaming pin *uGrpChatBlockID*.



(a) Internal behavior of *GroupChat Service*.



(b) ESM of *GroupChat Service*

Figure 5.10: GroupChat Service behavior and ESM.

In the *GroupChat Service* behavior shown in Fig. 5.10, the Generic Service uses the service name "GroupChat" to register the service with the Registry. The block will receive data via the *userGrpChtBlockID* pin from the City Guide Service instance created for all successfully logged in clients. The data received is the client's username and its address obtained by the *Block_Address Service Proxy* (explained in 5.2.1). The purpose of this information is to register each client's address with the *GroupChat Service*. The *GroupChat Service* performs an operation *updateBlockAddr* to do this registration. In the operation a hash map is kept for each client entry. A key/value relationship is maintained which stores each client's username with

its address. The ESM of the block will be such that in the active state, it can receive tokens via the *userGrpChtBlockID* pin hence new users can register their addresses with the *GroupChat Service* for as long as the server is running.

The Generic Service receives remote messages from clients subscribed to the *GroupChat Service* via the *receive* pin. The client is the user wanting to send chat messages to the other group participants. The received message is set as a variable in the block and fed as input to the Java operation *multicastMessage*. It is in this operation that the central logic of the *GroupChat Service* is executed.

Before the details of the *multicastMessage* operation are explained, it will be important to explain the structure used by a remote client to send a Group Chat message. A Group Chat message sent by a user follows the structure: SENDER#MESSAGE where the SENDER value is the username of the sender of the message and the MESSAGE value is the actual information sent by the user. The Java code used in the *multicastMessage* operation is shown in Appendix A.

The function of the operation is to forward the chat message from one user to the other users using their addresses which have been registered with the *GroupChat Service*. After receiving the message and extracting the message payload, it is split from its SENDER#MESSAGE structure. A *for* statement is used to iterate over the hash map containing the usernames and their respective addresses. What this part of the operation does is to fetch the entry value of each user in the hash map. This value is the address which is stored as a *String* value. This value is converted to the *Address* data type. A new message is composed (*response*) with the payload being the same message received from the sender. In composing the new message, its *set.Receive()* value is then set as the *Address* of the hash map entry value. An internal notification is then sent to the *GroupChat Service* block using the *sendToBlock()* method with the signal "SEND" and value of the message *response*. The *GroupChat Service* block receives the signal and forwards the message to the Generic Service via the *send* pin.

Whenever a new message is received by the multicastMessage, the *for* loop will be executed for all entries in the hash map, except for the address of the sender of the original message. This ensures that when a message is sent by a user, it will be forwarded to the other group participants except the sender itself. In summary the *GroupChat Service* can be seen as a message router which receives incoming messages from various users and forwards the messages to other users based on their receiver addresses previously registered in its hash map table.

5.3 Client Components of Group Chat Service

In this section, the set of building blocks on the client side that collaborate to perform the group chat service collaboration are presented. The new building block components are introduced within the *City Guide UI* block of the *City Guide App* system. This is so because the *City Guide UI* is the main block executed after a successful authentication of a client by the server and since the group chat service is only usable by logged in clients, it is logical to locate its components within the *City Guide UI* block. The new building block introduced is called the *GroupChat App*. Figure 5.11 gives an overview of the *City Guide App* with the *GroupChat App* block.

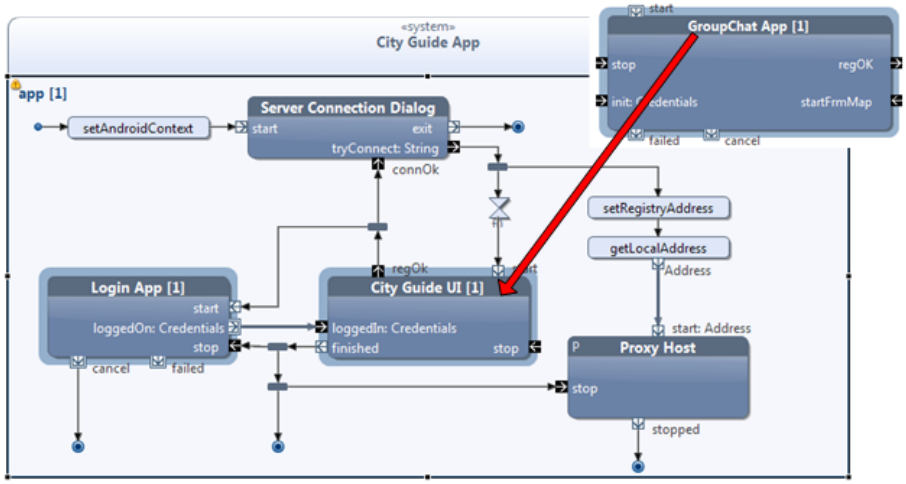


Figure 5.11: City Guide App showing the location of GroupChat App.

The *GroupChat App* block encapsulates the functional behavior of the group chat collaboration involving several other components. Its details will be explored subsequently. In the *City Guide UI* block shown in Fig. 5.12, the *GroupChat App* has parameter nodes with which it uses to interact with other components in the block. The *GroupChat App* is initialized via its *start* pin, in the same process that starts the City Guide UI block. This is so because the *GroupChat App* contains a component which is implemented as a *service provider* which requires to be registered immediately the client side application is started.

When the client is logged in successfully, the user credentials received via the *loggedIn* pin are also provided to the *GroupChat App* via its *int* parameter

node. With this data, the username can be obtained which is used to identify each client taking part in the Group Chat collaboration.

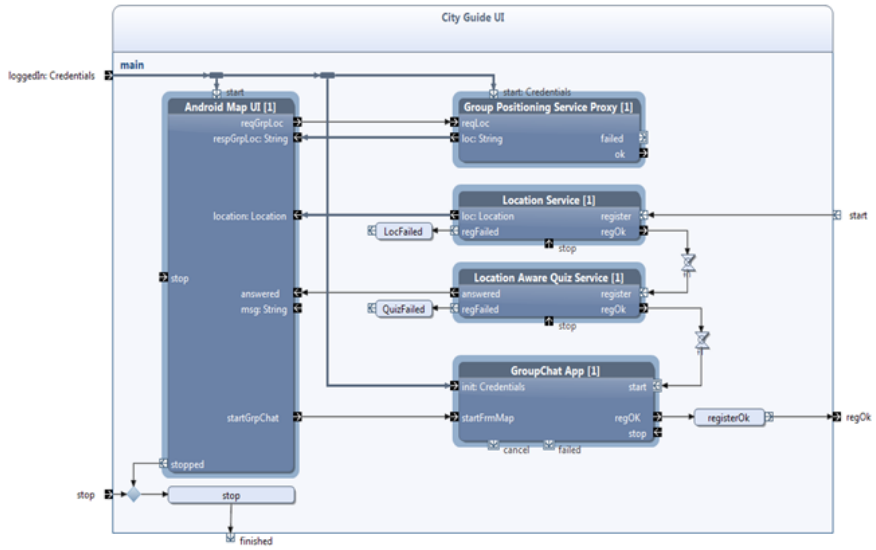
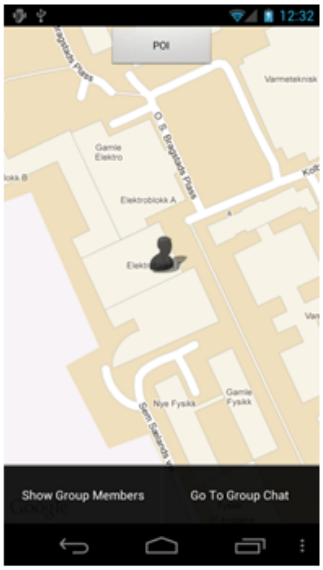
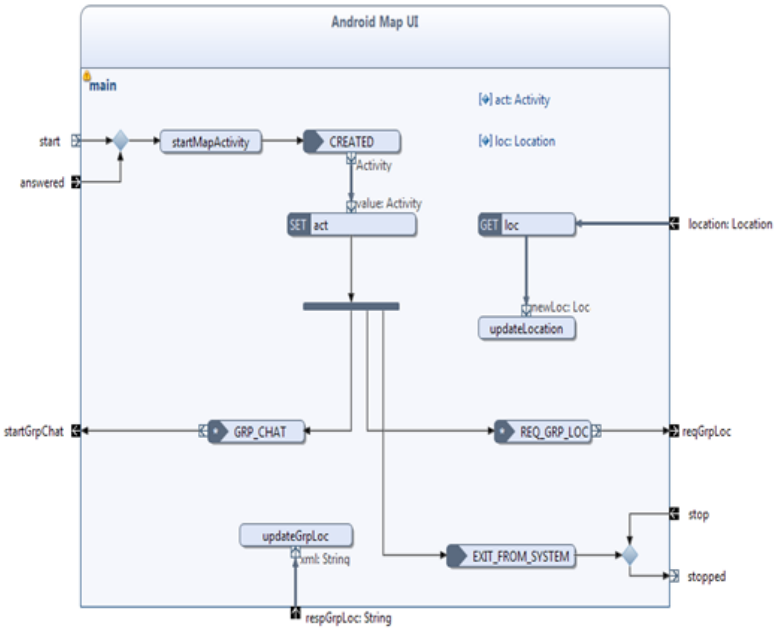


Figure 5.12: City Guide UI block with GroupChat App block.

The *Android Map UI* block which encapsulates the main map user interface presented to the user during the running of the application, interacts with the *GroupChat App* by sending a control flow through its streaming parameter node *startGrpChat* to the *startFrmMap* pin of the *GroupChat App*. On the map display, the client uses its options menu to select the item "Go To GroupChat". This sends a notification to the *Android Map UI* block which is received by the reception signal "GRP_CHAT". This initiates a control flow which is emitted via the *startGrpChat* pin. The purpose of this action is to open the Group Chat user interface which is a component block located in the *GroupChat App*. A screen shot of the map display with its options menu and the internal behavior of *Android Map UI* showing the new reception signal "GRP_CHAT" is shown in Fig. 5.13.



(a) "Go To Group Chat" is a menu item pressed to open the Group Chat UI.



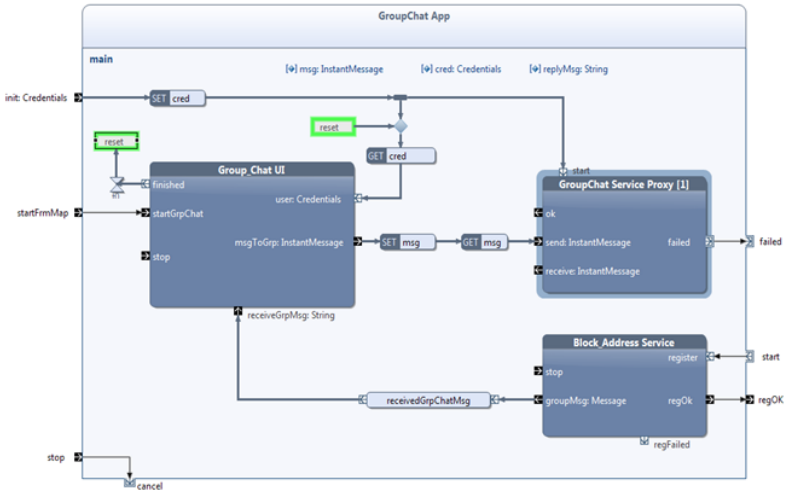
(b) Android Map UI block with new signal reception event "GRP_CHAT".

Figure 5.13: Options menu item of map display and Android Map UI block behavior.

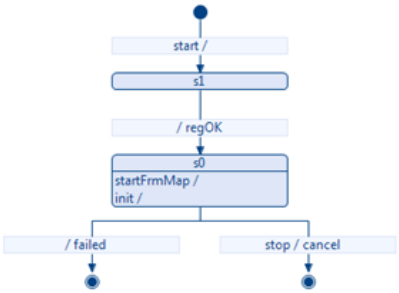
5.3.1 GroupChat App

The internal behavior and ESM of the *GroupChat App* is shown in Fig. 5.14. It consists of three building blocks namely *Group_Chat UI, GroupChat Service Proxy and Block_Address Service*. These building blocks are designed such that there is separation of user interface behavior from actual service behavior of the group chat service. The *Group_Chat UI* encapsulates all interactions to and from the user. The *GroupChat Service Proxy* and the *Block_Address Service* components are used in defining the service behavior of the group chat service collaboration.

From the ESM of the *GroupChat App*, the block is initialized via the *start* pin to purposefully register the only service provider component in the block. After a successful registration of the *Block_Address service*, a token emitted via *regOK* pin transitions the block from state *s1* to *s0* where the block will remain active until it is terminated via *stop* or *failed* pins.



(a) Internal behavior of *GroupChat App*.



(b) ESM of *GroupChat App*.

Figure 5.14: GroupChat App behavior and ESM.

5.3.2 Group Chat UI

This block models the user interface presented to the user during the Group Chat process. The behavior of user interfaces is normally triggered by distinct events such as triggers from signal receptions and timeouts, as well as direct actions from users like the tap on a button [KKB09]. Figure 5.15 shows the layout the Group Chat user interface designed in Android.

The layout file is created using graphical editor provided by the Android SDK. The screen layout shown is called Android activity. The user interface has two text fields. The larger text field is where ongoing messages will be received. These will be events that will update the user interface. The smaller text field is where the user writes messages. The "send" button is used to trigger the service action of sending messages.

The lifecycle of the user interface activity is controlled from within the *Group_Chat UI* and it is only the ESM of the *Group_Chat UI* that constrains the events affecting the lifecycle of the user interface activity. The ESM of the block is shown in Fig. 5.16. It was important in our design to keep the *Group_Chat UI* block in its *active* state at all times because it is in this block that the chat UI can be opened and used to send messages. In addition messages can be received at any time and the chat UI will be updated with the received messages.

From the behavior of the *GroupChat App* block and the ESM of the *Group_Chat UI*, the *Group_Chat UI* can only terminate via the *finished* pin. In the event that the *Group_Chat UI* block terminates, a data flow through a zero timer is used to set a *connector merge* named *reset* (shown in green). The second connector merge is placed purposefully so that it can reinitialize the *Group_Chat UI*. This is because the connector merge is in the same state and the two can be viewed as a single point which allows a token flow to start the *Group_Chat UI*. This means that each time the *Group_Chat UI* block terminates, it is restarted again and so will always be in the *active* state.

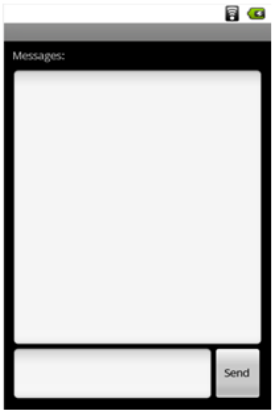


Figure 5.15: Layout of Group_Chat UI.

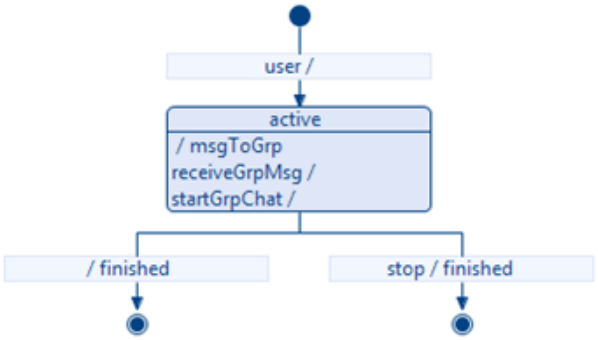


Figure 5.16: ESM of Group_Chat UI.

(a) Client Starts a Group Chat with other users

The internal behavior of *Group_Chat UI* is shown in Fig. 5.17. The block is initialized when it receives user credentials through the *user* pin. The UI is opened via *startGrpChat* pin which performs the *startActivity* operation. This operation launches the main activity class (*MessagingActivity Class*) of the Group Chat UI (see Appendix B). The Group Chat UI then opens on top of the map display, with the map UI which was originally displayed moved into the background.

After the launch of the Group Chat UI, an internal notification "CREATED" is received by the block and this sets the newly created Android activity as a variable. After this stage a number actions could be perform which are based on the client's use of the service. An operation *setUsername* is executed to

pass the user's credentials (username) to the UI so that it can be identified as the sender of a message. A user who wants to send a Group Chat message composes the message in the lower text field and presses the "Send" button. A series of events take place simultaneously.

Immediately the message is sent, the user will see the message appear in the upper text field in a format *username: message*. The username will be his own name and the message will be what he has sent. The lower text field is subsequently cleared and can be used to compose more messages.

In the same action, an internal notification with signal "SEND" is sent to the *Group_Chat UI* block from the UI activity. Along with the notification is data received from the UI activity. The data received is composed in the format SENDER#MESSAGE and set as a payload for the data structure *InstantMessage* which is forwarded through the *msgToGrp* node.

Reply messages sent to the client from other clients are forwarded to the *Group_Chat UI* via the *receiveGrpMsg* pin. When the message is received, a *Boolean* operation *checkSession* is performed to check whether the activity variable is set or not. Since the UI is active and can be seen on the phone screen, the operation returns a *true* value which enables the *ActiveSessionMessages* method to be executed. This operation displays the received message on the UI.

A user can close the Group Chat UI by pressing the back button on the Android terminal. When this is done the activity class of the Group Chat UI sends an internal signal "EXIT" to the *Group_Chat UI* which performs the *destroyAct* operation and terminates the block via *finished* pin. In the *destroyAct* operation, the activity variable of the Group Chat UI is destroyed by setting its value to *null*. Doing this means that the client is no more engaged in any group chat session with other participants.

(b) Client receives a Group Chat message from another client

While a user might be busy doing other things with City Guide application such as exploring points of interest or engaged in the interactive quiz, he can also receive Group Chat messages from other group participants. Already the *Group_Chat UI* block would be initialized via the *user* pin when the application is started. Messages sent to the client will be received by its *Block_Address Service* and forwarded to the *receivedGrpMsg* pin. The *checkSession* operation in this instance would return a *false* value. This is because the Group Chat UI has not been started so its variable in the *Group_Chat UI* would be *null*. The next action followed is to forward the message to the *Notification* block via its *start* pin.

The *Notification* block is taken from the Arctis library of building blocks.

Its internal details would not be discussed here. The function of this block is to add an icon to the system's status bar and a notification message in the notifications window. The notification is also configured to alert the user with a sound. When the user selects the notification, a token is emitted via the *pressed* pin and a control flow sent to the *merge* node causes the *Group_Chat UI* to open on top of whatever activity is being displayed on the phone screen.

When the UI opens, it will update the upper text field with the received message. To do this it performs an operation *checkIsFirst* which is a *Boolean* logic preset to check if the received message was initiated as a result of the action from the *Notification* block. if a *true* value is returned, a notification signal "SHOW" is received by the *Group_Chat UI* to execute the *updateChatUI* method. The UI is thus updated with the received message. The user can subsequently continue to send and receive messages as explained previously.

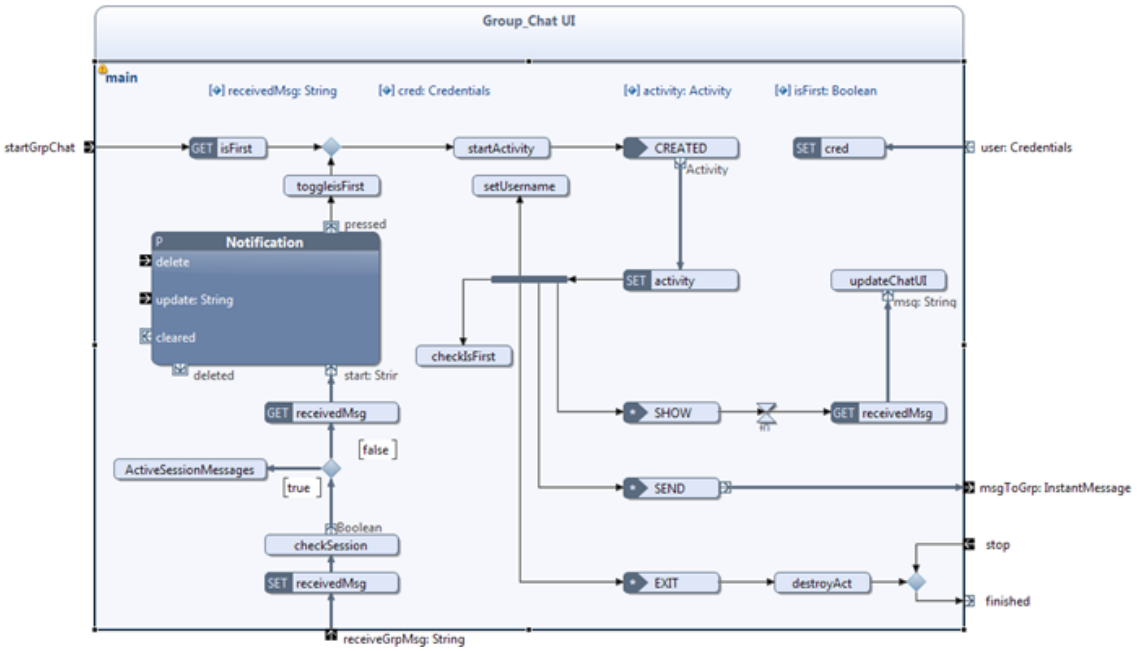


Figure 5.17: Internal behavior of Group_Chat UI.

5.3.3 GroupChat Service Proxy

This is a service discovery block and is what is used by the client in the group chat service collaboration. Its logical equivalence in the block diagram

is the GCp role of the client system. The discover service block uses the service name "GroupChat" to discover the *GroupChat Service* at the Registry. The main function of this block is to send messages to the *GroupChat Service* located in the server. These messages will be composed in the *Group_Chat UI* by the user and passed to the *GroupChat Service Proxy* for onward transmission to the *GroupChat Service*.

5.3.4 Block_Address Service

This component is used in the address service collaboration. It has a logical equivalence to the GLp role of the client system in the block diagram. Its functions are in two parts. The first is to collaborate with the *Block_Address Service Proxy* of the client in the server and provide it with the block's address. The second function is to receive messages from the *GroupChat Service* which are intended for the user. The internal behavior of the block is shown in Fig. 5.18.

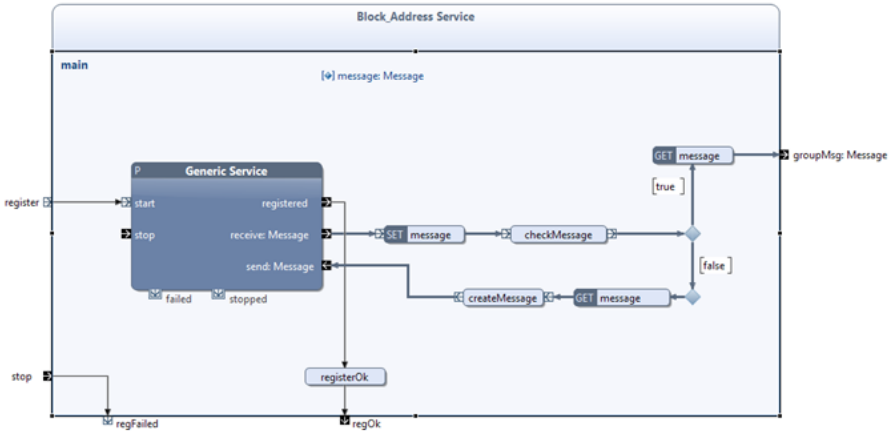


Figure 5.18: Internal behavior of Block_Address Service Proxy.

The Generic Service registers this service with the Registry using the service name "Block_Address". The *Block_Address Service* block is among the first components started in the client system along with other service provider components. The first message received by the block would be from the client's *Block_Address Service Proxy* in the server. An operation *checkMessage* verifies this and returns a *false* value. With this an operation *createMessage* returns a message to the *Block_Address Service Proxy* with the payload of the message containing the local address of the *Block_Address Service*.

Messages that are sent to a user by the *GroupChat Service* will be received in this same block. This time around the *checkMessage* operation verifies the message and returns a *true* value. The received message is then emitted as a data flow via the *groupMsg* pin which is then forwarded to the *Group_ Chat UI* (see Fig. 5.14).

5.4 IM Service Components

In this section we present the IM service components for both the server system and the client system. The architecture is similar to the Group Chat service but with some differences which are explained.

5.4.1 Server Components of IM Service

The IM service has two components in the server system namely, the *IM Service* and the *MyAddress Service Proxy* building blocks. Figure 5.19 gives a pictorial overview of how these blocks are combined with the other *City Guide Server* components.

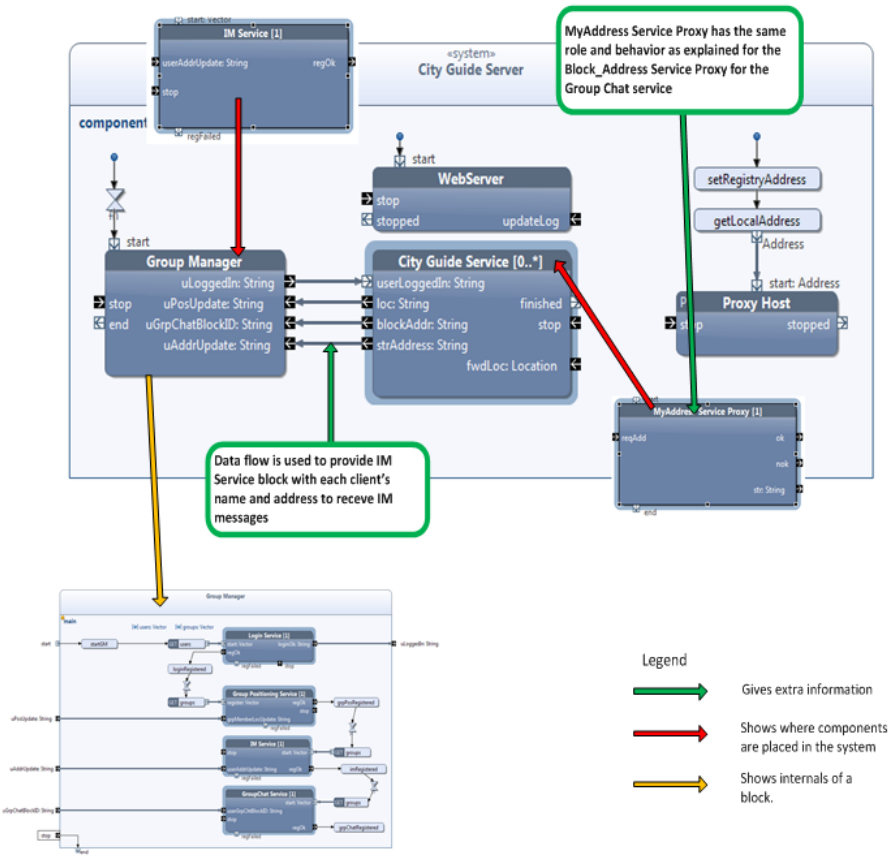


Figure 5.19: City Guide Server with components for the IM service collaboration.

IM Service

The IM service is shown in Fig. 5.20. Its internal behavior is similar to the *GroupChat Service*. Clients logged in are able to register their username and address in the operation *updateAdd* by passing data to the block using the parameter pin *userAddrUpdate*. The Generic Service will receive messages sent from the *IM Service Proxy* of the various clients. The IM message sent is of the structure SENDER#RECEIVER#MESSAGE. As can be observed, the structure used for the IM service differs from the one used in the Group Chat service. A RECEIVER value is added for the IM service. This value will be the recipient of the IM message.

The received message is fed as input to a Java operation *sendIMMessage*. In this operation, the received message is split and the RECEIVER value is extracted. It is then used to search the username/address hash map which

stores the username and address of clients registered for the IM service. The address found will be that of the recipient of the message. A new message is composed with the recipient address set in the *set.Receiver()* method of the new message. The message action is received in the *IM Service* block as an internal notification from the "SEND" signal with the value of the message. This is forwarded to the Generic Service via the *send* pin. The payload of the message sent to the recipient by the IM service will be the same as received from the sender. The *IM Service* is only to route the message to the correct recipient. It does not alter the contents of the message in any way. This is also true about the *GroupChat Service*. The *GroupChat Service* only forwards the message to other participants and does not alter the message structure. Extracting the actual message is done by the client itself. The Java code of the *sendIMMessage* operation is shown in the Appendix C.

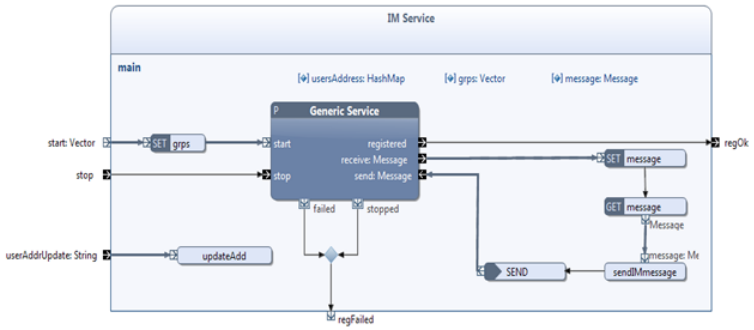


Figure 5.20: Internal behavior of IM service.

5.4.2 Client Components of IM service

Figure 5.21 shows how the *IM App* block, which encapsulates all the IM service functionality on the client system, is integrated with other components. Its relationship with the *Android Map UI* block which encapsulates the map user interface is also shown in the *City Guide UI*. In Fig. 5.22, the internal behavior of the *Android Map UI* is shown. In here, we have added a new reception signal "START_IM" which is used to select a participant displayed over the map to start an IM session with. The user is able to see on his map icons representing other group participants by using the Group Positioning Service (see Sec. 4.2.1 and 4.3.3). We adapted this to indicate presence or availability of a participant.

The user can start an IM with another participant by simply tapping on the icon of that participant. When this is done an internal notification "START_IM" is received by the *Android Map UI* block. The signal received

also carries data, value of which is the name of the participant the client intends to start the IM with. The signal with this data is passed out of the block via *startChat* pin which is then used to open the IM user interface.

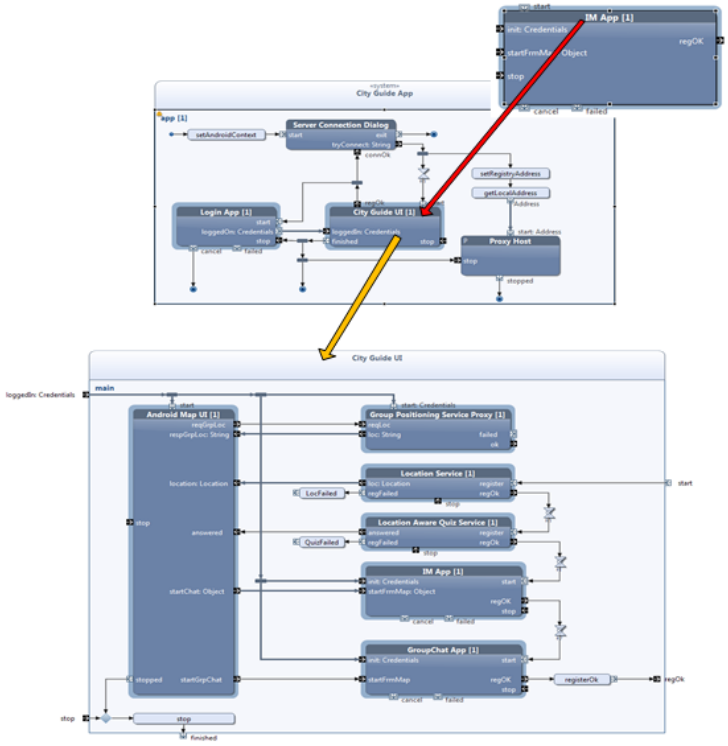


Figure 5.21: City Guide App showing components for IM service collaboration.

The internal behavior of the IM App is shown in Fig. 5.23. A UI is opened for the user where an IM message could be composed and sent. The *Instant Messaging UI* is given the username of the recipient of the IM message when it is started via the *startMap* pin. The message composed has the structure SENDER#RECEIVER#MESSAGE. The message sent in this format will be adequately handled by the *IM Service* in the server to route it to the correct recipient.

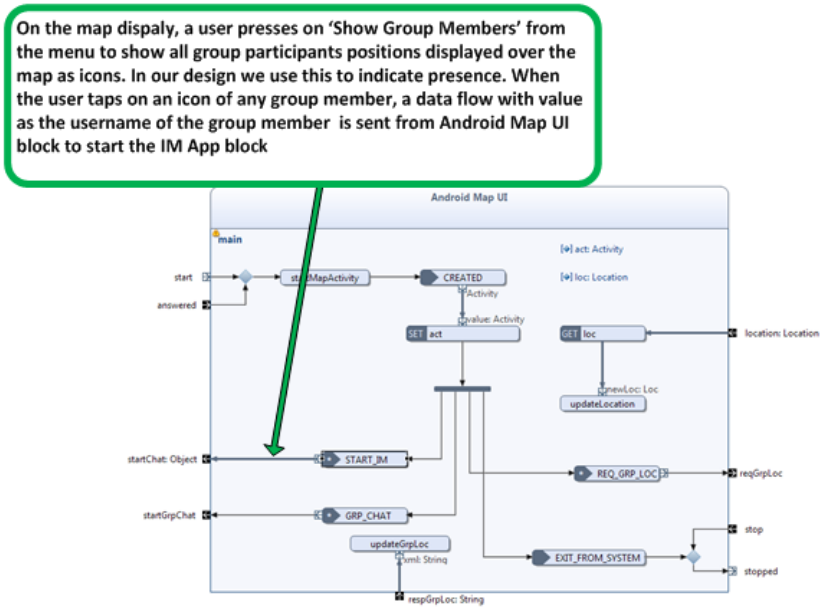


Figure 5.22: Internal behavior of Android Map UI with signal reception event to handle starting an IM.

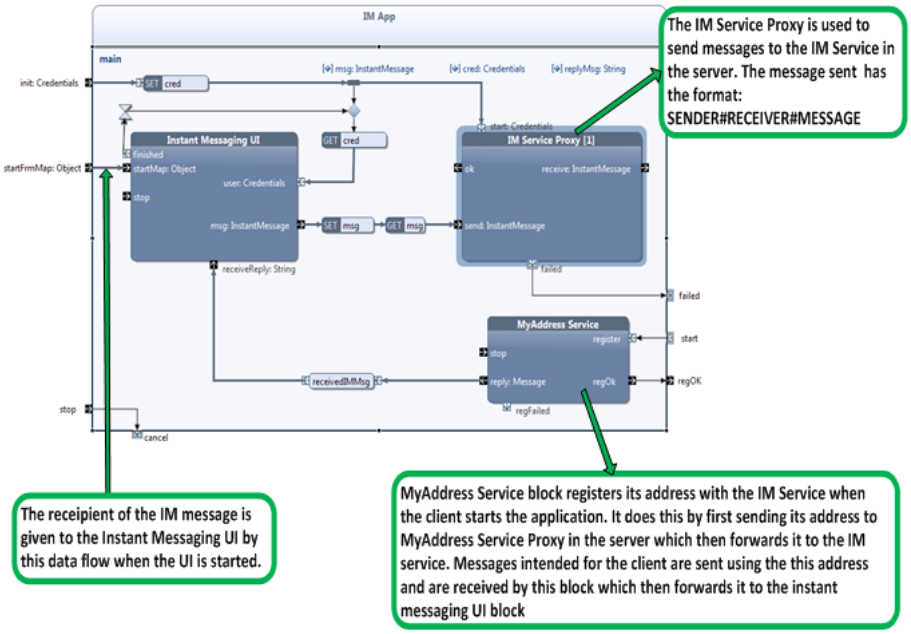


Figure 5.23: Internal behavior of IM App.

Chapter 6

Discussion

This chapter presents an analysis of the City Guide application based on the service discovery mechanism used. In Sec. 6.1 we discuss the application as a self-adaptive system and we identify limitations of how it currently works. We also discuss some alternative service discovery mechanisms that could be used for such an application. In Sec. 6.2, we identify and discuss some issues the application using a service discovery mechanism could face in the mobile environment. In Sec. 6.3, we take a critical look at our design of the IM and Group Chat services and discuss alternative solutions to our design.

6.1 Self Adaptive Computing System

The City Guide application uses the principles of self-adaptive systems. By this we mean systems and components that configure themselves and dynamically adapt to changing environments with minimal human participation. The City Guide application uses static discovery (explained in Sec. 2.5.4, *DA Discovery*). The Registry which is the centralized point of the system is configured along with the system components to ensure discovery and use of services.

Parameter adaptation is made in the Proxy Host of each of the three systems (see Sec. 3.3.3) before the application is deployed. Parameter adaptation modifies program variables that determine behavior [MS+04]. By this we mean changing values without changing components or algorithms. We achieved this by statically setting the Registry address in the client system and server system Proxy Hosts'. This ensures connectivity between the Registry, the client and the server systems.

6.1.1 Limitation of City Guide application Architecture

The system uses a centralized service description (explained in Sec. 2.5.4, *Operation Mode*). The Registry is the directory agent, the City Guide App deployed on Android devices acts as client and the City Guide Server acts as the server. Through parameter adaptation [MS+04], the client and the server set the IP address and port number of the directory agent at design time.

After a service provider registers its service with the Registry and a service consumer discovers this service with a service proxy, communication between the service provider and its service consumer still goes through the Proxy Host in the Registry system. This means that all forms of data exchange between the client and the server pass through the Registry system. Take the UML sequence diagram for the login service collaboration as an example shown in Fig 3.7. After the *login service* is registered and the *login service proxy* has successfully discovered the service, the user's login credentials are first sent through the Registry system before they are forwarded to the *login service* in the server. Similarly the response from the *login service* passes through the Registry system before being forwarded to the *login service proxy*. We have identified this as not an optimal way of using the service discovery mechanism. The Registry system should only be used for service registration and discovery and the communication between the service provider and service consumer should not be routed through the Registry system.

To optimize the implementation, communication between the service provider and its service consumer should be direct after initial registration and discovery of a service at the Registry. Figure 6.1 shows a UML sequence diagram of a login service collaboration which is based on the proposed implementation. After initial discovery of the *login service*, the user sends the login credentials directly to the server without passing through the Registry system. Response messages are then also forwarded directly to the user. Hence in this scenario, only the initial service registration and discovery goes through the Registry system.

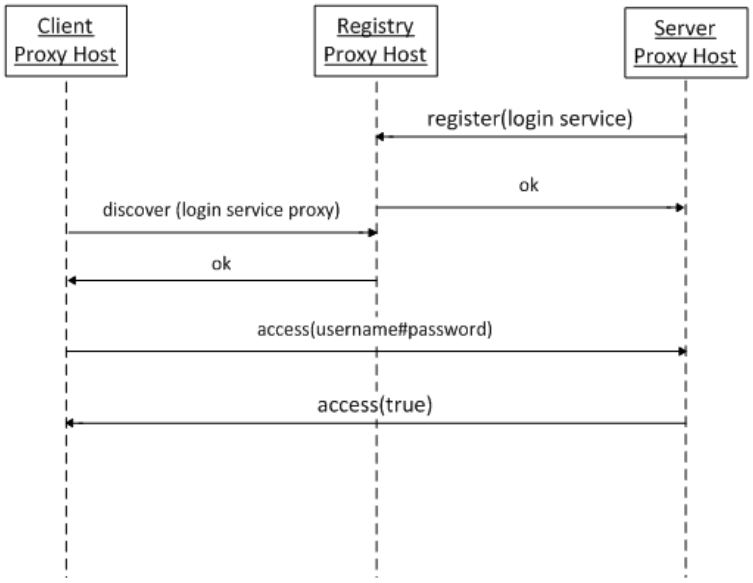


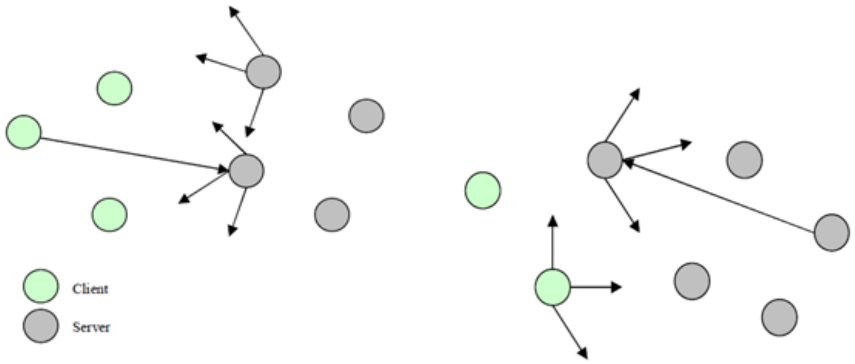
Figure 6.1: Login Service collaboration sequence diagram.

6.1.2 Decentralized service description for City Guide application Architecture

The main disadvantage of centralized service description is that there is a single point of failure. This breakdown point occurs at the DA. In the case of our application, the breakdown point is at the Registry. Take for example that the City Guide application is running smoothly. In the event that the Registry system fails the entire application will breakdown. This means that the client and server components have their functionality bound to that of the Registry.

A decentralized service description could be used in the implementation of the City Guide application. With the decentralized service description, the Registry would be absent. The server and client must find a way to discover services from each other without any central directory. In general, service providers must spread their descriptions so that the service consumers are informed about their services or the service consumers must spread their requests of service providers searched [ONT05]. Two methods by which this can be achieved are shown in Fig. 6.2.

- push model
- pull model



(a) push model (b) pull model

Figure 6.2: Decentralized service discovery with push and pull model [ONT05].

In the push model, the server periodically publishes in multicast service descriptions in the network. This makes it possible for the clients to find the available services. The clients can then contact the server with a unicast message for the service of their choice. In the pull model, the client makes requests in multicast in the network. The servers in accordance with the requests respond using a unicast message.

6.1.3 Tradeoffs between Centralized and Decentralized service description

In implementing the City Guide application using the central service description, the Registry assumes the main role of service discovery which would have otherwise been the responsibilities of the client and server systems. Service providers publish their service descriptions to the Registry and service consumers consult the Registry for the published services. Had the system been implemented using a decentralized service description, the service consumers must listen to service descriptions broadcasted directly from the service providers, or the service consumer must broadcast a request for a service description that the service provider could respond to.

Using the centralized service description would result in generally faster responses. Using static discovery, the client and server will have reference to the Registry so can connect with it directly. In addition, multicast messages that need to be circulated in the network are reduced hence there would be less use of the network bandwidth. In larger network environments, the centralized service description is scalable and supports adding more DAs to serve as duplicate repositories of service registrations [Gu99]. On the contrary

however, there is always the chance of failure with the Registry which would result in breakdown of the application since components cannot be configured and adapted in terms of service registration and discovery.

The decentralized service description is suitable for smaller networks. By smaller network, we mean that the number of users the application is intended for is small. Furthermore, since the Registry system is not involved, there is no central point in this architecture and hence the number of system components susceptible to failure is reduced. The challenge also here is that although the decentralized service description seems like a simple architecture, it involves higher consumption of bandwidth and increase in network traffic [ONT05]. Deciding on which of the two service descriptions to implement would therefore border on factors such as the size of the network and how robust the application should be.

6.2 Issues: City Guide application, Service Discovery Protocol in the mobile context

A major component of the City Guide application, the *City Guide App* system is deployed on mobile devices. Service discovery protocols were traditionally not intended for mobile contexts [ONT05] and so implementing such protocols with requirements on memory and storage capacity in the devices could prove challenging. Inherent mobile network features such as heterogeneity, limited bandwidth and frequent disconnections can pose limitations on these protocols. In this section, we discuss the pertinent problems faced in the mobile context viz-a-viz how the service discovery protocol used in the City Guide application adapts with these challenges.

6.2.1 Frequent disconnections

There are frequent disconnections for devices in a mobile environment. This is primarily due to limited energy in the device. When using the City Guide application and the mobile device loses connectivity, all interactions between the service provider components and the service consumer components in either the client or server are stopped abruptly. The server system however maintains the state of the client details indefinitely. For example the *Group Positioning Service* in the server still keeps the last known position of the client in its entries after the client loses connectivity. When the client gets connectivity back, it is prompted to restart the application and sign in again. The *Login Service* uses this new login credentials as entry for the client and the system operates as normal.

When a user signs in, the Login Service only seeks to authenticate the user's credentials and does not monitor which user is authenticated or what device is being used for the login procedure. This appears suitable for frequent loss of network connectivity of devices since the client could always reconnect with the server. In contrast, there is no way for the server to keep track of which user and what device it is signed on with. This limitation has been identified for the City Guide application and would need to be resolved to handle the situation of a user logging in with multiple devices.

6.2.2 Limited resources in power and memory management

A well-known fact about mobile devices is that they have weak memories, energy and low storage capacities. The Android operating system has its own mechanisms of managing system resources such as killing off some activities in low memory conditions. The *City Guide App* deployed on the mobile device should however not use too many background services [OHA12] that would require the spending of resources.

The Location Service is for instance one service which puts constraints on battery consumption of the device. The service requires that the GPS or WiFi is constantly being used to fetch location values for as long as the application is in use. The Location Service is run as a background service in Android and so in extreme low memory conditions could be terminated. The *ActorRouter* block found in the *Proxy* Host is also run as a background service since it is continuously interacting with other actors deployed in the distributed platform of the application. These two blocks identified are the heaviest in terms of use of resources by the City Guide App. To ensure successful running of the application, it is advisable to close other power consuming applications on the device while running the City Guide application.

The rest of the design of the City Guide application is such that most of the service intensive operations requiring the use of heavy logic such as the *Login Service, Group Positioning service, IM Service and GroupChat Service* which are implemented as service providers, are handled in the server system of the City Guide application. The server system is deployed on local host machines (PC) with far greater processing powers than the mobile device hence will be capable of creating and managing all the resources neededfor the application.

6.2.3 Heterogeneity of the mobile devices

The City Guide application is intended for Android platforms and has been tested on different devices running API version 7 and above with good success. However, a significant observation worth noting is that the appearance of the layout of the UI of the chat applications (*Group Chat* and *IM*) varies slightly when deployed on the varied devices. This is because of the different screen sizes offered by the devices. The core functionality and behavior of the UI does not however change when used across the various devices.

Android provides the developmental environment across devices and handles adjusting an applications UI to the screen on which it is displayed however for a better optimization, the Android platform provides APIs which allow for the control of application UI specific screen sizes and densities. Our implementation did not focus on user interface optimization. [OHA13] explains how Android supports applications deployed on devices with different screen sizes.

6.3 Critical Assessment of Components

In this section, the IM and Group Chat components are examined and the design aspects and implementation decisions that could otherwise been done differently are explored.

6.3.1 Proposal of a new Chat Application Architecture

The architecture used in the implementation of the IM and Group Chat communication platforms uses a relay point in the system to deliver the messages to the various clients. The *GroupChat Service* and *IM Service* components located in the server system are the relay points used. It would be ideal to route messages between players directly without the use of intermediaries such as the one in the server system. Eliminating intermediaries will reduce traffic in the network, improve the quality , lower latency and improve interactivity [HK+10].We will now propose an architecture that supports direct communication among the clients without any relay point. We will use the Group Chat platform to illustrate this.

From the knowledge of the Group Chat architecture already explained, each client is able to send and receive messages from the other clients in real time. The sending and receiving processes for the client are independent of each other. It is in that sense that each client registers a unique address with the *GroupChat Service* in the server to receive messages with. In the proposed model, the functional behavior of the Group Chat will not change

from the perspective of the client. This means that the client will send and receive messages as normal. What is different here is that, all the Group Chat application specific logic is moved onto the user client such that it is run from the user client directly. We explain this idea further with a sequence diagram of a simple chat service in Java shown in Fig. 6.3.

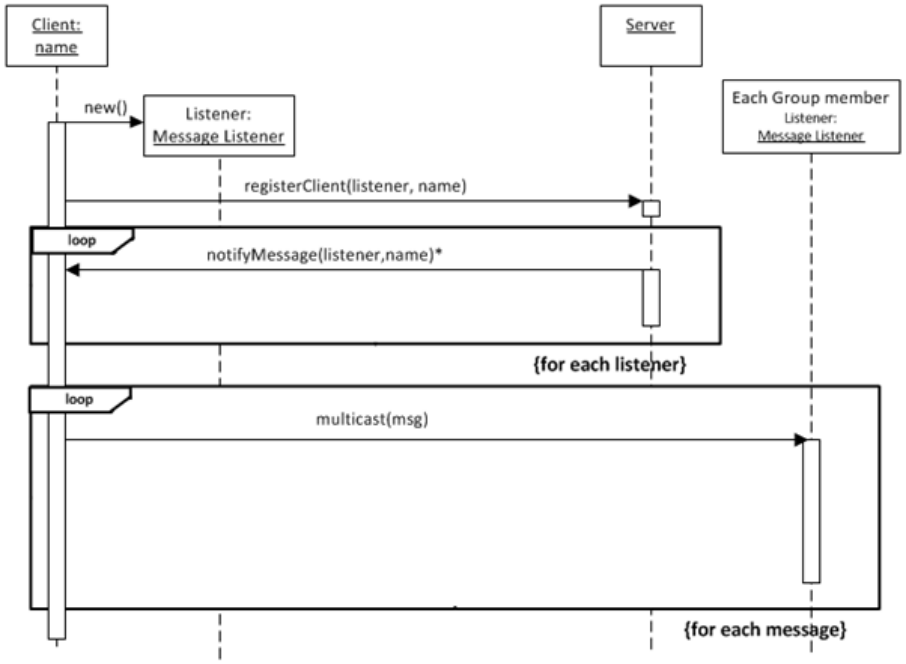


Figure 6.3: Group Chat sequence diagram of proposed model.

In this simple chat service, each client creates a listener object which is registered with the server. The server in this case forwards a reference of the listener objects of other group participants to the client in a *notifyMessage*. This means that each client will hold a copy of the listener objects of the group provided by the server. When the client wants to send a message, it uses the reference list of the listener objects to forward the message in a *multicast* message. The message is received by the other clients through their listeners. Since each client has a reference to the listeners of all other clients, the communication among the clients is direct and the server plays no role in the ensuing group chat communication.

We now proceed to illustrate using block diagrams, how the proposed architecture could be implemented in a distributed framework. Figure 6.4 shows a Group Chat communication involving three users. In each client system, there is a *GCp* role which takes part in the Group Chat collaboration.

The behavior of the GCp role is the same as shown in the previous block diagrams in Chap. 5, which is to send messages out from the client. However its implementation in this new model would be different since it would not necessarily require any corresponding role in the server system. The reusable roles *GLp* and *GLs* which model the *Address Service* collaboration is still relevant in this model because each client should be able to register an address with the server. In the server system we introduce a new role CCs whose behavior will be to establish a dynamic relationship with the GLs role in the server. This then provides a reference list of GLp's from various clients to all other clients subscribed. Hence client A for example will contain a reference to the GLp of client B and C (indicated as GLp_B andGLp_C).

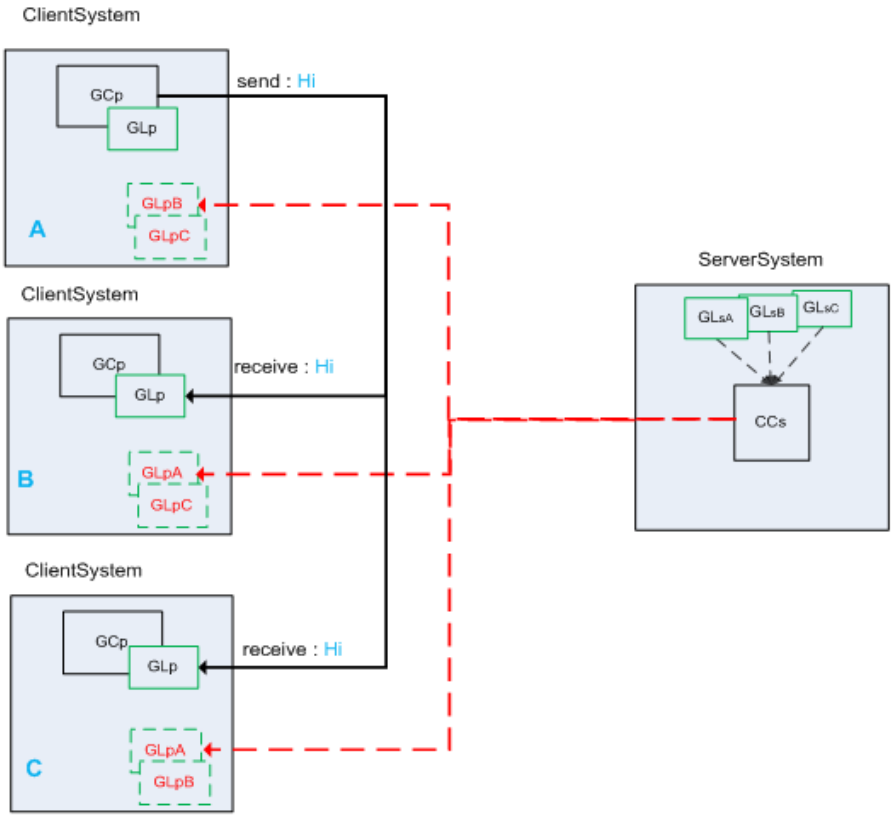


Figure 6.4: Block model of proposed architecture.

In Fig. 6.4, A sends a Hi message directly to B and C. To achieve this, A uses the reference of the GCp role of B and C and addresses the message directly to them. The message is routed through the network and the GLp component in B and C are notified therein. When B wants to send a message,

the principle is the same however the GLp reference held by B will be for client A and C (GLp_A andGLp_C). The sent message will thus be received by A and C only. The knowledge about group membership for the Group Chat service is thus distributed among the clients.

Comparing this new architecture to the one implemented, it is clearly seen that with the new model, sending and receiving messages is executed directly between clients without the use of any intermediary. In the previous model, messages were sent first to a server before being forwarded to the clients. This new architecture decreases the number of messages circulating in the network as well as reduced latency. In addition the need to use servers that can scale enough to handle a larger domain of users for a larger network is reduced. In contrast however, the new model brings high requirements on the mobile device itself. This is because all the application specific logic for the implementation of the Group Chat functionality is run on the client terminal and its network connection. Implementing this model would require the usage of more resources such as power and memory from the mobile devices. Robust implementation of such a model would require using high capacity mobile terminals with good requirements in power, memory management and signal strength.

6.3.2 Components needed for the implementation of proposed model

The new model will require changes to be made when composing building blocks for the Group Chat functionality. The GCp role for instance does not require any corresponding role in the server system. This means that the service provider-service consumer relationship is not required for this role. In implementing, the GCp role could be modeled as a service provider only and registered with the Registry. A service proxy for this service is not required since it does not need to be discovered by any other components. Alternativelty, a new building block could be composed and its functional behavior would be to establish a unilateral relationship with the Registry with which it can use to send messages from within a client. The behavior of the GCp role would however be more complex in the proposed model since it would be required to do multicast in sending messages to other clients. In addition it would have to handle and manage group data (addresses) that it would be updated with from the server.

The new role CCs could be implemented using a service provider-service consumer behavior. The behavior of this role is to notify each client of all the client addresses that have been registered with the server. The service

provider component of the CCs role would be located in the server system while the service consumer component would be located in the client system. This means that each client would be subscribed to this service and would establish a relationship such that, it would periodically be updated of the addresses from other clients that have been registered with the service provider in the server.

Chapter 7

Concluding Remarks

This chapter summarizes results achieved in this thesis. Based on our discussion chapter, some recommendations are also proposed as future work.

7.1 Summary of Results

In this thesis, we have presented the City Guide application and shown the architecture behind its design and implementation. The distributed system is decomposed into a set of building blocks wrapped up as three separate systems consisting of the client system which is deployed on handheld terminals, a server system and a Registry system which are deployed on a local host machine. In modeling the collaborative services which have a cross-cutting behavior among the several components in the distributed architecture, a service discovery technology was used. With this mechanism, each service is decomposed into a service provider role and a service consumer role. The service provider publishes its service using a known service description with a Registry and the service consumer discovers the service at the Registry. With this, there is a dynamic linkage between the collaborating roles to perform the service.

The communication model used in routing between the three distributed systems has also been examined. Each system has a Proxy Host which serves as the router. In the Proxy Host, we use ActorRouter to communicate among the distributed systems. The routing table in the Proxy Host for each system gets updated as components within each system communicate back and forth with components in other systems. Components within each system communicate with the Proxy Host using two local router proxies called Client Proxy and Server Proxy which are encapsulated in the *Generic Service* block and

the *service proxy* block.

Two new services, *IM and Group Chat* have also been introduced with their components developed and integrated with other system components. The IM service and Group Chat service have been recognized as basic services that should support an SCLS domain. The basic services could be reused in other application specific services. These two services enhance the social aspects of collaborative learning when using the application since there is a feeling of connectedness among the players of the game. In addition, by using these services, group members are able to perform shared tasks while distributed.

7.2 Conclusion

The platform used to compose the City Guide application provides an efficient way of composing other systems as well. Having knowledge of the development tool, Arctis and its use is necessary to achieving this. In addition having understanding of service discovery protocols and technology is essential. Thanks to the reusable building blocks in Arctis, new applications and services can rapidly be developed.

Composing the new services of IM and Group chat was flexible using the service discovery technology and the communication principle involving proxies and Proxy Host for the distributed architecture. The architecture itself enabled communication among distributed components hence developing an IM and Group chat service for the City Guide application required understanding the communication principles and developing suitable components for the client and server to run a successful IM and Group chat service.

The City Guide application is still in active development. The following are considered as further development and extensions of the City Guide application.

Implementation of new services The City Guide application should be extended to include new services that are important for situated collaborative learning. Other basic services such as picture or file sharing and SMS could be implemented. Application specific services such as scoring and clues are also yet to be implemented.

User Interactivity for composing of learning services Currently composing learning activities such as adding players, groups and learning content is done in XML as part of the server configurations. The system could be improved by making it more user-interactive with a graphical

user interface where game configurations would be loaded. This will create a good user experience for the administrator or composer of the game.

Bibliography

[BA12] Bitreactive AS 2012,*LaTeX: Arctis* , Available at http://reference.bitreactive.com/, Accessed April, 2012.

[BR00] C. Bettsetter and C. Renner, *LaTeX:A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol.2000*

[Cas08] Humberto Nicholas Castejon, *LaTeX: Collaborations in Service Engineering. Modeling, Analysis and Execution. PhD thesis, Norwegian University of Science and Technology, 2008.*

[GM03] Geir Melby, *LaTeX: Using J2EE Technologies for Implementation of ActorFrame Based UML2.0 Models* Agder University College, May 2003.

[Gu99] Erik Guttman, *LaTeX:Service Location Protocol: Automatic Discovery of IP Network Services, IEEE Internet Computing, vol. 3, no. 4, pp. 71-80, July-Aug. 1999.*

[HK+10] Huang Te-Yuan, Kok-Kiong Yap, et al., *LaTeX: Phone-Net: a Phone-to-Phone Network for GroupCommunication within an Administrative Doman,Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds, 2010*

[IC09] Ilaria Canova Calori, *LaTeX:Supporting Different Social Structures in City-Wide Collaborative Learning, IADIS International Conference Mobile Learning, 2009,NTNU.*

[JJH98] D.W. Johnson, R.T. Johnson, and E.J Holubec, *LaTeX:Cooperation in the classroom* Interaction Book Company Edina, Minn, 1998.

[Kat12] Surya B. Kathayat, *LaTeX:On the Development of Situated Collaborative Services* . Doctoral thesis , NTNU, February 2012.

[KB09] Surya Bahadur Kathayat and Rolv Bræk, *LaTeX:Platform support for situated collaborative learning* In International Conference on Mobile, Hybrid, and On-line Learning, 2009. ELML09. IEEE Press, 2009.

[KBH09] F. A. Kraemer, R. Bræk, P. Herrmann. *LaTeX: Compositional Service Engineering with Arctis, Teletronik, 2009.*

[KKB09] F. A. Kraemer, S. B. Kathayat and R. Bræk, *LaTeX:Unified modeling of service logic with user interfaces.* Proceedings of the first international workshop on Model driven service engineering and data quality and security,2009

[Kra08] Frank A. Kraemer, *LaTeX: Engineering Reactive Systems. A Compositional and Model-Driven Method Based on Collaborative Building Blocks* PhD thesis,NTNU, August 2008.

[KSH09] F. A. Kraemer,V. Slttem, P. Herrmann, *LaTeX: Tool support for the rapid composition, analysis and implementation of reactive services* Journal of Systems and Software, 82(12):2068-2080, 2009.

[LH02] C. Lee and S. Helal, *LaTeX: Protocols for Service Discovery in Dynamic and Mobile Networks. International Journal of Computer Research, Volume 11, Number 1, pp.1-12, 2002.*

[MS+04] Philip K. Mckinley, Seyed M. Sadjadi, et al. *LaTeX:Compossing Adaptive Software. Dept. of Computer. Sci.& Eng., Michigan State Univ., USA July 2004.*

[OHA11] Open Handset Alliance, *LaTeX: What is Android,* Available at http://developer.android.com/guide/basics/what-is-android.html, Accessed April, 2012.

[OHA12] Open Handset Alliance, *LaTeX: What is a Service,* Available at http://developer.android.com/reference/android/app/Service.html, Accessed April, 2012.

[OHA13] Open Handset Alliance, *LaTeX: Supporting Multiple Screens,* Available at http://developer.android.com/guide/practices/screens_support.html, Accessed April, 2012.

[OHA14] Open Handset Alliance, *LaTeX: android.location,* http://developer.android.com/reference/android/location/package-summary.html, Accessed April, 2012.

[OMG09] Object Management Group (OMG), *LaTeX: UML 2.2.1 Superstructure Spec., February 2009.*

[ONT05] S. Ousliha and N. Nouali-Taboudjemat, *LaTeX: Service Discovery Protocols, Available at http//http://mnet.skku.ac.kr/data/2005data /JCCI2005Spring/Papers/Paper/TM22/TM22-1.pdf, 2005.*

[PG99] C. Perkins and E. Guttman, *LaTeX: DHCP Options for Service Location Protocol, IETF, RFC 2610, June 1999. Available at http://www.ietf.org/rfc/rfc2610.txt*

[Ri00] Golden G. Richard III, *LaTeX: Service Advertisement and Discovery: Enabling Universal Device Cooperation, Available: http://computer.org/internet/, 2000.*

[SB08] Haldor Samset and Rolv Bræk, *LaTeX:Dynamic Service Discovery Using Active Lookup and Registration services, pp.545-552, 2008 IEEE Congress on Services - Part I, 2008*

[SC99] Salutation Consortium, *LaTeX: Salutation Architecture Specification Version 2.0c Part 1, The Salutation Consortium, June, 1999* Available at http:// http://systems.cs.colorado.edu/ grunwald/MobileComputing/Papers/Salutation/Sa20e1a21.pdf

[TA08] Tellu AS, *LaTeX: ActorFrame Technical Documentation Draft V-01, June 2008*

[TT08] TTM4115, *LaTeX: Engineering Distributed Real Time Systems, Laboratory Exercise 4, 2008.* Available at :http://www.item.ntnu.no/fag/ttm4115/exercises/2008/Labs/4/Lab_4_Problem.pdf, Accessed June 2012.

[TT11] TTM3, *LaTeX: Design of Self-Adaptive Systems, laboratory* Available at :http://www.item.ntnu.no/academics/courses/ttm3/start, Accessed June 2012.

Appendix A

GroupChat Service Class

Listing A.1: multicastMessage operation in GroupChatService Java Class

```
package no.item.ntnu.arctis.examples.cityguide.groupchatservice;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import no.ntnu.item.arctis.library.proxies.Address;
import no.ntnu.item.arctis.library.proxies.Message;
import no.ntnu.item.arctis.runtime.Block;

public class GroupChatService extends Block {

      public java.util.Vector<no.ntnu.item.arctis.library.objects.user.Group> grps;
      public no.ntnu.item.arctis.library.proxies.Message message;
      public java.util.HashMap<java.lang.String,  java.lang.String> address= new HashMap<String, String>();


      public void multicastMessage(Message message) {

      String msg = (String) message.getPayload();
      String mess[]= msg.split("#");

      for (Map.Entry<String, String> entry : address.entrySet()) {
             System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
            if(!(mess[0].contains(entry.getKey()))){
                  System.out.println("KeyLoop = " + entry.getKey() + ", ValueLoop = " + entry.getValue());
                  String payloadStr[] = entry.getValue().split("-");
                  Address addr = new Address();
                  addr.setSessionID(payloadStr[0]);
                  addr.setIP(payloadStr[1]);
                  addr.setPort(payloadStr[2]);

               Message response = new Message("Group_Chat Service");
                  response.setReceiver(addr);
                  response.setSender(message.getReceiver().getCopy());

                  Object s =  message.getPayload();
                  response.setPayload(s);

                  sendToBlock("SEND", response);
          }
     }
}

      public void updateBlockAddr(String upd) {
       String str[] = upd.split("#");

            System.out.println("Group Manager BlockID_AddressToStr "+ str[0]+ " "+str[1]);
            address.put(str[0], str[1]);
       }
}
```

Appendix B

Group Chat Messaging Activity (User Interface) Class

Listing B.1: Group Chat activity class for user interface: Messaging Activity Class

```
package no.ntnu.item.arctis.examples.android.cityguide.group_chatui;

import no.ntnu.item.arctis.android.R;
import no.ntnu.item.arctis.examples.android.cityguide.androidmapui.AndroidMapUI;
import no.ntnu.item.arctis.library.objects.quiz.InstantMessage;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.KeyEvent;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;

public class MessagingActivity extends Activity implements OnClickListener {
	private EditText messageText;
	private EditText messageHistoryText;
	private Button sendMessageButton;
	private String username, chatPartner;

	@Override
	public void onClick(View v) {
		// TODO Auto-generated method stub
		//check is button is clicked
		boolean ischecked = (v.getId()== R.id.sendMessageButton);

		if(ischecked){
			// get message and send to block
			System.out.println("Send button is clicked.....");
			CharSequence message;
			message = messageText.getText();
			if (message.length()>0) {
			InstantMessage msg = new InstantMessage();

			System.out.println("Username is....."+ username);

			appendToMessageHistory(username, message.toString());
			messageText.setText("");
			String msgText = "message is sent..";
				msgText = message.toString();

				String newmsg = username+"#"+msgText;
				msg.setMessageText(newmsg);
				Group_ChatUI.getInstance().sendFromActivityToBlock("SEND", msg);
			}
		}
```

```
	}

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		// TODO Auto-generated method stub
		super.onCreate(savedInstanceState);
		setContentView(R.layout.messaging_screen); //messaging_screen);

		this.sendMessageButton = (Button) findViewById(R.id.sendMessageButton);
		sendMessageButton.setOnClickListener( this);

		this.messageHistoryText = (EditText) findViewById(R.id.messageHistory);

		this.messageText = (EditText) findViewById(R.id.message);

		messageText.requestFocus();

		Group_ChatUI.getInstance().sendFromActivityToBlock("CREATED", this);
	}

	@Override
	protected void onStart() {
		// TODO Auto-generated method stub
		super.onStart();
	}

	public void showNewMessage(final String msg){
		 Runnable r = new Runnable() {

			public void run() {
				CharSequence message;
				String str[] = msg.split("#");
				message = str[1];
				chatPartner = str[0];
	appendToMessageHistory(chatPartner,message.toString());
			}
		};
		runOnUiThread(r);
	}

	private void appendToMessageHistory(String username, String message) {
		if (username != null && message != null) {
			messageHistoryText.append(username + ":\n");
			messageHistoryText.append(message + "\n");
		}
	}

	public void parseUsername(String user){
			username = user;
	}

	public void replyMessage(final String msg){
		 Runnable r = new Runnable() {

			public void run() {
				String str[] = msg.split("#");

				CharSequence message;
				message = str[1];
				chatPartner = str[0];
				System.out.println("First Other partner issssss "+ chatPartner+ "and"+str[0]);
				appendToMessageHistory(str[0],message.toString());

			}
		};
		runOnUiThread(r);
	}

	 //kill the Instant Messaging UI block
		public boolean onKeyDown(int keyCode, KeyEvent event) {
		 Log.v("KEYCODE","" + keyCode);
			switch (keyCode) {
		 case KeyEvent.KEYCODE_BACK:
			Log.v("","Exit?");
			finish();
			Group_ChatUI.getInstance().sendFromActivityToBlock("EXIT");
			break;
		 }
		 return super.onKeyDown(KeyEvent.KEYCODE_0, event);
	  }

}
```

Appendix C

IM Service Class

Listing C.1: sendIMMessage operation in IMService Java Class

```
package no.item.ntnu.arctis.examples.cityguide.imservice;

import java.util.HashMap;
import java.util.Iterator;
import no.ntnu.item.arctis.library.proxies.Address;
import no.ntnu.item.arctis.library.proxies.Message;
import no.ntnu.item.arctis.runtime.Block;

public class IMService extends Block {
      public no.ntnu.item.arctis.library.proxies.Message message;
      public java.util.Vector<no.ntnu.item.arctis.library.objects.user.Group> grps;
      public java.util.HashMap<java.lang.String,  java.lang.String> usersAddress = new HashMap<String, String>();

      public void extractMessage(Message message) {
      System.out.println("IM Service is Registered.."+ message);
      String mess = (String) message.getPayload();
      String mess2[]= mess.split("#");
      System.out.println("Message split is as folowwwwwwwwws.."+ mess2[0]+mess2[1]+mess2[2]);
            Iterator<?> i = usersAddress.entrySet().iterator();
            while(i.hasNext()) {
                  java.util.Map.Entry<String, String> me = (java.util.Map.Entry)i.next();

                  if(mess2[1].contains(me.getKey())){

                  String payloadStr[] = me.getValue().split("-");
                  Address addr = new Address();
                  addr.setSessionID(payloadStr[0]);
                  addr.setIP(payloadStr[1]);
                  addr.setPort(payloadStr[2]);

         Message response = new Message("IM Service");
                 response.setReceiver(addr);
                  response.setSender(message.getReceiver().getCopy());

                  Object s =  message.getPayload();
                  response.setPayload(s);
                  sendToBlock("SEND", response);

                  }
      }
      }

      public void updateAdd(String upd) {
            String str[] = upd.split("#");

            System.out.println("Group Manager MyAddressToStr "+ str[0]+ " "+str[1]);

            usersAddress.put(str[0], str[1]);

      }
}
```