Department of Computer Science

Hamilton, New Zealand

Anonymous Mobile Service Collaboration

Chao Zhao

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science at The University of Waikato.

Abstract

Since the advent of mobile devices, a large amount of Applications (Apps) have been released to offer compatible services to satisfy the different needs of users all over the world, especially location-aware services, which have become more and more popular in today's market. Normally, these Apps need to collaborate with other service providers, for example, the App needs the map service to show the location to users, and the map service needs to be supported by the location-based service to offer the coordinate data. In some situations, an App may find more than one service offering the same type of data; they may be located in different places and be in, or on, different networks, so the App service needs to consider which service provider to use. Quality is the most important factor to identify how good a service provider is. It includes many different factors, like the response time, accuracy, reliability, also security and privacy, because users may not want a third party to know who uses this service and where the user is. In this way, the mobile service collaboration has to be anonymous. In this project, an event-based, context-aware service collaboration is implemented, and it is on a publish/subscribe basis, also anonymity and quality are focused on as the most important factors in the implementation.

Acknowledgments

First of all, I would like to thank my supervisor, Mrs. Annika Hinze. This thesis may not exist without her valuable advice and monumental support. She has always offered an open office door for me to discuss any ideas and questions. I would also like to thank all the people in the Department of Computer Science of the University of Waikato, especially the members of ISDB research group. Thanks also to Michael Rinck for his explanation when I first time received this topic. My deepest thanks go to my parents, for their understanding and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, mobile devices have become one of the most important communication tools, used across all generations, in today's society. With technology constantly upgrading and the increasing information requirements of people, many new applications have been released to offer a wide variety of services to users. Therefore, the mobile device is not only a simple communication device anymore, but is now an advanced and useful tool which can help people to do more than just talk.

Currently, many mobile Apps need to be collaborated with other service providers to offer the service to users, like my earlier example of the map service needing to be supported by the location service to identify the user's position. The service collaboration, based on mobile devices, presents challenges for existing service-oriented architectures (Hinze, Rinck, & David, 2010). This is because some services are only available locally (or the service can disappear at any time due to distance/reception or other factors which may affect it). In this way, a service may be constantly changing collaboration partners when it offers their service to users. To avoid this problem of constant changing, selecting the collaboration partners becomes an important issue.

When services communicate with each other, information is shared between them. This leads to another important issue - protecting user information. Services (Apps) need to collaborate with other service providers without revealing their or their user's identity. Therefore, a middleware (local broker) is added to solve this problem when each service transmits data to others (Hinze, Michel, & Eschner, 2009). The purpose is that each service connects to a trusted middleware; it will handle all requests and transmits the data back to the requesting APP (Rinck, 2010). For example, the publishing service (publisher) will deliver their data to the local broker (middleware), and then the data is sent to the service who subscribed to this data. In this way, the subscribing service (subscriber) cannot get any restricted information off the

publisher. On the other hand, the publisher also cannot get the information from the subscriber due to the local broker; therefore the information about both Apps users is protected.

This project will focus on the problems, which have been described previously, creating an infrastructure for mobile service collaboration, and then an application is implemented to show how it will work. The application is developed in Java programming language, and it is running on the mobile device in which the Android operating system is running. In this chapter, some scenarios are described to give users a closer look at why this project is needed and how the application runs when utilised in mobile service collaboration. In chapter2, the concept infrastructure design of this project is discussed, and the implement level design is described in chapter 3. In chapter 4, the detailed information about the data transfer between each service is described, and some examples are given to make it clearly. Chapter 5 is the documentation of the application implementation. Then the conclusion and the future work are discussed in chapter 6.

1.1 Scenarios

In the scenarios, users are supposed to use a tourist application which is installed on their mobile when they are travelling. This application collaborates the map service and the location service via the local broker to not only show the location on the map where the user currently is, but also some tourist information around that location. Besides the local broker, observer is another middleware, which is used to help subscribers to check the quality of the data which is published from publisher.

Scenario 1

In this scenario, a tourist travels to Hamilton city for the first time, and he finds out from his tourist application that the Hamilton Museum is one of the nice places to go visit, and it is not very far from where he currently is. When he reaches the museum and moves inside, he finds the original location service (GPS) does not work as well as before due to the user's mobile device finding it hard to receive the signal from the satellite. However, the museum offers an

indoor location service in RFID (Radio-Frequency Identification) (Technovelgy LLC, 2011), which is also offering the location data, and the quality of it is growing the closer he gets to the museum. In this situation, the local broker starts to communicate with the RFID location service, and then the local broker subscribes the location data from it (RFID) instead of the GPS location service.

Scenario 2

This situation is similar to scenario 1; however, there is no indoor location service that can be used when the user moves into the museum. Due to the quality of the GPS service dropping, the observer stops transmitting data to the local broker. This is because the publishing data from the GPS service cannot satisfy the rule of the map service. In this way, the condition is broken.

Chapter 2

Concept of the Infrastructure

In this chapter, some detailed information on the concept of the infrastructure will be described, and the related work for the software architecture will be discussed.

2.1 Broker

In the service collaboration, the broker is designed as the middleware to help data transfer between the publishing services and the subscribing services. The purpose of it's used to help subscribers to find, with the publisher, who offers the data, and the quality of it, and can satisfy the requirements despite the limitations, of the subscriber. It is shown in Figure 1.

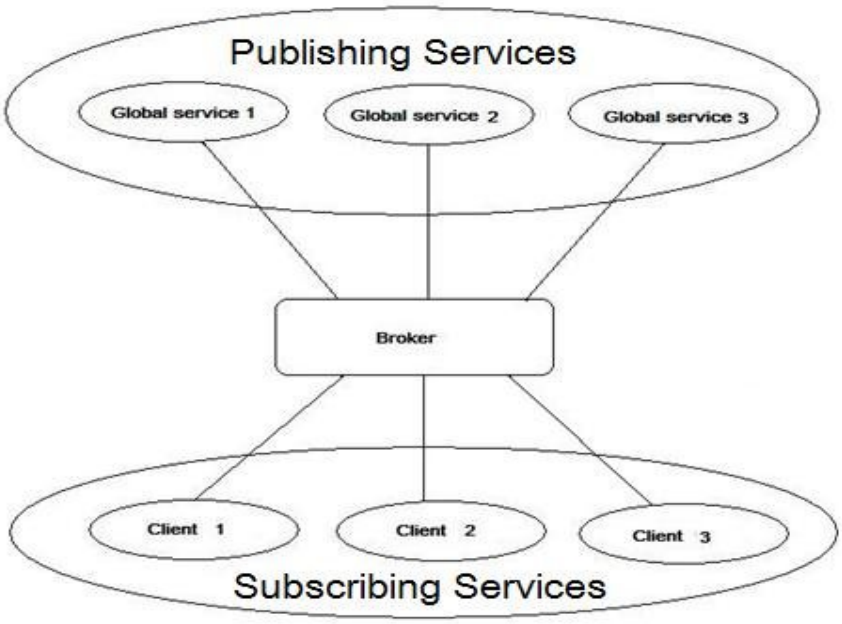

Figure 1: Concept of the connection of the broker with different services

In the broker design, three different approaches of the broker have been described, which are: the included broker, the remote broker and the intelligent gateway – distributed broker. Each design has its own advantages and disadvantages when it is used to handle the data transmission between

each publishing service and subscribing service. The detailed information is described below:

2.1.1 Included Broker

The concept of the included broker is that the broker locates on the user's mobile device. Services which are outside of the device are only accessible through this broker to connect the services which are located in this mobile device too (see Figure 2). Therefore, it may respond quite fast between the local broker and the local services, also it offers the maximized security. The disadvantage of this type of broker is that it may consume too much computation power of the user's mobile device; this is because the broker always communicates with new services which are not located in the mobile device. Also many pieces of useless information would be delivered to the local broker, which may also cause more traffic to the user's mobile device as well as taking up valuable space.

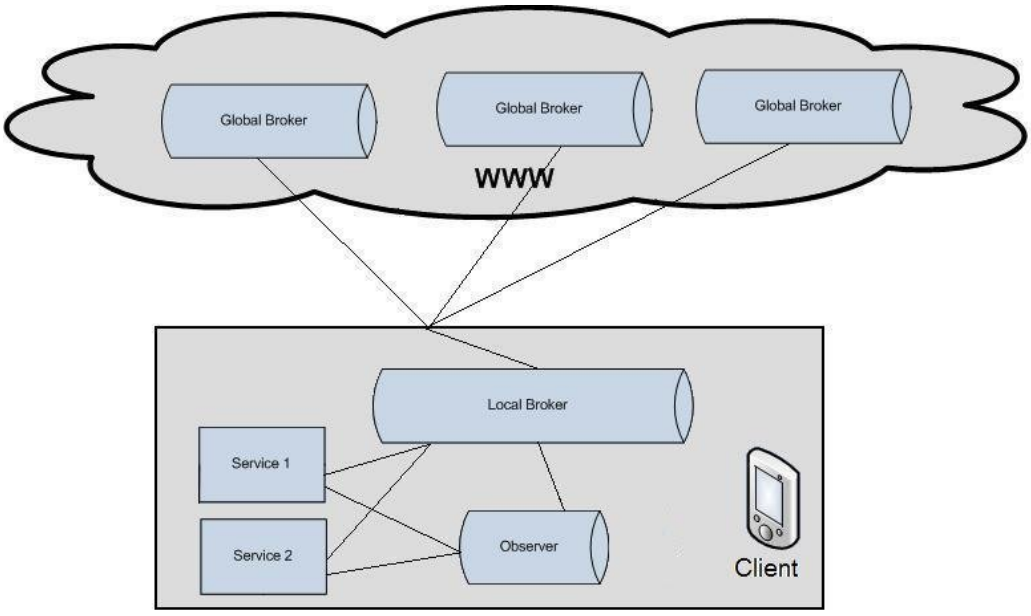

Figure 2: Included broker architecture

2.1.2 Remote Broker

The concept of the remote broker works in a similar way as the included broker, but is located on a private server instead of user's mobile device (see

Figure 3). All connections and events are routed thorough this broker, and it only sends events which are needed by subscribers. In this way, it may save the power and traffic of the mobile device; also, the device can be better supported by the large database. Therefore, this concept is really good for some businesses. On the other hand, there are some disadvantages for this type of broker. Firstly, the quality of the service may affected by the distance between the broker and the user's mobile device, which means the user may not be able to get the required services whilst travelling any distance. Secondly, the security level is lower than the included broker.

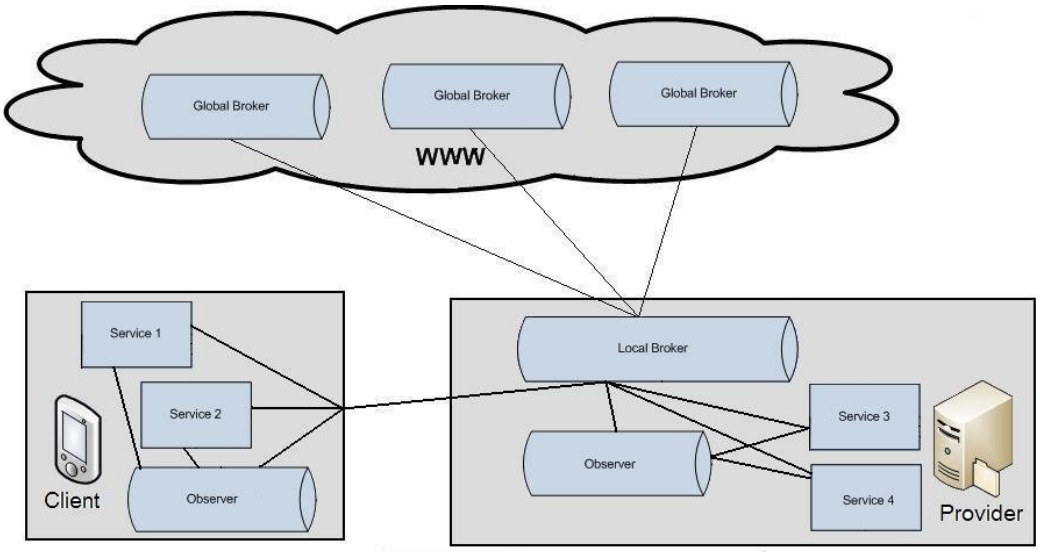

Figure 3: Remote broker architecture

2.1.3 Distributed Broker

This type of broker aims to keep the advantages of both the included and the remote brokers, and is designed with two brokers (instead of one) (see Figure 4). The first is located on the user's mobile device, and it consists of some basic capabilities, for example, it can help local services collaborate with each other. The second broker is located in the same area as the remote broker, which is used to communicate with the services outside the user's mobile device. In this way, it extends all the advantages of both included and remote brokers. But it also extends the problems which are described in the previous concepts.

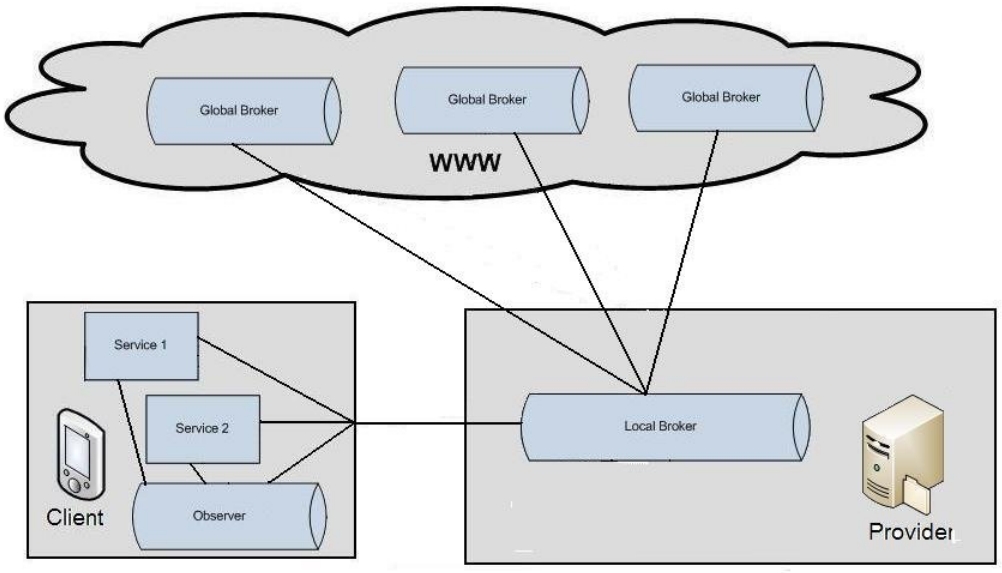

Figure 4: Distributed broker architecture

2.2 Conceptual Architecture of Included Broker

As the report described in 2.1, the included broker model is chosen as the target broker, and its concept will be used in this project when creating the broker. The reason for this is that the included broker concept seems to be the best in terms of anonymity and usability in different scenarios.

The conceptual architecture of the client side includes following parts: the connector, the broker and the display. The connector is used to connect with the different services, the client broker is used to analysis the received message and also it to protect the user's personal information when requests are made to and received by each server. The last part is the display, which will display the message to the user. The architecture diagram is shown in Figure 5.

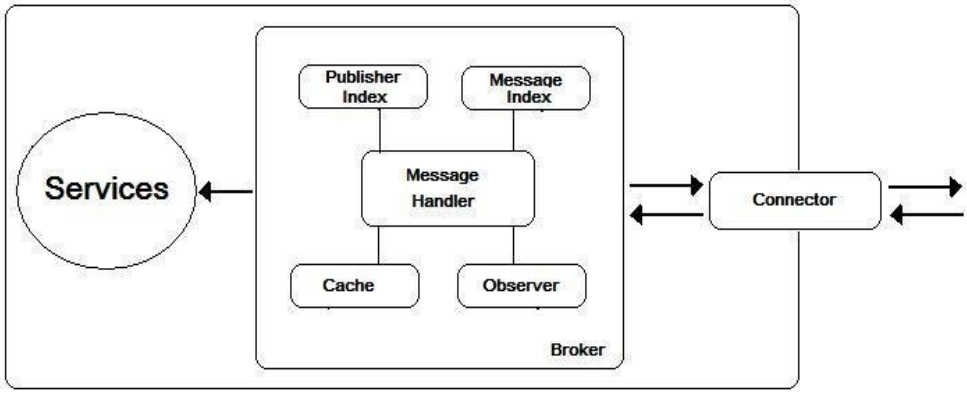

Figure 5: Conceptual architecture of client side

Connector

Connector of the client side is used to connect to each different service, and it also follows their own protocol to send and receive messages.

Display Part

In this part, the received data will be built up by the index of each message, and then displayed in the user's mobile device.

Broker of the Client Side

This is the main part of the client side, and it is designed to do two main jobs for users, which are: the send/receive event and protecting user's information. In this part, it includes a message handler and some other functions. Message handler is used to receive and broadcast events, when it receives a new event, it passes the event into the observer to check the quality of it, and observer will respond when the quality goes down. Cache is a temporary area to save the continuous events, and it will resend it to the display when all events are received. The publisher index is used to save the publisher information; this may be used later when the quality of the current service goes down. The last function is the message index, which is used to save the event information.

2.3 Conceptual Architecture of the Communication

In this system, the client's broker may communicate with many services when the user starts the system, and then the best quality one will be selected as the target service to request information. The rest of the services will still be considered if the quality of the current service goes down. The communication flow charts will be shown in Figure 6, Figure 7 and Figure 8.

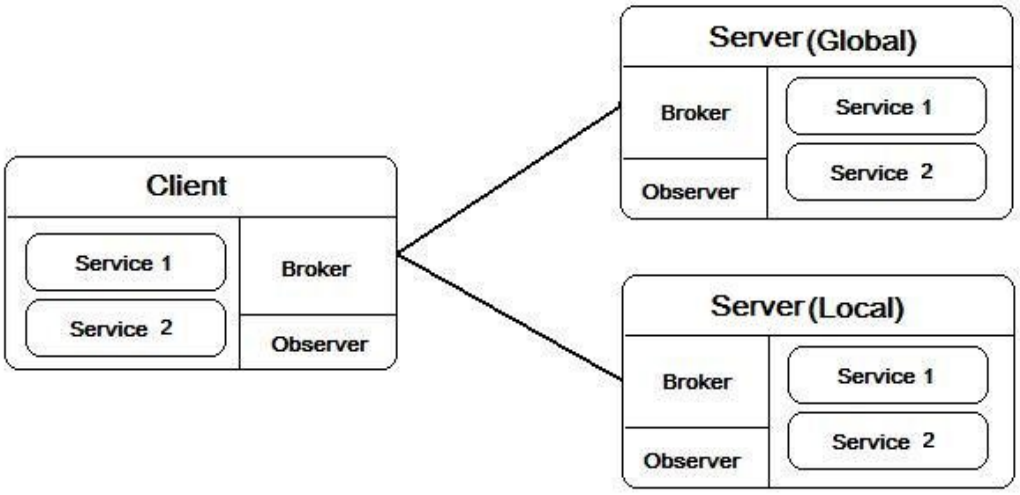

Figure 6: Client's broker communicate with both servers

Figure 6 shows the client's broker finds two services which offer the same data when the user turns on his mobile device. One service is offered by a global server, and other is offered by a local server. In this point, the broker of the client will decide which server to request the information from. Normally, the quality of the service is the main factor for the broker to choose the server, but it depends on different issues, like: the distance of the server, speed of the data transfer, response time, accuracy, security, etc. After the broker selects the server, it only responds to that server and starts to receive the information. Figure 7 shows the process.

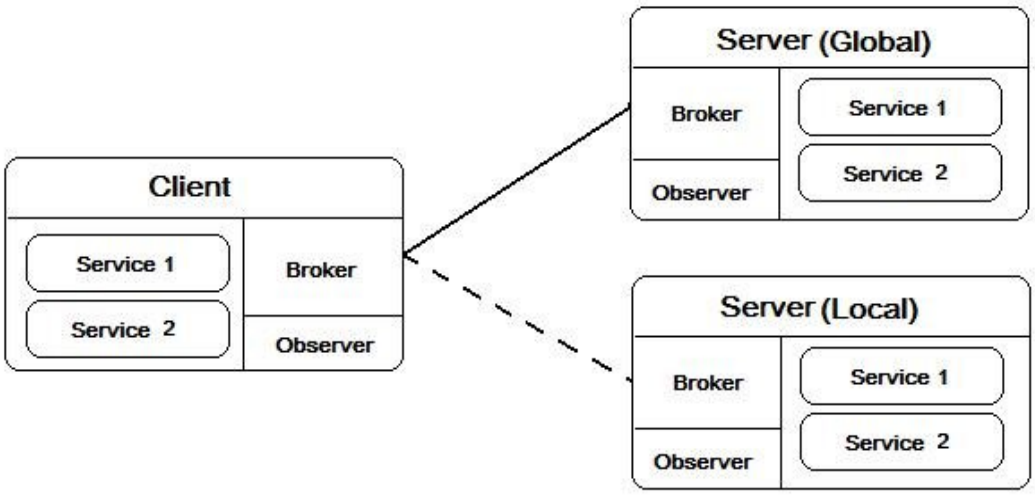

Figure 7: Client's broker stops communicating with the local server

In Figure 7, client's broker stops communicating with the local server when it finds the global server can offer the better service. But it does not mean the client's broker will never communicate with this local server again. Like the chart is shown in Figure 8, the client's broker re-communicates with the local server and stops communication with the global server when the quality of the service from the global server goes down.

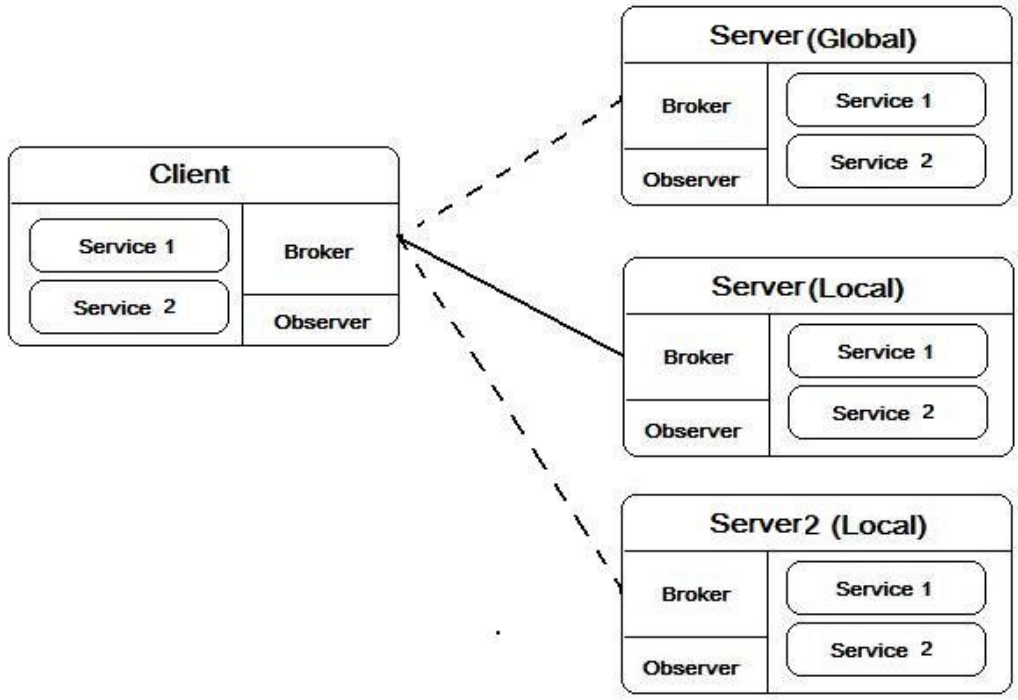

Figure 8: Client's broker re-communicates with the local server

In Figure8, the broker of the client side starts to re-communicate with the local server instead of the global server due to the quality of the data from the global server reduces. When the user's mobile device is running, some new services will also be found by the client's broker that will be saved and used at a later point.

Chapter 3

Implementation Level of Infrastructure

In this chapter, more detailed information about the implementation level infrastructures will be discussed; also some diagrams are used to make explanations more clear.

3.1 Basic Implementation on No Services is Available

In the previous scenario (see Section 1.1), we have talked about how the system automatically connects to an alternative service and starts to receive data from it when the quality of the current service goes down. But, there may not be an alternative service available in some situations. If this happens, the application may stop working due to the publishing service not satisfying the requirements of the rules and conditions which are defined by the subscriber. The sequence interaction of this situation is shown in Figure 9.

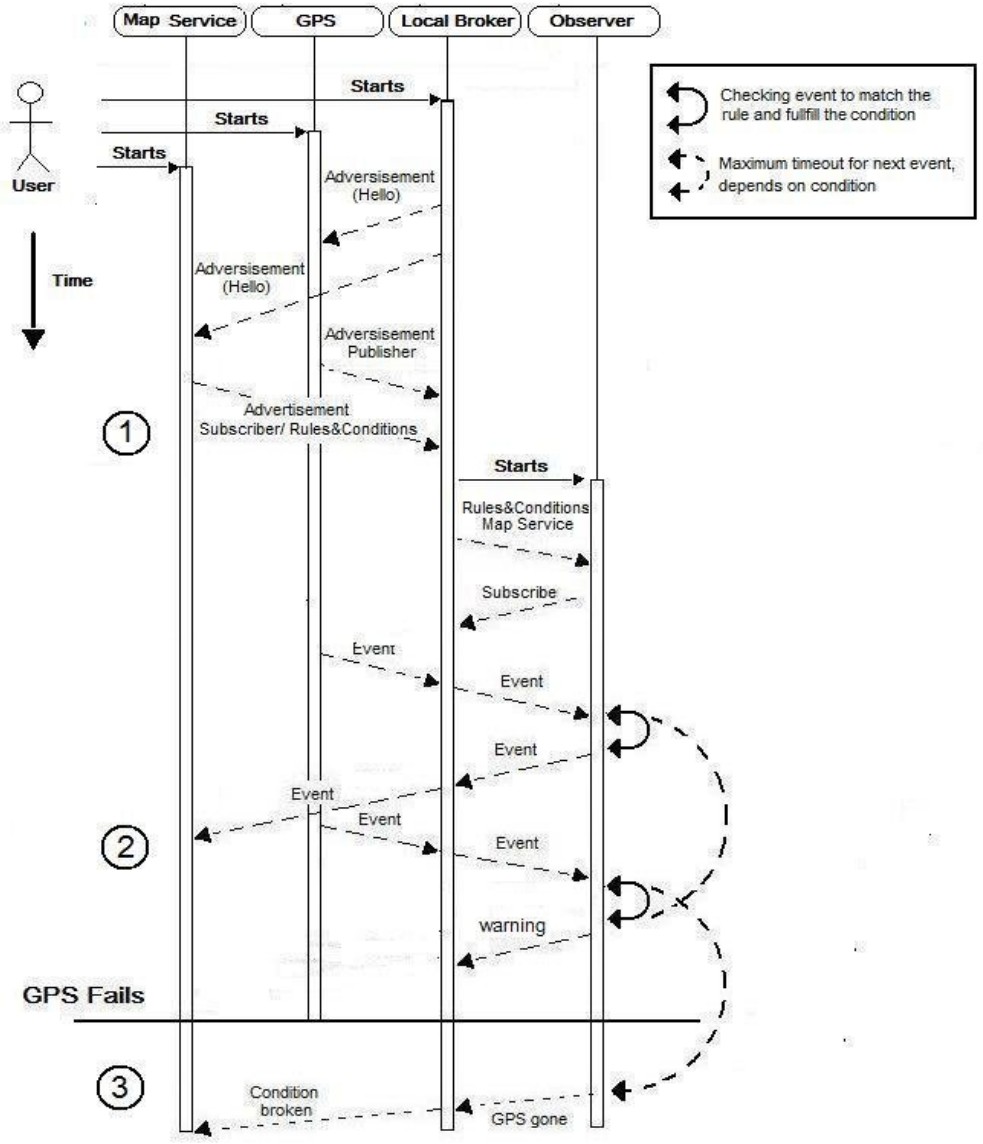

Figure 9: GPS fails and no alternative location service available in basic implementation

In this sequence diagram, it has been divided it into three parts, which are: (1) registering; (2) communicating and (3) conditions broken.

In the register part, it shows the local broker sets up connections with each service. As the mobile device is switched on, the broker, map service and the GPS service start at the same time. Firstly, local broker sends a “Hello” advertisement to the map service and GPS service. This advertisement contains a list of available event data type, which means each service can

both subscribe and publish these types of data with local broker. In some cases, the subscriber can also subscribe to the uncontained type of data from the local broker, and that type of data will be forwarded to the subscriber when local broker finds the publisher who publishes that type. After each service receives the message and checks it, a respond event is sent back to the local broker. For each service, it can be a subscriber, publisher or both. Publisher means a service which publishes a type of data, and the subscriber receives data from them. Like this diagram, the GPS service is a publisher, because it only publishes the location data, and no data are required by it. The respond event from publisher and subscriber is different due to the subscriber also needing to give the rules and condition information to the local broker, because the subscribers need to tell the local broker what kind of data they want, and in what situation. An example of the respond message is shown below (Figure 10).

Figure 10: Example of the respond message

While the local broker already connected with the map service and GPS, the observer starts at this time, and an advertisement from the local broker is sent to it. In this advertisement, all subscriber's rules and conditions information are included, this is because the observer is designed in this application to help subscribers to check the quality of each event to satisfy those rules and conditions or not.

The second part of this diagram shows how each service works with the local broker. In Figure 9, the GPS service already starts to send the location data after the first communication. Because this diagram is designed to focus on the quality of the service, therefore, the local broker will send every event to the observer to check the rules and conditions. If the quality of the event satisfies requirements of the subscriber, the observer sends back the event to the local broker, and it is then delivered to the map service.

In the last part, the observer checked the quality of the data which is published from the GPS service and as it is lower than the requirements from the subscriber. The observer stops to transmitting data to the local broker, and a warning message is sent to instead to inform it of the situation. The result of which is the map service stops working and waits for the local broker to find new publishers.

3.2 Alternative Implementation in Observer/Local Broker

In this section, a diagram with a different design of the data transmission between the broker and the observer is shown, and it is the same situation as the diagram in Figure 9. In this design, the local broker also sends every event to the observer to check the quality, but it does not wait for the response from the observer, and every event is directly resent to subscribers when the local broker receives it from the publisher. In this way, it reduces the time cost on the data transmission. The diagram is shown in Figure 11 below.

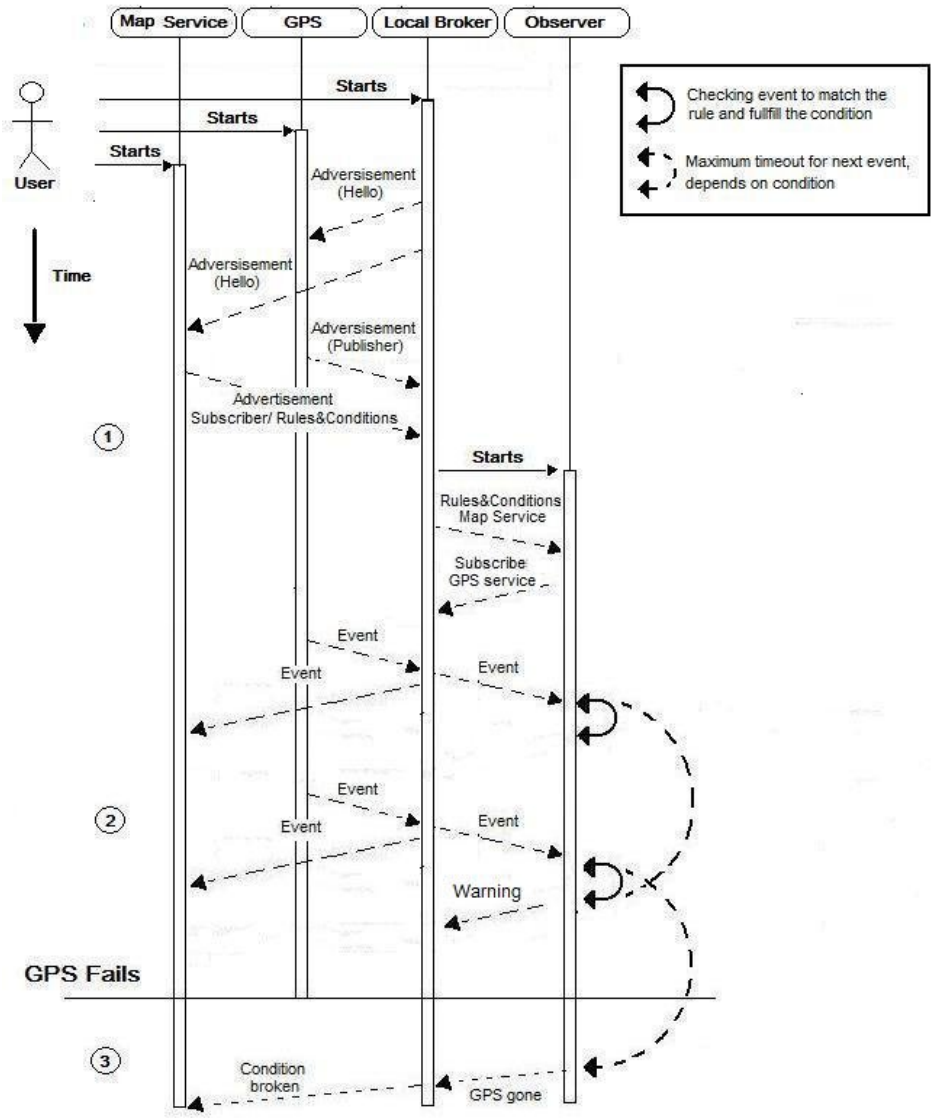

Figure 11: GPS fails and no alternative location service available in alternative implementation

The idea of this design is to save time in the data transmission between each service. For example, in Figure 9, every time the local broker sends the GPS event to the observer to check the quality, it then needs to wait the response from the observer. Therefore, the time-lapse may be costly using this operation. In a real situation, publishers can offer the good quality service most of the time, and so the observer is designed in this part to only respond to local broker when the quality of the event cannot satisfy the rules or conditions of the subscriber. Therefore, each GPS event is directly sent to the

map service after the local broker receives it (in this design), but the observer still checks the quality of that event, and it only responds to the local broker if the quality is reduced.

To compare the diagrams of the Figure 9 and Figure 11, they are designed in the different ways to solve each event, and each of them has their advantages, like it shows in Table 1 ("++" means really good "+" means good, "-" means normal).

Condition	Basic Implementation	Alternative Implementation
Speed	+	++
Quality	++	+

Table 1: Compare the speed and quality of both implementation

Due to the quality of service is most important factor that cared in this project, therefore, the basic implementation is selected as the architecture for this project.

3.3 The Basic Implementation on alternative service is Available.

In some museums, for example, they may offer their own indoor location service (RFID service) for their visitors. Therefore the system may use the indoor location service instead of the GPS service when the quality of it is reduced. The Sequence diagram is shown in Figure 12.

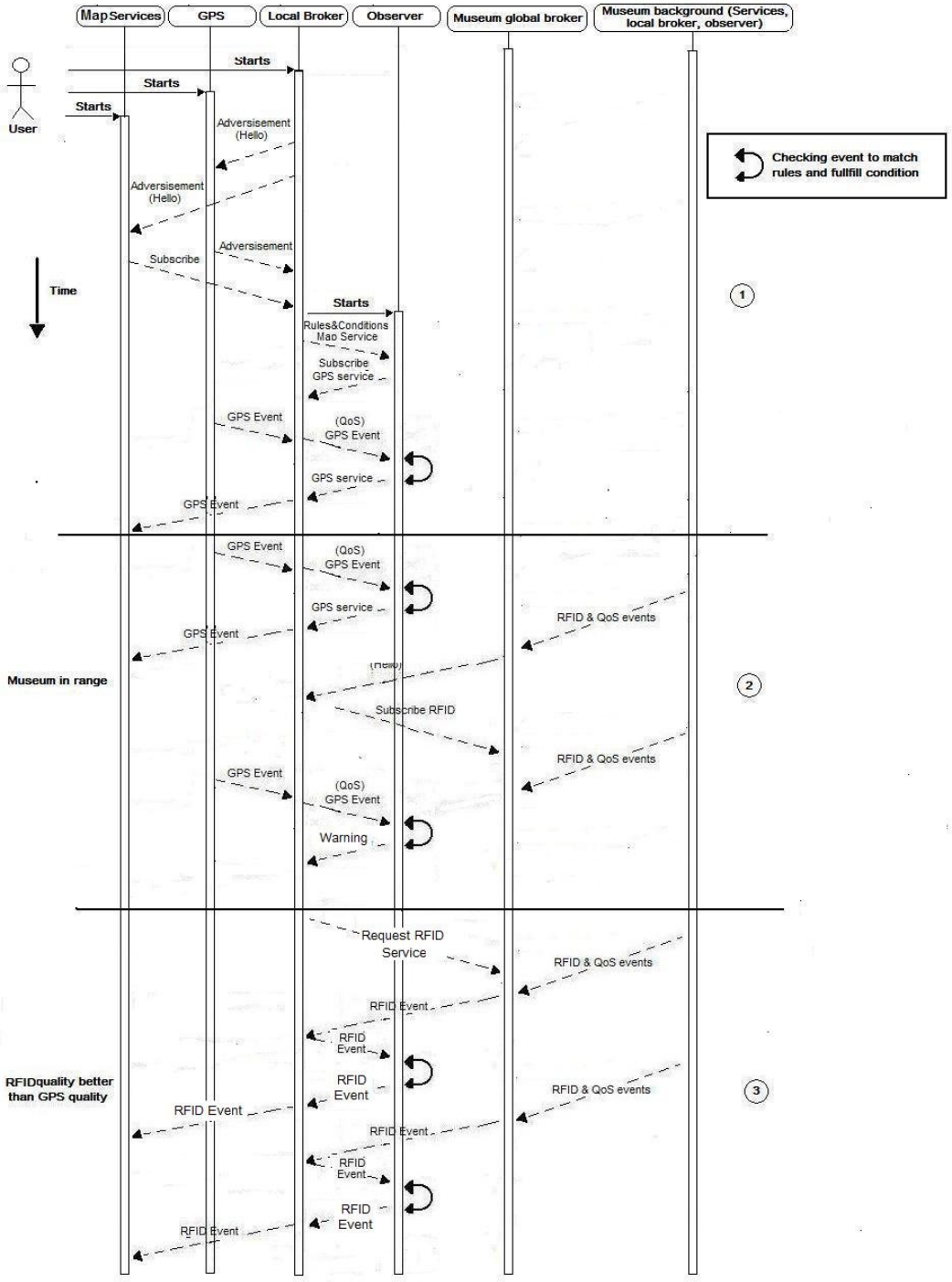

Figure 12: GPS fails alternative RFID location service available in alternative implementation

In this diagram, it also has been divided into three parts, which are the (1) set up, (2) the subscribe RFID service and (3) the RFID service instead of the GPS service.

The set up part in this diagram is similar to the previous case (see Figure 11), which also shows how the local broker sets up each service communication with others, and then the service of the publisher starts to send the data to their subscribers. Comparing this part with Figure 11, it shows some changes. Firstly, an alternative service is shown in this diagram, and it broadcasts their advertisement at anytime to the brokers which are in the range. Because the local broker of the user's device is not in range in this part, there are no communicates between the local broker and RFID services.

Then, an indoor location service is available when the user is near the museum, and so user's local broker starts to exchange advertisements with the museum's indoor service, and then it subscribes the location data from the indoor service. At this time, the local broker has not been started to receive the location data from RFID service because the quality of The GPS service is still good enough, therefore, the local broker will carry on working with the GPS service, but now has a backup service ready in case the quality reduces. Shortly, the observer checks the quality of the service from GPS goes down, an warning message is sent to local broker by the observer to tell it needs to find a new publishing service instead of the GPS service.

In the last part, it shows the local broker sends a message to RFID service to request the location data and same time it stops receive data from the GPS service. In this way, the local broker currently only receives location data from the RFID.

Chapter 4

Communications and Data Transfer

This chapter will describe the communication and data transfer between each service in detail, then the detailed information about each communication is discussed.

4.1 Data Type and Event Details

In this part, firstly it will explain the nature of data that may be communicated between services in event messages, and some examples are used to make descriptions more clear. Then it will show how the data is described in event messages and how rules can be defined for the handling of event messages.

4.1.1 GPS Data

In this application, it will follow the NMEA 0183 communication protocol (Scientific Coponent, 2010) to transmit the location data. Under this protocol, there are many different types of sentences that can be used to get the location data, like the GPRMC, GPGSV, GPGSA, GPGGA, and GPGLL (Robosoft, 2011). After comparing them, the GPGGA sentence is selected to use in this application, because it is the standard and most popular one. GPGGA sentence normally includes 14 different attributes to indicate the location. The GPGGA sentence structure and an example are shown below:

GPGGA sentence structure

- $GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>*xx<CR><LF>

GPGGA sentence (Example):

- $GPGGA,123519,4250.5589,S,14718.5084,E,1,04,24.4,9.7,M,,,,0000*1F

Name	Example	Description
Sentence Identifier	$GPGGA	Global Positioning System Fix Data
UTC Time	12:35:19	Hhmmss.sss
Latitude	4250.5589	Ddmm.mmmm
N/S Indicator	S	N=north or S=south
Longitude	14718.5084	Dddmm.mmmm
E/W Indicator	E	E=east or W=west
Fix Quality: • **0=Invalid** • **1=GPS fix** • **2=DGPS fix**	1	Data is from a GPS fix
Number of Satellites	03	3 Staellites are in view
Horizontal Dilution of Precision (HDOP)	20.4	Relative accuracy of horizontal position
MSL Altitude	9.7 (Meters)	
Units	Units = Meters	
Geoid Separation	Units = Meters	
Units	Units = Meters	
Time since last DGPS update	Blank	Null fields when DGPS is not used
DGPS reference station id	0000	
Checksum	*1F	Used by program to check for transmission errors

Table 2: Example of the GPGGA sentence

4.1.2 Events

Event is the ‘carrier’ of the system, which is used by each service to send the data. For example, the event from the GPS service includes four attributes, which are the event ID, name, service type, data type and the contents. The example shows this below.

The event from GPS service (Example 1):

- Service ID: GPS_001
- Service Name: GPS Service
- Service Type: Publisher
- Data Type: Location data
- Event Content: (123519,4250.5589,S,14718.5084,E)

In this example, the service ID and service name are used to remind the local broker who published this event. This is a quite an important attribute, because the local broker can use this attribute to find who sent this event. Service type for this event is used to tell the local broker the type of this service, and the local broker can use this attribute to connect subscribers and publishers together, also, the local broker will follow this attribute to decide if the event from this service needs to be checked or not by the observer. The third attribute is the data type, which is another attribute that the local broker uses to connect publishers and subscribers, and it also shows the type of the data that this service offers. The last attribute of the event is called the event content, which stores the information of this event. This example is the event from the GPS service, so some location data are stored there.

GPS events are normally sent continuously if any subscriber subscribes location data. Figure 13, is an example of how the GPS service sends location data to help people find the Hamilton Museum.

Figure 13: Example of the location data

In Figure 13, user's mobile device is supposed switch on at the position A, and he plans to go to the Hamilton museum. After he enters the location of the museum into the application, the way is shown to him. In this time, the local broker helps the map service to subscribe the location data from the GPS service. As we can see, the GPS starts to send the location data, and the map service shows the location data on the map to the user.

When user moves into the museum, the GPS service no longer works well, and the system finds that an alternative location service (RFID) is available. Therefore, the new service is subscribed by the user's device, and then starts to receive the location data from it. The example is shown in Figure 14.

Figure 14: Example of switching location data

In this example, we can see the first three points (A, B and C) are the location data from the GPS service, but the point D shows that the RFID location data is used by user's device instead of the GPS location data, the reason is that the quality of the GPS location data in point D is lower than the data which is published by RFID service. Therefore, local broker starts to communicate with the RFID location service at position D.

4.1.3 Rules and Conditions

In last chapter, we talked about the subscriber sending rules and conditions to the local broker when it subscribes information. Like the diagram in Figure 10, it showed the attributes that are contained in those rules and conditions. In this section, the detailed description of each attributes will be discussed.

4.1.3.1 Rules of the Subscription

The rule of the subscription is normally used to tell the local broker what kind of data the subscriber wants. For each rule, it usually contains three attributes, which are the rule type, rule factor and the rule information.

The rule type is used to identify what kind of things are quite important to this subscriber, like the example shows below, the response time is given to show that the response time is the most important thing for this subscriber. Rule factor normally have two types value, which are the (greater than">" and lesser than "<"), and it is normally used with the rule information together, like this example gives the rule factor is "<" and the rule information is ten. It means the response time between each event from publisher must be less than ten seconds.

The rule of map service (Example 1):

- Rule Type: Respond Time
- Rule Factor: <
- Rule Information: 10

4.1.3.2 Conditions of the Subscription

The conditions are also defined by the subscriber, and are used to determine the terms under what situation the event is accepted by the subscriber. Condition information normally includes two parts, which are the condition type and the condition information. The example is shown below.

The condition of map service (Example 1):

- Condition Type: Connection Number
- Condition Information: 1

In this example, the condition type is used to describe the factors that are important to the subscriber, and the details are stored in the condition information. Like this example, it means the subscriber only wants to connect one publisher at any time.

4.2 Description of Each Operation of the Infrastructure

In section 3.3, we have given the diagram to show how the system switches to an alternative service when the quality of the current service goes down (see Figure 12), and the process of it has been described. In this section, events of the diagram will be described in detail to show how the publisher and subscriber transmit data, and what data they transmit to each other via the local broker. To make descriptions more easily understood, numbers have

been added to each event (see Figure 15), and we will follow the number to describe them.

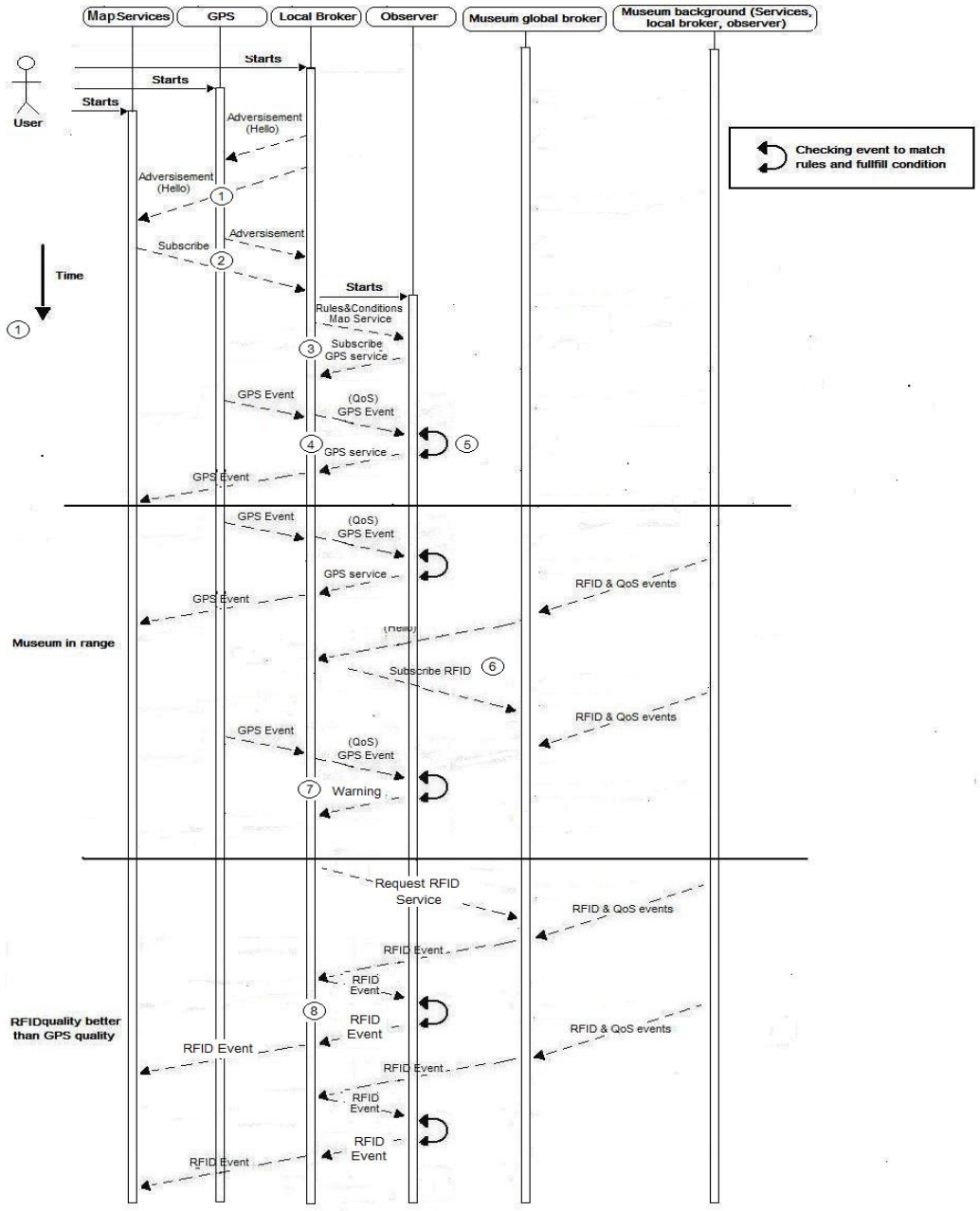

Figure 15: GPS fails and an alternative RFID location service is available

1. Advertisement from local broker

As we have talked about before, the local broker sends a “Hello” advertisement to each service when the user’s mobile device is switched on.

After this event, local broker connects with each different service and wait them to subscribe or publish data. (See example 1)

Event from local broker (Example 1):

- Event ID: Event_Localbroker_001
- Service Name: Local Broker

2. Responds from each service

The response message from each service is different, which depends on if the service is a publisher or a subscriber. Like this example, the GPS service is a publisher, which publishes the location data and it does not require any data from another service. Therefore, the response message from the GPS service is like an advertisement too, which includes the data types that the GPS service offers, and the local broker can subscribe any of those types of data from it (See example 2)

Event from GPS service (Example 2):

- Event ID: Event_GPS_001
- Service Name: GPS Service
- Service ID: GPS001
- Service type: Publisher
  Data type: Location data (Latitude, N/S direction, Longitude, W/E direction)

If the service is a subscriber, it will respond to the local broker what type of data that is needed by this service; also the rules and conditions are included in this response event too. This is because the subscriber has to tell the local broker under which situation the service wants that data. Like this example, the rules and conditions from the map service shows that the response time between each event from publisher needs to be less than ten seconds, and only one location service can be connected at a time (See example 3). In future, the local broker will follow these rules and conditions to send data to this service.

Event from map service (Example 3):

- Event ID: Event_Map_001

- Service Name: Map Service
- Service ID: Map001
- Service Type: Subscriber
  Data type: Location Data (Latitude, N/S direction, Longitude, W/E direction)
- Rule Type: Respond Time
- Rule Factor: <
- Rule Information 10
- Condition Type: Connection number
- Condition Information: 1

3. Communication with the observer

Once rules and conditions are submitted to the local broker, the observer starts. At this time, the rules and conditions from the local broker are forwarded to it. The reason is that the observer is designed in this application to help each subscriber check the quality of the event which is sent to them. (See Example 4)

Event from local broker (Example 4):

- Event ID: Event_Localbroker_002
- Service Name: Local Broker
- Subscriber Name: Map Service
- Rule Type: Respond Time
- Rule Factor: <
- Rule Information: 10
- Condition Type: Connection number
- Condition Information: 1

After the observer gets all the information from the local broker, the rules and conditions information are saved by the observer. Then a response event is sent back to local broker (see example 5). In this event, the observer also subscribes to the types of data which are requested by each subscriber.

Event from observer (Example 5):

- Event ID: Event_Observer_001
- Service Name: Observer
- Service Type: Subscriber
- Data Type: Location Data (Latitude, N/S direction, Longitude, W/E direction)

4. Data transfer

Each services start to work after they finish first communicating with the local broker. Like this diagram, the GPS service starts to send the location data to the local broker (Example 6) and the map service waits for the location data from local broker.

In this application, due to offering the best quality of service is the goal of this application, local broker sends every event to the observer to check the quality of it, and then send to the subscriber if the quality of this event is good enough.

Event from GPS (Example 6):

- Event ID: Event_GPS_002
- Service Type: Publisher
- Data Type: Location Data (Latitude, N/S direction, Longitude, W/E direction)
- Event Content: (37.422006, N, 122.084095, E)

5. Checking rules and conditions

As we have described in the previous section, every time local broker needs to send the event to the observer to check the rules and conditions. And then the event is resends to the subscriber. (See example 7).

Event from local broker (Example 7):

- Event ID: Event_Locatbroker_002
- Data type: Location Data (Latitude, N/S direction, Longitude, W/E direction)
- Destination: Map service
- Event content: (37.422006, N, 122.084095, E)

In this example, we can see the destination is added to this event to show the observer which subscriber will receive this event, because the observer will follow this attribute to find the relative rules and conditions that have been set. In this system, if the data of given event does not match the rules and conditions of their subscriber, a response message will be sent back to the local broker. (See Example 8)

Event from Observer (Example 8):

- Event ID: Event_Observer_002
- Service Name: Observer
- Problem Event ID: Event_Locatbroker_002
- Situation: False

In this event, the Problem Event ID is used to tell local broker what event cannot match the rule and condition.

6. RFID service

As we have described previously, the new service may be available when the user went to a new area, like the diagram shows in Figure 15, we can see an RFID service is offered when the user nears the museum and its global broker sends an advertisement to the user's mobile device. This event is similar to the advertisement which is sent from the GPS service to the local broker, it also includes a list type of data which the local broker can subscribe to (See example 9). At the same time, the user's local broker evaluates those types of data and then sends a subscribe event back to the museum's global broker (See example 10).

Event from the RFID service (Example 9):

- Event ID: Event_RFID_001
- Service Name: RFID Service
- Service type: Publisher
- Data type: Location Data (Latitude, N/S direction, Longitude, W/E direction)

Event from the local broker (Example 10):

- Event ID: Event_Localbroker_003
- Service name: Local Broker
- Service Type: Subscriber
- Data type: Location data (Latitude, N/S direction, Longitude, W/E direction)

7. Communicating with the GPS service

The museum's global broker starts to send the data after the user's local broker subscribed to data from it. But the local broker does not receive this

data, this is because the observer did not send the warning event to request the local broker to change the publisher, therefore, the local broker also communicates with the GPS service. (See example 11).

Event from the RFID (Example11):

- Event ID: Event_GPS_003
- Service type: Publisher
- Data type: Location Data (Latitude, N/S direction, Longitude, W/E direction)
- Event Content: (37.422007, N, 122.084195, E)

8. Connect with the RFID service

In this part, the observer checked the quality of the GPS service is not good enough to publish the data to the Map service, therefore, an event is sent back to the local broker to tell the situation. (See Example 12)

Event from the observer (Example 12):

- Event ID: Event_Observer_002
- Service Name: Observer
- Problem Event ID: Event_Locatbroker_003
- Situation: False

After local broker receives it, a subscribe event is sent to the RFID service to request the location data (See example 13). And then the local broker connects with the RFID service to request data and stops communicating with the GPS service.

Event from the local broker (Example 13):

- Event ID: Event_Localbrker_004
- Service Name: Local Broker
- Service Type: Subscribe
- Data type: (Latitude, N/S direction, Longitude, W/E direction)

Chapter 5

Implementation

This chapter focuses on the implementation details of the system. The first part is to mention the requirements when implementing the application, and some examples are given to make descriptions more clear. Then the database design will be described. Finally, the programmed code will be discussed.

5.1 Hardware/Software Requirements

In this part, the hardware and software which are required when developing the application are showed below.

5.1.1 Mobile Device

Currently, smart-phones have become a trend far surpassing the traditional cell phones. It offers more services to people, like browsing web pages, handling official business like emails etc, as well as some people installing software that have developed by themselves. Therefore, this project will implement an application based on smart phones.

For smart-phones, there are four popular systems that are mainly in used currently, like the Symbian Operating System, Windows Mobile, Palm Operating System and the Android Operating System (Ziff Davis Inc, 2011). Each system has its own advantages, and each of them is used by different brands of mobile phones.

In this project, the mobile device with the Android operating system is chosen as the platform to run the application for this project. This is because it is a nice operating system for mobile devices, like smart phones, PDA and some tablet computers. Also it establishes an open platform for all developers to build innovative mobile applications (Google Inc, 2011).

5.1.2 Android SDK for Java

The Android Software Development Kit (SDK) is necessary when people develop applications on the Android platform using the Java programming language. Normally Android SDK includes development tools, emulator, required libraries and some sample projects (QuinStreet Inc, 2011). In SDK, emulator is a quite useful tool to help people develop applications for the Android system. That is because it replicates a typical interface of the Android system to a user's PC; therefore, people can run the Android applications on their PC instead of the real android phone. For emulator, it still has some limitations that people have noted, like some applications that have touch movements may not work correctly.

5.1.3 Radio-Frequency Identification Tag (RFID)

Radio frequency identification (RFID) is a technology that uses radio waves to communicate between a reader and an electronic tag (see Figure 16) (Wikipedia, 2011), Because its ability to track moving objects, it has been used by thousands of companies currently. Normally, the data transmitted by the tag is dependent on the information that the tag has stored, it may be the identification or location information, or specifics about some product, like the price, color, etc (AIM, 2011). In this way, the RFID tag is used to store the location data for this project, and it is used to offer the indoor location service.

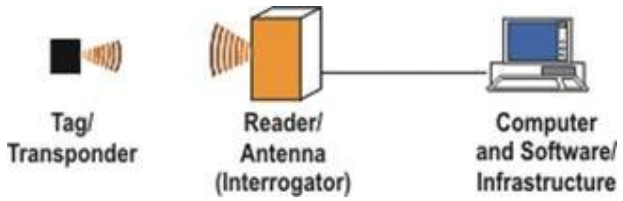

Figure 16: Example of the RFID

5.2 Database Design

In this project, SQLite has been chosen to set up the database for this application. It will store the information of each service; also the events that are transmitted between the subscriber and the publisher. SQLite is a

compact, high efficiency, high reliability, embeddable SQL database engine, and currently it is used in a wide range of commercial software products and electronic devices, because it is lightweight, fast and open source. Therefore, today, SQLite is found today in many mobile phones, MP3 players and some PCs (SQLite, 2011).

In figure 17, it is the ER-diagram of this project's database, which consists of seven entities and many relationships to store the general information of each service. The detailed information of each entity and relations will be described below; also, some examples will be given.

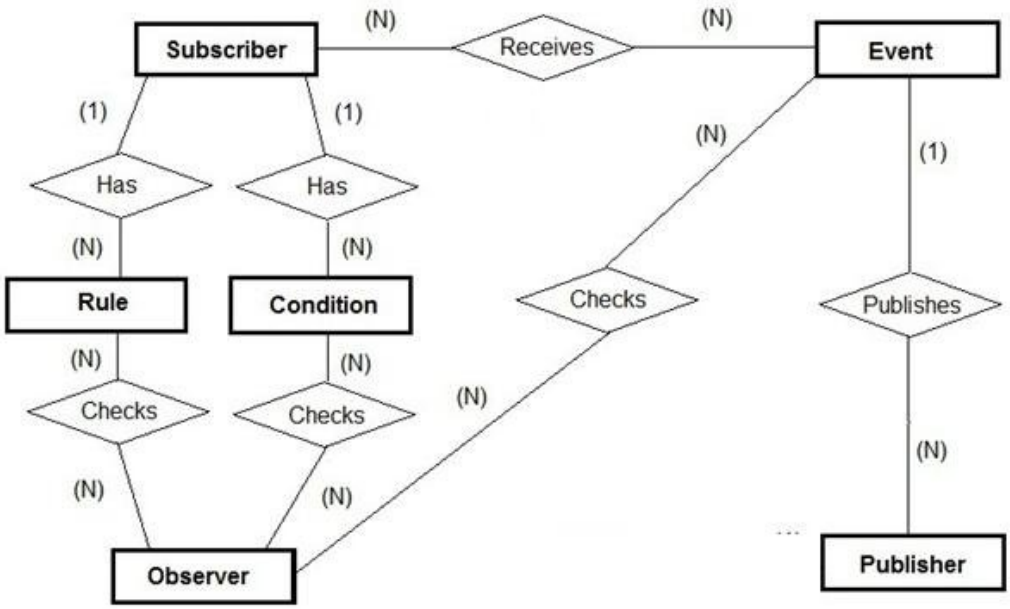

Figure 17: ER Diagram of the Database

5.2.1 Entities of Database:

1. **Entity of subscriber**

   The subscriber's table is to store subscriber's information. The key attributes of this entity are the service ID number, service name, service type, the data type, Rule ID and the Condition ID. The Service ID is the primary key of this table, and each attribute is shown below:

- **Service_ID Text -- Primary Key**
- **Service_Name Text—Not Null**
- **Service_Type Text -- Not Null**
- **Data_Type Text -- Not Null**
- **Rule_ID Text -- Foreign Key**
- **Condition_ID Text -- Foreign Key**

2. **Entity of rule**

The rule table is designed to store the rules of the subscriber, which will be used when observer checks the rules of an event. The key attributes of this entity are the rule ID, rule type, rules Factor and the Rule Information. The type of each attribute is shown below:

- **Rule_ID Text -- Primary Key**
- **Rule_Type Text -- Not Null**
- **Rule_Factor Text—Not Null**
- **Rule_Information Integer -- Not Null**

Rule factor includes two different conditions in this table, which are the (more than '>' and less than '<'). The rule information is a numeric value here, which is normally used with rule factor together. Like the example is shown below, it means the transmitting time between each event form publisher have to be less than five seconds.

Example of rule table

- **Rule_ID: Rule_001**
- **Rule_Type: Response Time**
- **Rule_Factor: <**
- **Rule_ Information: 5**

3. **Entity of condition**

The condition table is designed in this database to store the conditions information of the subscriber. This table is also used by the observerer when it checks the condition of an event. The key attributes of this entity are the Condition ID and condition information. The type of each attribute is shown below:

- **Condition_ID Text -- Primary Key**

- **Condition_Information Integer -- Not Null**

Condition information is the numerical value in this table, and it describes that how many publishers are allowed to communicate with this subscriber at a time. For example, the subscriber allows two publishers to communicate with this subscriber if the value of this attribute is 2.

4. Entity of observer

The observer table is to store the observer's information. In this table, subscriber's rule and condition information are also included due to the observer will use these attributes to find the related rule and condition information when it checks the rule and condition information of the latest arrive event. The key attributes of this entity are the observer ID, subscriber's ID, Name, data type, rule ID and the condition ID. The type of each attribute is shown below:

- **Observer_ID Text -- Primary Key**
- **Subscriber_ID Text -- Not Null**
- **Subscriber_Name Text—Not Null**
- **Subscriber_Data_Type Text -- Not Null**
- **Subscriber_Rule_ID Text -- Not Null**
- **Subscriber_Condition_ID Text -- Not Null**

To make the explanation clearly, the example below shows how the observer stores information of the map service.

Example of observer table

- **Observer_ID: Observer_001**
- **Subscriber_ID: Map_001**
- **Subscriber_Name: Map Service**
- **Subscriber_Data_Type:(Latitude, N/Sdirection, Longitude, W/E direction)**
- **Subscriber_Rule_ID: Rule_001**
- **Subscriber_Condition_ID: Condition_001**

5. **Entity of event**

   The event table is designed to store the information of every event, and it can be reviewed later. The key attributes of this entity include the event ID, event owner, and the content of the event. The type of each attribute is shown below:

   - **Event_ID Text -- Primary Key**
   - **Event_Owner Text -- Not Null**
   - **Event_Content Text -- Not Null**

6. **Entity of publisher**

   The publisher's table is used to store the publisher's information The key attributes of this entity are the publisher's ID number, name, service type and the data type that publisher publishes. The type of each attribute is shown below:

   - **Service_ID Text -- Primary Key**
   - **Service_Name Text -- Not Null**
   - **Service_Type Text -- Not Null**
   - **Data_Type Text -- Not Null**

5.2.2 Relationship between entities

1. **Relationship of subscriber has their rule**

   The relationship between the entity of the subscriber and the rule is that every subscriber can have just one rule, but any rule could be suited by many subscribers. Therefore, it is the relationship of one-to-many.

2. **Relationship of subscriber has their condition**

   The relationship between the entity of the subscriber and the condition is also one-to-many. Every subscriber only has one condition, but any condition can be used by many subscribers.

3. **Relationship of subscriber receives event**

The relationship between the entity of the subscriber and the event is that the subscriber can receive different events at a time, also every event can be sent to different subscribers. Therefore, this relationship is many-to-many and a table is designed for it. The attributes include the event ID and subscriber's ID, also both attributes are the primary key of this table.

4. **Relationship of publisher publishes event**
   The relationship between the event and the publisher is designed to be one-to-many, which means every event has a certain publisher, but the publisher can publish many events.

5. **Relationship of observer checks rule of each event**
   The relationship between the entity of the observer and the rule is many-to-many. The reason is that every rule can be checked by different observers, and the observer can check any rule which is stored in the rule table. Thus, it is a many-to-many relationship.

6. **Relationship of observer checks condition of each event**
   The relationship between the entity of the observer and the condition is many-to-many too, because any condition can be checked by different observers, and the observer can check any condition which is stored in the condition table. Thus, it is a many-to-many relationship.

5.3 Classes and Methods

This part focuses on documentation of the application implementation. The details of each class will be described, which include the declared variables and some important methods.

5.3.1 GPS Class

The purpose of the GPS class in this application is designed as the publisher to publish the location data to other services via the local broker. It therefore

contains the service name, ID, type of service, data type, latitude data, longitude data and some variables that are required in this class.

```
private String ID = GPS001;
private String ServiceName= "GPS_Service";
private String ServiceType = "Publisher";
private String DataType = "Location Data ";
private double latPoint, lngPoint;
private Event GPSEvent;
private LocationListener locationListener;
public static GPSservice mGps;
```

The service name is declared to represent the name of the service; the local broker will read this variable to know which service is currently publishing data. In this application, ID is quite useful when more than one service subscribes location data, and also observer will use this variable to find the matched rule to check the quality of it. Type of the service shows the type of data that this service provides, and local broker will use this variable to match the subscribers and publishers at first time. For example, map service and GPS service will be connected together via local broker, because map service needs the location data that is published from GPS service.

As the report talked in Figure 14, local broker will start to broad advertisement to each service when user starts running this application, and every service will respond that advertisement to publish or subscribe data. In the response event, the basic information about this service is included in it, like the data type, service type, etc. if the service is a subscriber, the rule and condition information are contained as well. Because the GPS service is a publishing service, therefore, it only responds the basic information about this service.

```
public Event ServiceSetUp(Event event){
   GPSEvent = new Event(ID, ServiceType,ServiceName,DataType);
   return GPSEvent;
}
```

Currently, location-based service is one of the key functionality that is used by many mobile applications. In this application, the GPS service class has used the LocationManager to get the location information.

```
public Event LoadCoords(){
   LocationManager myManager = (LocationManager) con.
   getSystemService(Context.LOCATION_SERVICE);

   locationListener = new LocationListener(){
      public void onLocationChanged(Location loc) {
         Log.d(TAG, "LocationListener onLocationChanged");
            if (loc != null) {
               Log.d(TAG, "LocationListener onLocationUnChanged");
             }
      }

      public void onProviderDisabled(String provider) {
         Log.d(TAG, "LocationListener
            onProviderDisabled  PROVIDER:" + provider);
         return;
      }

      public void onProviderEnabled(String provider) {
         Log.d(TAG, "LocationListener
            onProviderEnabled PROVIDER:" + provider);
         return;
      }

      public void onStatusChanged(String provider,
         int status, Bundle extras) {
            Log.d(TAG, "LocationListener onStatusChanged STATUS:"
            + status + " PROVIDER:" + provider);
         return;
      }
};

   myManager.requestLocationUpdates(
      LocationManager.GPS_PROVIDER,
      0,
      0,
      locationListener);

   this.SetLatPoint(myManager.getLastKnownLocation(myManager.
   getProviders(true).get(0)).getLatitude());
   this.SetLngPoint(myManager.getLastKnownLocation(myManager.
   getProviders(true).get(0)).getLongitude());

   GPSEvent = new Event(this.GetLatPoint(),
      this.GetLngPoint(),this.GetServiceID(),
      this.GetServiceName(), this.GetService(),this.GetTime());
   return GPSEvent;
}
```

In this segment of code, LocationManager and LocationListener are initialized at the start. Then the LocationManager.requestLocationUpdatas is used to get the location data, it includes four parameters, which are the provider information, minimum time interval for notifications, minimum distance interval for notifications, and the location listener. This application has chosen the

GPS provider to provide the location data, which also can use other type of provider when get the location data, like the new work location provider. The second and the third parameters are used to set up the minimum time interval between notifications and the minimum change in distance between notifications. In this application, both of them have been set to zero, which means this application requests location notifications as frequently as possible (Android developers, 2011). The last one is the LocationListener, which receives call backs for location updates. After get the location data, the LocationManager.getLastKnowLocaiton is used to get the latitude and longitude data, and then these data are passed to the event class to publish to local broker.

In this application, the goal is to provide the best quality of service to users, and the response time from publisher is one of the most important factors to determine it. Therefore a method has been written to get the system time, and it will be sent to local broker too when GPS service every time sends location data.

```
private String GetTime(){
   Date date = new Date();
   int hour = date.getHours();
   int minute = date.getMinutes();
   int second = date.getSeconds();
   String curTime = hour + ":"+minute + ":"+ second;
   return curTime;
}
```

5.3.2 Map Service Class

Map service is designed as subscriber in this application, which is used to show the location data into map to make user view easily. Because this class is defined as the subscriber in whole application, it has some variables about the rule and conditions to tell local broker what kind of data are required.

```
private String ServiceName= "Map_Service";
private String ServiceType= "Subscriber";
private String DataType = "Location Data";
private String RuleType = "Time";
private String RuleFactor = "<";
private String RuleInformation = "10";
private int Condition = 1;
```

The first two variables are the service name and service type, which are the same as the GPS class, and they are used to tell local broker the name and the type of this service. Data type is declared in this class to tell local broker what kind of data that map service asks for; also local broker will use this variable to find the publisher who publishes location data as well and then help map service to subscribe data from it. Last of four variables are designed for the rule and condition of this service. Because subscribers need to tell local broker what kind of data they want, and under what situation. In this class, rule type shows that the response time is quite important for this service. Rule factor and rule information show the response time that has to be less than ten seconds between each published location data. The condition variable shows that map service only wants to connect with one publisher in any time.

Due to the map service is designed as the subscriber in this application, so the rules and conditions are respond to local broker at the first communicate with the local broker.

```
public Event ServiceSetUp(Event event){
   this.AdvertisementEvent = event;
   Event MapEvent = new Event(ServiceName, ServiceType,
      DataType, RuleType,RuleFactor, RuleInformation,Condition);
   return MapEvent;
}
```

In this application, map service uses Google map (Google Inc, 2011) to show location data to users, also some related functions are offered to make people use it easily. Like the location and service information, zoom in, zoom out, etc. (See Figure 18)

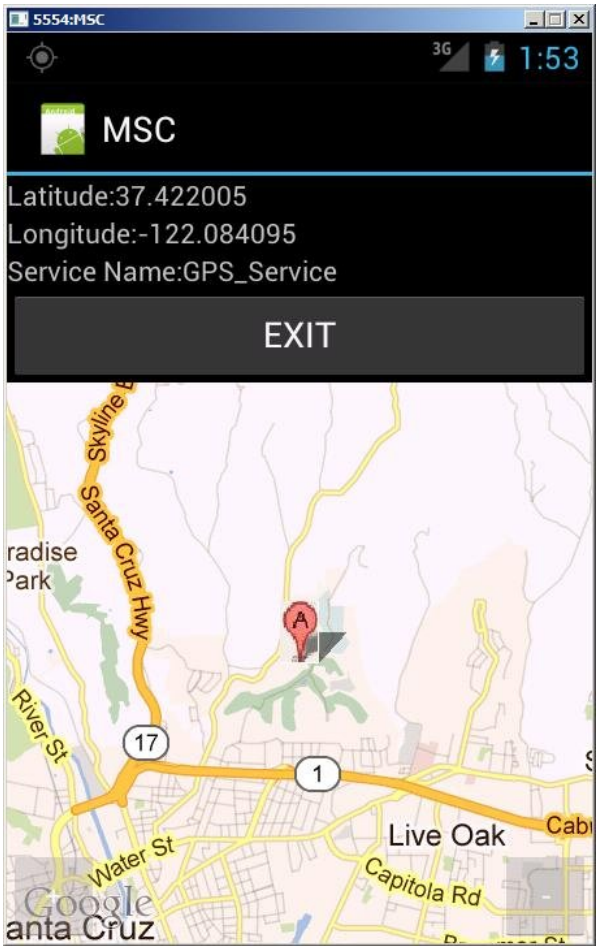

Figure 18: Interface of the application

As the interface design is showed in Figure 18, the related codes about the design are showed below.

```
private void SetUpInterface(){
   final Button exitButton = (Button)findViewById(R.id.ButtonExit);
   exitButton.setOnClickListener(new Button.OnClickListener() {
      public void onClick(View v){
         System.exit(0);
      }});

   final MapView myMap = (MapView) findViewById(R.id.myMap);
   final MapController myMapController = myMap.getController();

   final Button zoomIn = (Button)findViewById(R.id.ButtonZoomIn);
   zoomIn.setOnClickListener(new Button.OnClickListener() {
      public void onClick(View v){
         ZoomIn(myMap,myMapController);
      }});

   final Button zoomOut = (Button) findViewById(R.id.ButtonZoomOut);
   zoomOut.setOnClickListener(new Button.OnClickListener() {
```

```
        public void onClick(View v){
           ZoomOut(myMap,myMapController);
        }});
    }
```

In this part of code, MapView is used to display the map, and MapController is used to pan and scale the map. In map controller, the zoom range is from one (smallest size) to twenty one (largest size), therefore, both the zoom in and zoom out method are required to determine the current scale level is out of range or not. In this application, user can scale the map size in one level every time if the zoom level is in the zoom level range.

```
public void ZoomIn(MapView mv, MapController mc){
   if(mv.getZoomLevel()!=21){
      mc.setZoom(mv.getZoomLevel()+ 1);
   }
}

public void ZoomOut(MapView mv, MapController mc){
   if(mv.getZoomLevel()!=1){
      mc.setZoom(mv.getZoomLevel()- 1);
   }
}
```

When map service receives the location data from local broker, it will process those data and then display them on the map.

```
MapView myMap;
public void SetCoords(Double Lat, Double Long,String servicename){
   TextView latText = (TextView) findViewById(R.id.TextView02);
   TextView lngText = (TextView) findViewById(R.id.TextView04);
   TextView serviceText = (TextView) findViewById(R.id.TextView06);

   latText.setText(Lat.toString());
   lngText.setText(Long.toString());
   serviceText.setText(servicename);

   myMap = (MapView) findViewById(R.id.myMap);
   int latInt = (int )(Lat.intValue() *1E6);
   int lngInt = (int)(Long.intValue() *1E6);
   GeoPoint myLocation = new GeoPoint(latInt , lngInt);
   MapController myMapController = myMap.getController();

   myMapController.setCenter(myLocation);
   setOverlay(myLocation);
   myMapController.setZoom(15);
}
```

In this application, it mainly focus on the mobile service collaboration, therefore, some related information, such as the coordinate data and the service information, are also displayed to users to make them clearly understand what service is being used currently. In the interface design, three text views are used to represent the details (see Figure 18), as we can see the start of this method; each text view is used to set the different information. Then the coordinate data are timed 1E6 before they are set into the GeoPoint, this is because the GeoPoint only accepts integer. Finally the GeoPoint is set on the centre of the map to display to users, and the initial zoom size is set in level fifteen.

Overlay is used in this application to mark the position that people currently locate, users can click the position balloon to see more detailed information around this point. In this application, it displays the coordinate data to users.

```
private void setOverlay(GeoPoint myPoint,
   String ServiceName, double lat, double lng){

   List<Overlay> mapOverlays = myMap.getOverlays();
   Drawable drawable = this.getResources()
   .getDrawable(R.drawable.ic_launcher);
   MapServiceOverlay itemizedoverlay =
   new MapServiceOverlay(drawable,this);
   OverlayItem overlayitem = new OverlayItem(myPoint, ServiceName,
   "Coordinate: (" + lat+","+ lng +")");
   itemizedoverlay.addOverlay(overlayitem);
   mapOverlays.add(itemizedoverlay);
}
```

Firstly, an Overlay list is declared to store the overlay item objects that are required to be put on the map, then an overlay image is loaded, because it will be showed on the map to mark the user's position. Then the related information of this overlay is added. The related information can be everything, like the tourist information around this location, traffic information etc, which is depends on what information that designer wants to show users.

5.3.3 MapServiceOverlay Class

The overlay class is used to mark the coordinate on the map to make user can view it easily. There are two variables have been declared, which is an overlay item list which is used to hold all overlay objects, and another variable is declared to hold the context information.

```
private ArrayList<OverlayItem> myOverlays =
   new ArrayList<OverlayItem>();
private Context myContext;

public MapServiceOverlay(Drawable defaultMarker, Context context)
{
   super(boundCenterBottom(defaultMarker));
   myContext = context;
}
```

In this class, an addOverlay method is defined to add the overlay into the myOverlays list, and then it performs all processing on that new overlay.

```
public void addOverlay(OverlayItem overlay)
{
   myOverlays.add(overlay);
   populate();
}
```

Because this class extends the ItemizedOverLay class, some methods need to override the method from their super class. Like the createItem method returns corresponding overlay items when this method is called, size method returns the size of it, and the onTap method which is used to show the related information about the overlay, and it is displayed in a dialog box.

```
@Override
protected OverlayItem createItem(int i)
{
   return myOverlays.get(i);
}

@Override
public int size()
{
   return myOverlays.size();
}

@Override
protected boolean onTap(int index){
   OverlayItem item = myOverlays.get(index);
   AlertDialog.Builder dialog =
   New AlertDialog.Builder(myContext);
```

```
            dialog.setTitle(item.getTitle());
            dialog.setMessage(item.getSnippet());
            dialog.show();
            return true;
        }
```

5.3.4 Event Class

Event class is defined in this application to help each service communicate with the local broker; it is like a ‘carrier’ to make the data transmission easily. Because each service has the different number of information when they send or receive data with local broker. Therefore, event class has many different constructors to suit that situation.

As we described in chapter 4, when application starts running, local broker sends an advertisement to each service to get each service’s information to connect related publishers and subscribers. At this time, event class is used to carry the advertisement information to every service.

```
public Event(String servicename){
    this.ServiceName = servicename;
}
```

In this message, it only contains the service name to show each service who broadcast this advertisement, but the response message concludes detailed information of each service. Like the map service is a subscriber in this application, and it wants to subscribe the location data from local broker. Therefore, in this message it not only respond some basic information about the service, but also respond the rules and conditions to tell local broker what kind of data are required, and under what situation.

```
private String ServiceName;
private String DataType;
private String ServiceType;
private String RuleType;
private String RuleFactor;
private String RuleInformation;
private int Condition;
public Event(String servicename, String servicetype,
    String datatype,String ruletype, String rulefactor,
    String ruleinformation,int condition){

    this.ServiceName = servicename;
    this.ServiceType = servicetype;
```

```
        this.DataType = datatype;
        this.RuleType = ruletype;
        this.RuleFactor = rulefactor;
        this.RuleInformation = ruleinformation;
        this.Condition = condition;
    }
```

The structure of responding message from publisher is simpler than subscriber; it only contains some basic information, like the ID of the service, the name of the service, type of the service and the type of the data.

```
    public Event(int serviceid,String servicename,
        String servicetype, String datatype){

        this.ServiceID = serviceid;
        this.ServiceType = servicetype;
        this.ServiceName = servicename;
        this.DataType = datatype;
    }
```

After the first communication, local broker connects the related services together, and then wait for publishers to publish data.

```
    private double Lat;
    private double Lng;
    private String Time;

    public Event(double lat, double lng, int serviceid,
    String servicename,String servicetype, String time){
        this.Lat = lat;
        this.Lng = lng;
        this.ServiceID = serviceid;
        this.ServiceType = servicetype;
        this.Time = time;
        this.ServiceName = servicename;
    }
```

When GPS starts to publish the location data to local broker, the variables of the event is different with the respond message that GPS service responds to local broker at the first time. The data type is removed, and it is instead by the coordinate data, also the sending time is included in this event.

After local broker receives the event, it sends the event to the observer to check the rule and condition of this event. In this event the destination service

is added, which is used to let observer know who subscribed this event, and observer will find the rules and conditions of this subscriber.

```
public Event(double lat, double lng, int serviceid,
   String servicename,String servicetype, String time,
   String destination){
   this.Lat = lat;
   this.Lng = lng;
   this.ServiceID = serviceid;
   this.ServiceType = servicetype;
   this.Time = time;
   this.ServiceName = servicename;
   this.DestinationService = destination;
}
```

Observer is designed to check the quality of the service in this application, so the checked result of this event is added in the response event form the observer to local broker.

```
public Event(double lat, double lng, int serviceid,
   String servicename, String typeofservice, String time,
   boolean checkedresult){

   this.Lat = lat;
   this.Lng = lng;
   this.ServiceID = serviceid;
   this.ServiceType = typeofservice;
   this.Time = time;
   this.ServiceName = servicename;
   this.CheckedResult = checkedresult;
}
```

5.3.5 Observer Class

In this application, observer is defined to help subscribers to check the quality of the service who publishes data to this subscriber. The variables of this class conclude some basic variables which are used to store the information of the rule and condition when observer is checking the quality of a service; also some variables are declared to hold the information which is used by the methods of this class. Like the destination service holds the name of the service who subscribes this event, LastSendingTime and LastPublisher are used to hold the information about who published last event and the sent time of it.

```
private String ServiceName;
private String ServiceType;
private String DataType;
```

```
private String RuleType;
private String RuleFactor;
private int RuleInformation;
private int Condition;

private String DestinationService;
private Event RuldandConditionevent;
private String LastSendingTime;
private String LastPublisher;
private MSCdb mscdb;
int num = 0;
```

When local broker sends advertisement to observer at the first time, subscriber's rule and condition information are included in that event, and observer will save those data into the database.

```
public Observer(Event event,Context con){
   this.RuldandConditionevent = event;
   SavingRuleandCondition(this.RuldandConditionevent, con);
}

private void SavingRuleandCondition(Event
   ruleandconditionevent, Context con){
   mscdb = new MSCdb(con);

   mscdb.insert_rule_table(ruleandconditionevent.GetServiceName()
      , ruleandconditionevent.GetRuleType()
      , ruleandconditionevent.GetRuleFactor()
      , ruleandconditionevent.GetRuleInformation());
   mscdb.insert_condition_table(
      ruleandconditionevent.GetServiceName(),
      ruleandconditionevent.GetCondition());
   num++;
}
```

After the first communicate with observer, local broker will pass the event to observer to check the rule and condition when the new event is published from publisher.

```
public boolean RuleandConditionCheck(Event event,
   String distinationservice){
   this.DestinationService = distinationservice;
   Cursor RuleCursor  = mscdb.select("rule_table");
   Cursor ConditionCursor = mscdb.select("condition_information");
   for (int i = 0;i<num; i++){
      RuleCursor.moveToPosition(i);

      if((RuleCursor.getString(1)).
         equals(this.DestinationService)){
         this.ServiceName = RuleCursor.getString(1);
         this.RuleType = RuleCursor.getString(2);
         this.RuleFactor = RuleCursor.getString(3);
         this.RuleInformation = RuleCursor.getInt(4);
```

```
            this.Condition = ConditionCursor.getInt(1);
         }
      }

         if((this.RuleCheck(event.GetTime())
            && this.ConditionCheck(event.GetServiceName())) == true){
            if(this.ServiceName.equals("GPS_Service")){
               return true;
            }else{
               return false;
            }
         }else{
            return false;
         }
   }
```

In this method, every time local broker passes the new event and the destination service's name to observer, the rule and condition information of the destination service are selected out from the corresponding tables. Then observer starts to check the event satisfy the rule and condition of the destination service or not.

```
private boolean RuleCheck(String sendingtime)
boolean CheckedResult=true;
if (LastSendingTime.equals("")){
   return true;
}else{

int TempSendingTime = this.GetHour(sendingtime) *
   3600 +this.GetMin(sendingtime)*60 +
this.GetSecond(sendingtime);

int TempLastSendingTime = this.GetHour(LastSendingTime)
   * 3600 +this.GetMin(LastSendingTime)*60
   + his.GetSecond(LastSendingTime);

   if(this.RuleType.equals("Time") &&
      this.RuleFactor == "<"){

      if((TempSendingTime - TempLastSendingTime  )
         <= this.RuleInformation){
         CheckedResult = true;
      }else{
         CheckedResult = false;
      }
    }
}
   return CheckedResult;
}

private boolean ConditionCheck(String servicename){
   if(LastPublisher.equals(""))
      return true;
   else{
```

```
            if(LastPublisher.equals(servicename)){
                return true;
            }else{
                return false;
            }
        }
    }
```

Because the rule of the subscriber focus on the respond time from the publisher, some methods are written in this class to help observer to calculate the time when new event is coming.

```
    public int GetSecond(String currenttime){
        String TempSecond;
        int length = currenttime.length();
        String TempString = currenttime.substring(length-2, length-1)
        if(TempString.equals(":")){
            TempSecond = currenttime.substring(length-1, length);
        }else{
            TempSecond = currenttime.substring(length-2, length);
        }
        return Integer.parseInt(TempSecond);
    }

    public int GetMin(String currenttime){
        String TempMin;
        int length = currenttime.length();
        String TempString = currenttime.substring(length-2, length-1);
        if(TempString.equals(":")){
            if(currenttime.substring(length-4,length-3).equals(":")){
                TempMin = currenttime.substring(length-3, length-2);
            }else{
                TempMin = currenttime.substring(length-4, length-2);
            }
        }else{
            if( currenttime.substring(length-5,
                length-4).equals(":")){
                TempMin = currenttime.substring(length-4, length-3);
            }else{
                TempMin = currenttime.substring(length-5, length-3);
            }
        }
        return Integer.parseInt(TempMin);
    }

    public int GetHour(String currenttime){
        String TempHour;
        String TempString = currenttime.substring(1,2);
        if(TempString.equals(":")){
            TempHour = currenttime.substring(0, 1);
        }else{
            TempHour = currenttime.substring(0, 2);
        }
        return Integer.parseInt(TempHour);
    }
```

5.3.6 Database Class

In this project, SQLite has been chosen to create the database for this application. As the report motioned in data base design (see section 5.2), it stores the information that are required when application is running. For instance the rule and condition are saved into database by observer when a new subscriber join in, and they will be used later when observer checks the quality of the service of the publisher who publishes data to this subscriber.

```
private final static String DATABASE_NAME = "MSC.db";
private final static int DATABASE_VERSION = 1;

//variables of the rule table
private final static String Rule_Table_Name = "rule_table";
public final static String Rule_ID = "rule_id";
public final static String Rule_Owner = "rule_owner";
public final static String Rule_Type = "rule_type";
public final static String Rule_Factor = "rule_factor";
public final static String Rule_Information = "rule_information";

//variables of the condition table
private final static String Condition_Table_Name = "condition_table";
public final static String Condition_ID = "condition_id";
public final static String Condition_Owner = "condition_owner";
public final static String Condition_Information
   = "condition_information";
```

The first two variables are the name and the version of the data base, which are used to create the database when application starts. The rest of variables are the attributes of the rule and condition table. Rule id and the condition id is the primary key of each table, and the owner represents who owns this rule and condition. The rest variables are used to save the corresponding information.

Every time when database class is called, the database will be created if the database does not exist, and the related table will be built.

```
public MSCdb(Context context) {
   // TODO Auto-generated constructor stub
   super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

 public void onCreate(SQLiteDatabase db) {
    String sql_rule = "CREATE TABLE " + Rule_Table_Name + " ("
```

```
            + Rule_ID + " INTEGER primary key autoincrement, "
            + Rule_Owner + " text, "
            + Rule_Type + " text, "
            + Rule_Factor + " text, "
            + Rule_Information +" text);";

        String sql_condition = "CREATE TABLE " +
            Condition_Table_Name + " (" +
            Condition_ID + " INTEGER primary key autoincrement, " +
            Condition_Owner + " text, " +
            Condition_Information +" INTEGER);";

        db.execSQL(sql_rule);
        db.execSQL(sql_condition);
    }
```

In database class, a method called ‘onUpgrade’ overrides the method from its super class, which is used to upgrade the data base when new version is coming. In this method, it drops all tables firstly, and then rebuilds the database.

```
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
        int newVersion) {

        String sql_rule_table = "DROP TABLE IF EXISTS "
            + Rule_Table_Name;
        String sql_condition_table = "DROP TABLE IF EXISTS "
            + Condition_Table_Name;

        db.execSQL(sql_rule_table);
        db.execSQL(sql_condition_table);
        onCreate(db);
    }
```

Every time if any class needs the information from any table, the ‘select’ method is called. The parameter of this method only requires the table name, then all information about that table will be returned, and the information is saved in a cursor.

```
    public Cursor select(String Table_Name) {
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor cursor = db
            .query(Table_Name, null, null, null, null, null, null);
        return cursor;
    }
```

When observer receives the event from local broker at the first time, all subscribers’ rule and condition are included in this event. In this time,

observer saves those rule and condition information into the data base, and use them later.

```
public void insert_rule_table(String rule_owner,String rule_type,
   String rule_factor, String rule_information)
{
   SQLiteDatabase db = this.getWritableDatabase();
   /* ContentValues */
   ContentValues cv = new ContentValues();

   cv.put(Rule_Owner, rule_owner);
   cv.put(Rule_Type, rule_type);
   cv.put(Rule_Factor, rule_factor);
   cv.put(Rule_Information, rule_information);
   db.insert(Rule_Table_Name, null, cv);
 }

 public void insert_condition_table(String condition_owner,
    String condition_information)
 {
    SQLiteDatabase db = this.getWritableDatabase();
    /* ContentValues */
    ContentValues cv = new ContentValues();
    cv.put(Condition_Owner, condition_owner);
    cv.put(Condition_Information, condition_information);
    db.insert(Condition_Table_Name, null, cv);
 }
```

The database class still provides the delete function and it will be used when subscriber changes their rule or condition in some case.

```
public void delete(String table_name, String owner)
{
   SQLiteDatabase db = this.getWritableDatabase();
   if(table_name.equals("Rule_Table_Name")){
      String where = Rule_Owner + " = ?";
      String[] whereValue ={owner};
      db.delete(table_name, where, whereValue);
   }else{
      String where = Condition_Owner + " = ?";
      String[] whereValue ={owner};
      db.delete(table_name, where, whereValue);
   }
}
```

5.3.7 Local broker Class

Local broker is like the agent of the application, which communicates with both publishers and subscribers to help them transmit data.

```
private GPSservice gpsservice;
private MapService mapservice;
private Observer observer;
private Event event;
OtherServices otherservice= new OtherServices();
public boolean switchservice = true;
Context context;
private String destinationservice;
private String ServiceName = "Local Broker";
Event dataevent;
ArrayList<Event> arraylist = new ArrayList<Event>();
```

When this application starts running, local broker broadcasts an advertisement to each service to set up the communication with them. From the response events, local broker selects the services that is subscriber, and then pass those events to the observer class to let it save the rules and conditions; also the communication between the local broker and observer is set up. After local broker sets up connection with all services, it starts helping subscriber to requests data from publisher, and observer starts to check every event as well.

```
private void Advertisement(){
   event = new Event(this.ServiceName);
   Event GPSRespondEvnet;
   gpsservice = new GPSservice(this.context);
   GPSRespondEvnet = gpsservice.ServiceSetUp(event);

   Event MapRespondEvnet;
   mapservice = new MapService();
   MapRespondEvnet = mapservice.ServiceSetUp(event);

   for(int i=0;i<arraylist.size();i++){
      if((arraylist.get(i).GetServiceType()).equals("Subscriber"))
      {
         observer.ServiceSetUp(arraylist.get(i));
      }
   }

   if(GPSRespondEvnet.GetDataType().equals
      (MapRespondEvnet.GetDataType()) &&
      MapRespondEvnet.GetServiceType() !=
      GPSRespondEvnet.GetServiceType())

      this.RequestData();
   }
```

In this application, GPS service is supposed as the best service to offer the location data, and local broker communicates with it when application starts.

In the real situation, sometimes it may be effect by some factors, so other services will instead it to publish the location data.

```
public void RequestData(){
   while(true){
      if(this.switchservice == true){
         dataevent = gpsservice.LoadCoords();
         mapservice.SetLocationData(dataevent);
         this.switchservice = observer.RuleandConditionCheck
            (dataevent, destinationservice);
      }else{
         Event OtherServiceRespondEvnet;
         OtherServiceRespondEvnet=
            otherservice.ServiceSetUp(dataevent);
         if(OtherServiceRespondEvnet.GetServiceType()
            .equals("Publisher")){
            dataevent = otherservice.LoadCoords(dataevent);
            mapservice.SetLocationData(dataevent);
            this.switchservice = observer.RuleandConditionCheck
               (dataevent, destinationservice);
         }
      }
            try {
               Thread.sleep(4000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
   }
}
```

5.3.8 Other Service Class

As the thesis talked about in previous, the RFID has been supposed as an indoor location service to offer the data. Due to the time problem, this project does not implement this part, and an example class which is called other service is defined to instead of it.

The variables of this class is similar as the GPS class, which includes the service id, service name, data type and the service type, as well as some variables which is needed for this class.

```
private int ServiceID = 1;
private String ServiceName= "Other_Service";
private String ServiceType = "Publisher";
private String DataType = "Location Data";
private Event OtherServiceEvent;

public OtherServices(){}
```

This class still has the ServicSetUp method which is used to receive the advertisement from local broker when it first time communicates with it.

```
public Event ServiceSetUp(Event event){
   OtherServiceEvent = new Event(
      ServiceID, ServiceType,ServiceName,DataType);
   return OtherServiceEvent;
}
```

When local broker starts requesting location data from this service, the LoadCoords() method is called, and it responds the same type of event as the GPS service to local broker. Like the coordinate and some information which describes this service.

```
public Event LoadCoords(Event event){
   this.SetLatPoint(event.GetLatitude());
   this.SetLngPoint(event.GetLongitude());
   OtherServiceEvent = new
      Event(this.GetLatPoint(),this.GetLngPoint(),
   this.GetServiceID(),this.GetServiceName(),
   this.GetService(),this.GetTime());
   return OtherServiceEvent;
}
```

In this application, response time from publisher is the main factor to affect the quality of the service. To make the service collaboration effect more clearly, the sending time of this service is increased in one second when it every time sends the event to local broker. For instance, we suppose the time range between the first sending time and the seconding time is three seconds, in this service, it will increase in one second between the second sending time and the third sending time. In this way, this service wills not satisfy the rules and conditions of their subscriber at a point, and the local broker will switch back to the GPS service to request location data from it.

```
int tempSecond = 1;
int second, minute,hour;

private String GetTime(){
   Date date = new Date();
   if(date.getSeconds() != 59){
      second = date.getSeconds()+tempSecond;
      hour = date.getHours();
      minute = date.getMinutes();
```

```
         }else{
            second = 0;
            if (date.getMinutes()!=59){
               minute = date.getMinutes()+1;
               hour = date.getHours();
            }else{
               minute = 0;
               if(date.getHours() != 23){
                  hour = date.getHours() +1;
               }else{
                  hour = 0;
               }
            }
          }
          String curTime = hour + ":"+minute + ":"+ second;
          return curTime;
        }
```

5.3.9 AndroidManifest.xml

In this application, the set up information is stored in an xml file. Some services that are offered by Android SDK are required to be given the permissions when it is used in this application. The permission of this application is showed below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.msc"
      android:versionCode="1"
      android:versionName="1.0">
   <uses-sdk android:minSdkVersion="14" />
   <uses-permission android:name=
      "android.permission.ACCESS_COARSE_LOCATION"/>
   <uses-permission android:name="android.permission.INTERNET" />
   <uses-permission  android:name="android.permission.ACCESS_FINE_LOCATION"/>

   <application android:icon="@drawable/ic_launcher" android:
      label="@string/app_name">
     <uses-library android:name="com.google.android.maps" />
       <activity android:name=".MapService"
         android:label="@string/app_name">
         <intent-filter>
           <action android:name="android.intent.action.MAIN" />
           <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
       </activity>
     </application>
</manifest>
```

In this application, the android version code and version name are both set in version one, the minimal SDK version of android is set in version fourteen, which means this application only can be compiled on version fourteen or

higher. Then some permission are set up for this application to get the location data when it is running, like the ACCESS_COARSE_LOCATION is declared to give the permission that this application can access the CellID or WiFi to get the location. Permission.INTERNET is used to allow this application to mange the application tokens in the windows manager. The permission.ACCESS_FINE_LOCATION is used to allow this application to receive the high quality of the location data, like the GPS. Finally some attributes about the application are set, like the application icon, name. Because the Google map is used in this application, therefore the library of the Google map is set here as well.

Chapter 6

Conclusion and Future Works

In the following, we present the conclusion based on what was accomplished and point out opportunities for future work.

6.1 Conclusion

Currently, the mobile devices offer many different services to their users, but some of them are only available locally, or the quality of the service becomes worse when the user is out of the service's range. Therefore, the services need to collaborate together to offer a good service to users, and it can not reveal the user's privacy.

In this project, three types of architecture have been designed to satisfy the service collaboration, which are: the included broker, remote broker and the intelligent gateway – distributed broker. To compare them, the included broker architecture was selected as the target architecture to implement due to it being the best in terms of anonymity and usability in different scenarios, and it also meets the requirements for quality of service. Quality of service normally uses rules and conditions to ensure the quality of the service collaboration in the selection of alternative services and to ensure liveliness of services when dealing with changing availability of collaboration services.

The java program with the Android SDK has been used to implement this application, and it is quite useful to represent the principle of the proposed service collaboration infrastructure. The application includes four parts, which are: the subscribing service, the local broker, the observer and the publishing service. The application runs as the process that is described in Figure 15.

6.2 Future Work

The application that we have implemented in this thesis can be considered successful to some extent. However, due to time and space constraints, there are still some parts that can be implemented better. We would like to point this out for future work.

6.2.1 Implementation

In this project, this application just implemented the basic functions. Some parts still can be implemented further, which are:

Support more Types of the Rules and Conditions

In this application, only one type of rule and condition is designed to satisfy the situation which is described in scenarios. In fact, the different types of services should have their own rules and conditions, so that more service can be a part of this application, therefore, more types of rules and conditions should be designed, and each service can choose their own type to restrict their data.

Full Database Implementation

In this project, the full data base has been designed in Section 5.2, but only parts of them are implemented in the application. In future work, the rest of the data base should be implemented. This would allow more functions to be offered to users.

Add more services

This application does not have the delay problem due to it only implementing two services. In real situations, there may be many services collaborating together; therefore, the data transmission may be affected due to too many services transferring data at the same time. In this way, more services should be added to this application to test it and improve its function.

Implement the RFID service

The RFID location service has been chosen as the indoor location service to offer the location data to support the map service to show users the location data when GPS becomes worse. Due to the time constraints, this part was not implemented in this project, in future work, this part should be implemented to show the real situation of the scenario which was talked about.

6.2.2 Architectures

There are three ways to place the broker within our architecture, and each of them has their own advantages in different situations. In this project, it only implements the included broker architecture. Therefore, the advantages of the other two cannot be shown in this project. In future work, implementing all of them, and then comparing them to get the most accurate idea of which one is best under what situation would be ideal.

Bibliography

AIM. (2011). *What is RFID*. Retrieved from http://www.aimglobal.org/technologies/RFID/what_is_rfid.asp

Android developers. (2011). Obtaining User Location. Retrieved from http://developer.android.com/guide/topics/location/obtaining-user-location.html

Google Inc. (2011). *Android everywhere*. Retrieved from http://www.android.com/developers/

Google Inc. (2011). About Google Maps. Retrieved from http://support.google.com/maps/bin/answer.py?hl=en&answer=7060

Hinze,A., Michel,Y., & Eschner,L. (2009). *Event-based communication for locationbased service collaboration*. In: ADC, vol. 92, pp. 127-136

Hinze,A., Rinck,R., and David,S. (2010). *Anonymous Mobile Service Collaboration: Quality of Service*. University, Hamilton. New Zealand.

QuinStreet Inc.(2011). *Android SDK*. Retrieved from http://www.webopedia.com/TERM/A/Android_SDK.html

Rinck.M. (2010). *Implementing an Event-driven Service-oriented Architecture in TIP* (Unpublished master's thesis). University of Waikato, Hamilton, New Zealand.

Robosoft. (2011). *NMEA 0183 Interface Standard*. Retrieved from http://www.robosoft.info/en/technologies/knowledgebase/nmea0183

Scientific Coponent. (2010). *NMEA 0183 and GPS: Decoding the NMEA 0183 standard in your GPS Software Project*. Retrieved from http://www.scientificcomponent.com/nmea0183.htm

SQLite. (2011). *Distinctive Features of SQLite*. Retrieved from http://www.sqlite.org/different.html

Technovelgy LLC. (2011). *What is RFID?* Retrieved from http://www.technovelgy.com/ct/Technology-Article.asp

Wikipedia. (2011). *Radio-frequency identification*. Retrieved form http://en.wikipedia.org/wiki/Radio-frequency_identification

Ziff Davis Inc. (2011). *Smartphone Operating System*. Retrieved from http://www.geek.com/smartphone-buyers-guide/operating-system/