Project Write-up

Project Title : Implementing a Local Mobile Web Server Gateway

Written By : Ndakunda Shange-Ishiwa (605n5057)

Supervisor : Mrs Madeleine Wright

Submitted in partial fulfilment of requirements of the degree

Bachelor of Science (Honours) in Computer Science

At Rhodes University

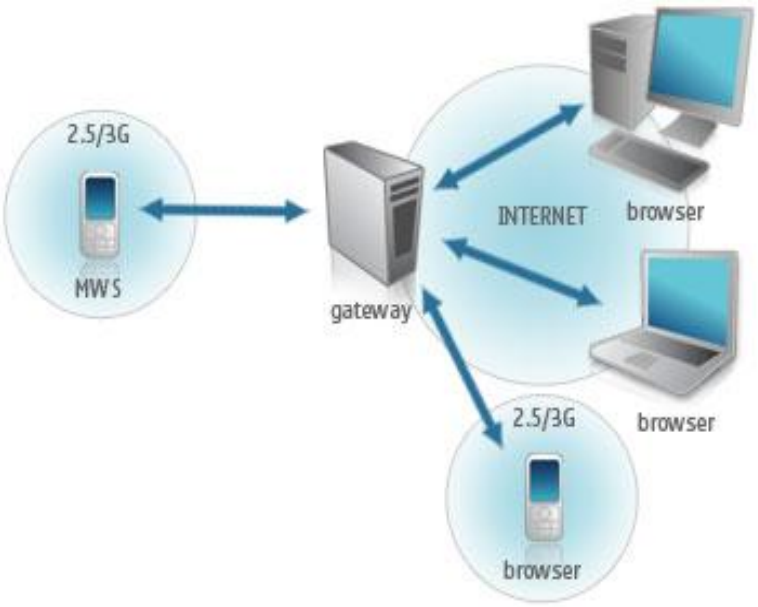

Diagram By Nokia Corporation - 2007

November 2008

Table of Contents

Table of Figures

Acknowledgements

I would like to thank my supervisor, Mrs. Madeleine Wright for all the guidance and support throughout the year. I would also, like to thank my friends and all my class mates especially, Curtis Sahd, Takayedwa Gavaza, Ray Musvibe, Flora Panjou-Tasse, Sinini Ncube, Bwini Mudimba the wonderful support they've given me the whole year support they gave me. I acknowledge the financial and technical support of this project of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University. Last but not least I would like to thank my parents and my family for being behind me all the time.

Abstract

With the ever increasing memory and processing power, mobile phones have become the latest breed of machines on which web servers reside. These servers cannot be accessed from outside a mobile operator's firewall unless there's a gateway that enables clients to connect to them. The main objective of the project is to make mobile web servers on 3G (Symbian series 60) phones both addressable and accessible to internet clients using a local gateway. The gateway discussed in this thesis allows for mobile web server registration, account management and routes requests for the web clients to the right web servers. This thesis looks at the challenges and dynamics involved into building a mobile web server gateway like the one set up locally.

Chapter 1

Introduction

1. 1 Background

Since the launch of the Internet, web servers have been used to serve HTML pages to requesting clients over the internet. These web servers, until recently, were mostly located on stationary computer systems with large processing capabilities, large executable memory and storage space. With increasing processing power and memory, mobile phones have come to be the latest breed of machines on which web servers reside. The introduction of mobile web servers means that the position of the server now matters. Responses to requests could depend on the location of the server and content generated dynamically on the mobile device. This has opened doors for new applications and opportunities for users to manage their web sites on their mobile devices.

Nokia initiated research projects which resulted in the development of the Raccoon and the Nokia Web Servers. The Raccoon web server is an open source product that is available for further development and modification by willing developers. The Nokia web server, on the other hand, is a finished product that has ready-to-use content creation applications and has grown from the continuous development of the Raccoon server.

Nokia manages the mobile web server connections with a gateway that facilitates connections with clients. The gateway manages domain registration, connections and also provides caches for static data to improve latency. It provides a single point of access to the mobile web servers. The proposed local gateway will perform all these functions. Additionally, the research effort will also focus on improving on them as well as finding and exploring new ways of using the gateway.

1.2 Problem Statement

The problem with mobile web servers on mobile phones is that they are not readily accessible or addressable from the internet. A gateway that is connected to the internet is needed as the intermediate point between the requesting web browsers and the web servers on a network run by a mobile operator. For security and efficiency reasons, operators use network address retranslation (NAT) and restrict access at their firewalls. The gateway circumvents these restrictions in conjunction with the mobile connector on the mobile phone to provide an efficient way of serving web resources to internet users.

1.3 Objective of Research

The objective of this research project is to develop a gateway for mobile web servers locally. Mobile devices serving web content cannot be directly accessed without a system that manages and updates their connection information. The local gateway, residing on a computer system, will perform network functions that will enable the web servers to serve their resources to requesting clients. Once registered on the gateway, the mobile web server's connection details are stored and updated periodically. This enables the web server to be accessed with a URL by clients on the internet. It will mainly handle domain registration, connection and access management, request routing and caching. This will afford mobile web server users, both local and international, the luxury of having their server information managed on the local gateway. This information can then be used to facilitate the flow of HTTP requests and subsequent responses between the mobile server and the client. The research effort will also look at Global Positioning System (GPS) to provide web server location information on the gateway.

1.4 Document Structure

This dissertation first discusses the technologies that are closely related to mobile web server gateways. These range from web servers to other gateways that are in this computer

networking area of study. From the third chapter, the dissertation goes on to discuss the gateway in detail. It takes a look at the issues that needed to be considered and properly understood before the gateway could be put together. The fourth chapter discusses the process of modifying and customizing the gateway before setup. The fourth chapter also explains the process of setting and building the gateway. Chapter five focuses on the implementation of the location based module. Testing is done in chapter seven to show that the gateway delivers web content from mobile web servers on cellular networks. The final chapter, seven, concludes the dissertation and discusses ideas that could be implemented in the future.

2. Chapter 2

Related Work

2.1 Introduction

In the early 1970s, The United States Defence Advanced Research Projects Agency (ARPA) developed a wide-area computer network known as ARPANET. ARPANET was a packet-switched interconnection of computers located at various Universities, research agencies in the States and a few selected NATO countries. By the end of the millennium, the network which had evolved into the Internet had millions of hosts connected to it [2]. This system of connected networks had vast amounts of information on just about anything. Social, commercial, technological and educational information resources were shared and made available to a world-wide audience. Central to this world-wide phenomenon, were web servers and client browsers. They facilitated the process of sharing Hyper-Text Transfer Protocol resources over TCP/IP connections.

Cellular systems have also evolved to using packet-switching technologies instead of the usual circuit-switching [13]. Third generation networks, as they have come to be known, have

brought along new opportunities of enlarging and growing the Internet. With them has come a totally new way of thinking. The fact that mobile phones are starting to have the storage and processing capabilities of earlier computers means there's no reason why web servers shouldn't reside on them. The purpose of this paper is to study the internet and web technologies that are related to mobile web servers and gateways. This will help in setting up and building a mobile web server gateway on the local campus.

2.2 Mobile Internet Protocol

Mobile Internet Protocol (IP) allows the use of the Internet on the move. A user with a mobile device whose IP address is associated with one network can stay connected when moving to a network with a different address. Expressed differently, a user can keep their IP address while moving between networks with different addresses. When a user leaves the network with which their mobile device is associated and enters the domain of the foreign network, the Mobile IP protocol is used to handle the connectivity issues [1]. As is stated in the book *Telecommunications Essentials* [3], the foreign network sends a message to the home network address, notifying it of a care-of address as discussed in [15]. The care-of-address is the foreign address which a mobile device may have access to if not in the home network. This address is where all the user's packets are sent [3]. Mobile IP is also the technology used for wireless data applications and for mobile networks like 3G and 2.5G for cellular systems. It is implemented in the packet equipment for packet-switched cellular networks. It works at layer 3 of the OSI stack as shown in the Figure 1.

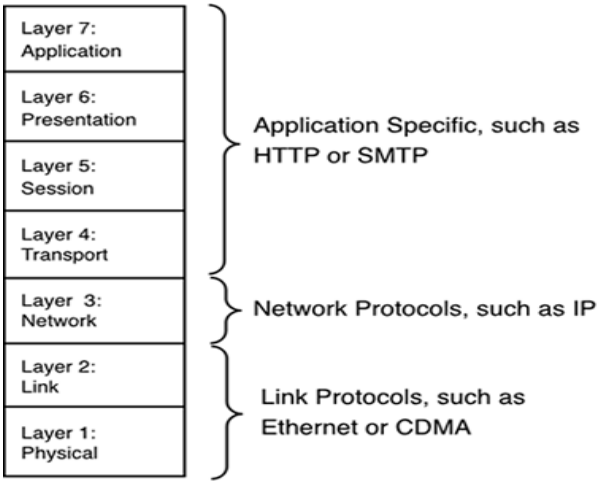

Figure 1: The OSI Stack [1]

According to *Communications Essentials* [3], layer one and layer two protocols are already implemented in mobile devices.

Nokia phones will be used in this project and the MTN and Vodacom networks use Code Division Multiplexing Algorithms for sending signals to mobile users. [13] explains that mobile operators are increasingly starting to use Wideband Code Division Multiple Access (WCDMA for 3G), a faster and newer variant of CDMA. Going one level up, Mobile IP is implemented in the networking equipment of mobile operators. They use this for normal Internet access by mobile users. The one problem there is according to [9], however, is the fact that the mobile operators have restrictions at this particular layer.

[9] further explains that for security reasons, the mobile operator firewalls allow only outbound traffic. Only mobile devices from within the network can initiate requests to the internet and not the other way round. According to [7] to be able to access a web server on a mobile phone this has to be overcome. Another problem stated by [9] at this layer is that mobile operators have introduced Network address translation (NAT). This is a networking technique that involves the circulation of IP addresses between hosts on a network. It is a

favourable option for both the operators and the whole computing society at large because there are too many mobile phones out there. Giving each of them a 32 bit IP address will eventually result in a shortage of these currently-running-out identifiers. It is also not economical as some users hardly access the Internet from their phones. Even those who do it do not do it consistently. [9] elaborates on the fact that for the solution to this problem to succeed it will be cheaper and more practical if it circumvents the current restrictions with minimal changes to the infrastructure already in place.

If a system that is both addressable and accessible is put on the Internet, it can solve all the problems posed by the operators [9]. On the system will be a reverse proxy that receives HTTP requests on behalf of mobile servers and forwards them to requested mobile phones. Before this could happen the mobile server on the phone will need to establish and keep-alive a connection to the proxy. [10] emphasises the point that mobile devices will need to be registered on the proxy or gateway for their access information to be stored and updated. When a request enters the reverse-proxy, its header is checked and the access details are used to determine the mobile server requested. Prior to this, a mobile phone would need to have made a connection to the gateway. [9] says that this should eliminate the need to know about the IP address of a phone because the phone is the initiator of the connection. This connection is also kept alive until it is voluntarily terminated by the owner of the mobile web server.

2.3 The Wireless Application Protocol (WAP) and the WAP Gateway

In June of 1997, major players in the mobile phone industry gathered to form the Wireless Applications Protocol (WAP) forum as stated by [5]. Cellular phone manufacturers, Nokia, Ericsson and Motorola were part of this forum. Another organization present was Phone.com (Wired Planet), the WAP creators. The WAP Forum is an industry group responsible for managing and extending the WAP standard and facilitating the adoption of WAP. WAP is a standard used for the transmission and subsequent presentation of wireless data to mobile devices. [5] further emphasises that WAP is based on HTTP and is easily interoperable with the Internet. The Wireless Mark-up Language (WML), a tag language based on XML, is used

for the presentation. WAP is mainly designed to integrate the Internet with a lightweight, low bandwidth system that is suitable for wireless devices. [4] says that for this interoperability to be achieved, A WAP gateway should be used. The Wireless Applications Protocol architecture (with the gateway) is shown in the Figure 2 .

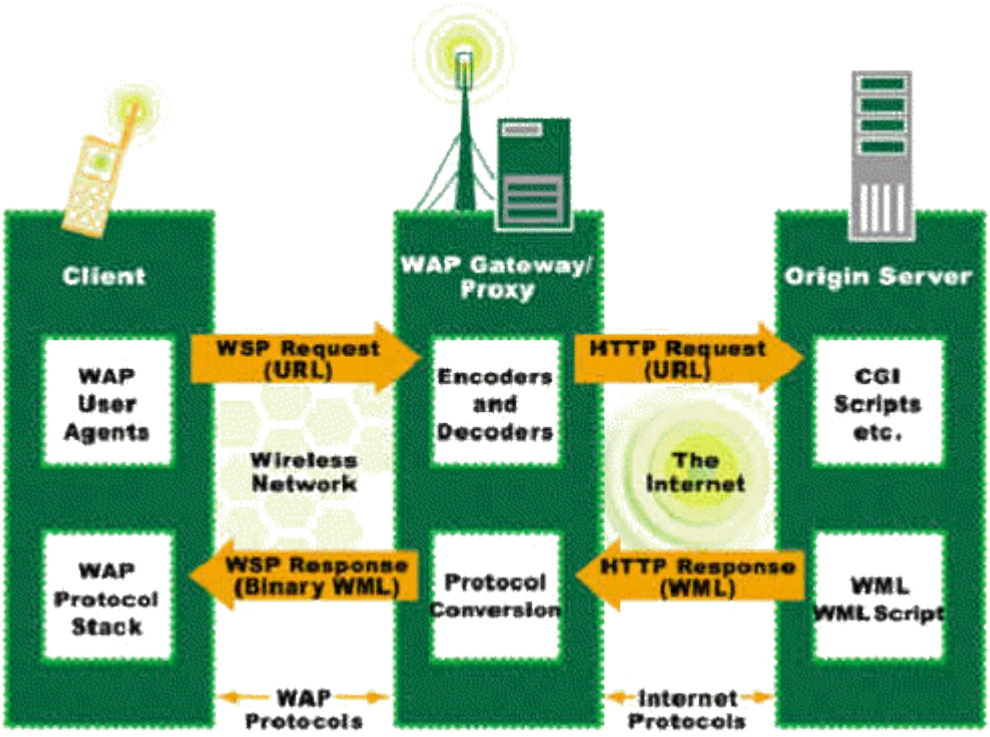

Figure 2: The WAP Gateway [4]

The WAP Gateway and Other WAP Components.

The WAP gateway, like the mobile Web server gateway, sits between the devices on the internet and those on a wireless network. It is responsible for encoding, decoding and for protocol conversion of the requests and responses that pass through it. A user starts the HTTP flow by making requests for web pages. By typing a URL into the browser on a mobile phone, the WAP user agent sends a WAP request (WSP) to the gateway. The gateway converts it to an HTTP request and does the necessary encoding and decoding operations. The origin server is the web server on the Internet containing the requested resources as

discussed in [4]. It services the request using scripts or by simply returning a static page. The server needs to have WML scripts to service mobile requests dynamically. Once serviced the response is sent out in HTTP format back to the gateway. The gateway then converts the response to WSP format and sends it back to the requesting browser [4, 5].

The WAP protocol stack takes care of the transmission of requests from the phone to the gateway and back to the phone again in binary format [4, 5]. According to [4, 6] the stack consists of three distinct layers: The Wireless Datagram Protocol (WTP), The Wireless Transaction Protocol (WTP) and the Wireless Session Protocol (WSP). Figure 3 shows the sequence of events involved in servicing a request.

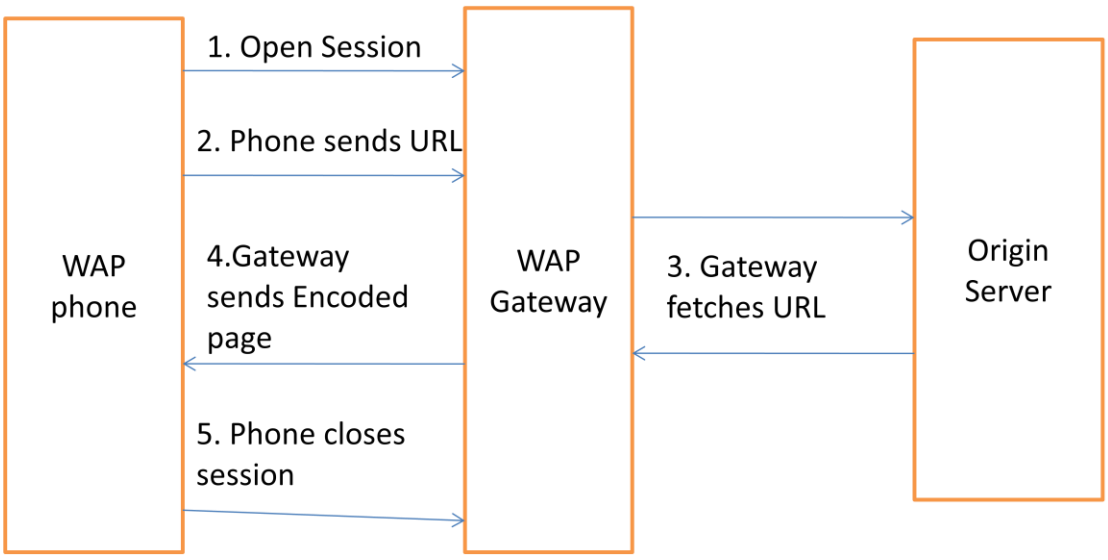

Figure 3: The Wap Gateway [6]

[6] explains the steps as follows:

1. Phone opens session. The features and HTTP headers to be used in requests gateway makes on behalf of the phone are negotiated. - WSP

2. Phone sends URL for the page the user has configured as his home page.

3. Gateway makes HTTP request, with negotiated headers.

4. Gateway encodes page in a binary form and sends it to the phone.

5. User shuts down the browser and the phone terminates the session.

The Wireless Transaction Protocol implements a single request-response pair. The Wireless Datagram Protocol is the lowest of the three layers. It implements the actual moving of packets from the phone to the gateway and back [6].

This means that the gateway being set up as part of this project needs a module to handle connections with another module residing on the phone as suggested in [9]. The two modules need to negotiate connections, keep them alive and make sure the two systems interface. The WAP gateway also handles user authentication and has some management functions such as billing customers as put by [6]. The Gateway to be implemented will also take care of mobile access data and will have information on the number of times a server has been visited, enable the sending of messages when the mobile web server is offline and possibly also have server location information. The next section will discuss the Kannel WAP Gateway which could be used as a guide to building the local gateway.

2.4 The Kannel Gateway

The Kannel WAP gateway is a product developed at Wapit Ltd., a company started in 1998 to develop products for mobile phones. A year later, as part of its strategy, the company started to develop tools and software for the then emerging WAP protocol. The project was launched as part of the WAP Forum in July 1999 [5, 4, and 6]. Kannel is widely used as a WAP gateway by mobile operators and corporate service providers. The author of this dissertation thought they would gain insight from studying this gateway. Next, the paper will

discuss the architecture of the gateway with the intention of gaining some insight on how to put the local mobile web server gateway together.

2.4.1 The Gateway Architecture

The Kannel WAP gateway is very similar to the desired mobile web server gateway. Studying its architecture and design gives some insight as to how to structure the latter. The overall structure of the Kannel gateway is depicted in the Figure 4 below which is taken from [6]:

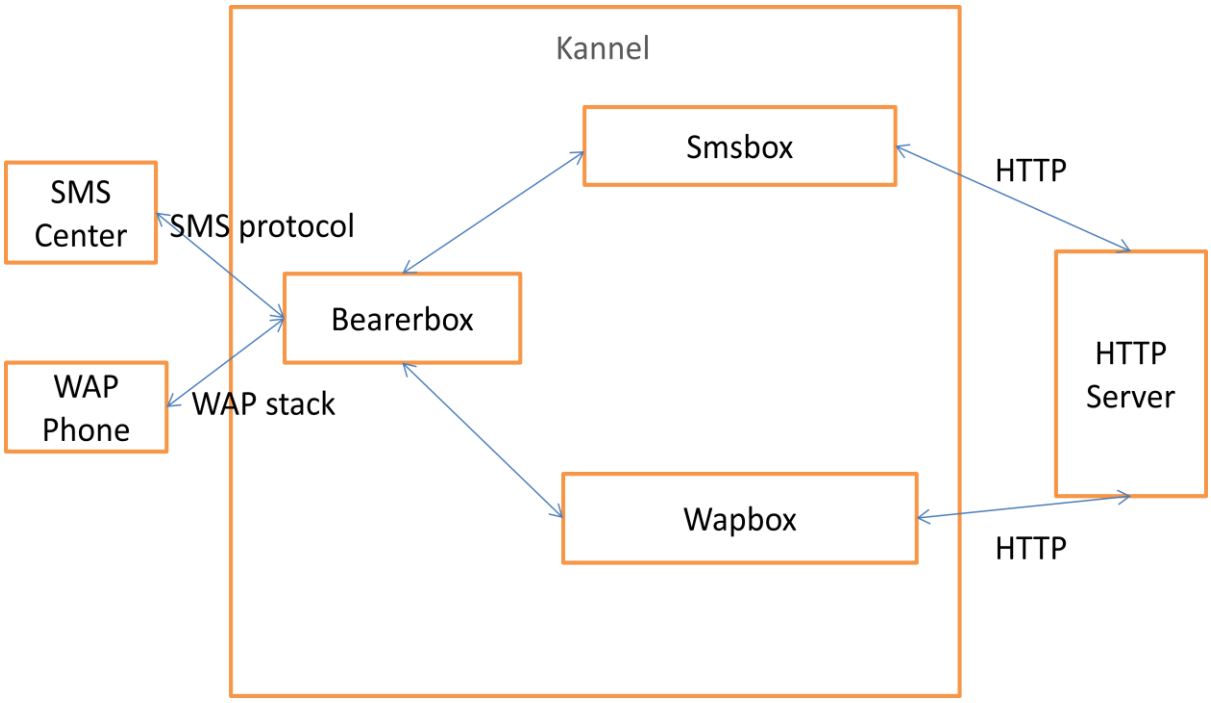

Figure 4: The Kannel Gateway Architecture

The diagram (figure 4), from [6], illustrates how the gateway has three interfaces on which interacting systems can communicate. This suggests that the gateway host has three ports on which to listen for requests. Since the Mobile Web Server Gateway will listen to two types of services, it will most likely have two ports open: One for HTTP traffic from web browsers and the other to communicate with the phone. The text in [6] goes on to clarify that the SMS Centre is a server responsible for SMS-related services which are not really part of the WAP

standard. The WAP phone is a mobile phone with WAP capabilities and can send and receive WAP messages. The HTTP server is a web server residing on the Internet. Inside the gateway itself are three modules: the Bearerbox, the Smsbox and the Wapbox. The Bearerbox component handles the incoming and outgoing low level Wireless Datagram Protocol (WDP) packets. The SMSbox receives SMS messages form the Bearerbox and translates them to service requests. It is also responsible for doing the reverse to the responses. The WAPbox module implements the WAP protocol stack and WAP Push services. It works with an application level protocol [6]. For the purposes of the project, the SMS components of the Kannel gateway are going to be ignored. The focus is on the WAP functionality that will enlighten us on how to build a gateway between the Internet and a wireless network. For this reason, the paper will discuss the Wapbox component next.

2.4.2 The Wapbox Module

The gateway documentation [6] deepens the insight by explains that, the Wapbox and all the other gateway boxes are internally multithreaded to allow for efficient request and response handling. It's responsible for fetching messages from the Bearerbox, maintaining state for each of the active clients and subsequently making HTTP requests for them. It is also responsible for the reverse process of sending responses back to the Bearerbox. According to the Kannel documentation [6], things get more complicated depending on the load being dealt with. The protocols implemented in the box are Wireless Transactions Protocol and the Wireless Session Protocol. To gain more insight on gateway building we shall concentrate on the Pull threading features of the WAPbox component.

2.4.3 The Thread Structure

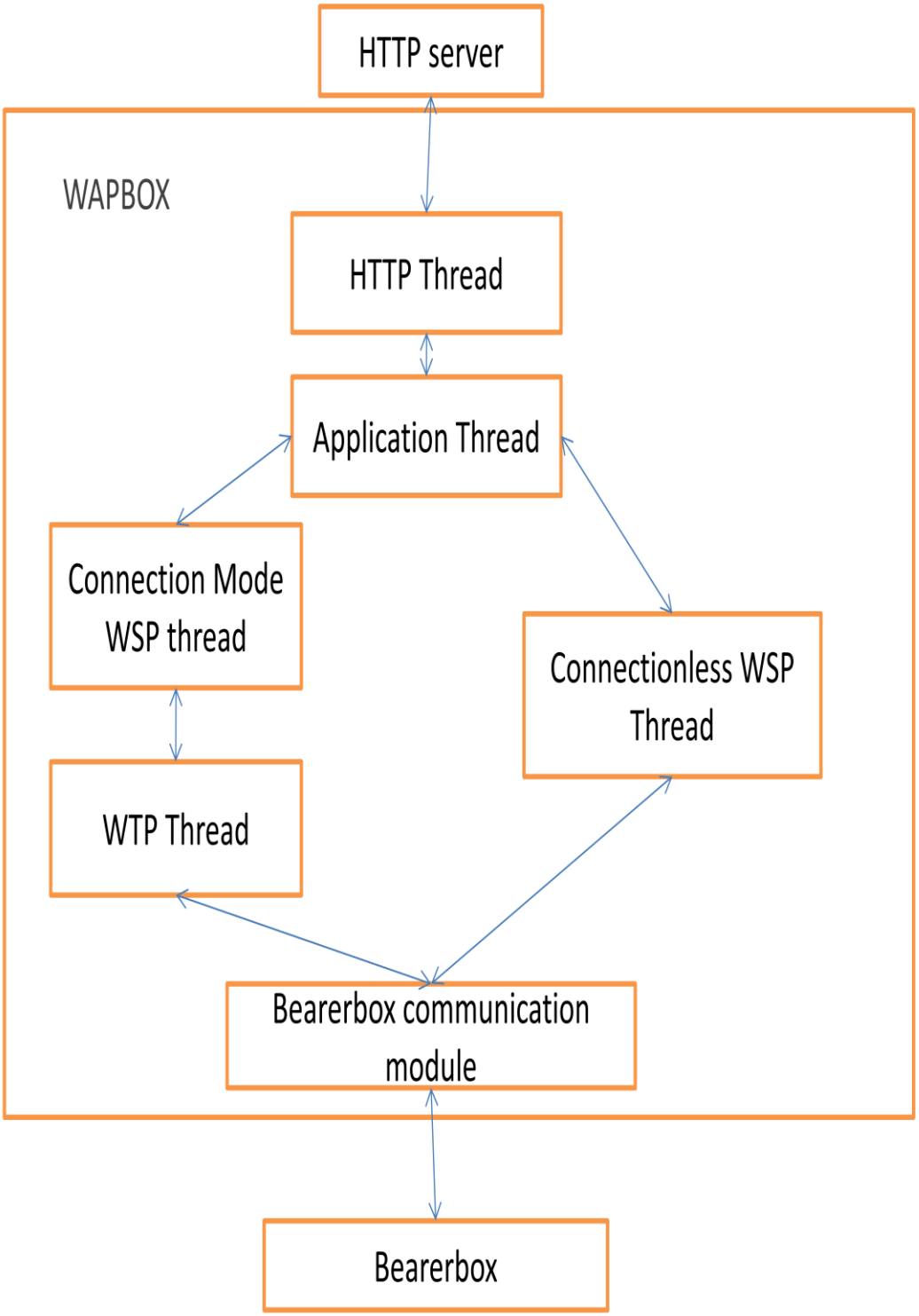

Figure 5: The Wapbox Architechture [6]

According to the Kannel documentation [6] and to the Figure 5, each WAP protocol stack layer has its own thread. This means that for every WAP and HTTP request serviced by the gateway a thread is spawned. This fact has some implications for the design of our gateway. There has to be a sub-module that services HTTP requests (as shown in the diagram). It also has to be threaded. There are many things to be considered at this interface. A request coming in from an HTTP browser will be serviced by a servelet or any other script. Its header contents can be used to determine which mobile web server needs to be connected to. Web servers already have threading capabilities so there woul be guaranteed threading for HTTP requests on the one port of the gateway.

Looking at the other end, the interface to the Bearerbox is also multi-threaded. This is analogous to the Local Mobile Web Server (MWS) gateway's interface to mobile web servers. In terms of Implementation this will be a port known to the mobile web servers, as is suggested in [9]. It is also threaded to service multiple requests efficiently. This will be implemented in the gateway by using java threading libraries. Each request will have its own thread to service it. The WAP gateway is a little more complicated. Apart from the interfaces, the required inner-workings of the MWS gateway are about accessing data from a database and maintaining connection information as described in [9]. There will be no protocol conversion at all. The main lesson learned form this gateway is that the gateways need to be multithreaded to handle the load of services being requested.

Although threading provides high request-handling efficiency, it comes with its costs. According to WapIT's documentation of Kannel [6], threads keep the implementation simple, but are expensive in terms of computation resources. According to [6] "*If there are ten thousand concurrent users each making a new request every fifteen seconds, on average, and each request taking one second, on average, there are about 670 concurrent requests at any one time. On Linux, each thread uses 8 kilobytes of kernel memory, minimum, so 670 threads would use over 5 megabytes of extra memory*". Additionally [6] explains that, starting and stopping threads and having lots of threads will cause more context switches, an additional CPU cost. As we are using a web server as the HTTP interface we might not need to worry about this too much. The Apache Tomcat Web server will be used for implementing the mobile web server gateway. Its web container already deals with request threading and will

provide access to HTTP requests as is stated in [8]. It will also be the interface through which responses will be sent.

2. 5 The Nokia Mobile Web Server

2.5.1 Details

The Nokia Mobile Web Servers (MWS) are the mobile web servers that will be connected via the gateway. Nokia Corporation introduced these web servers that are written in Symbian C++ for mobile phones running the Symbian operating system as stated in [7]. The Mobile Web Server Book [7] describes the invention as something that will change the way people view the Internet. We have come to know it as a network where the servers are stationery machines in mysterious locations. The mobile web server, however, makes it possible for users to generate their own information and serve it on their mobile phones. This opens up opportunities for users and developers to do more creative things. It is in line with the new vision of the web where content is generated by the users themselves as described by [12]. This includes dynamic-content creation, interactive photography and location-based services [7]. It will also facilitate new ways of communication as well as new ways of fulfilling mobile phone functions like answering messages. This section of the paper discusses the details of the Mobile Web Server Technology that are relevant to the implementation of a gateway.

As mentioned earlier and further strengthened by [10], Mobile Web Servers lie behind a mobile operator's firewall and this makes them impossible to access from the internet. The gateway will be the means of communication between the web servers and the requesting clients. This is shown in the Figure 6. The diagram also shows the main lines of communication between the gateway and the web server. In step one the owner of the MWS connects to the gateway. According to [11], this is, of course, after it has been registered. The gateway and the phone keep this connection alive. When a user wanting to connect to the web server with a web browser comes along in step two, the gateway already has a connection to the phone. [7] says that the Domain Name System (DNS) serving the browser

host returns the address of the gateway for all the MWS requests. In step three the server identifies the web server and checks if it's online and sends requests for a channel to communicate HTTP traffic. Finally, in step four the HTTP traffic is communicated to the web server and the response is relayed through the gateway. The operator firewall is not an issue of concern because the requesting connection is initialized from the mobile device through a legal port [9].

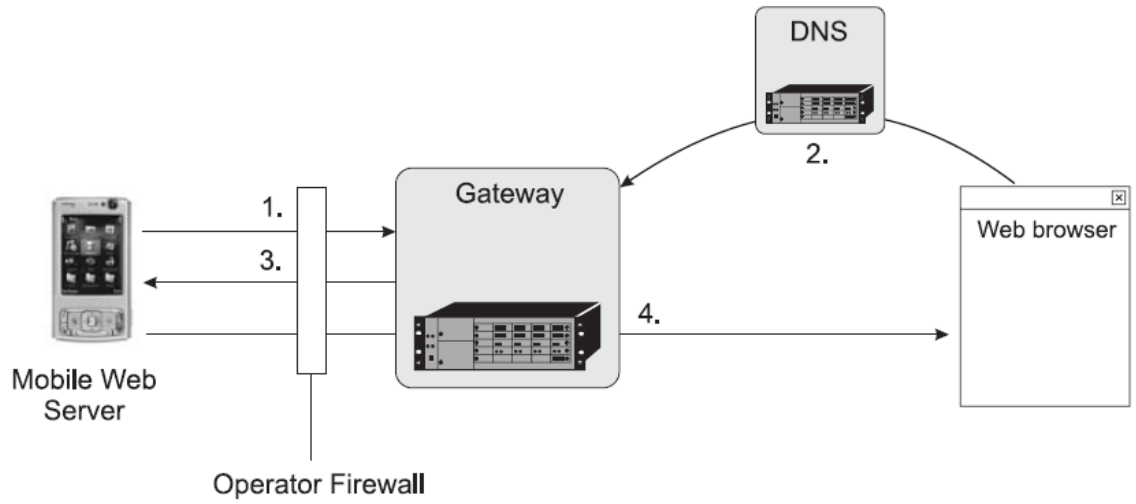

Figure 6:The mobile web server technology [7]

2.5.2 The MWS Components

The MWS is a ported version of the Apache Web Server. This port may be the Nokia Mobile Web Server or PAMP which is a package containing the Apache Web Server, PHP and a MySQL database. It also has a connector and user interfaces to facilitate the use of the web server. The apache web server was chosen because it is the most popular web server in the world and it is free. Implementation-wise, it also turned out to be relatively easier to port as stated in [7]. The modular structure of The Apache Web Server makes it easier to port to mobile devices as modules can be included and excluded if needed. This, as common sense would dictate, decreases the memory footprint. On mobile devices, both space and processing

power are limited so the smaller an application the better. In addition the memory used decreases with the number of modules loaded.

As shown in the Figure 7 and explained in [7], The Apache HTTP daemon (HTTPd) runs on top of the Apache Runtime (APR) [7, 8]. This is a runtime library that provides a standard API to the underlying platform-dependent implementations. According to Nokia, the only task required was to port the platform-specific parts of the HTTPd and APR. This were translated to work with the underlying Symbian operating system (the operating system on Nokia phones). To make the job slightly less difficult, some Unix-specific parts could use the Symbian POSIX. The Figure seven shows a graphic version of the ported version.

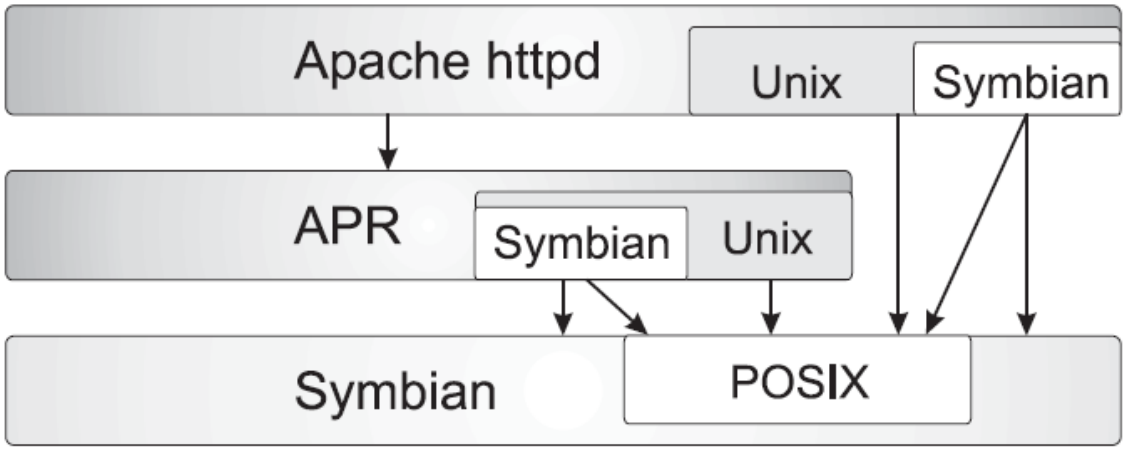

Figure 7: The ported Apache Server [7]

The most important component for the purposes of this paper is the connector. This is the subsystem that is going to communicate with the gateway. It has an interface to the web server described above and to the remote gateway. The connector takes care of all the connectivity issues as far as the mobile device is concerned [7]. It establishes and keeps alive a connection with the gateway. The channel for establishing and keeping the connection alive is known as the control channel, and the one for HTTP traffic is known as the data channel [11]. The connector establishes a connection to the gateway by using the required connection details: the username, the password and the gateway address. Other things to set include the

keep-alive interval, the keep-alive latency, the gateway port the maximum number of connections that can be made to the server and the local web server port. The control channel is over UDP datagrams. This is favourable because of the overhead. It therefore uses less bandwidth and battery power [11]. This channel carries a custom-made protocol that is meant to facilitate the formation and maintenance of a connection between the web server and the serving gateway. The Data channel goes over TCP and it is the one the carries the actual HTTP data when a request or a response is in transit [10]. The gateway (to be implemented in this project) needs to interconnect with the web server through this connector. Figure 8 shows the messages and protocols between the mobile connector and the gateway.

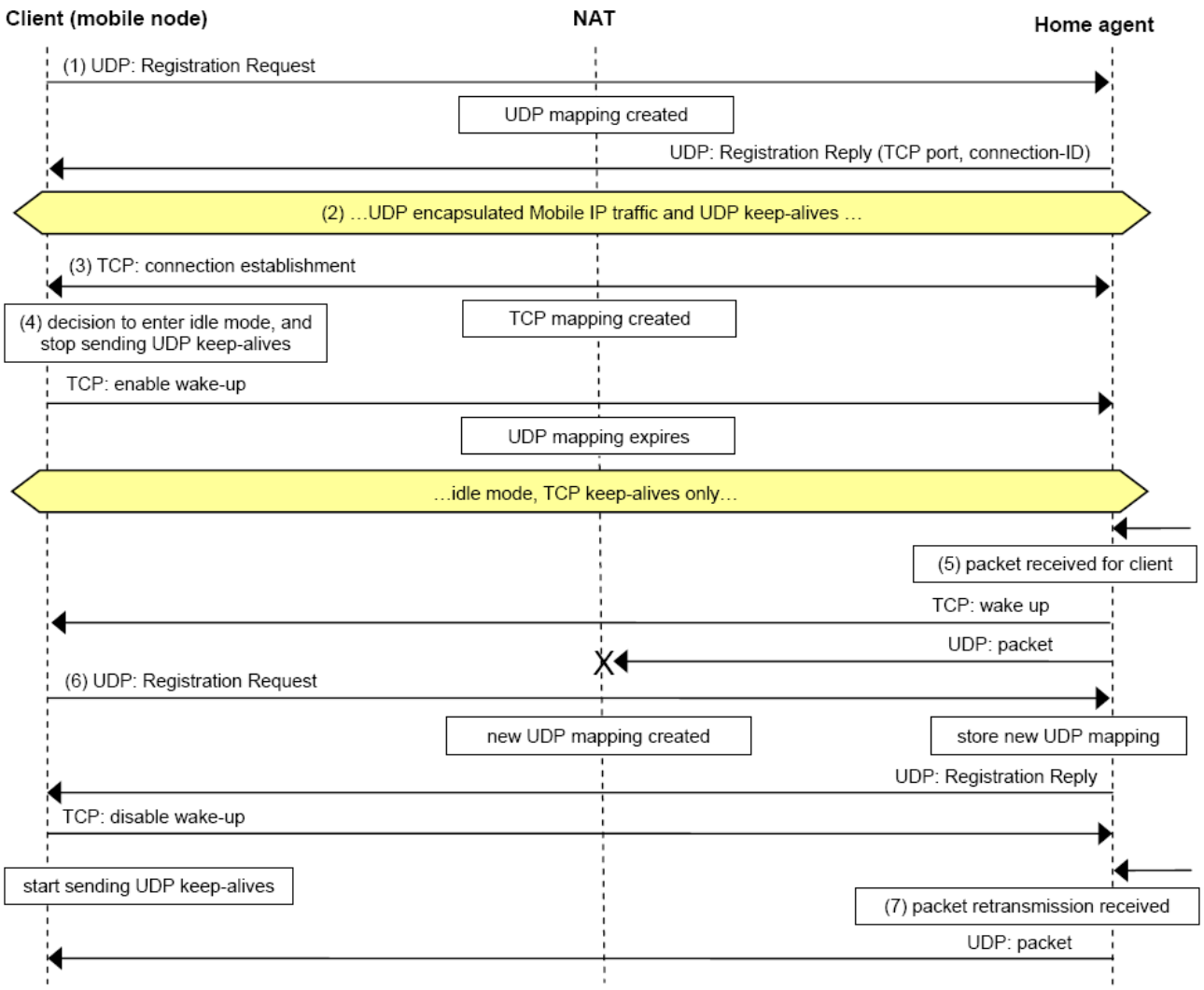

Figure 8: The communication streams between the gateway and the client (phone) connector [11]

2.6 Conventional Web Server Technology

2. 6.1 Introduction

A web server is a server process running at a website which sends out web pages responses on a particular port to web requests from local or remote web browsers as described in [14]. A web server resides on a host computer which can be stationery or mobile [41]. A web host is a computer that runs a web server and provides web space and bandwidth to those who wish to publish web sites. The website owner is given space on a host machine to which they can upload static and dynamic HTML pages as well as text and multimedia. In this section, the paper briefly looks at conventional web servers. These web servers are relatively easy to access as they are not restricted by operator firewalls or Network address Translation (NAT).

2. 6.2 Conventional Web Servers on Stationery or Wired Hosts

On a stationery host that has a wired internet connection, web servers are easily accessible and addressable as the host has a fixed IP address [14]. Even with network address translation, the host with the web server is still accessible on the network through its domain name. The web server daemon runs in the background waiting for requests for web pages [14]. Because of the complicated nature of the web servers, they are usually installed and administered by people with technical skills and knowledge of operating them . This means that website owners may have to upload content to the web space on their host using File Transfer Protocol if they happen to be in remote locations.

Web access on these type of hosts usually involves a user typing in a uniform resource locator (URL) in a browser to get Hyper Text Transfer Protocol (HTTP) responses or other resources form the server. The domain name system then translates the domain name to an IP address which then results in the HTTP request getting routed to the computer hosting the web site with the requested resources as stated in [14]. Because the web server and the

associated web resources are usually on a host with a predetermined IP address and geographical location, it is relatively easy to locate and address web sites hosted on it and to request web resources. The web server receives the request on a particular port (usually port 80 or 8080) and serves it by responding with static or dynamic HTML pages as well as other content [14]. According to [14] the server can respond in different ways: by using static HTML which is usually just plain HTML with no server side scripts executed; or by using dynamic responses which involves the execution of PHP, JSP, ASP or some other scripts that could include accessing databases and other resources.

The administration and hosting of web sites on stationery or wired hosts is advantageous because it allows for easy and fast access. It is also good because the hosts are usually always close to a power source increasing the chances of the server always being on (reliability). However, with the emphasis of web 2 technologies of personal or user content creation as described in [12], it is useful for web servers to be close to the content creators. This can also allow for on- the-move or dynamic content creation. It will also lead to the web becoming ubiquitous as pages are served by the users or the web site owners themselves [7]. The objective of the project is to build a gateway that will ensure the accessibility and addressability of such web servers. This web servers are easy to use and configure for normal people and will forward the agenda of Web 2.0 technologies.

According to the Wireless Internet Handbook [13] wireless, systems started back in the 1970s. These systems went through different generations based on different access technologies. The first wireless systems were analogue, circuit-switched networks that were used for voice transmissions only. They were based on the Frequency Division Multiplexing (FDMA) air interface. Second Generation wireless systems were digital and used different and more efficient multiplexing techniques (Time Division Multiplexing (TDMA) and Code Division Multiplexing (CDMA)). 2G systems were followed by 2.5G Systems [13]. This involved a combination of both packet- and circuit-switched technologies. Circuit switching was used for voice and packet switching for data services such as multimedia messages (MMS) and Wireless Application Protocol (WAP) Internet access [13]. The third generation of mobile systems, however, is entirely packet switched [16]. This means that the traditional

Transport Control Protocol and Internet Protocol (TCP/IP) can now be used for mobile phones for internet access and also for service provision.

The introduction of 2.5G saw the introduction of Internet web browsing on Mobile phones. The circuit switched nature of the data services for 2.5G systems was made to cater for this. Network connectivity for mobile devices like mobile phones is provided by an operator. The mobile operators had to introduce Wireless Access Protocol Gateways to enable this kind of web browsing as is discussed in [5]. An Internet request was made from inside the operators firewall to a website hosted on a stationery or mobile host. It would then passes through a WAP gateway to negotiate the differences between the protocol used for WAP browsing and the protocol used for the normal internet. Apart from the client or the web browser being located on the phone, the architecture of this kind of system was the same as that of ordinary wired internet browsing. Third Generation wireless Systems, however, opened new doors. Web servers can now be hosted and accessed from mobile phones.

2.7 Chapter Summary

In conclusion, the author has discussed the technologies that are related to the project involving the implementation of a mobile web server gateway. The gateway facilitates the connections between mobile web servers and client-web browsers. The WAP gateway technology has a number of similarities with the desired MWS gateway and has been discussed in this paper. The Mobile Web Server which is the endpoint to the gateway connections is also discussed. Last but not least, conventional web servers are also discussed because the gateway itself will use the Tomcat Web Server to fulfil its functions. These related technologies will give some insight on the issues involved in putting a mobile web server gateway together.

Chapter 3

Understanding the Gateway

3.1 Overcoming Addressability and Accessibility Problems

[9] states that two important solutions have been suggested to solve the problem posed by operator's firewalls and NAT. One of the solutions involves the use of a firewall control protocol (FCP), which works by allowing a trusted third party to dynamically control the operators firewall. The third party decides on what firewall ports to open at what time and how to use them. These ports will be open for a short period of time and should also have a way of choosing the users allowed to use the specified ports. The problem with this approach is that a great deal of investment by operators will be needed to get it off the ground. Making firewalls dynamically configurable might also necessitate new web browsers that work with them. It will also need a way of keeping track of the temporary Internet protocol addresses as they change over time. Additionally operators need to route requests from other devices from within the firewall, a functionality, according to [9], most of them do not offer. A better solution is one that does not need much of the operator's involvement and needs no change to the existing infrastructure.

The most viable solution involves setting up a gateway somewhere on the internet. This gateway will allow for users to register their web servers and allows surfers to address and access these servers. Having this gateway will ensure that the mobile phones or other devices inside the operators firewall always have a connection to a device on the internet. The HTTP server on the mobile phone initiates the connection from within the firewall to the gateway when it is started. This connection is then kept open for as long as the mobile server administrator wishes. At that time the HTTP daemon can serve requests. The gateway computer with a gateway server installed, acts as a device in the middle that is connected to the running servers. The gateway runs a daemon of its own waiting for connections from web servers. As soon as a connection request comes in, the gateway program checks if the requested web server is running as explained in [7]. If it is, the HTTP request is sent to the

right Mobile Web Server (MWS) through the connection established by the server (MWS) when it started running. Since it's the mobile server that initiates the connection to the operator's firewall this is normal (because it's an outbound connection as explained in [9]). The web server on the mobile phone then gets the request from the gateway and responds to it accordingly. After the server has put together its response, its response is relayed to the originating web browser through the gateway. The HTTP requests and responses pass through the operator's firewall as if they were legitimate third Generation services (3G) requests and responses (which they are). This means they could go through a special port, if the operator reserves one, or through the normal Third Generation services port [7] used for Internet access for example. This solution works harmoniously with the existing infrastructure and needs little operator involvement, if any at all.

3.2 How the Gateway Will Work

The gateway host has the gateway server installed on it. The gateway server opens up two ports on the host. One port is for incoming connections from requesting web browsers; the other is for the connecting to web servers. The web surfing port should be a well known port (e.g. port 80) for HTTP requests to be mapped to it. The host should also be registered on the Domain Name Server to allow all the requests with URLs ending with a particular name to be directed to it, resulting in its IP address being returned. The second port should be a special port on which mobile devices establish connections to the gateway. The web server (MWS) on the mobile phone listens on port 80 like a normal web server. Even thought, the server waits for HTTP requests on this port it cannot be addressed or accessed by any other device without going through the gateway [7]. The connections between the server and the gateway are established and managed by software programs know as connectors. Both the gateway and the mobile web server will have their own connectors through which connections will be managed and kept alive [8].

The connector on the gateway host opens up a port to receive and maintain connections with the mobile web server. When the connector on the mobile phones opens a connection it gives the identity of the hosting phone and the gateway connector authenticates the request before negotiating the connection. This is one of the reasons why the mobile web server has to be

registered on the gateway before connections from it can be accepted. As in most other communication protocols, there are two types of interchange between the connector on the gateway and the peer connector on the mobile phone: One for controlling and managing the connection and the other for the actual transmission of traffic, in this case, HTTP data as mentioned in [10]. These are respectively known as the control channel and the data channel. Initially, when the mobile connector opens a connection to the port reserved for connections on the gateway, it establishes a Transport Control Protocol (TCP) connection [10]. This happens in the control channel because it just opens a line that can be used to relay HTTP requests later on. When the need arises, the gateway connector asks the mobile connector to initiate data channels in addition to the control channel that keeps the connection alive when there are no other activities. Conceptually, the two channels then co-exist as two separate TCP connections on the same port. The problems associated with Network Address Translation do not matter any more as the gateway connector maintains a connection with the mobile connector. This means that whenever the IP address of the mobile host changes the connector sends control data through the firewall using the new address. The gateway connector, therefore, just needs to access the control open control channel and associate any subsequent requests with it [8].

3.3 Keeping the Connection Alive

Once a connection is established by the mobile host, it has to be kept alive to ensure that it is accessible from the other side of the firewall. The mobile and the gateway connector, therefore have to exchange control data for as long as the web server is running. If this connection dies, only the mobile host can initiate it again because the gateway does not have the authority to re-establish it through the firewall as mentioned earlier. During this period (when the connection is idle) only the control channel is kept alive. In implementing this technique, one needs to consider seriously the most efficient way to keep the connection alive. This includes sending as little data as possible (to keep the network and bandwidth costs low) and making sure the process does not overuse the mobile host's power resources (batteries) as discussed in [11]. For this reason, the two connectors negotiate a certain time period that should elapse between two keep-alive messages. If the period expires before a

signal is received from the other side, the connectors just assume that the other host has given up the connection. Otherwise, the connection is kept alive and the mobile web server may be accessed from the internet. The gateway connector is assigned the task of sending regular keep-alive messages to the mobile connector over a "stale connection" (when there's no other HTTP traffic to be exchanged). In the author's opinion it would have been much more reasonable to give this responsibility to the mobile connector because it would then readily reflect the state of the connection. However, research has shown that this would have resulted in a more complex state machine for both end points as indicated by the literature in [11]. After the initial handshake, the gateway connector is the only one that sends keep-alive messages over the control channel. If a request from a web browser comes in from the "popular port," a data channel is requested. If the mobile connector fails to respond to the keep-alive message within the negotiated time period, the gateway connector assumes the host has stopped running and it is rendered offline.

A decision the timing and regularity of the keep-alive messages have to be sent also has to be made. The gateway connector has to figure out an optimal time period after which keep-alive messages are to be sent to the peer connector [11]. For this reason, the keep-alive messages are not empty packets; they also specify parameters such as when the next control packet will be sent. If the mobile connector does not receive this packet by the specified time period it assumes that the connection is broken and it tries to fix it. It establishes a new control channel and tries to get a "handshake" for it. The gateway discovers the favourable keep-alive period by simple trial-and-error. It sends the control messages at increasing time intervals until it gets a complaint from the mobile connector indicating that a due packet was not received. After this, the connector reverts to the last keep-alive period that was favourable. Determining the keep-alive period dynamically is important because different operators have different infrastructures and different qualities of service. The geographical location of the two negotiating entities can also play a role. This is because the connection has to go through different media (with different bandwidth), network devices such as routers and firewalls that implement different technologies. If any changes are to be made to how the keep-alive massages are sent, they affect only the gateway connector. This makes the system easier to maintain.

3.4 Mobile Web Browsing Using the Gateway

Figure 9 shows an example of a browsing session. A web surfer types in the domain name of the desired server, in this example John.doe.raccoon.net. The request is then sent to the Domain name server where every name ending with “raccoon.net” returns the IP address of the gateway host. The browser then sends the HTTP request to the returned IP address. Upon receiving the request on the “popular port”- port 80- the gateway analyses the request header to determine which server to send it to. If the server is running, and a connection has been kept alive between the two peer connectors, the gateway searches for it from the pool of connections.

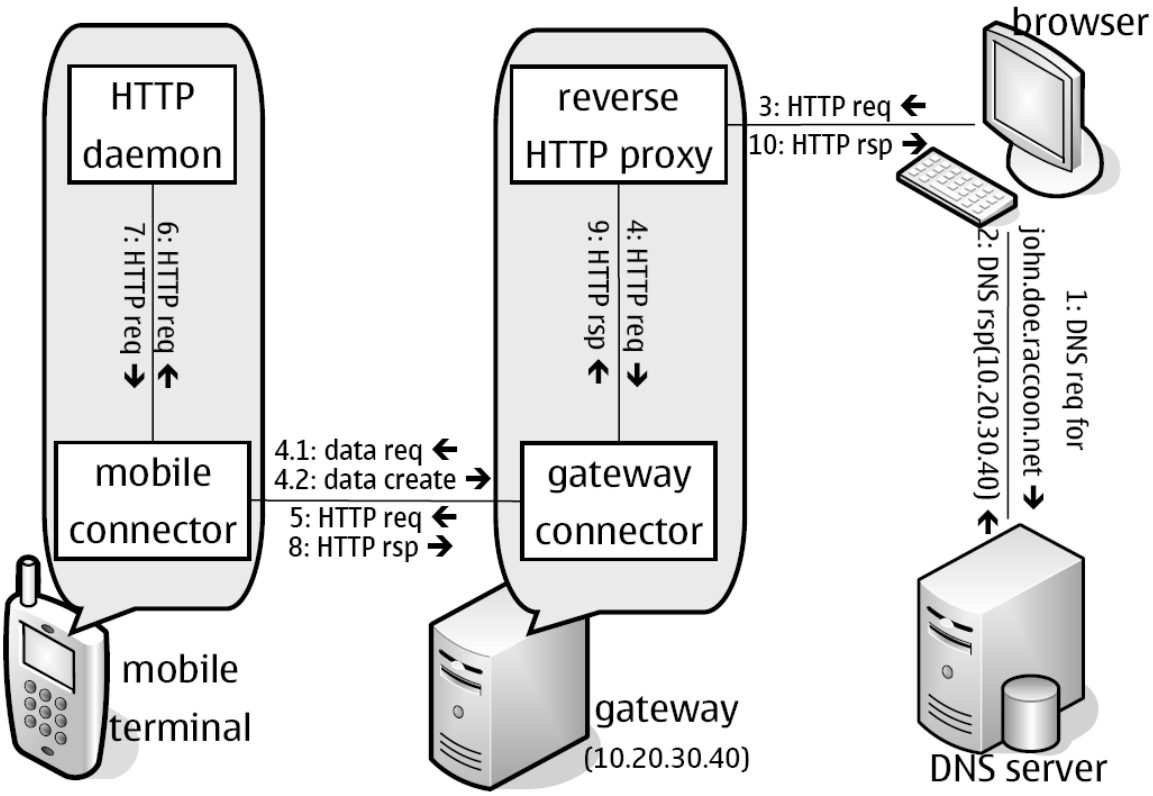

Figure 9: The Request Response Sequence(from source [10])

After analysing the identity of the mobile devices by looking at the opened control channels, the required mobile connector is identified. The mobile connector for the requested mobile host is instructed to open up a data channel for relaying the HTTP requests. Once it's up and running, the gateway connector sends the HTTP request to the mobile connector over the TCP channel. The mobile connector then passes the HTTP message to the local web server. After the server has processed the request, the response is forwarded to the connector and goes to the gateway through the data channel used for the request. The data channel is kept open for future HTTP traffic. Since data channels are not kept-alive, if the keep-alive time period elapses the channel is killed [11]. However, if the channel is still active, it is revived to keep the data flowing.

3.5 Security

The fact that every connection to the mobile web server has to be made through the gateway provides a single point of access and control. At the gateway, users can be authenticated and servers can be accessed based on a quota system. This eliminates the threat of denial of service and other kinds of attacks that conventional web hosts suffer [7]. Gateways may also block or alert users to update their servers to more recent versions if they happen to be out dated. Figure 10 shows a general idea of how security is inherently easy to implement in the system.

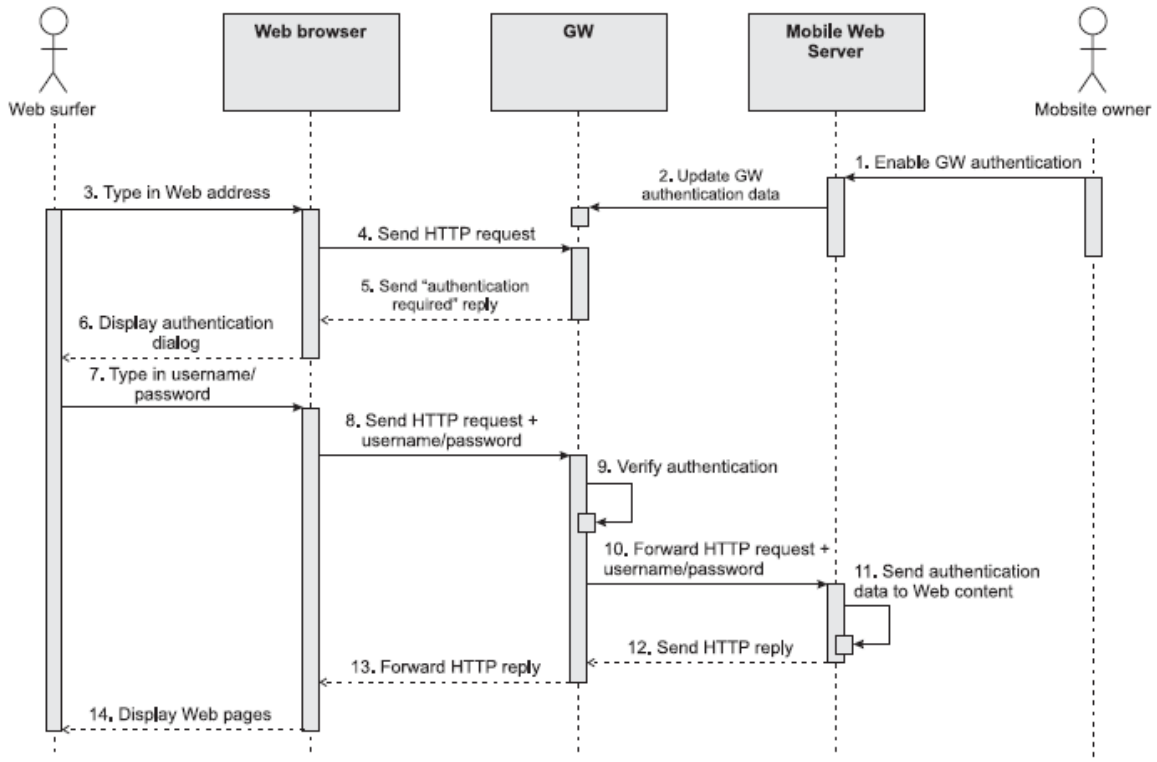

Figure 10: Gateway Security (from source [7])

3.6 The Architecture and Functionality of the gateway

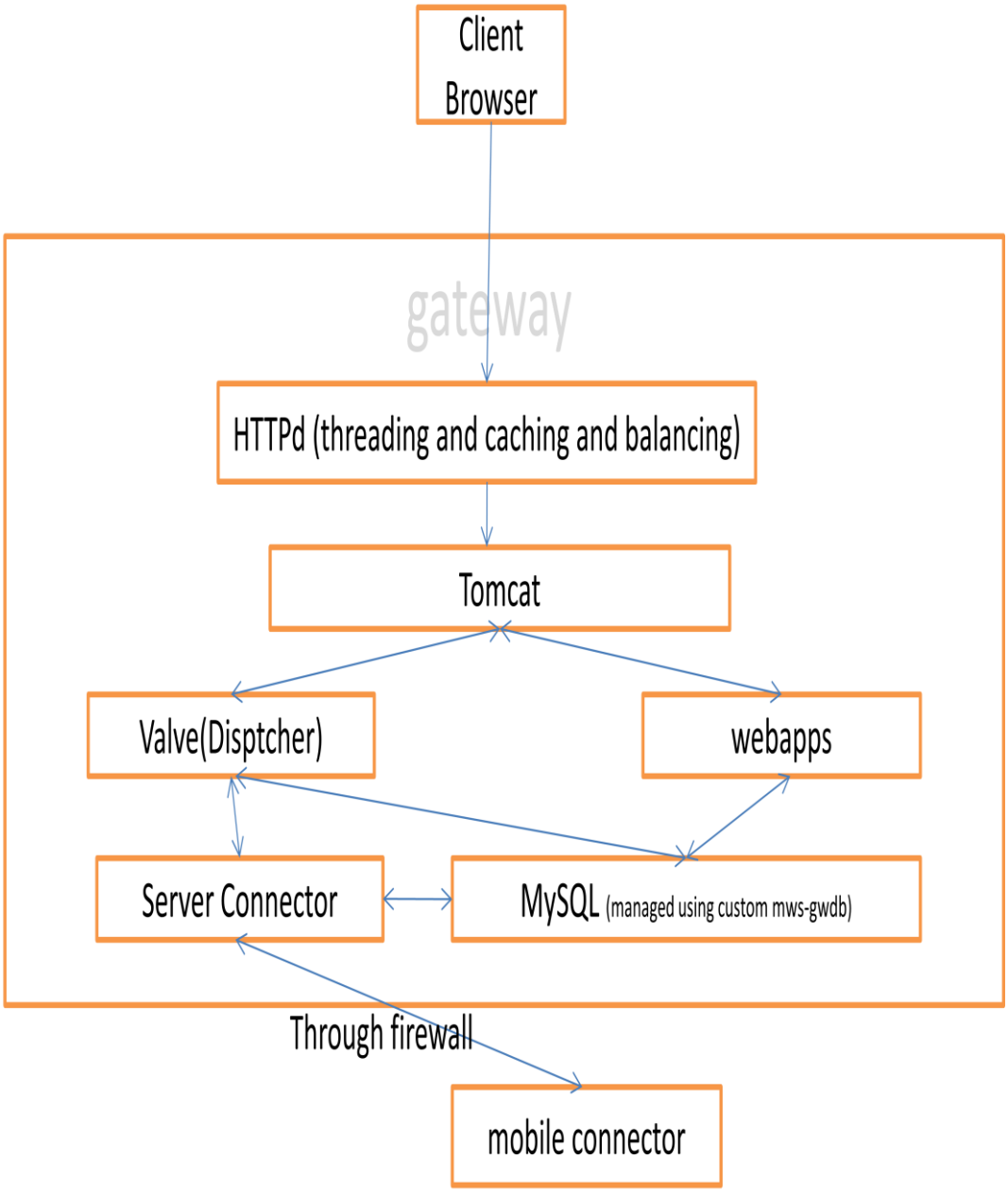

Figure 11: The Mobile web Server Gateway Architecture

The gateway was implemented using the Apache Tomcat Web Server. When the Tomcat server is started, the web server daemon starts running and waits for incoming web requests directed to the server. Upon receipt of a request (on the popular port), the server forwards it to the dispatcher or valve. As shown by the Figure 11, the dispatcher runs as part of the tomcat server. The gateway's internal interface to the Tomcat server is called the 'valve', the reason for that being that it is the entity that can capture and redirect HTTP requests. The valve or the dispatcher captures all the incoming requests as they are received on port 80 by the tomcat daemon. It (the valve) is implemented using a Java HTTPServlet that looks at the request headers to determine where to forward the requests. After this is done the valve (or dispatcher) is also responsible for checking if the connector of the requested mobile host is registered (by querying a MySQL database). If it is registered the HTTP request is forwarded to the gateway connector (described in the previous section). If the requested server does not correspond to any of the registered servers, it is forwarded to the 'webapps' and web services components. This components deal with the request accordingly by returning an error page that indicates that the host is not registered with the gateway.

The MySQL database tables are accessed using a custom persistence module namely 'gwdb.' It is basically an abbreviation for 'gateway database'. The module is responsible for ensuring database connectivity; access to the tables at runtime; enforcement of constraints for database queries and transactions; logging and exception handling. Within the modules are objects that have methods to enable persistent database transaction execution at run-time. The objects should be available to all the other gateway components that access the database. It is basically the single point of access to the database for the web applications, the valve and the connector.

The web applications, labelled the 'webapps' in the diagram, are Java web applications. They allow the administrator to manage the gateway, for users to register their accounts and manage them. Additionally, the also allow for users to see other people who are registered. Moreover, he or she could check who is online and who isn't. This also facilitates messaging amongst users. Through these web applications, the administrator can create, view and delete

user accounts. They can also play around with the settings; send emails and list online and offline web servers. As the web applications are online, administrators can carry their gateway management duties whenever they have an internet connection. Mobile users can also register their accounts on the gateway from anywhere. They fill in their user names, passwords, names, e-mail addresses, the preferred domain name and other important details. Once registered, a user can play around with account properties and can change their log in details if need be. Users also have access to other web applications such as the number of times their servers were visited, the names of the servers online and others registered with the gateway. Users can also send messages to each other. This is implemented on a store-and-forward basis so that mail is always delivered. The MySQL database stores the data used by the gateway. This includes the user account information that is queried to provide information needed for the proper functioning of the gateway [11].

3.7 The Protocols

As shown in the Figure 12, the protocol between the web client and the gateway (which is a web server in its own right) is HTTP over TCP. The requests between these two entities go through the internet (it is normal web browsing). The protocol between the two peer connectors is a proprietary protocol invented for the purpose of opening and maintaining connections. It is an XML-based protocol that is used to communicate the state of the connection on either side. For the data channel, however, the protocol is still HTTP over TCP. This is possible because Third Generation (3G) cellular systems entirely use TCP/IP and packet switching. This channel is used for sending HTTP data between the mobile web server and the gateway. It still goes through the connectors.

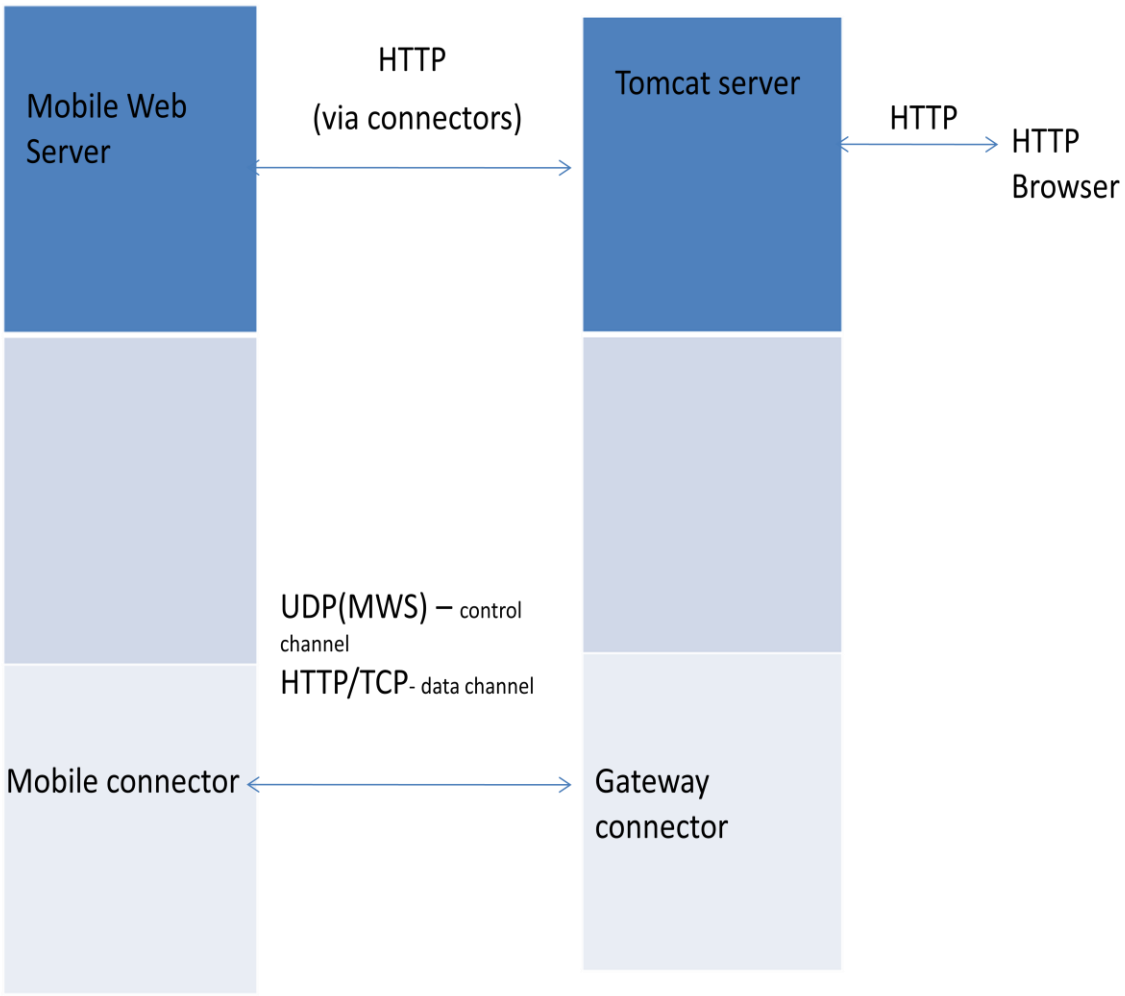

Figure 12: The Mobile Web Server Gateway Protocols

3.8 Chapter Summary

In conclusion, the author has discussed the basic functionality of the gateway, its architecture and the protocols used to communicate with web browsers and mobile connectors. The gateway has a connector component that runs as part of the Tomcat web server and it handles connection issues with the mobile connectors. Another component known as the 'valve' looks at the headers of incoming requests and decides where to forward them. The gateway's database in a MySQL database server is accessed and transacted with using a data persistence module, 'gwdb.' For the interface, web applications are used for gateway registration, account management and social networking.

Chapter 4

Building and implementing the gateway

4.1 Introduction

The gateway for the project is implemented by building on the basic open-source gateway connection management modules made available by Nokia. These modules are responsible for accepting and maintaining connections with the mobile phone connector (control channel) and provide objects that contain information about connection states. They are also responsible for opening up data channels when a request is to be serviced. Another part of the package is a database persistence module, *gwdb*, that manages run-time transactions and queries as mentioned in chapter three. Provided are also skeleton web applications that show how information can be accessed from the *gwdb* module at run time. This chapter discusses the customization and the building of the package to make a unique functional gateway.

4.2 The Structure of the Package

The gateway package is organized into eight modules namely *connector*, *db*, *gwdb*, *iapi*, *util*, *webutil*, *valve*, and *webapps*. All these modules perform tasks that collectively add up to a system that is capable of receiving and servicing mobile web server requests. The connector module handles the functions of the gateway connector as mentioned in chapter three. It is the part of the gateway that is responsible for receiving connection requests from the mobile web servers. It keeps these connections alive by using the XML protocol that is custom-made for this purpose (through the control channels). When requests for a connected mobile connector come in the gateway connector opens up a data channel as mentioned in the previous chapter.

All this is performed through the Tomcat web server and the details will be shared in the Tomcat section. The *db* and *gwdb* modules are collectively responsible for database setup and run time transaction management. The *gwdb* module is where the tables are defined and the objects to access them reside. The 'db' module is specifically responsible for creating the database and for creating the configuration file (with passwords and user accounts) that is used to make connections at run time. It is also the module that is responsible for runtime transactions. The *gwdb* module on the other hand has table specifications and methods of accessing or querying them. Setting up these two modules, however, was no walk in the park as expected. This will be explained in the next section.

Next is the *iapi* module. It is responsible for functions related to web requests such as showing the offline page when the requested mobile web server is not online. It also has interfaces that are related to. The other package is *util* which basically has classes that perform utility functions for the other packages. These include functions like opening streams, converting from xml to strings and vice versa (for the connector module) and generating random identifiers. For the *db* and *gwdb* modules, *util* has classes that assemble the requests into final SQL statements and executes them after making a connection to the database.

The *valve* module is a collection of classes that define objects for serving requests as they come in form web browsers. As mentioned in chapter three, the valve captures all the requests incoming on port 80 of the Tomcat server. The headers of the requests are then analysed by the request analyser to determine where to forward the request and the response objects. If the request URL for the request matches one of the musers' chosen web server URLs it is forwarded to the connector module. The connector checks if the requested server is online and forwards the request to it. If the requested server is not online then the server offline page is shown. The *webapps* module has web applications that present the gateway data to the visiting users. The web applications take care of account management and to view gateway connections. The web applications are discussed in later sections. The valve runs as part of the Tomcat server and this is discussed in the section 4.3 of this chapter

Compiling the Package

Each of the modules in the package are compiled and built using the *Ant* build tool. By issuing the command 'ant' in the main directory of the module the files are compiled and a jar file is produced as a product of the compilation when the 'ant jar' command was issued in the main directory of the modules. The *util, db, aipi, connector,* and *valve*, modules were built and compiled at this point since no changes were focused. The *gwdb* and the *webutil* modules on the other hand needed changes to accommodate new data for pictures messages and GPS updates. The web Applications also needed changes as discussed later.

4.2 Adding New Data Fields and Setting Up the Database

4.2.1 The *gwdb* Module

The *gwdb* and the *db* modules as mentioned earlier manage the connections and transactions to the database. To add new features to the package, new fields were added to the tables and necessary changes were made to ensure that they were accessible through the *gwdb* module. The module has table definitions and schemas to facilitate access. The changes were made to the classes and interfaces that provided access to the tables in the *gwdb* database. New features introduced include pictures that are seen in the web applications and location-based services that show the location of the mobile web servers on a map.

The *gwdb* module has ten classes in its *mysq*' package (path:\gw-GW_0_6_1\gwdb\src\com\nokia\mws\gwdb\mysql). This package is used for accessing the MySQL database of five tables namely *gwdb*. It has interfaces and classes that are used for installing and accessing the *admin*, *musers*, *musernames*, *muserpropr*' and the 'scopes' tables. The classes in the package are *MuserImpl* that has the table definition for the 'musers' table and the methods to access and change the data in it. '*AdminsImpl*' for the '*admins,* table; *MuserNamesImpl* for the *musername* table; *ScopesImp*' for the *scopes* table and *MuserPropsImp* for the *muserprops* table. All these classes also have interfaces that are in the

module's main directory,' \gw-GW_0_6_1\gwdb\src\com\nokia\mws\gwdb\.' Adding new fields to the *musers* tables necessitated the changes to the *MuserImpl* class the *Musers* interface and the other classes that have objects that use the *MuserImpl* class. This includes classes in the *webutil* package which are used by the web applications to query table objects. The diagram below shows the important classes in this package and their relationships with the database tables.

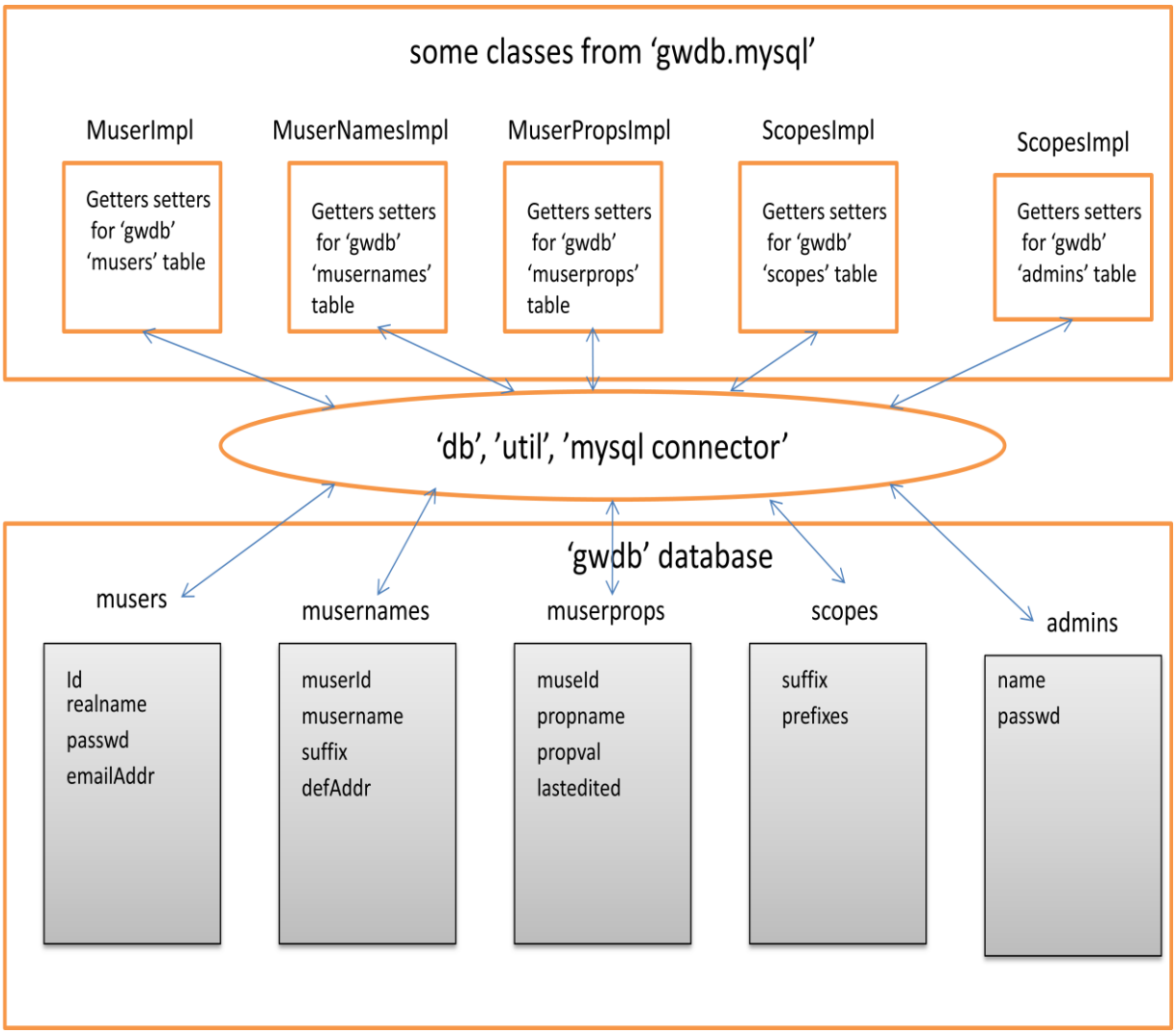

Figure 13: The Gateway Database and Data Access

As shown in figure 13, the *db*, *util*, classes have objects that are called by the objects in the *gwdb* module. These objects perform the actual transactions by connecting to the database through the *MySQL connector*.

The *musers* table stores data related to the mobile web-server users. It stores the identifier, the name, the password and the email address of the user. The *musernames* table stores information that is used to construct the URL that is used to identify the users' mobile web servers. The *musername* field holds the name that will be used to identify a user in a URL and the *suffix* stores the part that suffixes the *musername* such as *mobile.ict.ru.ac.za*. As it will be explained in the testing chapter however the functionality of addressing a user with a URL starting with the *musername* was not implemented. This is because the Domain Name Server was configured to only recognize URLs starting with *http://mobile.ict.ru.ac.za.* Instead the system used was that of adding the username at the end of the URL like *http://mobile.ict.ru.ac.za/~alex*, where *alex* is the *musername*. The *defAddr* field basically holds the address that should be used as a default.

Moving on to the next table in the diagram, the *muserprops* table stores properties for the mobile users. The properties stored in this table are those defined in the *MuserPropetyTags* class which is also located in the *gwdb* module. The properties stored include background pictures and messages for offline pages and the number of hits the page was paid. The values are stored in a field of type 'blob' namely *propval*. The *scopes* table stores the domains that are used for constructing URLs. It is used for synthesising the URLs that are stored in the suffix field of the *musernames* table. The *admins* table is used to store the details of the administrators, their names and passwords.

4.2.2 Changing 'gwdb' to add New Features.

The fields needed for the new features are those that hold GPS data, the message displayed on the maps, and for the profile pictures. The GPS data only needs three fields to keep the longitude and the latitude coordinates as well as the message from the last GPS update. The pictures are not stored in the database as 'blobs'. They were rather stored in a designated directory on the hard drive. The path of the picture is the one stored in the database. All these features are unique to every mobile user and every user will only have one at a time. This suggests a one-to-one relationship between the user and these data values. The most suitable table for this is the *musers* table which has four fields namely *ID*, *REALNAME*, *PASSWD* and *EMAILADDR*. The four fields to be added are *PICTURE*, *LATITUDE*, *LONGITUDE* and *MAPMESSAGE* in that order.

To accomplish this in the *gwdb* package, the interface for the *musers* table (*Musers*) was changed. The five methods added where:

```
String getPicturePath(String muserId) throws DbExc;

String getLongitude(String muserId) throws DbExc;

 String getLatitude(String muserId) throws DbExc;

void getMapMessage(String muserID) throws DbExc;

void changePicturePath(String muserId,

                String newPicturePath) throws DbExc;
```

These methods are implemented in the *muserImpl* class which implements the *musers* interface. The sample code for getting data used to implement the '*get*' methods in the *MuserImpl* table is as follows:

```
public String getLongitude(String muserId) throws DbExc {
    try {
                    String noData =  "not";
      String[]
        longitude = parent_.selectRow(TABLE,
                        new String[]{LONGITUDE_COL},
                        new String[]{ID_COL},
                        new String[]{muserId});
      return (longitude == null) ? noData : longitude[0];
    } catch (SQLException sqlExc) {
      throw new DbExc("Could not query muser "
            + muserId
            + ": "
            + sqlExc.getMessage(),
            sqlExc);
    }
  } //
```

Evidently the code is for the getting the longitude coordinates from the *musers* table. The method used is the *db.TableBase.selectRow* method which is located in the 'db' package or module. It basically takes care of selecting the fields from a given table. The parameters that it takes in are arrays for the column(s) to be returned, the column(s) to be matched and the values to be matched. In this case the variable *table* holds the value *musers*; the column to be returned is the longitude column and the column to be matched (the 'where' clause) is the *id*

column. It returns the longitude value for the column that match the given *muserId* (identifier). All the other get methods follow this basic format. The web applications and other modules access this information by calling these methods from the objects of this package (*gwdb*). The fields also needed to be added to the table definition in *MuserImpl.java*. The column names, their types, lengths, default values and whether they are null-able or not was specified in the table specification in the *muserImpl* class. An example code is as follows:

```
ColumnSpec musersLongitude =
        new ColumnSpec(musers_,
                LONGITUDE_COL,
               "VARCHAR(" + 16 + ')',
                true,  // non nullable
                'not'),  // no default value
```

The *db.ColumnSpec* class is located in the *db* package and it hold column specifications that were supposed to be used for setting up the database and for accessing it. The setting-up part however did not go so well. Figure 14 shows the edited database and the relationships between tables. The tables where related to ensure data integrity. If a user removes their account the deletion is to be cascaded to the *musernames* and *muserprops* tables to make sure all the user's data is deleted from the database.

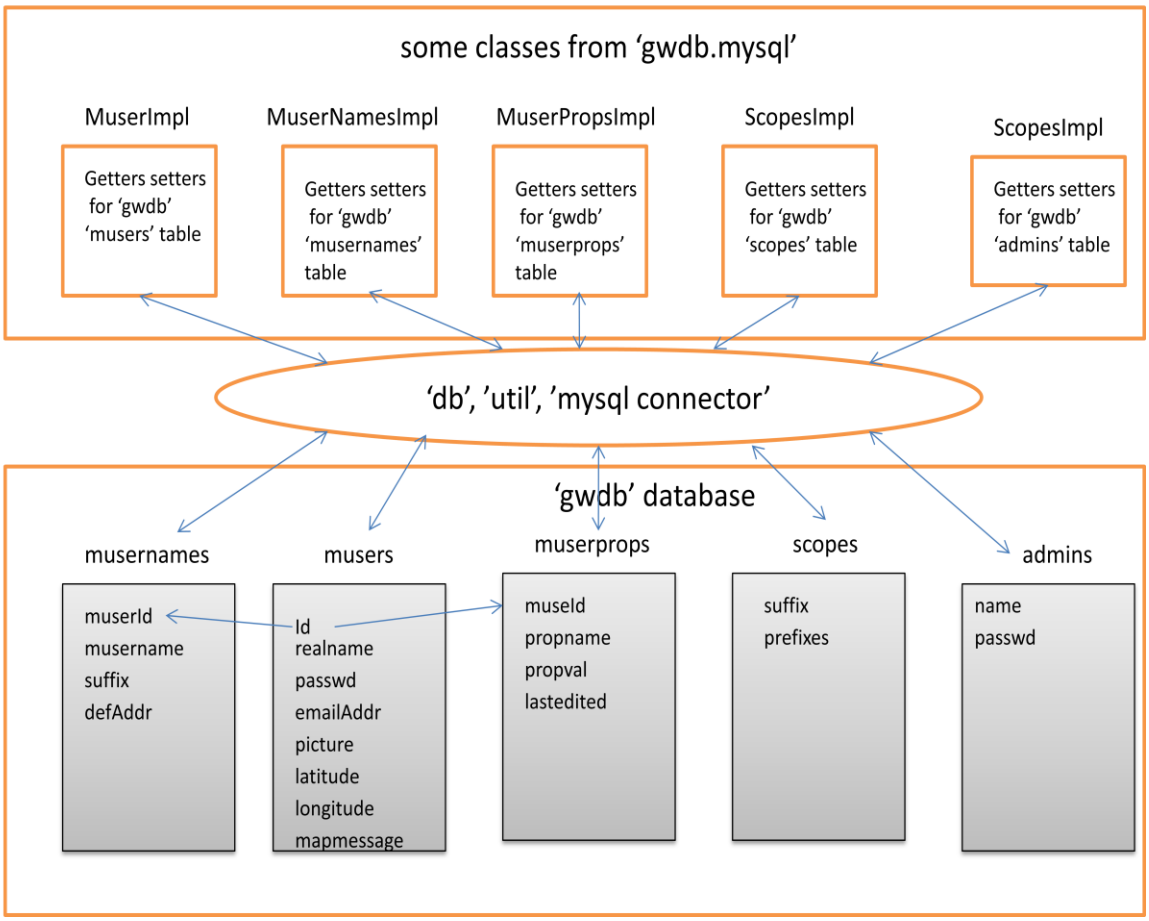

Figure 14: The Gateway Database and Data Access (modified)

4.2.3 Putting the Database Together

The package is set up in such a way that the person wishing to install it issues commands to install all the tables. The commands involved execute Bash shell scripts. The Bash scripts are also supposed to add the necessary references to the *classpath*, to set up the database and to generate a configuration file that is used to configure connection variables used for run-time connections.

Because the gateway is being setup on the Windows operating system, bash commands as well as scripts did not execute as expected. The tool used was 'bash'. The bash tool is an

emulation tool for the Bash shell on windows. Instead of producing the above-mentioned deliverables the bash script execution returns class-not-found errors. Attempts to correct the errors proved futile. The tasks had to be performed manually by studying the code and the scripts. This was also desirable as the code for setting up the database did not have any table relationship information at all. Since the table definitions were given from the code that is supposed to set the *gwdb* database up, the mission was not difficult at all. The relationships were also setup on the MySQL database in addition to what was in the code. The database name was set to *gwdb*, there are two user accounts *gwusr* and '*root.*' The usernames passwords for logging onto the MySQL server where provided to the gateway application by putting them into the *gw.build.properties* file.

4.2.4 The Database Configurations File

The database configurations file, namely *mws-gw-db.cfg*, is a file that is used to configure the database whenever the access details change. It is used to keep the database connection details current. The file was supposed to be a product of the bash-script execution mentioned earlier. It had to be put together (by studying the code that was supposed to set it up in the *db* package) to ensure database connections. The information contained in this package is the name of the database; the name of the database user that the package will use to access the database and the corresponding password. Also in the file was the URL to accessing the database. The configurations were as follows:

db.name = gwdb

db.url = jdbc:mysql://localhost:3306/gwdb

db.user = gwusr

db.passwd = 673367

The file is located in the ‘db’ package of the in the ‘bin’ folder. This file is used by the db package to establish connections with the *gwdb* database.

Chapter 4

Building and implementing the gateway

4.1 Introduction

The gateway for the project is implemented by building on the basic, open-source, gateway connection-management modules made available by Nokia. These modules are responsible for accepting and maintaining connections with the mobile phone connector (through a control channel) and provide objects that contain information about connection states. They are also responsible for opening up data channels when a request is to be serviced. Another part of the package is a database persistence module, *gwdb*, that manages run-time transactions and queries as mentioned in chapter three. Provided also are skeleton web applications that show how information can be accessed from the *gwdb* module at run time. This chapter discusses the customization and the building of the package to make a unique functional gateway.

4.2 The Structure of the Package

The gateway package is organized into eight modules, namely: *connector*, *db*, *gwdb*, *iapi*, *util*, *webutil*, *valve*, and *webapps*. All these modules perform tasks that collectively add up to a system that is capable of receiving and servicing mobile web-server requests. The

connector module handles the functions of the gateway connector as mentioned in chapter three. It is the part of the gateway that is responsible for receiving connection requests from mobile web servers. It keeps these connections alive by using an XML protocol that is custom-made for this purpose (through the control channels). When requests for a connected mobile connector come in the gateway, *connector* opens up a data channel as mentioned in the previous chapter. (All this is performed through the Tomcat web server and more details will be given in the Tomcat section.) The *db* and *gwdb* modules are collectively responsible for database setup and run-time transaction management. The *gwdb* module is where the tables are defined and the objects to access them reside. The 'db' module is specifically responsible for creating the database and for creating the configuration file (with passwords and user accounts) that is used to make connections at run time. It is also the module that is responsible for runtime transactions. The *gwdb* module on the other hand has table specifications and methods of accessing or querying them. Setting up these two modules, however, was not as easy as had been expected. This will be explained in the next section.

Next is the 'iapi' module. It is responsible for functions related to web requests such as showing the offline page when the requested mobile web server is not online. It also has interfaces that are related to. The other package is *util* which has classes that perform utilty functions for the other packages. These include functions like opening streams, converting from XML to strings and vice versa (for the connector module) and generating random identifiers. For the *db* and *gwdb* modules, *util* has classes that assemble the requests into final SQL statements and executes them after making a connection to the database.

The 'valve' module is a collection of classes that define objects for serving requests as they come in from web browsers. As mentioned in Chapter 3, the valve captures all the requests incoming on port 80 of the Tomcat server. The headers of the requests are then analysed by the request analyser to determine where to forward the request and the response objects. If the request URL matches one of the musers' chosen web server URLs, it is forwarded to the connector module. The connector checks if the requested server is online and forwards the request to it. If the requested server is not online, then the server offline page is shown. The '*webapps*' module has web applications that present gateway data to visiting users. The web applications take care of account management and enable the viewing of gateway

connections. The web applications are discussed in later sections. The valve runs as part of the Tomcat server and this is discussed in the section 4.3 of this chapter

Compiling the Package

Each of the modules in the package was compiled and built using the Ant build tool. When the command *ant* was issued in the main directory of the module, the files were compiled. A jar file was produced as a product of the compilation when the *ant jar* command was issued in the main directory of the modules. The *util*, *db*, *aipi*, *connector*, and *valve*, modules were built and compiled at this point since no changes were focused. The *gwdb* and the *webutil* modules on the other hand needed changes to accommodate new data for pictures messages and GPS updates. The web applications also needed changes as discussed later.

4.3 Setting up Tomcat

4.3.1 Setting up the Containers

The container for the mobile web server gateway is the Tomcat web server. For this project the version used is tomcat 5.5. Tomcat can be broken down into a set of containers, each with their own purpose [17]. These containers are configured by using the 'server.xml' file. The hierarchy for the containers is shown below as it appears in the 'server.xml' file format:

```
<Service>
 <Connector />
 <Engine>
  <Host>
```

```
      <Context />
    </Host>
  </Engine>
 </Service>
</Server>
```

The *service* tags for each container are contained within the *server* tags. The tags have attributes that describe the behaviour of the container when servicing requests. The Tomcat server is the interface for receiving all web requests for the gateway. It also has the responsibility for receiving and sending requests, as well as responses on behalf of the gateway connector. This is where the containers come in. The web requests are serviced by the Tomcat's Catalina container as explained in [19]. (The service name for this container is 'Catalina'). The connector used for this project is the non-SSL HTTP/1.1connector. The port reserved for this connector for the purposes of the project is port 80, as mentioned in preceding Chapter 3. The Catalina service is always configured by default when installing the Tomcat package. It is therefore not hard to set it up at all as it is the main Tomcat service. Below is the configuration of the connector from the 'server.xml' file:

```
<Connector  port="80"   maxHttpHeaderSize="8192"
        maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
        enableLookups="false" redirectPort="8443" acceptCount="100"
        connectionTimeout="20000" disableUploadTimeout="true" />
```

For Tomcat to service requests and other traffic for the gateway connector, a new container should be created. The service for this container should listen on the gateway host's port 15001. When the requests come in from the mobile web servers for connection establishment, the container needs to parse the stream coming in using a protocol. After this is done, the

requests need to be processed. The Tomcat container must know where to forward incoming data. This is shown in the figure 15 which is from Alex Hanik's presentation entitled *Hacking Tomcat* [18].

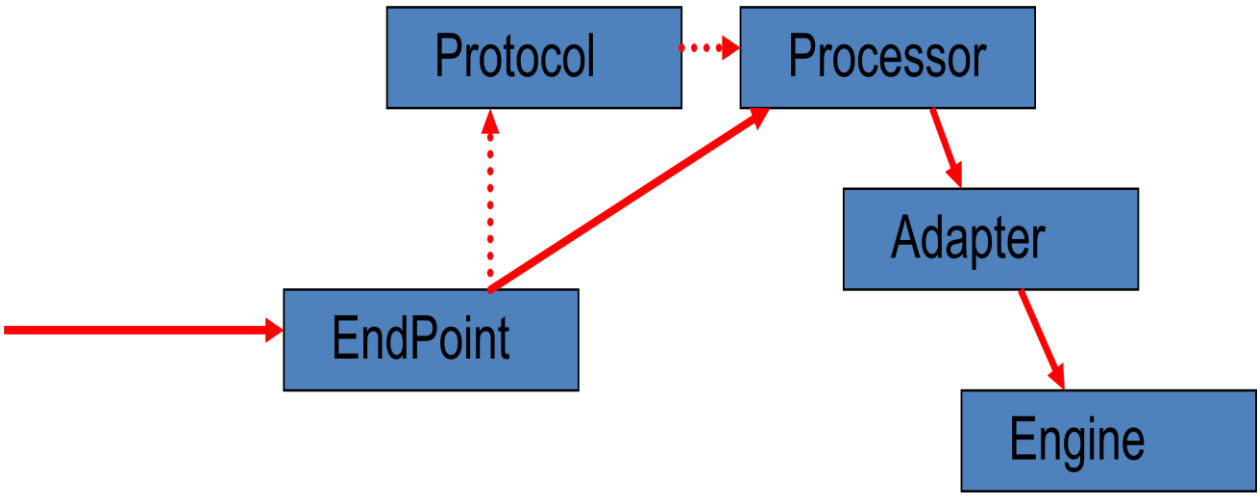

Figure 15: Tomcat Containers and request Flow

The *endpoint* is the receiver and sender of traffic. According to Alex Hanik all java.io, java.nio/apr and socket logic form part of the *endpoint* illustrated in Figure [18]. For the desired connector the important thing to do here is just to give the port, which is 15001. This is the port on which the gateway will communicate with mobile web servers. The protocol and the processor components should contain logic for parsing the data that comes in. The *Processor* component is responsible for setting up buffers for passing streams and for parsing the actual stream against the 'Protocol' component. For the gateway this is the *MuserProtocolHandler* which is an object from a class in the *server* package of the *connector* module. This package contains classes that parse incoming and outgoing XML data. For the control channel the UDP data could be for connection establishment, connection keep-alives and data channel requests (when the gateway requests a data channel from the mobile web server).

The *Adapter* component, according to Hanik, handles passing on the requests and responses, once parsed, to the container's engine. According to Apache's documentation of Tomcat, the container's engine " receives and processes all requests from one or more Connectors, and returns the completed response to the Connector for ultimate transmission back to the client". The configuration of the mobile web server connector container in Tomcat is as shown below (partially from [22]):

```
<Service name="Mws">
  <Connector  protocol="com.nokia.mws.connector.server.MuserProtocolHandler"
processorChainSpec="com.nokia.mws.connector.server.RequestStreamer:com.nokia.mws.connector.server.HitCounterResponseStreamer"  logLevel="INFO"
muserObservers="com.nokia.mws.connector.server.observer.Messenger:com.nokia.mws.connector.server.observer.EverOnline"
port="15001"/>
  <Engine name="Mws" defaultHost="MwsHost"/>
 </Service>
```

The *processorChainSpec* attribute shows the container the objects that use the container. The *Requestreamer* (in the connector module's server package) receives information about which mobile web servers are requested and serves content depending on whether the requested server is online or not. The *hitcounterResponseStreamer* is responsible for updating and incrementing the 'hits' property of the *muserprops* table every time a mobile web server is accessed successfully. The *muserObservers* attribute has references to the observer objects. These are the *Messenger* and the *EverOnline* classes or instances. Both are part of the 'observer' package. The *EverOnline* observer updates the *everOnline* property of mobile web servers (also in the *muserprops* table). This property indicates whether the user (mobile web server) has ever been online. The 'Messenger' observer on the other hand handles the

messaging. It sets the *msg* property in the *muserprops* database table. The observer handles the sending and reading of messages by the mobile users depending on the connection state, online or offline.

4.3.3 Setting up the Valve

According to the Apache Tomcat documentation [21] *A Valve element represents a component that will be inserted into the request processing pipeline for the associated Catalina container (Engine, Host, or Context).* As mentioned in chapter three, the *valve* or *dispatcher* for the gateway is the part that determines where to forward the web requests as they come into the gateway. It either forwards the requests to the web applications (the Catalina HTTP 1.1 container) or to the gateway (Mws) connector, which is also a part of Tomcat (as explained in the previous section). There needs to be a reference in Tomcat pointing at the class that implements the valve. This is accomplished by placing the line below in 'server.xml' outside all the containers:

```
<Valve className="com.nokia.mws.valve.MuserValve"/>
```

It's outside all the connectors because it is the one that directs incoming requests to either the web applications or the web servers. Phrased differently, it determines which container to forward a request to as shown in the diagram below. If the incoming request is for a mobile web server that happens to be offline, an offline page is generated and the 'out of site' web application is invoked. This is shown in Figure 16 with the line from the mobile web server container to the web applications.

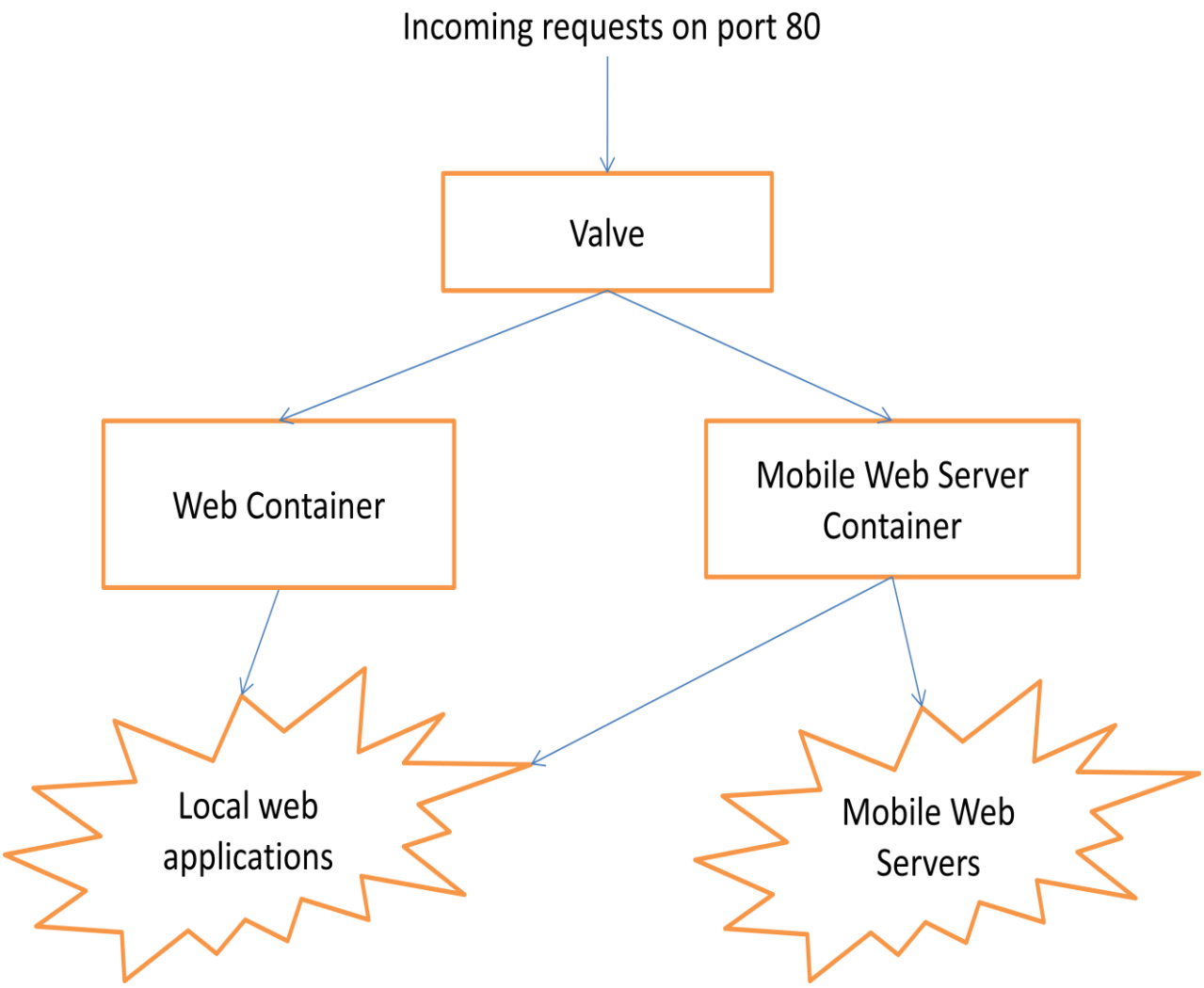

Figure 16: The Valve and the Containers

4.4 The Web Applications

The gateway would not be complete without the web applications. The web applications handle, amongst other things, user registration and user account management. For registration

a user only really needs to create an account with log-on details. The user should then be able to manage (change details, delete their accounts) when they so wish. The web applications also enable the administrator to manage mobile web server accounts by adding new users or deleting on editing details of existing ones. They also enable the web visitors to see the mobile users registered on the gateway and see whether they are online or not. The web visitors also get the benefit of seeing the number of hits web visitors have paid to a particular account. The web applications are the face of the gateway, they are all visitors really ever see and the presentation as well as the user-friendliness will determine the visitors' impression of the service. The web applications that come with the Nokia gateway package are far from complete and are minimal. The next section describes the process of customizing, adding functionality and setting them up.

The web applications for this project are implemented using JavaServer Faces and in some cases JavaServer Pages and Servlets. This enables applications to access other backend Java code to access databases and other runtime information such as the connection state of the mobile web servers. JavaServer Faces applications use the Ant build tool and have their directory structure organized in standard Ant application structure. The main web application folders contain the subfolders: *build*, *dist*, *docs*, *src* and *web*. In the main directories are also ant build files. Ant build files (build.xml) perform functions such as importing all the needed library jar files, for dynamically constructing the *classpath* for the project, compiling as well as installing the web applications into Tomcat using the *Catalina-Ant.jar* file. The first thing to do for any web application is therefore to set up this file. The logical flow of data in the web applications for this project is depicted in the Figure 17:

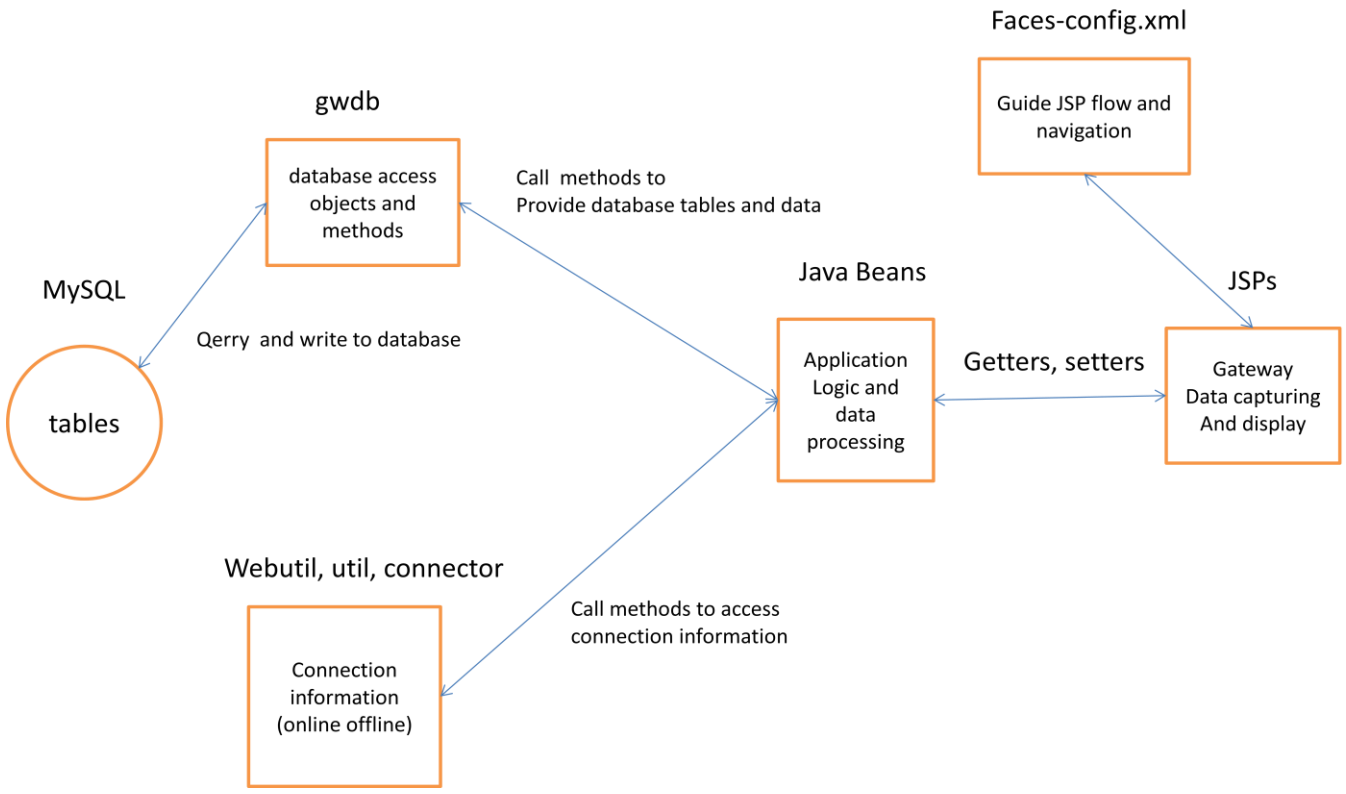

Figure 17: Web Applications and Data Access

The JavaServer pages capture information and display it to the user. They forward or get the data from the backing beans by using the beans' getter and setter methods. The flow and navigation of the JavaServer Pages is guided by the 'faces-config.xml' file. The backing beans have libraries and import the database persistence module objects and packages. If data from the database is to be queried, these objects are used to access it by calling their methods. The beans also process the data and perform functions like sending email messages. The *webutil*, *util* and the *connector* packages help in accessing connection information for mobile users. By instantiating objects and calling their methods, the JavaBeans can obtain information on which mobile web server users are online and which ones are offline. The next sections will concentrate on the challenges faced while setting up and customizing the web applications.

4.4.1 The User Registration Web Application

The gateway package did not come with a user registration web application and one had to be put together to allow web users with mobile web servers to register for the service. The application captures user information that is crucial to successful web-server-gateway communication. It also sends an email to the registrant confirming the registration event to them. The application should also have error handling to make sure that the details entered are valid for successful user registration.

The only data that is important to the functionality of the gateway is that for the user account: a user name and a password; and that for the access URL (that is the URL with which the web server will be identified). With this a user can connect their mobile web server to the gateway by putting their log-in details into the mobile PAMP connector. PAMP is combined Apache, PHP and MySQL on a mobile phone (running a Symbian operating system). However, there is other information that is needed for the web application presentation and user management. This data includes pictures for display and an email address to contact user when necessary. All this information should be solicited from the user with a clean, user-friendly interface.

JSP Pages

The web application first produces a user registration page that has a form with fields to be completed. Upon successful submission, the application flows to the success page after sending an email. If, however, the form was not completed successfully the application displays the error at the top of the page (if it is an internal error such as database access) or next to the field involved. If a cancel button is pressed, another page thanking the user for trying, is displayed. This flow logic is all implemented in the 'config.xml' file in the '/web/web-inf' folder. Also declared in this file are references to the backing beans which do the processing for the application. The main JSP Page and the final fields for user registration are shown in Figure 18:

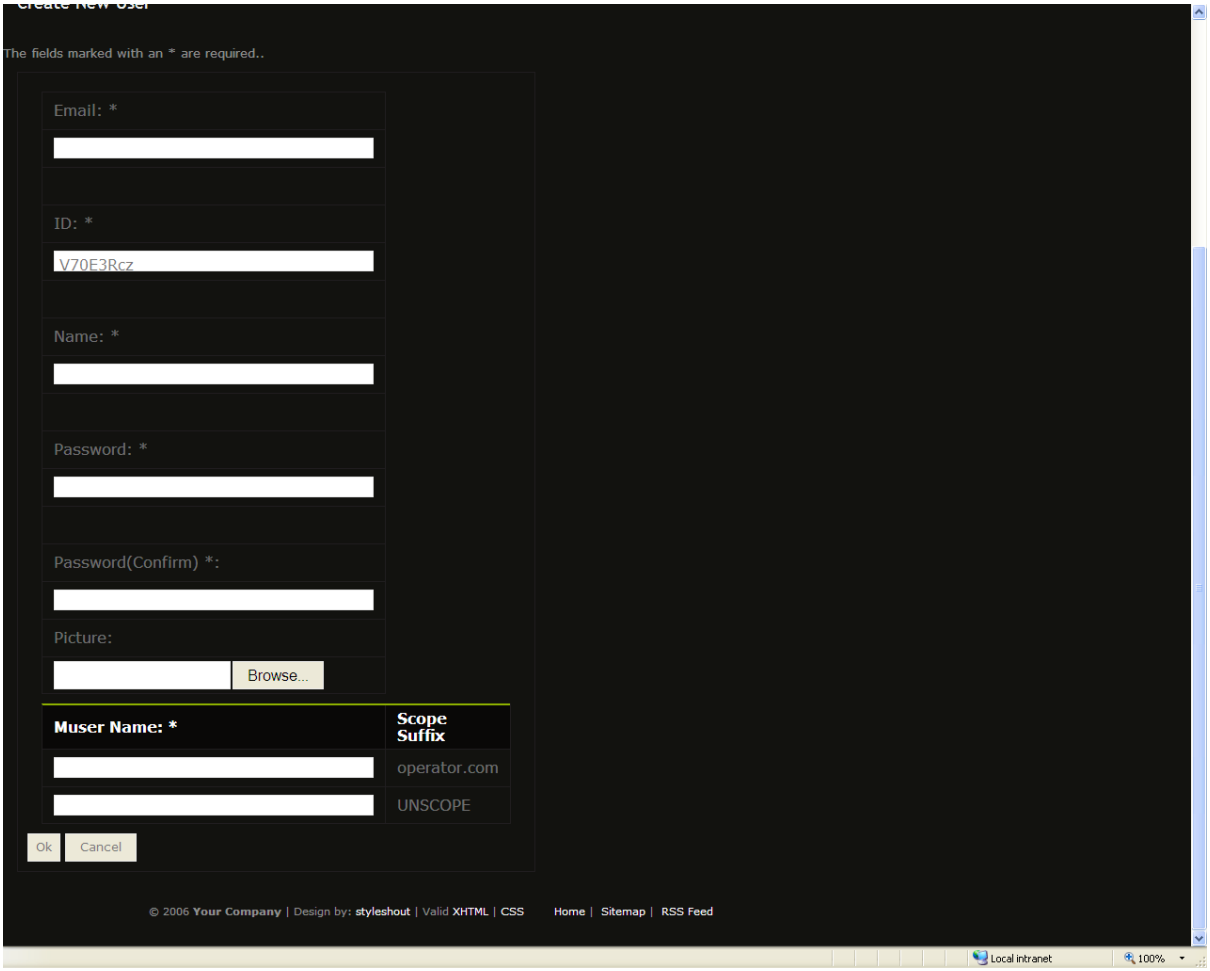

Figure 18: The Registration Page

There are tags from the JavaServer faces library for text, password and file upload fields. The file uploading tag had to take in images only. An attribute named *accept* which specifies the MIME type of the file had to be added to ensure this. The code extract from the JSP page for accomplishing this operation is shown below:

```
<h:outputText value="Picture:"/>

    <t:inputFileUpload id="myUploadedFile" accept= "image/*"

     value="#{muserCreator.myUploadedFile}"/>
```

The string *accept= "image/*"* ensures that only image files are uploaded as part of this operation.

The Backing Beans

The main aim for JavaServer Faces was to separate presentation from processing logic and styling code. The backing beans handle the processing for a java web application and the JSP pages handle the presentation. The backing beans are the *MuserCreator.java* and the *NameScopepair.java* classes. The *MuserCreator.java* class has getters and setters for the fields obtained from or displayed on the main registration form. It accesses the database calling methods from the *gwdb* module. The bean ensures that the user identifier is unique and registers the user after making sure everything is perfectly fine. After that, it sends an email message affirming the registration by sending the user the details they had registered with. The *NameScopepair.java* class is responsible for getting information about the domain names used for constructing the mobile web server users' URLS. The *gwdb* objects used for the registration are those for accessing the *musers, musernames* and the s*copes* tables.

The challenge encountered during the implementation of this application was the uploading of files. The 'muser' table in the database had no field for pictures. At first a 'blob' field was added to it for this purpose. However, storing picture in databases requires expensive operations and memory. A lighter approach was taken that stores the picture on a directory on the disk. The file path was then stored in the database. The pictures are all loaded in the same folder with the user identifier as their names. This ensures that the pictures have unique names and paths and can be overwritten easily.

The pictures are delivered from the interface using the *org.apache.myfaces.custom.fileupload.UploadedFile* class. The object holding the picture that is obtained from the interface is instantiated from this class. The picture is then stored in

a directory on the hard drive by using the operation below shown in the following code extract from the *MuserCreator.java* bean:

```
public void sendFile(){
try{
//first create dir for file - not needed ofcourse
        File fileOnServer = null;

//create empty file with specified name and path
     filePath = "e:/pictures/" + muserId_ + ".jpg";
     fileOnServer = new File(filePath);
          filePath = muserId_ + ".jpg";

// save uploaded file into new one
        BufferedOutputStream os = new BufferedOutputStream(new
FileOutputStream(fileOnServer));
        BufferedInputStream is = new
BufferedInputStream(myUploadedFile_.getInputStream());
 byte[] buffer = new byte[1024];
        int count = 0;
        while ((count = is.read(buffer)) != -1) {
          os.write(buffer, 0, count);
        }
 fis = new FileInputStream(fileOnServer);
        os.close();
        is.close();
}catch(Exception e){
  e.printStackTrace();
}
```

}

The *filePath* variable is then stored in the database. It is guaranteed to be unique since it is named after the user identifier. After all the input variables have taken in data from the input form, the *createMuser()* method of the bean is called. It invokes all the operations for creating a user.

4.4.2 The User Account Web Application and Offline Properties

The user account web application was set up but what the mobile users could manage on their accounts was very limited. The application allowed the users to change the identifiers (usernames), passwords and real names. However there was a lot in the application that the user could change. Additional data fields and forms were needed for changing pictures and messages shown when a user is offline. A user also need to set if they wanted to be visible to others on the 'all-users' or online-users page. The *gwdb* objects used in the beans are for accessing user information from the *musers* tables. However, the three fields for offline data and visibility are part of the *muserprops* table which stores properties for users. Also, session tracking and password recovery functionality had to be added, but this is fairly standard and the code is shown in the extracts at the end. The main final form for this application is shown in the Figure 19:

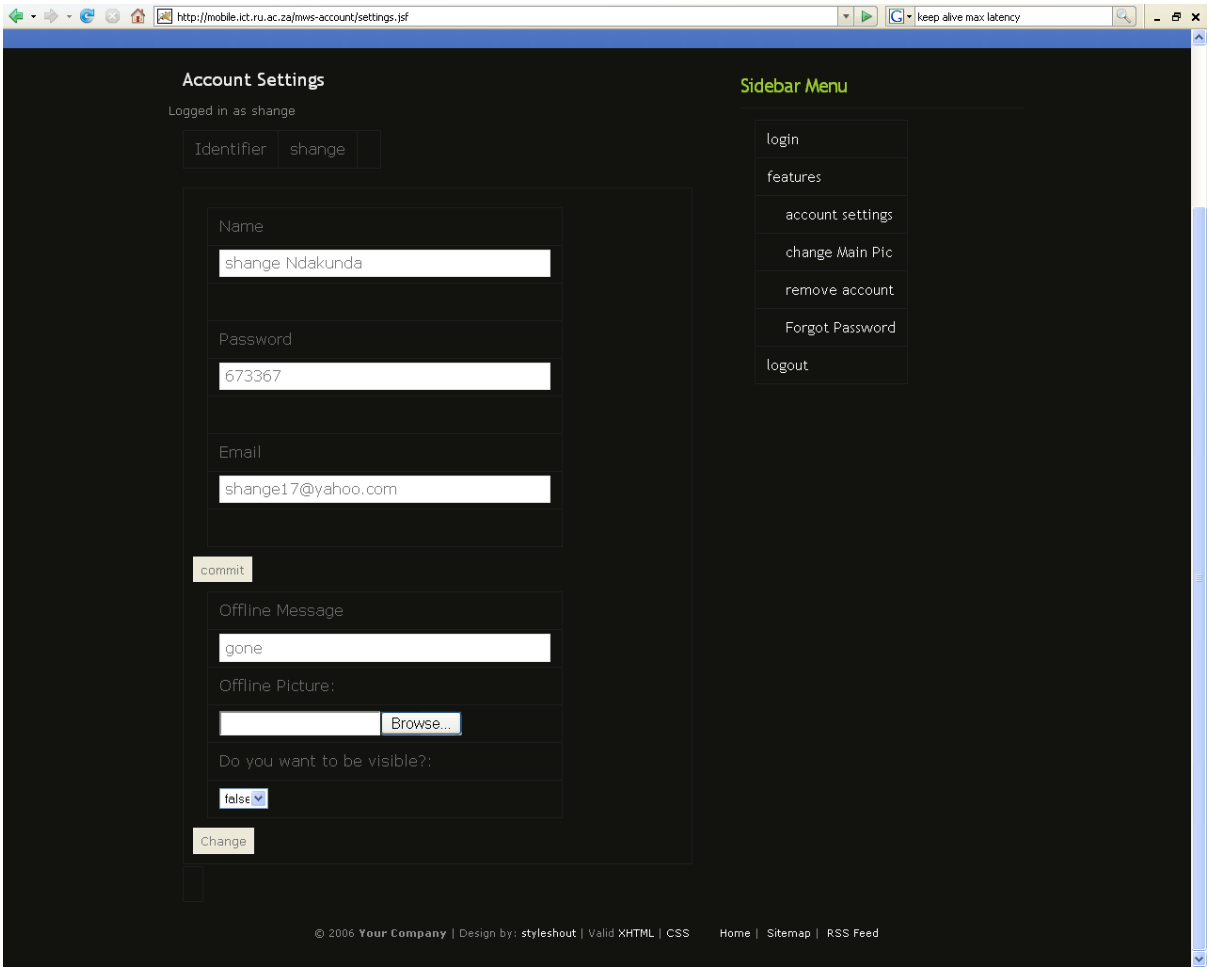

Figure 19: The Settings Page

A *success.jsp* page was also added to confirm the event of successful account data changing. Necessary changes had to be made to the *faces-config.xml* file to add the page to the overall application-flow logic.

Allowing Users to Change Offline Message and Picture (in the Backing Bean)

In the main backing bean (*Settings.java*), the methods used to change the offline fields are those of the *MuserProps* interface in the *gwdb* module. The properties have codes. To set these properties, the define method from the *MuserProps* interface was called. And the value

of the properties set to those input from the form in the JSP page. An extract of the most important code for accomplishing this (from the *Settings.java* bean) is shown below:

```
MuserPropTag  mpt = MuserPropTagMgr.getMuserPropTag("oos-text");

 muserProps().define(muserId_, "oos-text", msgProp_);
```

oos-text is the code for the offline message, *muserId_* is the user that is currently logged on and *msgProp_* is the input data from the form. The *muserProps()* method returns the *muserprops* table from the database for writing and querying. The offline picture is stored as a '*blob*' in the database (*muserprops* table). An overridden method of the define method takes in *InputStream* types instead of strings. This method is called for uploading the offline picture as shown in the code extract below (from the *Settings.java* bean):

```
MuserPropTag  mpt = MuserPropTagMgr.getMuserPropTag("oos-fg");

 muserProps().define(muserId_, "oos-fg", fis);
```

oos-fg is the offline picture tag code in the *muserprops* table; *fis* is the *FileInputStream* object that contains the image uploaded from the input form. The file, however, does not come here directly it has to be put in a *FileInputStream* object first. Simply taking the file object from the interface and converting it to a *FileInputStream* object directly does not work. The uploaded file was in the form of the *org.apache.myfaces.custom.fileupload.UploadedFile* object and calling the *getInputStream()* method used for this was not rendering the desired results. To get around this, the file had to be stored on the disk (as done in the *sendFile()* operation for registration) and converted to a *FileInputStream* object (*fis* in the code extract). This worked as expected. An example of an offline page is shown in Figure 20 with a picture and message left by the user.

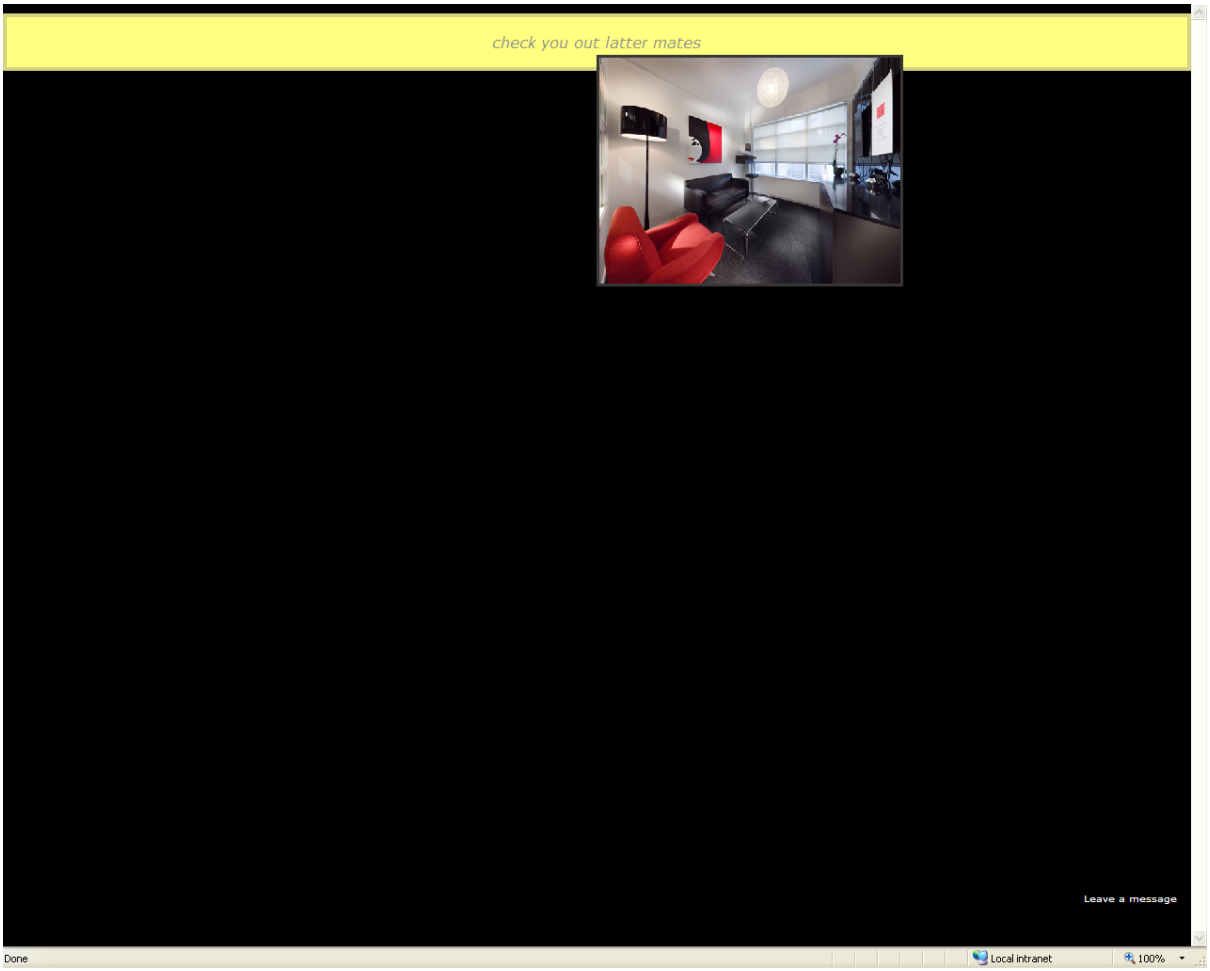

Figure 20: The Offline Page

Allowing users to set their visibility

Another feature added is that of allowing users to set if they want their status to be shown online. They could set the combo-box on the screen to either true or false. If the user chooses 'true' their status are made 'public' and everyone could see when they are online and when they are not. If it is set to false their status is not set to false. This is set in part by using the extract of code below:

MuserPropTag

```
mpt = MuserPropTagMgr.getMuserPropTag("show-on-portal");
```

muserProps().define(muserId_,

"show-on-portal",

visibleProp_);

Allowing Users to Change Profile Picture

Another issue was uploading another picture (for the main profile picture). A JavaServer Faces error about the clashing of variable names of the uploaded files when they were different in both the forms and the beans was being displayed. To overcome this, the picture had to be put on another JSP page (*changePic.jsp*) using a separate backing bean (*ChangeFile.java*) to clear out these errors. Necessary changes had to be made to the *faces-config.xml* file to reference the bean and to add the page to the overall application-flow logic.

Session tracking and Password Recovery

Session tracking using cookies was added to the application. This was done by adding the cookies to a *javax.faces.context.FacesContext* object (*facesContext*). The following is an extract of the main code used to add the cookies:

((HttpServletResponse)facesContext.getExternalContext().getResponse()).addCookie(username);

((HttpServletResponse)facesContext.getExternalContext().getResponse()).addCookie(password);

This is a standard way of adding cookies to JavaServer Faces web applications.

For the password recovery feature, a user just needs to enter their user identifier in a textbox. Their email address is obtained from the *musers* table and the corresponding password is sent to their mailbox. Necessary changes had to be made to the *faces-config.xml* file to add the password-recovery page to the overall application-flow logic.

4.4.4 The Admin Web Application

The administrator web application allows the administrator to manage the gateway from the web. The functions that an administrator can perform are adding new users, deleting existing ones and changing their details. The application also has a page for showing all the registered users. Another page also shows the users that have their web servers online. The registered users' page may be seen by the administrator only. For social networking purposes however this is not favourable and the application was modified to be seen by all the web visitors. The online users' page was also made visible to the public so that they might network with them.

The application was edited to display pictures uploaded by the users. This meant using the methods of the '*gwdb*' objects to get the picture name of the mobile user. The task, however, was not as simple as initially anticipated. The pictures were stored on the hard disk on a folder that was outside the web application's context path. If the images were put in the context path of the 'admin' web application it would have deprived other applications access rights. A solution had to be found to make sure the pictures were visible from outside the web applications' context paths. The solution was using a third party servlet (*com.jsos.image.ImageServlet*) that was in form of a jar file. The jar file was placed in the library folder of the web application. In the *web.xml* file a servlet and its mapping was added. This was done with this XML extract:

```
<servlet>
    <servlet-name>Image</servlet-name>
```

```
    <servlet-class>com.jsos.image.ImageServlet</servlet-class>
  </servlet>

 <servlet-mapping>
    <servlet-name>Image</servlet-name>
    <url-pattern>/servlet/Image</url-pattern>
</servlet-mapping>
```

In the backing bean (*MuserInfo.java*) the picture path is obtained from the 'gwdb' objects using the *musers.getPicturepath* method. The picture path is then rendered to the JSP using a getter method. On the page the *graphicImage* tag of the *http://myfaces.apache.org/tomahawk* library was used. The image's path is referenced in the *url* attribute of this tag. The URL is basically that of the *com.jsos.image.ImageServlet* servlet as specified in the *web.xml* (*/servlet/Image*). The parameter taken by the servlet (through the URL) is the file path and that is *e:\pictures\#{muserInfo.picturePath}. #{muserInfo.picturePath}* is expression language to access the *picturePath* property of the *muserInfo.java* bean. The code extract is as follows:

```
<t:graphicImage url="/servlet/Image?e:\pictures\#{muserInfo.picturePath}"
                border="1"
                alt="image not available."
                width = "90" height = "120"
                />
```

This successfully displays the picture from outside the context path of the application, that is in the folder located on the hard disk (e:\) in the pictures folder. Figure 21 is a screenshot of the mobile web server gateway users' page.

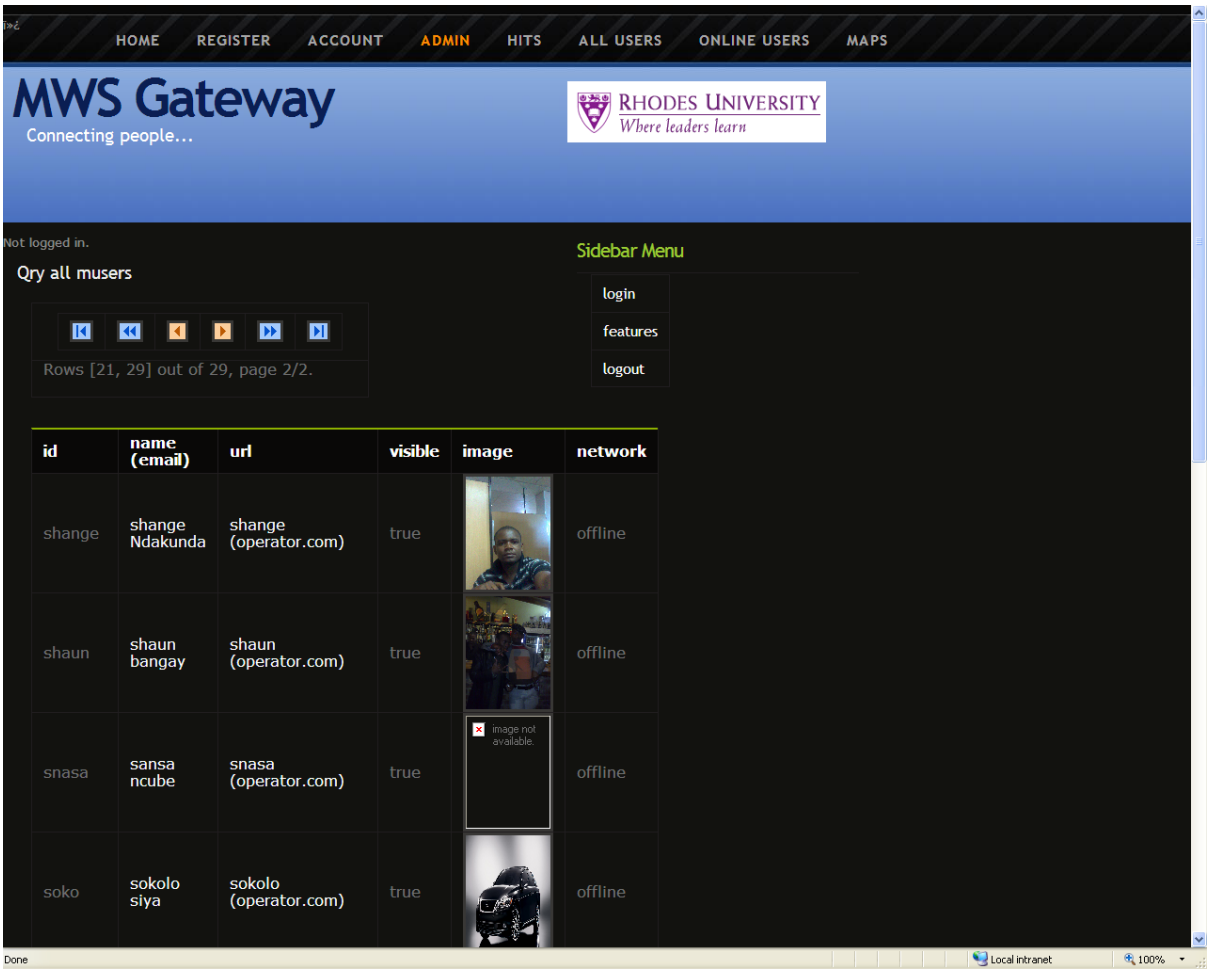

Figure 21: The All Users Page

4.4.5 The main Page and Styling of the Web Applications

The web applications needed to be connected to other pages and to a main page (the initial page that connects all the others). Styling also needed to be changed for the application to look more attractive and consistent. The implementation had to be simple and minimal to allow for easy editing. A Cascading Style Sheet (css) file was used to style the web applications (basic.css). Each application has its own 'basic.css' file in the web folder. The styles are applied using four files because of they are needed in different places in the JSP file. *<@include>* tags are used to include the files into the pages at compile time. The three files contain content to format the header, the body and the footer. The file extracts fro the four *'.inc'* are shown in the appendix, but just for clarity's sake the files contain links and tags that are included at four distinct places in the main file were they styles are needed. Including the four files required the tags following tags (in the JSP files).

<%@include file="inc/head.inc" %>

<%@include file="inc/subBody.inc" %>

%@include file="inc/subBodyFin.inc" %

<%@include file="inc/footer.inc" %>

The 'head.inc' file contains the header information and the reference to the 'basic.css' file. The other '.inc' files have '* *' style tags for the different parts of the JSP files such as the navigation list the body, and the footer. After application the results is the styling for the pages shown in the preceding sections with the links to all the web applications and a highlighted link (orange) link for the current one. The Main page basically has the link to all the web applications; software downloads of the web-server and other supporting software that works with the gateway.

4.4.6 Setting up the Web Applications

After everything is edited and finalized for the web applications an Ant command is issued in the main directory of the web application to compile it (the default action in the 'build.xml' file). Once this is done, Tomcat is started and the 'Ant install' command is used to install or deploy the application. This uses the *Catalina-ant.jar* file which is used by 'Ant' to deploy applications. Once this is done the web applications can be visited using their context URLs or by going to the *index.html* page (the main page) that has all the links to all the other web applications.

4.5 Chapter Summary

In conclusion, the author has discussed the relevant issues encountered during the process of building and setting up the gateway. The chapter explain how the gateway was set up using the open source Nokia gateway. It describes how the database was installed and how Tomcat was configured. The chapter also explains how the web applications where modified to add new functionality and to make them more attractive

Chapter 5

Design and Implementation of the Location Based Services for the gateway

5.1 Introduction

One of the project objectives was to design and build a new feature that works harmoniously with the other parts of the gateway. The new feature implemented is the gateway location based service that gets GPS information from the mobile phone serving mobile content and shows the location of the web servers on a map. Since location is a sensitive issue, the mobile user would need explicitly to log on to upload their coordinates to the server. They should also be able to delete the coordinates if they so wish. This chapter will discuss the design and the main points of implementation encountered during this phase of the project.

5.2 Application Specifications and Development Considerations

The application residing on the mobile phone needed to be capable of soliciting authentication information and for obtaining GPS data from the GPS device on the phone. The application was designed to communicate its information with the server and receive a reply stating whether the GPS coordinates were to be updated at a certain interval. The information was meant to be sent be sent over a TCP connection which is terminated after the updates to make sure power use on the mobile phones was minimal. Contrary to expectations, however, the implementation of TCP connections using streams was not so simple. The approach taken was that of establishing a TCP connection to the server and opening streams to communicate through this connection. The program worked very well on the emulator and on the Local Area Network. On GPRS networks it was a totally different story. The TCP connections were being established but the streams were not being established. Instead the

program hung at that point until it was stopped. The simplest solution to the problem was to use UDP connections instead of the originally proposed TCP. This is discussed in the implementation section of this chapter.

The GPS application was designed to be invoked explicitly by the user of the mobile phone, separately from the use of the web server and its connector. This application can in future be easily modified and used for other features if implemented separately from the mobile web server code. Another reason why the application was implemented separately is that it was much easier to implement it using the Java Mobile environment (JME) rather than Symbian C++ or Python for Symbian. Both these languages are currently very inconvenient for implementing applications in as they need to be Symbian-signed before they can do anything useful like accessing the GPS data. For signing one can either apply for a publishers' key (given only to publishers!) or upload the application to Symbian Signed which enables one individual to run the application on only one phone[1]. Both approaches are relatively inconvenient compared to the use of Java applications on the mobile phone.

There is a server application on the gateway that receives information from the mobile phones and updates the location information in the *gwdb* database. This application was implemented in Java and it listens on a specific port for any incoming connections. Its operations are simple: for each incoming request (thread), it reads the message and divides it into tokens; checks if the user exists and updates their GPS coordinates and map message (the message that will show on the map). If a user does not exist or their logon details do not match, they are ignored. The GPS coordinates and the map message are accessed later by a JavaServer Pages web application through the *gwdb* module. The web application then displays the most recent updates at the given coordinates on a Google map. The service is delivered as shown in Figure 22.

[1] Nokia's recent purchase of Symbian Signed may change this situation, which has caused much concern and anger in the mobile development community.

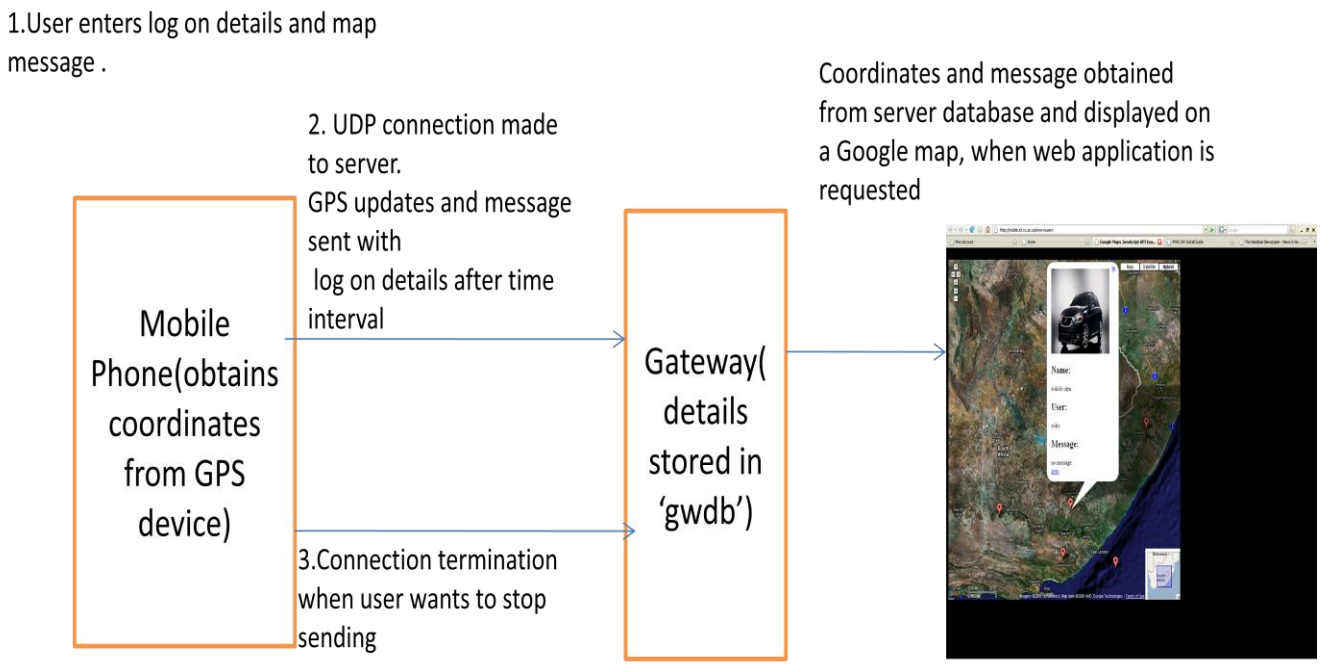

Figure 22: The Gateway's Positioning Feature

5.3 Designing and Implementing the Applications

As shown in the diagram in section 5.2, the application should have a client-server architecture, with the gateway hosting a server that awaits requests from mobile clients. The application on the server has to be multi-threaded and has to access the database through a single connection. This is much more efficient than creating new objects and database connections as connections come in from the clients. Since the client application would be behind the operator firewall it would need to make the connection to the server. It does not matter if the connection is unreliable because the messages are resent after a number of milliseconds. Therefore employing UDP for the job is the best option because packet loss is not a major problem in this instance.

5.3.1 The Mobile Phone Application

The application on the mobile phone has a form that takes in three values: the username, the password and the message to be displayed on the map. In the background, it opens a UDP connection to the server on a known port. The GPS coordinates are obtained from the phone's GPS device by a class that implements the *javax.microedition.location.LocationListener* interface. This interface requires that the classes implementing it subscribe to three methods: the *locationUpdated*, *locationStateChanged* and the *run* methods. In the *run* method preferred GPS accuracy, and battery usage are specified. This method is responsible for obtaining the GPs coordinates from the GPS device. It calls the *javax.microedition.location.LocationProvider.getLocation()* method that gets the coordinates from the GPS device. Also set in this method is the interval between which the application prefers to get the GPS coordinates.

The other method that classes should subscribe to is the *locationUpdated* method. This method contains all the tasks that are to be carried out after a successful GPS update, such as. sending the coordinates to the server. The message is constructed using a special string delimiter, *!;* (an exclamation mark and a semicolon) to separate the tokens. The components of the message are (as previously mentioned) the username, password, latitude and longitude coordinates and the message to be displayed on the map. The message is then converted to bytes and inserted into a datagram. It is then sent to the server on port 55555 using the connection established earlier. This is repeated after every ten thousand milliseconds until the user terminates the connection. The actions are illustrated in the Figure 23:

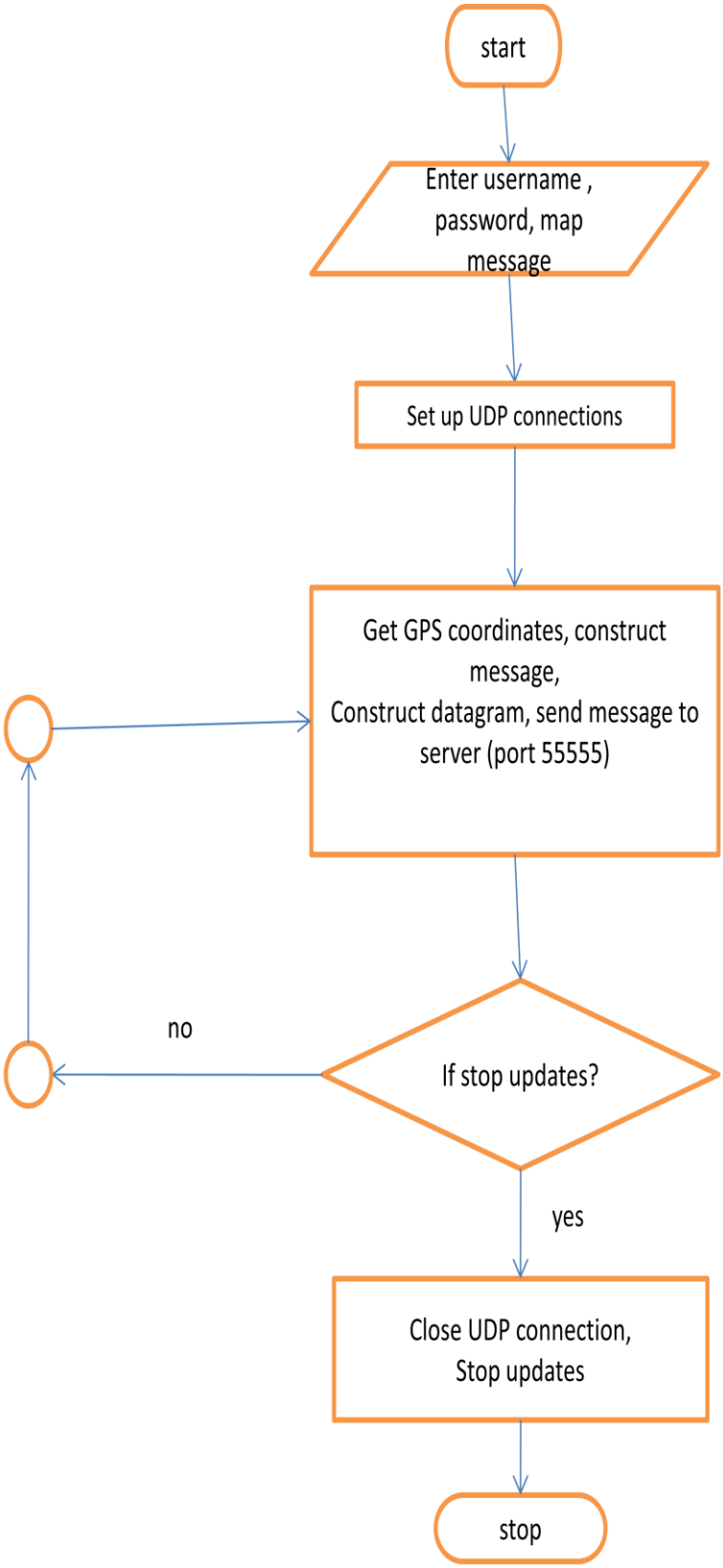

Figure 23: The Client Application's Flow Chart

While the application is updating the server, it displays a form that tells the user that the updates are currently being made every ten thousand milliseconds. If the user presses the back button on the server, the *locationUpdated* method is stoped.

5.3.2 The Server Application

The application that resides on the gateway server sits and waits for incoming connections and messages. When started, it establishes a connection to the *gwdb* database and waits for datagrams to come in on port 55555. When a connection comes, it is serviced by a thread. The thread reads in the datagram and converts the contents to a string. The string is parsed through a tokenizer to get the separate parts of the message into variables. The username and the password parts of the message are then matched against those in the database. If the user exists the updates are made to the *gwdb* database's *muser* table. The fields overwritten are the *latitude*, *longitude* and the *mapmessage*. The connection thread runs a while loop that listens for incoming connections on the designated port. The loop is terminated when the server is stopped.

5.4 The Web Application

5.4.1 Introduction

The GPS service would not have been complete without the front end, the web application. The web application's functions are simple. It takes the coordinates and the message from the database through the objects provided by the *gwdb* module and displays the markers of those that have GPS coordinates at their location on a Google map. When a marker is clicked, the an information window should pop up showing the profile picture of the mobile user, the username, their real name, their message and a link to their mobile web servers. The *gwdb* methods used for obtaining the data from the database are those of the *musers* table added to the database in section 4.2.1.

5.4.3 Server-Side and Client-Side Value Exchange

Google maps can be used by getting a key from ‘maps.google.com.’ Once a key is given it can be used for maps in a particular Domain or sites using the registered URL. The maps can be manipulated by using JavaScript code that can call Google maps-specific objects and methods. The methods and objects can be used to draw maps on a page, to put markers at specific coordinates and to put information windows. The information windows can also be displayed after clicking a marker (known as an overlay) after adding an action listener to the marker. This application needed to be displaying markers at positions determined from the coordinates stored in the *gwdb* database. Also, the markers are displayed only when there are values for them in the database. All the users that have coordinates in the database need to have their markers displayed at their positions. Moreover, when the markers are clicked they display the picture of the user, their names, their map-message and a link to their mobile web server. Before this was done one crucial thing had to be understood, the server-side and client-side values exchange.

5.4.3 Server-Side and Client-Side Value Exchange

The web application was implemented using a servlet that constructs a new page through a *java.io.PrintWriter* object. The reason why this approach was taken was that it is much easier to integrate with Google maps JavaScript code and to iterate over it in a manner that yields the desired points on the map. The challenge involved in rendering the Google maps to the client for this application is that it should be a combination of server-side and client side code. The two sets of code are executed on the server and the browser respectively. Although the aspect of Google maps using server-side Java values is not well documented, what was noted was that if the values (from server-side variables) are rendered to the browser as part of JavaScript code they are executed anyway. An example doing this is as follows:

```
pw.println("point = new GLatLng(" + lat + ", " + lon + ");
map.addOverlay(createMarker(point, " + count + "," + capt + ")); ");
```

The bold variables are server-side variables that ('lat' for latitude coordinates from the database, *lon* for longitude coordinates) are used by client-side Google maps JavaScript code (in italics). On the server the variables are resolved to their current values and are executed at the client as if the were normal client side values.

By writing the values of the variables on the server to be executed in JavaScript code, a technique to add markers, information windows and listeners to the maps was realized successfully.

5.4.4 The Implementation

The *gwdb* package was imported to access the coordinates, pictures and messages. The application was first coded separately (JavaScript and Java) and then merged to produce a map from the database. The program basically has a Java loop that iterates over all the users and checks if they have left any GPS coordinates and messages. If they have, the values for the current user are put into variables that are then put into JavaScript code to form markers and information windows. The code for obtaining the values is shown below:

```
while (musers.hasMoreElements()) {
        String muserId = (String)musers.nextElement();
        String realName = gwTables().getMusers().getRealName(muserId);
        String pic = gwTables().getMusers().getPicturePath(muserId);
        String lat = gwTables().getMusers().getLatitude(muserId);
        String lon = gwTables().getMusers().getLongitude(muserId);
      String message = gwTables().getMusers().getMapMessage(muserId);
```

Once the values were put into the variables they had to be put into JavaScript code in such a way that the coordinates determined the position of the markers or overlays and the other information was put in the information window. This was done with the following code:

For the information window (put in a string variable first):

```
String capt = " \" <img src='http://mobile.ict.ru.ac.za/mws-musers/servlet/Image?" + pic + "' " + "width='200' height='200'/> <br/>\" + \"<h2> Name: </h2> " + realName + "<h2> User: </h2>" + userId + "<h2> Message: </h2>" + message + " <br/>\" + \"<a href=''> goto </a>\" ";
```

For the image, the approach was different from that used in the admin web application: instead of specifying the directory and the file path in the code, this information was put in the 'build.xml' file as shown in the following code extract:

```
<servlet>
    <servlet-name>Image</servlet-name>
    <servlet-class>com.jsos.image.ImageServlet</servlet-class>
    <init-param>
     <param-name>dir</param-name>
     <param-value>e:/pictures</param-value>
    </init-param>
   </servlet>
```

To place the overlay in the position determined from the coordinates the *GlatLng* javascript method was called. To create a marker with an overlay, a custom JavaScript function ('*createMarker'*) had to be called:

```
pw.println("point = new GLatLng(" + lat + ", " + lon + ");
map.addOverlay(createMarker(point, " + count + "," + capt + ")); ");
```

The variable *count* holds the number of iterations the loop has suffered. It is passed to the *createMarker* method for uniquely identifying the marker. The *capt* variable is that shown in the earlier code extract and has information that will be shown in the information window.

The *createMarker* method was defined using this code:

```
pw.println(" function createMarker(point, number, toSay)");

pw.println(" {var marker = new GMarker(point); marker.value = number;
GEvent.addListener(marker, \"click\", function() {var holder = '';  ");

 pw.println("holder = toSay");

  pw.println("map.openInfoWindowHtml(point, holder);  });return marker;}" );
```

As is evident in the code, the method creates a marker at the coordinates given using the Google *GMarker* method. The marker is then given the value that was input as the value in the *count* variable when the method was called. The *GEvent.addListener* method then adds a listener for clicking actions to be picked up when a user wants to see information about a particular marker.

The results of the application are best shown in Figure 24 which is showing a screenshot of the requested map getting the values from a database:

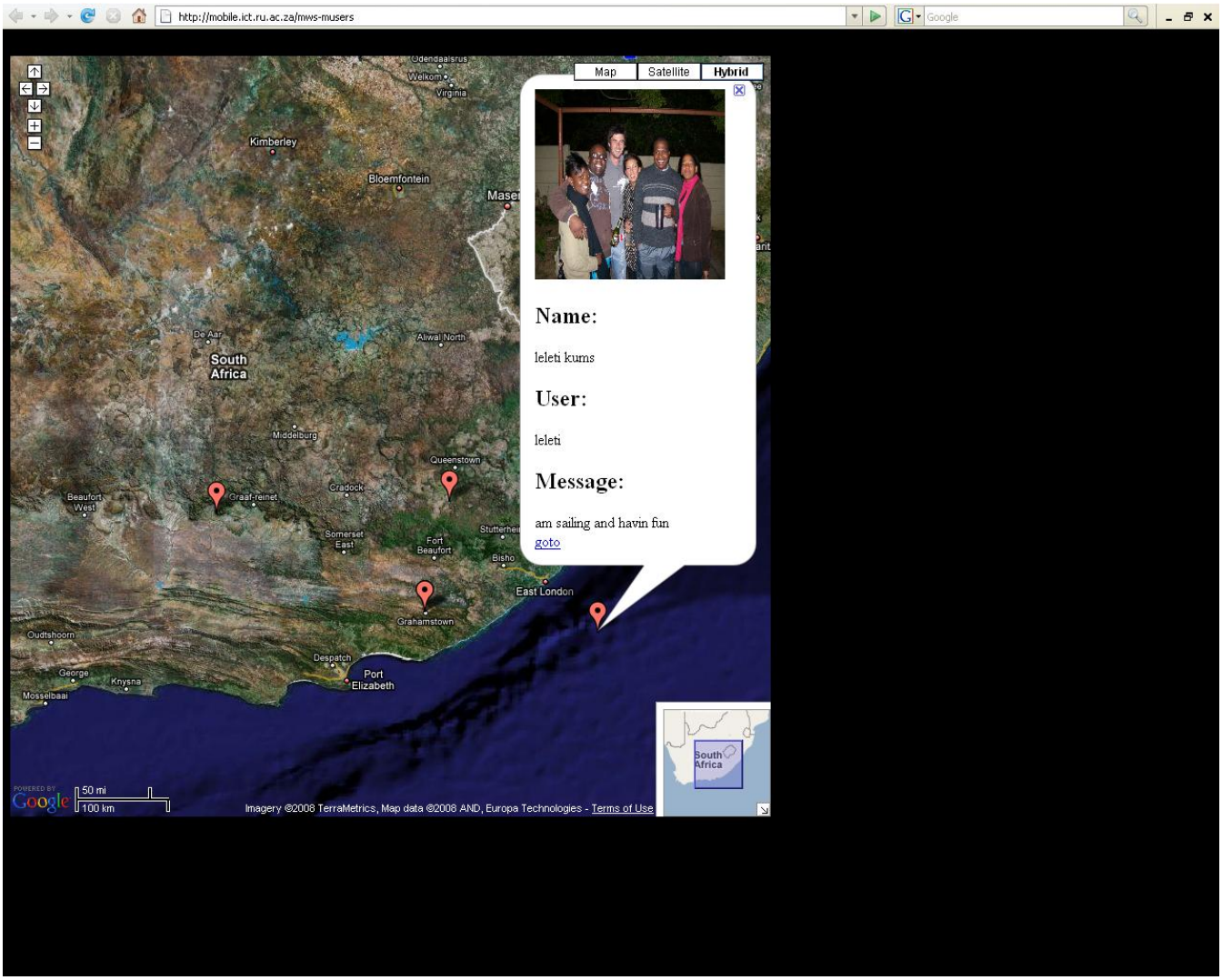

Figure 24: The Positions of the Mobile Web Server Users

5.5 Chapter Summary

In conclusion, the author has discussed the relevant issues encountered during the process of adding the positioning feature to the gateway. It discusses the client and the server applications and how they communicate to deliver location on a web application. A google map is used to display the location of the mobile web servers and the offline messages left by the mobile web server users.

Chapter 6

Testing the Gateway

6.1 Introduction

Now that the gateway was set-up the last thing to do is to test that it is working properly. The gateway as introduced in chapter one should be an intermediate point between the mobile web servers on phones on a cellular network and HTTP clients on the internet. With the gateway the mobile web servers can be addressed and accessed whenever they are online. The problems related to the mobile operator firewall are solved by the gateway without any harm or help from the operators themselves. The primary focus of this chapter is on testing the gateway. Although for the gateway to server its main purpose it had to be visible on the internet, it was first tested in the Local Area Network. This was done using a Nokia N95 phone with the web servers installed on it. After these tests were completed successfully the gateway was finally put on the internet with the help of the Information Technology Department. Further tests were performed to make sure that the gateway was providing access by having the mobile web servers addressed through it. The tests where performed using the Nokia N95and the Nokia N82 mobile phones with web servers installed on them. The three phones laid on the MTN and the Cell-C networks.

6.2 Testing the Gateway in the WLAN

6.2.1 The Tools

The tools used for this test were:

1. The Nokia N95 that was registered on the local wireless network

The software that was installed on the phone was the PAMP mobile web server (specifically *PAMP_with_htdocs_on_c.sis*). PAMP is a mobile web server which is a combination of *mod_PHP*, *mod_ Apache* and *mod_MySQL*. The *mod* prefix means that they are the ported versions of the originals (ported to work on the Symbian operating System). The version of PAMP used for the tests (*PAMP_with_htdocs_on_c*) comes with a mobile connector that can be configured to connect to a particular gateway on a specified port. The software required for PAMP to work were the *openc_ssl.sis* and the *pips_s60.sis*

Also installed on the phone was the Nokia Mobile Web Server (version 122). The Nokia mobile web server is a ported Apache web server as explained in chapter 2. The server-side scripting for web applications of this web server are done in Python for S60. Therefore the supporting python runtime was installed using the *PythonFor60.sis* file.

2. The Hamilton Lab wireless access point

3. A virtual host on an Intel Xeon 2.4 GHz processor with 512MB of RAM. The gateway was assembled on this host.

4. Web browsers installed on the Hamilton labs' computers (these were used to browse to the connected mobile web server).

6.2.1 Configurations

Before testing commenced the tools and the software installed on them needed to be configured. The Mobile connector which was part of the PAMP installation had to be configured to work with the gateway. By going to the menu and selecting the settings, the were set as follows:

- The identifier (the identifier of an account on the gateway): was set to *test*

- The password (the password of the gateway account being used for the test): was set to *ptest*
- The access point: was set to the local Hamilton labs access point.
- The Gateway address: was set to the IP address of mobile.ict.ru.ac.za (146.231.121.211)
- The gateway port: was set to 15001(the port where the gateway listens for the data and control data)
- Max connections (the maximum number of incoming connections that could communicate with the web server): 5
- Keep-alive interval (the time that should pass before the web server is considered orphaned if it does not receive a keep-alive message from the gateway): set to 0 which means it will be adjusted by the gateway connector dynamically.
- Keep-alive max latency (the time that should pass after a web server is orphaned and the when the mobile connector should stop waiting for the gateway to start sending keep-alive messages again): was set to 10 seconds (however this is ignored because the keep-alive interval was set to 0)

- Reconnect Interval (if the connection is broken and cannot be re-established immediately, the value settings specifies how many seconds the terminal should wait before attempting to re-connection): was set to 20.

- Server (the mobile web server host to which the connector should forward incoming data channels): set to 127.0.0.1 (since the mobile web server was on the local phone)

- Server port (the server port to which the connector should forward incoming data channels): was set to 80 (the port on which local mobile web servers listen for requests).

- On the Nokia Mobile Web Server, the password was set to *testwb* and the password to *testwbp*. This was the password used for logging onto the default web applications that come with the Nokia Mobile Web Server.

6.2.1 Tests and Results

The gateway was started by starting the Tomcat web server. The two connectors (containers in Tomcat) were started and they were waiting for connections. At this point, the mobile connector was started with the configurations explained in the configurations section. When this was done, the connector went into discovering state then it exchanged a handshake with the server. It was then declared online after this sequence of events. When the *netstat* command was run at the command prompt, the connection showed the results as shown in the extract below:

Proto	*Local Address*	*Foreign Address*	*State*
TCP	*mobile:3389*	*hons04.ict.ru.ac.za:3526*	*ESTABLISHED*
TCP	*mobile:15001*	*ict-coeph05.wlan.ru.ac.za:51422*	*ESTABLISHED*
TCP	*mobile:5555*	*ict-coeph05.wlan.ru.ac.za:4572*	*ESTABLISHED*

The Mobile Web Server was then started on port eighty of the mobile phone. The settings were set to run the web server on the local phone without any network connections. In this case the daemon (HTTPd) runs locally on port eighty and the mobile connector forwards the incoming HTTP requests to the mobile web server. To see the contents of the mobile web server the URL *http://mobile.ict.ru.ac.za/~test* was used by the five testing web browsers. This, in about five seconds and a half, displayed the web contents (hosted applications) served by the mobile web server. Going a level deeper the web application hosted on the phone required the visitors to log on to gain access to the features provided by the web application. The password *testwbp* was used with the username *testwb* to log into the web application and the results are shown in the Figure 25:

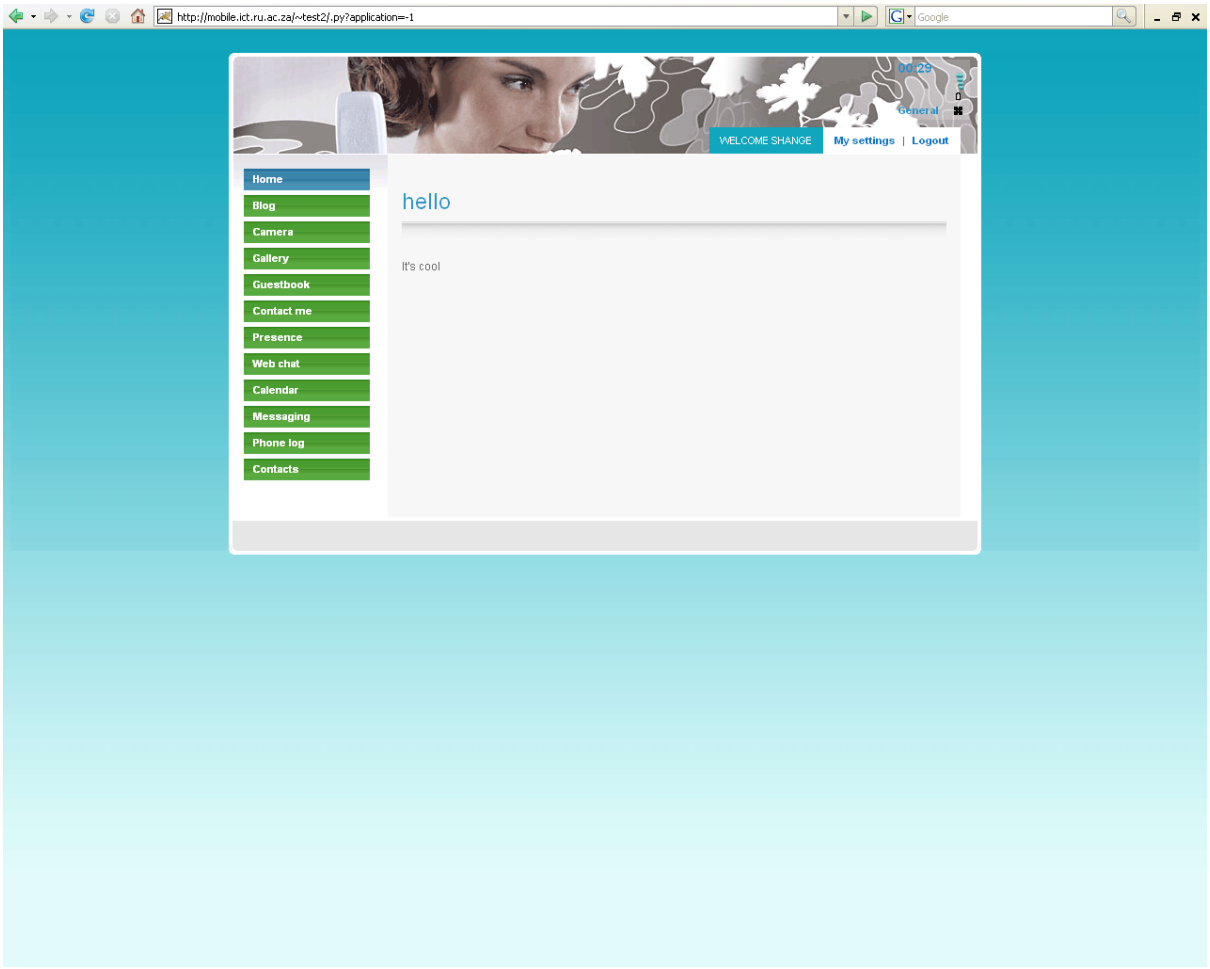

Figure 25: The Mobile Web Server's Main Web Application

Next the test efforts went to the *account* web application in an attempt to change the message and the picture for the offline message. The mobile web server and the connector were put offline and browsing to the URL again resulted in the offline page being displayed with the offline message and the picture that was setup for the user in the account web application.

6.2 Testing the Gateway on the Internet

6.2.1 The Tools

The gateway was made visible on the internet through the Rhodes university firewall. The only two ports permitted were the standard HTTP port eighty and the port for control and data channels (port 15001 for communication with mobile web servers). The tools used were the same as those used for the WLAN test except that the mobile phones were connected to the gateway from a GPRS network. This was the main purpose of the gateway. The MTN phone was using the MTN GPRS access point, the Cell-C phone on the Cell-C GPRS network. The account used was the test account mentioned earlier an a new test account with the name 'test2' an password 'ptest2'. The following extract shows the results of the *netstat* command that shows the hosts that are connected to the mobile host:

Proto	*Local Address*	*Foreign Address*	*State*
TCP	*mobile:15001*	*41.157.10.20:4569*	*ESTABLISHED*
TCP	*mobile:15001*	*41-208-11-176.mtnns.net*	*ESTABLISHED*

At he very bottom there are two TCP connections. The first TCP connection , 41.157.10.20, was from the Cell-C network. The bottom most one was from 41-208-11-176.mtnns.net on port 15001 this is the connection from the MTN GPRS network. By browsing to the *http://mobile.ict.ru.ac.za/mws-admin/onliners.jsf* the online users are shown as expected. This were 'test' and 'test2' as expected. This is shown in Figure 26.

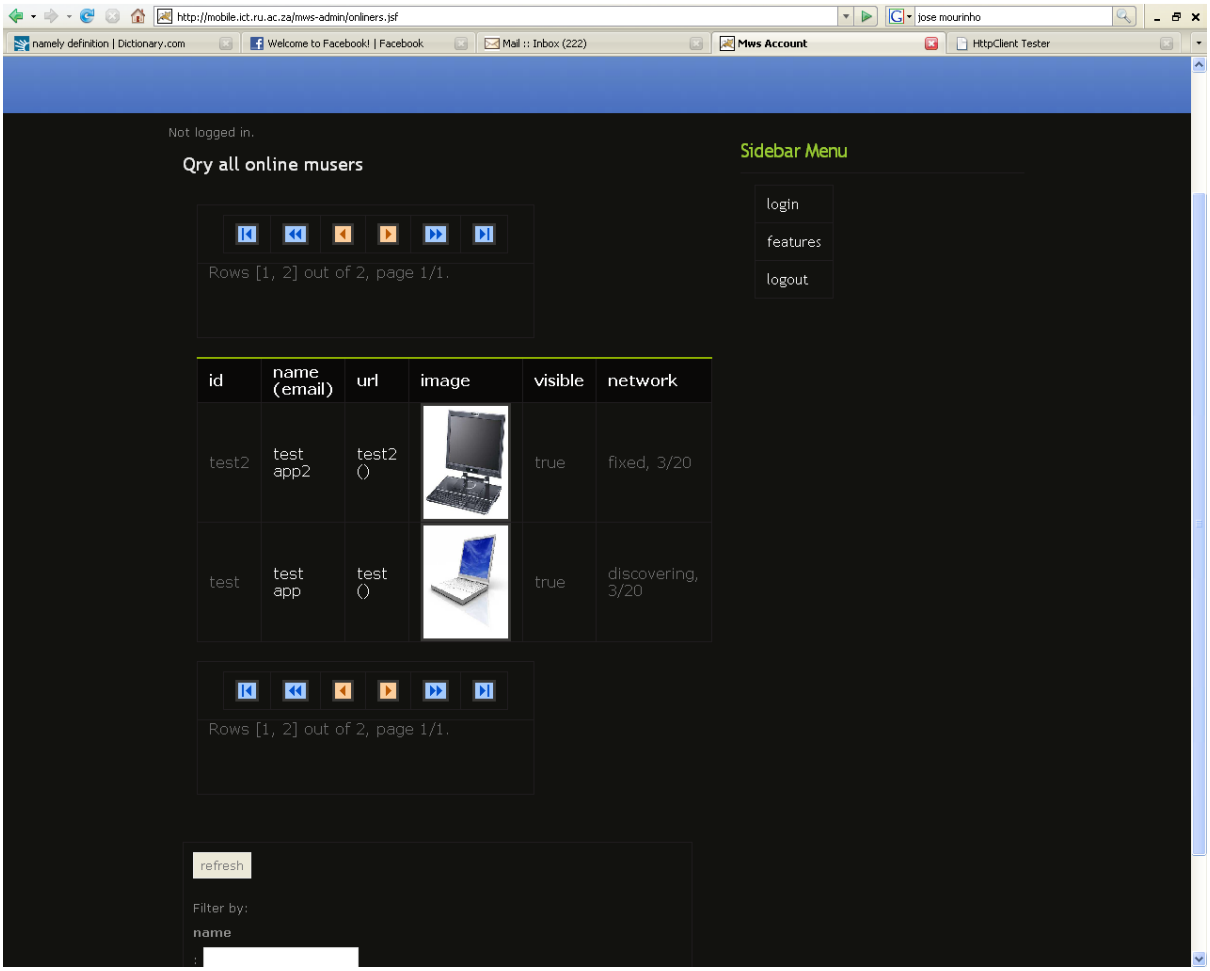

Figure 26: The Online Users (Testing)

By going to the URL *http://mobile.ict.ru.c.za/~test2* the web application in the diagram below was shown. The web applications on the two phones were identical and one could see the general outcome in Figure 27:

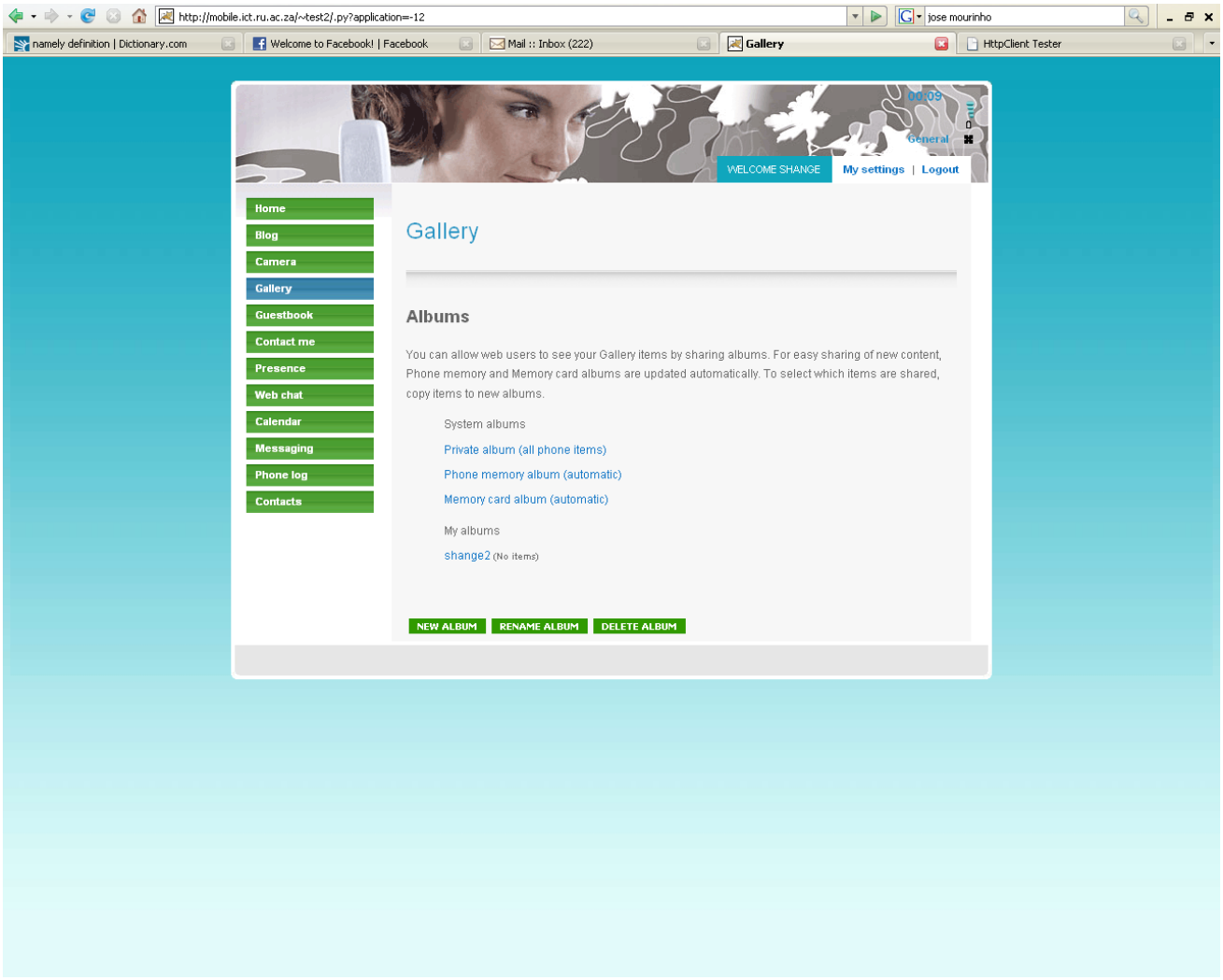

Figure 27: The Mobile Web Server Page

6.3 Chapter Summary

In conclusion, the author has discussed the process of testing the core functionality of the gateway. The gateway is tested with mobile web servers, first, within the local area network and later on the internet. The phones are accessed from the MTN and The Cell-C GPRS networks. The results prove that the gateway works as expected and the responses are acceptable.

Chapter 7

Conclusion and Future Work

This chapter concludes the project and discusses, briefly, extensions that could be added to the project in future.

7.1 Conclusion

Having web servers on mobile phones is a very exciting idea. Its implementation, however, is not so easy because of the firewalls that mobile operators use to protect their cellular networks. Also, the mobile phones do not have unique internet addresses with which to access them. A gateway host needs to service the people wishing to request services from the mobile web servers. The gateway host allows access to the web servers and gives them a means of being addressed by the clients wanting to use their services.

The aim of the project was to put together a gateway that provided a way for addressing and accessing Nokia Mobile Web Servers. The solution to the problem used other provided programs from Nokia (The Nokia Open Source Gateway) to achieve the most optimal results in the time frame given. The implementation of the project has shown that it is feasible to set up one's own gateway and it is highly recommended as it familiarizes one with the technology involved in implementing mobile web servers.

The gateway performs the same functions as those performed by the Nokia Mobile Web Server Gateway and with imagination one can make it even more attractive. To make it more social, the registered mobile users' access details were made public. Also an application for seeing other users on the map and displaying the messages they have left gives the gateway a unique dimension.

The gateway worked as expected and the objectives set initially have been met. Overall it was an exciting challenge to work with the concepts and ideas in this domain.

7.2 Future Work

The project should be extended to include more social networking into the world of mobile web servers and gateways. The users need to find out more about others and have access to information based on relationships established between users. Adding functionality that enables people to add others as friends and to see their profiles as web applications hosted on their phones will be a self –propagating idea. This will make the gateway more unique and more popular than the Nokia Gateway.

Chapter 8

References:

[1] Stefan Raab; Madhavi W. Chandra; Kent Leung; Fred Baker. *Mobile IP Technology and Applications*. Cisco Press, May 2005.

[2] Hauben, Michael. *History of ARPANET*. [on-line] Available: http://www.dei.isep.ipp.pt/~acc/docs/arpa--1.html. Last accessed: 27 June 2008

[3] Lillian Goleniewski; Kitty Wilson Jarrett . *Telecommunications Essentials, Second Edition: The Complete Global Source*. Addison Wesley Professionals, October 2006

[4] Service Stratergies Inc. *Most Popular and Widely Used WAP Servers.* [on-line] Available: http://www.ssimail.com/WAP_gateway.htm, November 2003 last accessed: 21 August 2008

[5] Matto Valtonen; Nokia Corporation. *The Story of WapIT* . . [on-line] Available: http://www.vesku.com/pdf/The_Wapit_Story.pdf Last accessed: 20 August 2008

[6] Lars Wirzenius (gateway Architect WapIt ltd.). *Kannel Architecture Design.* [on-line] Available:http://www.kannel.org/download/1.3.2/arch-1.3.2/arch.pdf last accessed: 24 August 2008

[7] Jani Ilkka; Carlo Vainio; Nokia. *Mobile Web Server Handbook*. Published by Nokia Corporation-2007.

[8] Apache Software Foundation. **HTTP Server** [on-line] Available: http://httpd.apache.org/. Last Accessed: 24 August 2008

[9] World Intellectual Property Organization. *(WO/2006/129182) SYSTEM AND METHOD FOR ACCESSING A WEB SERVER ON A DEVICE WITH A DYNAMIC IP-ADDRESS RESIDING A FIREWALL.* [on-line] Available: http://www.wipo.int/pctdb/en/wo.jsp?wo=2006129182&IA=WO2006129182&DISPLAY=DESC. Last Accessed: 16 May 2008

[10] Johan Wikman, Ferenc Dósa Rácz. *Providing HTTP Access to Web Servers Running on Mobile Phones*. Nokia Research Center article - May 24 2006 [on-line] Available: Last accessed: 27 June 2008

[11] Pasi Eronen, Nokia Research Centre. *TCP Wake-Up: Reducing Keep-Alive Traffic in Mobile IPv4 and IPsec NAT Traversal.* Nokia Research Center article – January 31 2008 Last accessed: 27 June 2008

[12] Ajit Jaokar; Tony Fish. *Mobile Web 2.0.* Published by FutureText Limited-London-15 August 2006

[13] Borko Furht - Ph.D., Mohammad Ilyas – Ph.D. *Wireless Internet handbook – Technologies Standards and Applications.* Published By CRC Press LLC - 2003

[14] Nancy J. Yeager; Robert E. McGrath. *Web Server Technology.* Published by Morgan Kaufman Publishers Inc. San Francisco California – 1996

[15] Subir Kumar Sarkar, T.G. Basavaraju. C. Puttamadapa. Ad hoc Mobile Networks: Principles, Protocols and Applications. Published by CRC Press - 2007

[16]Timo Halonen, Javier Romero, Juan Melero. *GSM, GPRS and EDGE Performance – Second Edition.* Published by John Wiley and Sons – 2003

[17] *James Goodwill, co-Founder of Virtuas Solutions, LLC. Embedding Tomcat Into Java Applications.* [on-line] Available: http://www.onjava.com/pub/a/onjava/2002/04/03/tomcat.html?page=1. Last Accessed: 20 October: 20 June 2008.

[18] Alex Hanik, Tomcat Committer / ASF member - Implemented NIO connector in Tomcat 6. Available: us.apachecon.com/us2007/downloads/ApacheConUS2007-**HackingTomcat**.ppt. Last Accessed: 20 June 2008.

[19] Tim Funk- Catalina Documenter. Apache Tomcat Configuration Reference: HTTP Connector. [on-line] Available: http://tomcat.apache.org/tomcat-6.0-doc/config/http.html

Last Accessed:18 June 2008.

[20] Tim Funk- Catalina Documenter. Apache Tomcat Configuration Reference: The Engine Container. [on-line] Available: http://tomcat.apache.org/tomcat-6.0-doc/config/engine.html. Last Accessed:18 June 2008.

[21] Tim Funk- Catalina Documenter. Apache Tomcat Configuration Reference: The Valve Component. [on-line] Available: http://tomcat.apache.org/tomcat-6.0-doc/config/valve.html . Last Accessed:18 June 2008.

[22] Johan Wikman, Ferenc Dósa Rácz, Installing The Mobile Web server Gateway, [on-line] Available: http://huono.info/gateway/installguide.html. Last Accessed:10 June 2008.