Master's Thesis

An Architecture for the Integration of Wireless Internet Services

Laura Carvajal, Erasmus

Department of Computer Science
Lund Institute of Technology
Lund University, 2004

ISSN 1650-2884
LU-CS-EX: 2004-16

An Architecture for the Integration of Wireless Internet Services

Laura Carvajal

Lund
June 27, 2004

Supervised by:

Lars Bendix
Academic Supervisor
Department of Computer Science

Lund Institute of Technology

Jens Jakobsen
Industrial Supervisor
Mondru AB

Ideon Science Park

Contents

1 Introduction **4**

2 Risk Assessment **6**
2.1 New Technologies . . . 6
2.2 Third Party Components . . . 6
2.3 Unrealistic Planning . . . 7

3 Analysis of the Requirements **8**
3.1 Requirements . . . 8
3.1.1 Requirements: “Musts” . . . 8
3.1.2 Requirements: “Shoulds” . . . 9
3.2 Use Cases . . . 10

4 Designing The Architecture **11**
4.1 Information Sharing . . . 11
4.2 Component Collaboration . . . 12
4.2.1 Direct Interaction with the Web Server . . . 12
4.2.2 Shared Database . . . 12
4.3 Back-up Strategies . . . 14
4.4 Redundancy and Load Sharing . . . 14
4.5 The Final Set-up . . . 15

5 System Functionalities **17**
5.1 The Prepaid Cards Process . . . 17
5.2 The SMS One-Time Password Process . . . 20
5.3 Authentication . . . 20
5.4 Accounting . . . 21
5.5 HotSpot Management . . . 22

6 Evaluation of the Prototype **23**
6.1 Evaluation of Scalability . . . 23
6.1.1 Storage . . . 23
6.2 Further Developments . . . 25

7 Conclusions **27**

Bibliography **28**

A Appendix A: Use Cases **29**
A.1 First Development Cycle . . . 29
A.1.1 Login . . . 29
A.1.2 Logout . . . 29
A.1.3 Authenticate User . . . 29

A.1.4 Register User . . . 29
A.1.5 View User List . . . 30
A.1.6 View User . . . 30
A.1.7 Modify User . . . 30
A.1.8 Change User Status . . . 30
A.2 Second Development Cycle . . . 31
A.2.1 View WISP List . . . 31
A.2.2 View WISP . . . 31
A.2.3 Modify WISP's Privileges . . . 31
A.2.4 Delete WISP . . . 31
A.2.5 Register WISP . . . 31
A.2.6 View WISP . . . 32
A.2.7 Modify WISP . . . 32
A.2.8 Register NAS . . . 32
A.2.9 View NAS List . . . 32
A.2.10 View NAS . . . 32
A.2.11 Modify NAS . . . 33
A.2.12 Delete NAS . . . 33
A.2.13 Register Administrator . . . 33
A.2.14 View Administrator List . . . 33
A.2.15 View Administrator . . . 33
A.2.16 Modify Administrator . . . 34
A.2.17 Delete Administrator . . . 34
A.2.18 Update User Activity Records . . . 34
A.2.19 View User Activity Records . . . 34
A.2.20 Register Operator . . . 34
A.2.21 View Operator List . . . 35
A.2.22 View Operator . . . 35
A.2.23 Modify Operator . . . 35
A.2.24 Delete Operator . . . 35
A.3 Third Development Cycle . . . 35
A.3.1 Activate Prepaid Card Bundle . . . 35
A.3.2 Generate Prepaid Card Bundle Information . . . 36
A.3.3 View Prepaid Card Bundle List . . . 36
A.3.4 View Prepaid Card Bundle Information . . . 36
A.3.5 Change Prepaid Card Bundle Status . . . 36
A.3.6 Change Prepaid Card Status . . . 36
A.3.7 Change Language . . . 37
A.3.8 Generate SMS one-time Password . . . 37
A.3.9 Change current Login Method . . . 37

Chapter 1

Introduction

Wireless Local Area Network (WLAN) standards such as IEEE 802.11b were originally developed for use within local area networks. Recently, however, WLAN has also gained large acceptance as a public Internet access technology. Locations that support public Internet access through WLAN are called HotSpots. Typical HotSpot locations include cafes, hotels, conference centers and airports [1]. By the end of 2003 there were approximately 40,000 HotSpots wold wide. The projection is that by the year 2007 the number of HotSpots will increase to 180,000 [2].

A basic HotSpot consists of the following components: one or many access points, an access point controller application, a radius [10] server to handle authentication and an internet connection. With this set-up any venture owner can provide its users with wireless internet access see Figure 1.1. However, at this point in time, this service is limited to authenticating users against a text file.

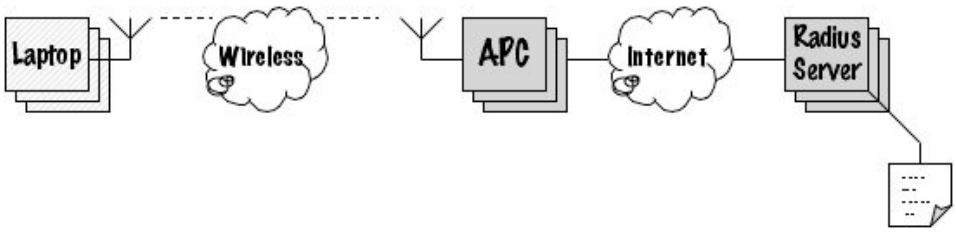

Figure 1.1: ***A basic HotSpot***

This project aims at studying the possibilities of broadening the spectrum of services that HotSpot owners can provide. Developed at Mondru AB, it has the main goal of determining the viability of a system architecture capable of the following interrelated goals:

- Integrating an existing access point controller application (APC) and a Radius Server to do user authentication and accounting using a Database.
- Providing HotSpot owners with management features by integrating a web server application to the set-up shown in figure 1.1
- Allowing security features like authentication using one-time passwords over SMS by integrating an SMS server (kannel).
- Providing HotSpot owners with the possibility to have pre-paid as well as post-paid users. Post-paid users are registered in the system and billed on a timely basis, and pre-paid users have access to the service with pre-paid cards. These cards are generated by the system for the HotSpot owner to sell to his users.

- Flexibility. The architecture needs to be flexible enough to easily adapt to the integration of new services in the future.
- Scalability. The architecture should provide for future development. This would allow the system to be used in an environment larger than that used to develop the prototype.

As with any project that ventures into new and unexplored territory there are many questions to be answered, the most important one being "can this be done?". We intend to answer this question by developing iteratively. In the initial cycle we will attempt to develop a prototype based on a simple architecture that carries out user authentication against a database. Then, two more iterations will be carried out to add on the rest of the functionality based on a more solid architecture.

Another important issue to address is the possibility of developing a commercial product based on this prototype. It is common for prototypes to be built in order to prove a theory and then be disregarded, building the final (commercial) system from scratch. In this project, however, it is of interest to use relevant parts of it in the final system. This emphasizes the importance of the last goal mentioned on the list above, the prototype must be *scalable*.

Due to time limitations and company policy we will use third-party components in the development of the architecture. This will add on certain restrictions on the design of the architecture. It will also add new risks to be considered and mitigated (see Chapter 2).

The proposed third party components are:

- ChilliSpot: The Access Point Controller Application. When a user attempts to connect through an access point in the HotSpot, ChilliSpot will capture the connection and ask for login and password. ChilliSpot transfers these to the radius server for authentication. If authorized, the user is allowed to connect to the internet. [4]
- FreeRadius: The RADIUS server. It is expected to authenticate users (pre-paid and post-paid) against the database, monitor user traffic and to forward accounting data. [5]
- Kannel: The SMS Server. It is expected to provide one-time passwords to requesting users (See Chapter 5). [6]

In the rest of this document we will address the risks involved in the development of this project and analyze the formal requirements and the functionalities expected of the system. Later we will study the different possible architectures and choose a one suitable for prototype development. Then we will describe the processes that carry out the most relevant system functionalities and explain their interrelations. Finally we present an evaluation of the prototype pointing out the possible bottlenecks and the recommendations for future development.

Chapter 2

Risk Assessment

Any development activity in a new project has a certain degree of uncertainty. It is this uncertainty that will lead to discovering new ideas and breaking down barriers into unknown territory. However, with uncertainty comes risk; no new project would be interesting enough if it didn't have an adequate degree of risk associated to it.

Since developing a project represents an investment of (usually limited) time and economic resources, risks need to be brought to a minimum. It might be possible to eliminate certain risks by changing the problem layout. However, when the risks cannot be avoided, mitigation plans must be specified in order to have some certainty that the project will be delivered. Worst-case scenarios must be considered and contingency plans must be elaborated for the most critical risks.

In this chapter we will analyze the risks associated with the development of this system. We will study the problems they may present and the impact they may have on both the development process and the system. We will also present mitigation and contingency plans for each risk.

2.1 New Technologies

Using the RADIUS protocol to authenticate users against a database is a pivotal requirement. However, establishing the communication between the radius server and the database is highly unexplored territory. This could cause delays on an optimistic project plan, as some time needs to be devoted to carrying out experiments and getting to know the tool and the protocol. The proposed tool for the radius server is FreeRadius.

Another area of development that may present similar problems is the SMS functionalities. Time should be set aside to study the SMS technology and to experiment with the communication protocols. The proposed SMS server is Kannel.

2.2 Third Party Components

Beta software may sometimes provide poor or incomplete documentation. They might also be unstable because they are evolving as products, but these instabilities will be transferred to the system. By using these tools we also run the risk of delaying the project, as additional time might be required to understand the tools and maybe even deploy them.

If the tool proves to be too unstable or impossible to figure out in the proposed time frame it should be discarded. In this case a new tool should be proposed as a replacement. This should be done in the first development cycle.

The latest versions of the FreeRadius server and ChilliSpot are both currently 0.9.3. Kannel is a stable release but the development activity on this application is somewhat erratic and uncertain [6].

2.3 Unrealistic Planning

Planning to develop system functionalities that might not be viable in the proposed time frame will delay the project. Another outcome could be having an incomplete product as of the final release. To mitigate this risk the project should be clearly defined in a realistic way during the early stages of development. Any low-priority functionalities should be left out, and picked-up for future versions of the software.

Chapter 3

Analysis of the Requirements

The first step when solving a problem is knowing exactly what the problem is. We have done that in Chapter 1. The second step is taking that information and expressing it in the form of *needs* or *requirements* expected to be fulfilled by any adequate solution [8].

In section 3.1 we translate the problem description into requirements. These express in a more formal way what the system is expected to do, removing any ambiguities and getting the most expressiveness out of the problem description. Then in section 3.2 we transform those requirements into actual functionalities that the system should perform. These will be the foundations on which the project is built.

3.1 Requirements

From a detailed problem description it is relatively simple to produce a list of system requirements. It's a process of iteratively brainstorming on the problem specifications until one gets a list of one-sentence requirements. The problem with this process is that it could go on forever if not monitored carefully. Both the development team and the project managers may be lured into adding too many requirements to the list, not considering the time and resources available [9]. So, knowing when to stop "wanting" functionalities from the system early in the game could prevent much frustration.

After brainstorming on the problem, and later pruning the results, the following requirements list was produced. It is divided into two sections: the *must haves* and the *should haves*, the latter being a shorter list of requirements that will probably not fully make it into development, but that could not be pruned away.

3.1.1 Requirements: "Musts"

1. Management Requirements
   (a) Allow the HotSpot owner to register his HotSpot in the system
   (b) Keep permanent record of all HotSpots
   (c) Allow the system operator to manage HotSpot permissions
   (d) Support HotSpot administrators [1] with different permissions
   (e) Keep permanent record of all administrators
   (f) Allow an administrator to register and modify a user
   (g) Keep permanent record of users and user activity
   (h) Allow an administrator to suspend a user in his HotSpot

[1] A HotSpot may have one to many administrators. The HotSpot owner is considered an administrator.

(i) Allow an administrator to consult user activity records
(j) Support multiple languages at all levels
(k) Provide login and logout capabilities
(l) Support several system operators with different permissions
(m) Keep permanent record of all operators
(n) Delete all data of former (deleted) users after n years
(o) Allow an administrator to register his HotSpot's NASes
(p) Keep permanent record of all NASes
(q) Keep all accounting data even after HotSpot deletion

2. Prepaid Cards Requirements

(a) Allow the operator generate the data to be printed on prepaid cards
(b) Save that data to a comma-separated file
(c) Keep permanent record of all generated prepaid cards
(d) Allow the system operator to view the info of a PPC bundle
(e) Allow a HotSpot administrator to activate a PPC bundle
(f) Allow the system operator to deactivate a lost/stolen PPC Bundle
(g) Render all prepaid-cards invalid after usage

3. SMS Requirements

(a) Accept a user request for an SMS one-time pasword (OTP)
(b) Store a newly generated OTP
(c) Allow a HotSpot administrator to turn on/off OTP authentication

4. Authentication Requirements

(a) Authenticate a user through the RADIUS protocol
(b) Disconnect user after time/volume on his prepaid card have expired
(c) Periodically update user activity records

3.1.2 Requirements: "Shoulds"

- Additional Requirements
  - Allow the user to consult his accounting data
  - Allow the user to change his password
  - Offer support for timezone management
  - Support MAC and SIM authentication methods
  - Support radius roaming between HotSpots
  - Allow for database redundancy for availability
  - Support network management for the system operator
  - Allow for a reseller to take over the system
  - Allow the reseller to use his own interface

3.2 Use Cases

Once a stable and viable requirements list was produced we could move on to define the use cases. These will be the basis of all further design of the system, since they draw out what the system should do in terms *user functionalities*. All use cases map to at least one system requirement and vice-versa. Below is a general list of the use cases. The fully detailed list of use cases can be found in Appendix A.

- **Management** These use cases include user, NAS, administrator and operator management. The term "management" refers to the use cases that have to do with viewing, removing, adding, modifying, suspending and deleting entities. For example, the use case"Modify User" would fall in this category.
- **Authentication** Use cases that involve authenticating users and choosing the authentication method (i.e. SMS one-time passwords)
- **Prepaid Cards** These use cases include generating the prepaid card codes, bundling the cards together, activating the bundles and inactivating individual (used up) cards and lost/stolen bundles.
- **Accounting** These are the use cases related to keeping track of user activity. Also in this category are use cases that involve generating statistics based on the user activity records and downloading them for billing purposes.

Chapter 4

Designing The Architecture

In chapter 1 we presented the overall need for an authentication service. In chapter 3.1 we described the basic requirements the system needs to fulfill. In this chapter we present a more detailed study of the different architecture set-ups by which such a system could be built.

As we mentioned in the introduction, this system is to be developed using some third-party components to cover some of the functionalities. There are two reasons for doing this instead of developing all the components from scratch. First, the development effort (and time) should be put mostly into creating the web server, as it is there where most of the core functionalities are. Second, there are several radius servers and SMS servers available in the market, so the key is finding the ones most suited for this project. FreeRadius [5] will be used as the radius server and Kannel [6] will be used as the SMS server as they are open source and both have the functionalities this system will need from them.

Another third-party application used in development is PostgreSQL [7]. There are many database managers out there. Out of the ones that have a free license, PostgreSQL is the only one that can guarantee data consistency after failure [7].

In the following sections we will look at the pros and cons of all the architecture possibilities, considering these third-party components. Then we will distinguish between different architectural choices and component redundancy. Finally, we will choose a set-up for development of the prototype and contrast it with the "ideal" set-up for product development.

4.1 Information Sharing

In the existing set up, the radius server uses a text file to authenticate users (see Figure 1.1). Therefore, one obvious solution would be to create a web server aplication that uses this file to read/write user data (see Figure 4.1). There are two advantages to this approach: one is that the current set-up woud not need to be modifyied as the radius server would still authenticate users agianst a text file, regardless of any other components modifying that data. Another advantage is that implementation is easy and quicker than designing a database.

One clear deisadvantage of this architecture is that it's not scalable. It would work for a small-scale prototype but it will have to be completely discarded when creating the commercial system. Since one of the requirements of this prototype is to set the basis of an architecture that can later be expanded to be used in the final product, this approach is not viable. The final system is supposed to suport hundreds and even thousands of simultaneous users, and a single text file would not support that kind of access.

We can conclude that the most efective way to share information in this kind of scenario is by means of a database, as it allows rapid access to large ammounts of data. In the next section we will analyze the various ways in which components can collaborate to efficiently share a database.

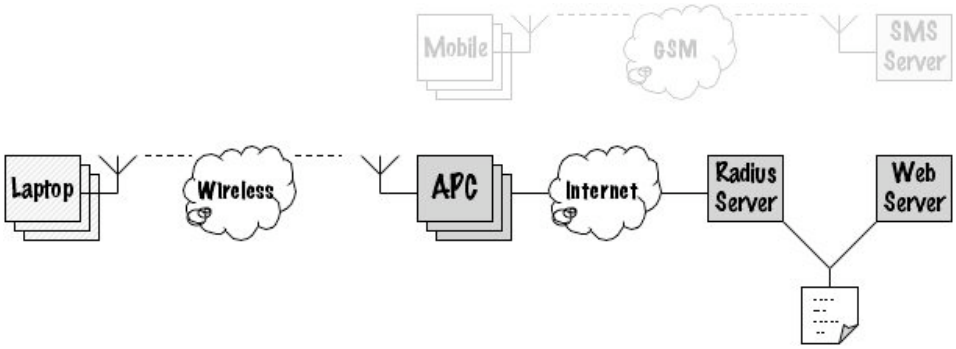

Figure 4.1: ***Architecture*** *The components share information through existing text file.*

4.2 Component Collaboration

We have determined that the information to be shared must be stored on a database and that this database must be accessible to all components of the system. There are several ways of accomplishing this. In this section we will study two possibilities in which the Radius Server, the Web Server and the SMS Server can share the database.

4.2.1 Direct Interaction with the Web Server

One way of providing access to the database to both the radius server and the web server is the following: connecting the web server directly to the database and establishing a direct line of communication between the web server and the radius server. This set-up is shown in figure 4.2. This way, all the radius server requests and replies would go through the web server, and it would be the web server who queries the database.

The case of the SMS server is analogous. It would communicate directly with the web server, and the latter would interact with the database.

The main advantage of this set-up is that most SMS servers come with capabilities for communicating with a web server. Kannel provides this after some extra configuration.

The advantage of this set-up is that access to the database is restricted to only one system component which would enhance security and data consistency. The draw back is that there is the added requirement of implementing an API that would establish the communication between these two components.

4.2.2 Shared Database

In this set-up the database is accessed by both the radius server and the web server, never communicating directly with each other. For example, the web server would add user data to the database, and the radius server would use that information to authenticate them. Also the radius server would add accounting data to the database and the web server would interpret it and display it to HotSpot owners.

This set-up (Figure 4.3) is possible when the selected radius server is capable of communicating directly with a database. In the case of freeradius this is possible after some additional configuration. So, one drawback of this implementation would be that if the radius server were to be changed in the future, it would need to be replaced by one with database support.

In the case of the SMS server this is not so simple. The database communication capabilities of the chosen SMS server is not largely developed. Unfortunatelly, turning that into something that can fulfill the system requirements would fall out of the scope of this project.

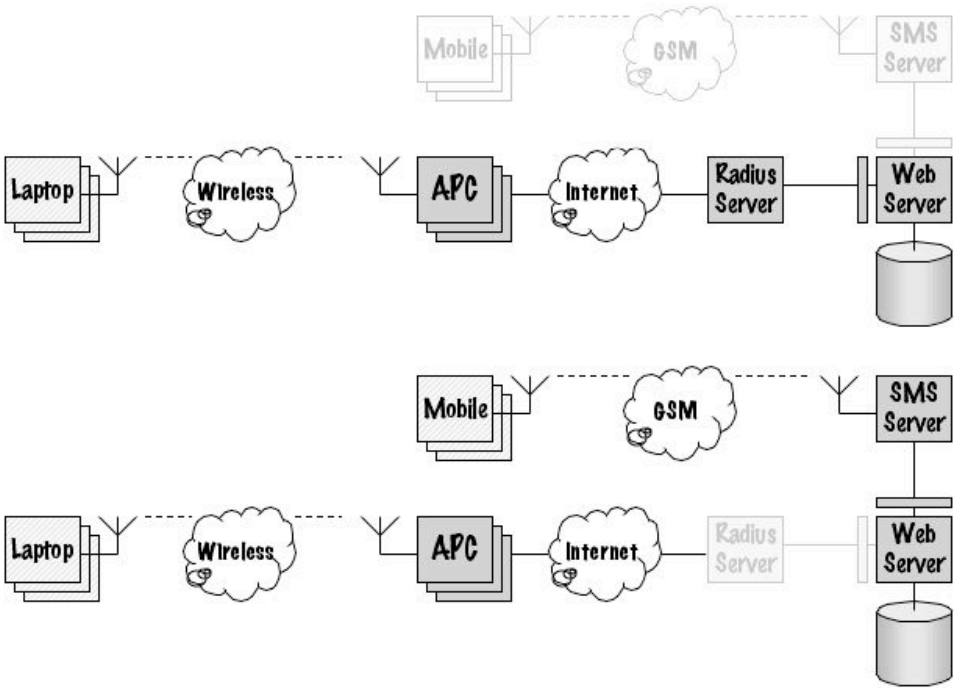

Figure 4.2: ***Architecture*** *Direct communication between the Radius Server and the Web Server (top) and between the SMS Server and the Web Server (bottom). The database is only accessed by the Web Server.*

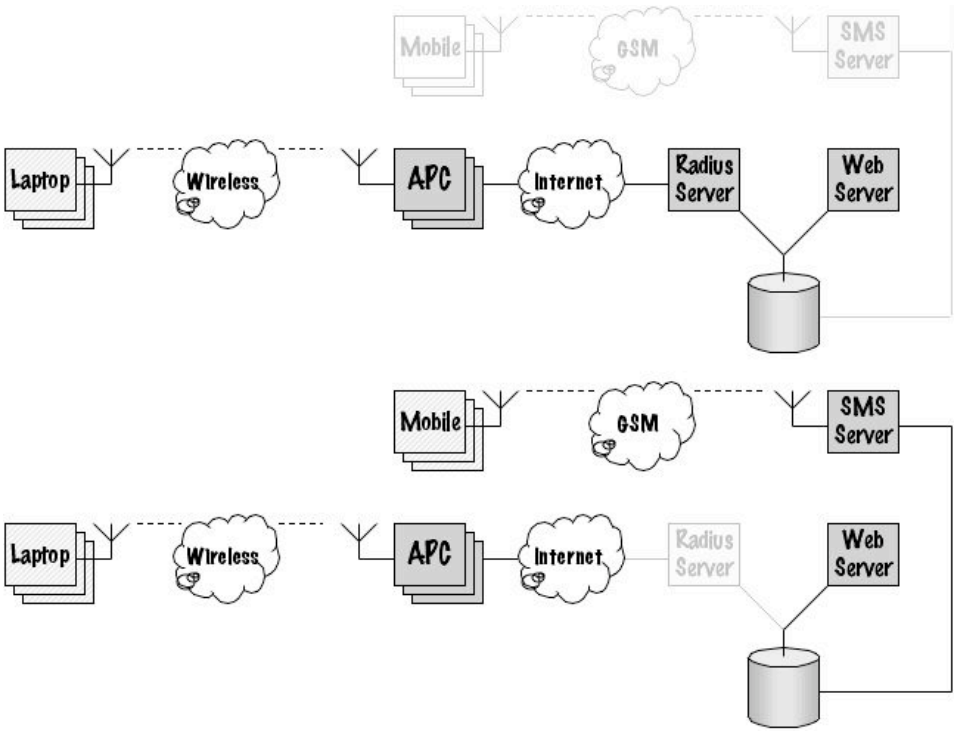

Figure 4.3: ***Architecture*** *The Radius Server and the Web Server share a database.*

When two components share a database, they must "agree" on the way that data is structured and organized. Each component may have certain requirements on the data that's shared and on the way it's stored. This can be a disadvantage, because it would require careful designing of the database so it can be used consistently by both components.

4.3 Back-up Strategies

Back-up strategies are ways of increasing the availability of the system. If one component fails, others could still work to some extent for a limited period of time.

In any set up all components are necessary and the system will only work partly, if at all, with any of them missing. The only case in which the system would come to a complete halt is when the database goes down. It would be very desirable to provide a strategy to cope with this scenario. In section 4.4 we present the possibility of having several redundant databases. A simpler and more cost-effective solution would be having a "buffer" on the side of the Radius Server and Web Server, with small amounts of *relevant* data from the database. This way, at least some activities could be performed by the servers while the database is brought back up. This set-up is depicted in figure 4.4.

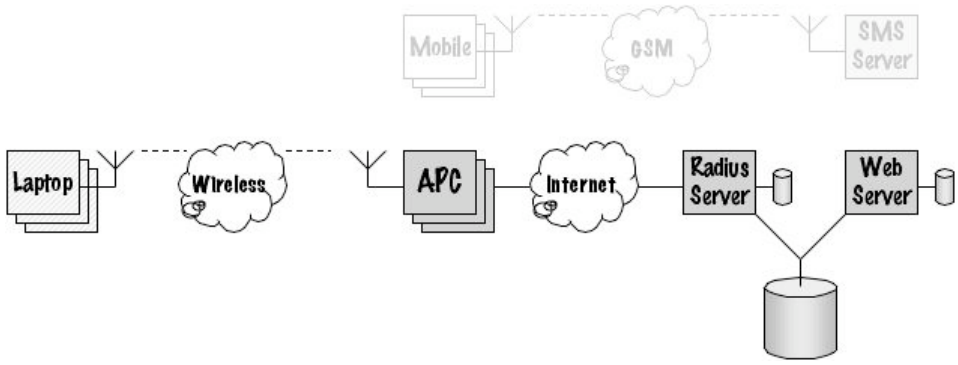

Figure 4.4: ***Architecture*** *Data "buffers" on the Web Server and the Radius Server.*

This set-up would require careful planning on exactly what data is to be kept in those buffers, and how to synchronize it with the actual database when it is is brought back up.

The case of a buffer on the radius server is perhaps the most important, as it would allow the authentication process to work to some extent. Users could still log in, and the radius server would authenticate them against the data in the buffer. This data however, is limited as it cannot be an exact replica of the database contents. Through a carefully designed algorithm, this buffer should contain only the data of those users that are "most likely" to connect when the database goes down. This is obviously impossible to guess, but an algorithm that comes close could be devised.

4.4 Redundancy and Load Sharing

As we have seen, system availability depends highly on back-up components: if one component fails, another, up-to-date, component of the same kind is available for replacing it. This, however, comes at a cost. Keeping several components of the same kind (i.e. several web servers) up-to-date requires double (triple or n-ary) maintenance. If the software is updated in one of the servers, it needs to be updated in the rest.

The issue of load sharing is also important in large-size systems. In this case redundancy would serve not only as a means of guaranteeing availability, but also could be used for splitting the workload. For example, the system could be set up to have 5 radius servers.

This would guarantee a much higher availability than just one server would, because if one fails, the next available one will replace it, and so on. However, several radius servers could also work *simultaneously*, so if there are ten user requests at one time, they could each tend to two of them, making authentication faster. So, combining these two possibilities the system would be not only available but also faster. An analogous approach can be taken with the web server and the SMS server.

There is one component for which it is a bit trickier to provide redundancy: the database. Double maintenance of redundant databases would not be as simple as multiple software updates as in the case of the servers. It would require updating the back-up constantly: for every modification in the original database, the copies should also be updated. This would guarantee perfectly mirroring databases, but as the number of back-ups grow, the cost of each operation grows as well. Another option would be to have back-up databases that are only updated a few times a day, instead of on every query. This would make back-ups lest costly, but they won't be entirely up-to-date all the time, incurring in some data loss if they were to replace the original one. So, some careful analysis would have to be carried out in terms of how often to update the back-ups and how many of them to have.

Load sharing can be even trickier when it comes to databases. It would imply partitioning the data and placing each partition on a separate disk. This partitioning however needs to be done carefully and the dependencies within the tables carefully observed. Partitioning is only possible if there are two or more sets of relations that are independent of each other. If it's possible to partition, then some queries would go to one disk and the rest would go to another (ideally on separate database servers) improving system performance.

4.5 The Final Set-up

Now that we have analyzed all the possible set-ups and their combinations, we can come up with an "ideal" architecture. This is shown in figure 4.5. An ideal set-up is what the architecture of a fully implemented commercial product should look like. However, for the prototype the chosen architecture is simpler due to time and resources restrictions.

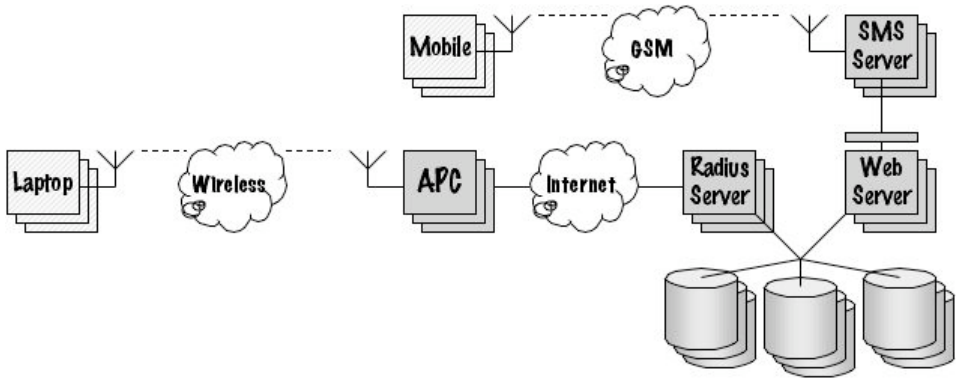

Figure 4.5: ***Ideal Architecture*** *The Radius Server and the Web Server share a database. The SMS Server communicates directly with the Web Server. There is redundancy in all of the components (no back-up strategies needed). The database is distributed.*

Designing the ideal architecture is easy, it's simply done by choosing "the best of everything". This ideal architecture would consist of redundant radius servers that share the load of requests and also provide back-up when one of them fails. Also, it would have redundant web servers and SMS servers for the same purpose. The database is shared between the web server and the radius server and the SMS server is directly linked with the web server. In the ideal set-up, the database would also be distributed.

After knowing what the ideal architecture should look like, and being aware of the fact that such an architecture is what the final product should evolve to, we have come up with an architecture for the prototype. This architecture should be simple enough to experiment with in the limited time available, but also robust enough to eventually evolve to the ideal. For this reason we have chosen the set-up shown in figure 4.6):

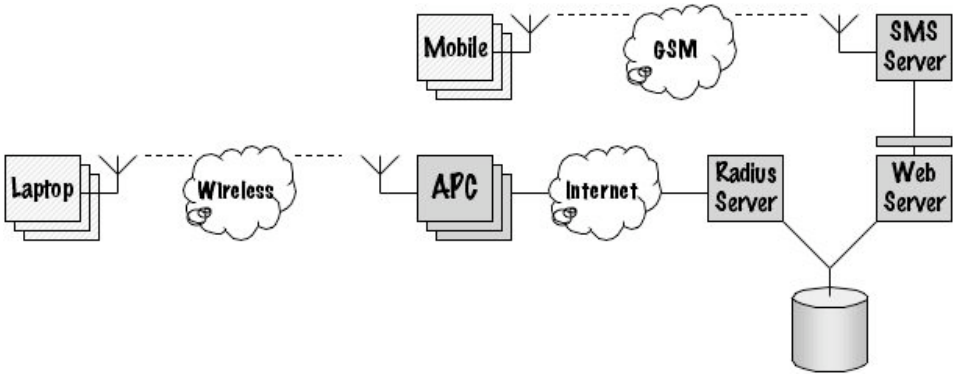

Figure 4.6: ***Prototype Architecture*** *The Radius Server and the Web Server share a database. The SMS Server communicates directly with the Web Server. There is no component redundancy and no back-up strategies. The database is centralized.*

There is no redundancy in any of the servers. While redundancy is important for availability, it can be easily incorporated later and is not vital at the prototype level. The radius server and the web server share a database, and the sms server communicates directly with the web server, which would work in the commercial product. Finally, there is a single database in the prototype, as time would not permit experimenting with redundancy or distribution. Nevertheless it would still be possible to add these two characteristics after some further development.

Chapter 5

System Functionalities

As we saw in the first chapters of this work, this system is made up of a wide variety of functionalities that allows the HotSpot owner to have flexible control over the service he provides. In this chapter we will go over the most relevant and complex functionalities and discuss how they were designed, why, and what other possibilities exist.

We will see how the different functionalities are interrelated, how they complement each other and how sometimes they may restrict each other and possibly future additions. We will look at the collaboration processes that were studied to design the Prepaid Cards and the SMS one-time passwords processes. Then will describe how authentication and accounting are carried out and why. Finally we will take a brief look at HotSpot management.

5.1 The Prepaid Cards Process

As we have seen there are two types of users: pre-paid and post-paid. The latter are registered in the system and are billed according to the HotSpot owner policies (i.e. monthly). On the other hand, pre-paid users are not registered in the system, they only "exist" to the HotSpot owner when they buy a pre-paid card and use it to gain access to the HotSpot for a limited amount of time (or traffic).

The Prepaid Card Process thus covers every system functionality from the moment the prepaid cards are created until the end user buys one to connect to the internet. At this point the process is immediately followed by Authentication (see Section 5.3)

The prepaid cards process implemented for this prototype is described by the collaboration diagram shown in Figure 5.4. Figures 5.1, 5.2, 5.3 show alternative collaboration models.

In the prototype the username-password combinations are generated by the system because the printing facility chosen by the company requires such a set-up, as they don't generate codes themselves. The bundles are activated [1] by the HotSpot owner instead of by the system operator for security reasons: the system establishes the bundle-HotSpot relation *only* in the moment they receive the cards. If the bundle is lost on the way, the system operator can simply render it invalid it in the system, after verifying it has never been activated by the HotSpot. On the other hand, if the bundle were activated by the system operator prior to delivery (meaning the cards in it are "live" and ready to be used before they reach the HotSpot), and later the HotSpot owner claims never receiving it the system operator would have no way of corroborating it.

The set-up shown in figure 5.3 although ideal, was not incorporated into the prototype due to time constraints.

[1] An bundle with status = active indicates that its cards can be used. Prior to activation, any login atempt will be rejected.

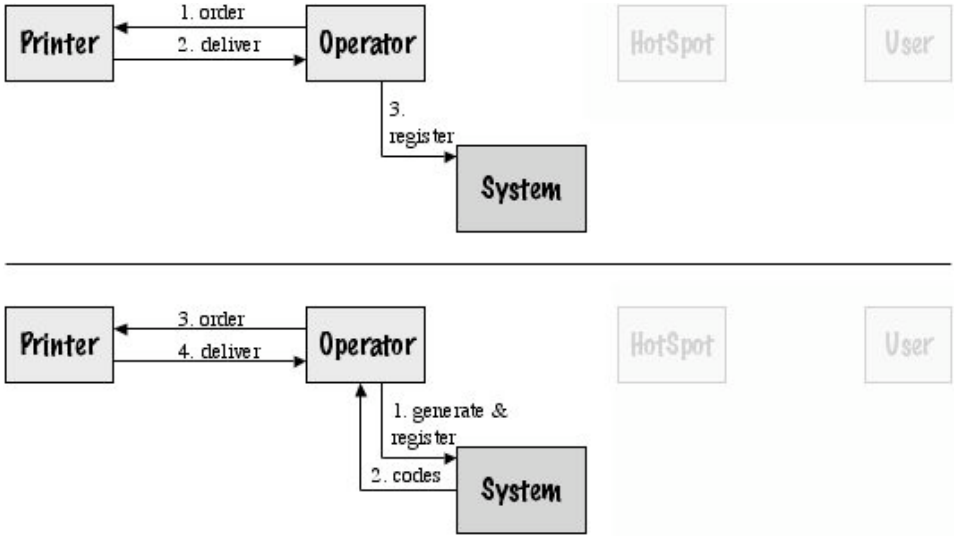

Figure 5.1: ***Printing Facilities*** *There are two types of printing facilities: those that generate the codes that go on the cards, and those that don't. The topmost figure shows the first case, where the system operator only orders the cards (1) and the printing facility generates them. Then the operator receives them (2) and registers them in the system (3). In the second case (bottom) the operator first generates the codes and registers them on the system (1) and the system returns the generated codes via a text file (2). Then the operator sends the codes to the printer (3). The printer then sends the cards to the operator (4). The cards are already registered.*

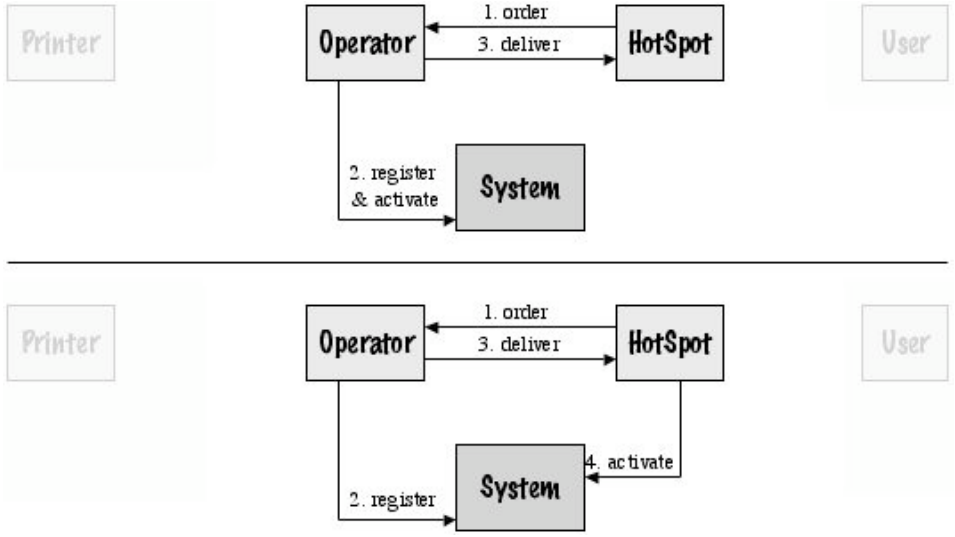

Figure 5.2: ***Prepaid Card Bundle Activation*** *Only when a bundle has been activated, the cards in it can be used. A bundle can be activated by the system operator before it is sent to the HotSpot that requested it (top), or it can be activated upon arrival to the HotSpot (bottom).*

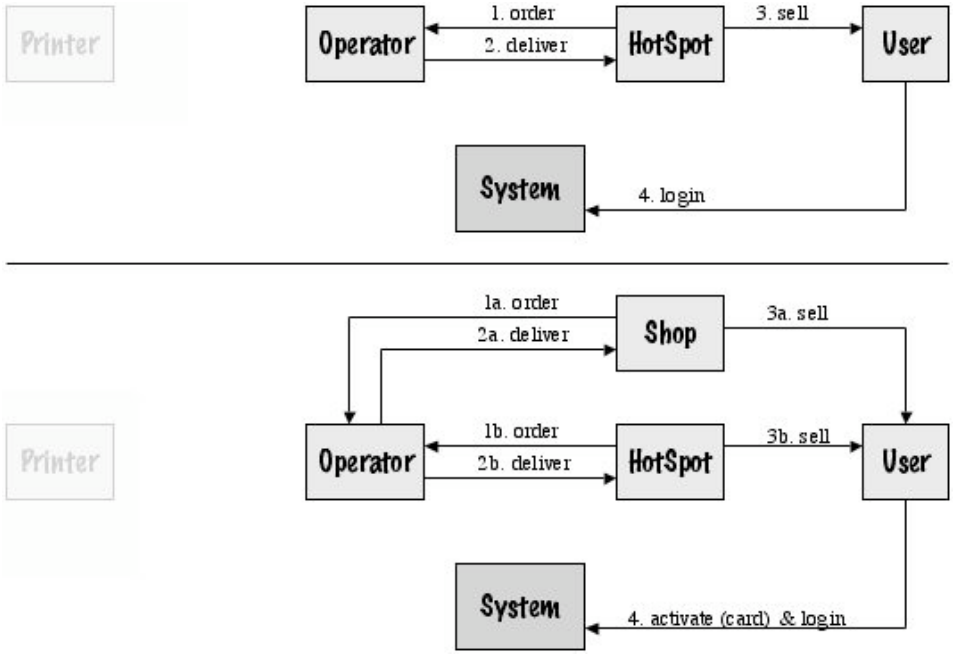

Figure 5.3: ***Reseller Shops*** *Instead (or in addition) to selling Prepaid Card Bundles to each individual HotSpot (top), they can also be sold to shops (bottom). These would sell them directly to users (3). The user then can choose which HotSpot to login from (4), or if roaming agreements exist between HotSpots, he could login at more than one with the same card (4).*

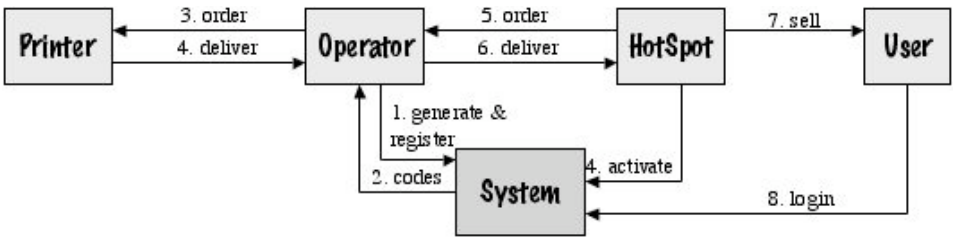

Figure 5.4: ***The Prepaid Card Process for the Prototype***

5.2 The SMS One-Time Password Process

Certain HotSpot owners may have a need of providing additional security to users regarding their passwords. Username-password authentication is a standard way of verifying someone's identity: a username is *unique* and it unequivocally identifies a user. A a password is *secret*, meaning only the owner knows it... ideally. In reality some users write passwords down, others choose easy-to-guess passwords, etc. This password vulnerability can be solved by changing the user's password frequently, as long as the system doesn't rely on the user having a good memory to remember each new one. The approach taken in this project is providing the user with a new password through SMS *every time they log in*. This way the user would not have to remember it, he would just need to type it in once as soon as he receives the SMS. One added advantage of this system is that the user is also being identified by his mobile number, enhancing security.

The scope of the SMS OTP process goes from the moment the HotSpot owner activates one-time password (OTP) authentication for a user and it ends when the user receives that OTP and uses it to *authenticate*. This process is immediately followed by Authentication (see Section 5.3).

Two possible set-ups for the SMS OTP process are shown in figure 5.5.

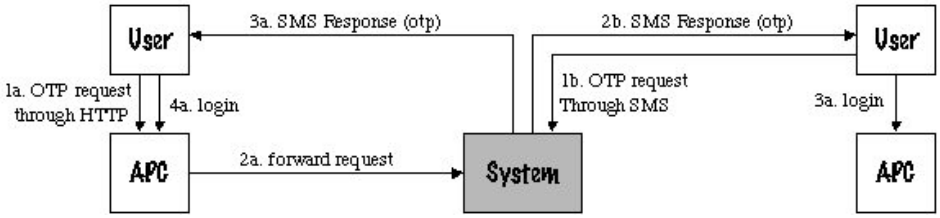

Figure 5.5: ***SMS one-time passwords****OTPs requested through HTML (left): the user enters the login page presented by the APC and clicks on the "Send OTP" button. This HTTP request is then forwarded to the system. The system then sends the one-time password to the user's mobile phone and he is able to log in. OTPs requested through SMS (right): the user sends an SMS to the system requesting a one-time password. The system responds with another SMS, containing the user's new OTP. The user logs in.*

When comparing these two possibilities it is obvious that both of them are viable. Nevertheless, the option shown in figure 5.5 would seem more natural to the user: clicking on a button in the HTML interface produces an SMS where he gets his new OTP. It would also be more cost efficient than the option shown in figure 5.5 as the user pays nothing to receive the OTP. However, sending a request through one channel (HTTP) and receiving an OTP through another (SMS) is a patented process which may include requesting permission to use the patented process [3]. This represents an added complexity on the business process. Thus, for the prototype, the alternative shown in figure 5.5 was chosen (request and reply are sent over the same channel: SMS).

5.3 Authentication

The process of authentication covers all the system events that occur from the moment the user enters his username and password in his laptop (at the hotspot). The process ends on the moment the user is allowed, or denied, to connect to the internet.

Regardless of the type of user, the authentication always is reduced to matching a username against a password. This is true not only for post-paid users (even those with one-time passwords) but also for pre-paid users, who authenticate with the username and password on their pre-paid cards.

The purpose of a username is to identify the user within the system, like a social security number identifies a person within a country. This would be very straight-forward to implement but, user satisfaction would be sacrificed. To illustrate this with an example imagine two HotSpots from the many that may be subscribed to the system, hotspotA and hotspotB. HotspotA registers a user with username `johan_ericsson`. HotspotB, perfectly unaware of the existence of hotspotA, tries to register his own `johan_ericsson`, but he finds out that the username is already taken. This makes sense, as that name already identifies the first Johan Ericsson within the system, but it will unacceptable to HotSpot owners and their users. One way out of this problem would be to somehow make the system identify a user not only by his name, but also by the name of his HotSpot.

One alternative that was studied is to identify users by usernames of the type "username@hotspotname". This way usernames are unique within the HotSpot, as well as within the system, but they can be seemingly repeated throughout HotSpots, allowing the two Johan Ericssons to register under the same username: `johan_ericsson`@hotspotA and `johan_ericsson`@hotspotB. This alternative, although better, may prove to be uncomfortable for users, who would need to type additional characters every time they log in.

Another alternative is to still authenticate users by their username and HotSpot name, but making it transparent to the user. Users would only need to type in their username, and the system would look at the name of the NAS they are login in in from, and then search the database for a match with a HotSpot name. This way the user is still authenticated by username and HotSpot name, but he does not need to provide the additional information, the system gathers it itself. This is the set-up used in the prototype.

5.4 Accounting

Accounting is the process of keeping record of user activity. This includes connection and disconnection times, session length, amount of data transmitted, etc. Following authentication, this process starts on the moment the user gains access to the HotSpot, and is connected to the internet. The process ends when the user is disconnected and is closely interrelated with HotSpot Management (see Section 5.5) where the HotSpot owner can have access to that data for billing and statistical purposes.

Accounting is carried out similarly for all types of users (i.e. pre-paid, post-paid). When the user connects and disconnects, the start and stop times of his connection are recorded, as well as the ammount of data transmitted in that period. This allows for the HotSpot owner to subtract the two and know how long he was connected for, for billing purposes. This approach, though very sound, is not enough. There is a problem with the so-called "long-lived connections", when the user connects and doesn't disconnect for a long period of time. If we were to count only on start and stop records these type of connections would never be accounted for, and therefor never billed. So another type of record needs to be kept, and it is called Interim Record. When the user connects, apart from recording the start time, the system sets an interval at which updates will be sent from the NAS to the radius server. This way, all connections would leave records consisting of one start time, several update times (and the traffic up to those times), and a stop time if there is one. This would allow the HotSpot owner to bill correctly, even if the user never logs off.

One problem with interim accounting is that it generates loads of additional traffic. Upon every update, a packet is sent with interim information. The more packets the more traffic, and excess traffic may cause the system to collapse. Thus, the interval at which this packages are sent should be set carefully: often enough to provide for correct billing but not to often, to prevent excess traffic from locking the system.

5.5 HotSpot Management

HotSpot Management covers all the functionalities to which the HotSpot owner has access through the web server. Processes like adding new users and registering the HotSpot's NASes are covered. Also accessing the data gathered during the accounting process and viewing statistics of user activity and enabling SMS OTP authentication and managing the prepaid cards.

As we can see, this part of the system is largely interrelated with the rest of the processes, as it allows the HotSpot owner to control and configure them. For example, the HotSpot owner can download a file with detailed activity records for a certain month. He can also view the peak hour of his HotSpot and the average number of daily connections through the graphical interface.

Chapter 6

Evaluation of the Prototype

As we saw on chapter 4 there were many architectural options for building the system. The one chosen for the prototype was simple enough given time frame. At the same time it contained all the necessary components to easily expand it at a later stage.

In this chapter we will evaluate the architecture of the prototype. Also, we will assess its performance in the prototype environment and make a projection on the commercial environment.

We will take a look at the areas of the prototype that would need expanding/replacing to reach the standards of a commercial product. More importantly we will roughly calculate *when* this need will arise. We will answer questions like "How many users can the system support before we need to add a second Radius Server?" and "How easy *is* it to integrate a new Radius Server?" or "How well will this algorithm perform when we get twice as much traffic?" by doing load analysis to identify the potential bottlenecks.

6.1 Evaluation of Scalability

6.1.1 Storage

Every HotSpot that is registered in the system implies several entries in the database: one for the hotspot itself, one for the HotSpot owner and one for every additional administrator and NAS registered. Further on, there will be entries for prepaid cards, one-time passwords, etc. Finally, every user that the HotSpot adds to the system generates several additional entries. As we can see, storage space can eventually become an issue to consider. We will try to calculate the size of the database in different environments, to identify the possible "bottlenecks" in this area.

The following list shows the different types of entries in the database and an estimation of amount of disk space they consume:

- WISP: The relation representing the HotSpot. Storage space: 25 bytes
- Admin: The HotSpot owner or administrator. There may be several of these for each HotSpot. The storage space required for one administrator is 75 bytes
- NAS: Information about the computer holding the access point controller aplication. There may be several of these, one occupies 63 bytes
- User: A registered HotSpot user. 215 bytes
- Operator. The system operator. There may be one to several for the whole system. One occupies 60 bytes.
- Operation: Defines the possible operations in the system, i.e. "Register User". Size: 30 bytes

- Administrator Privilege: A relation between an operation and an administrator. It defines which operations different administrators can perform. Each privilege takes up 30 bytes.
- WISP Privilege: 30 bytes.
- Operator Privilege: 30 bytes.
- User Activity Record. Each record can indicate the start or stop time of a user's connection, as well as intermediate times. Each contains information about data transfer, IP address, etc. Size: 290 bytes.
- Monthly Activity. This table contains a maximum of 36 rows, each representing a month in the past 3 years. It contains information on the average monthly user activity. Each record is 51 bytes.
- Hourly Activity. This table contains a maximum of 31x24 entries, one for each hour of the day of the past month. It contains information on the average hourly user activity. Each record is 54 bytes.
- One-time password. Relation between a one-time password and a user's mobile. 38 bytes.
- Prepaid Card: Contains the username and password of a prepaid card as well as information relating it to a certain HotSpot and user. 47 bytes.
- Prepaid card bundle: A group of prepaid cards. 31 bytes.

Type sizes are defined in the PostgreSQL documentation [7]. Calculations of storage sizes were done by assuming mid-range values for strings (i.e. if a string is allowed a max length between 1 and 30 characters, 15 is assumed). It is important to note that these calculations are not exact due to limited resources, and further, more detailed studies are recommended. However, with this information we can roughly approximate the storage space required by different kinds of HotSpots.

The Small HotSpot One NAS, one administrator and five users that connect an average of 3 times a day for two hours every day. No one-time passwords. This kind of HotSpot would occupy 434 KB after one month of using the system. After one year it would take up 4.8 MB.

The Medium HotSpot Three NASes, three administrators and fifty users that connect an average of 3 times a day for two hours every day. One-time passwords. This kind of HotSpot would occupy 4 MB after one month and 48 MB after one year.

The Large HotSpot Ten NASes, ten administrators and one thousand users that connect an average of 3 times a day for two hours every day. One-time passwords. After one month: 78 MB. 1 GB after one year.

The Very Large HotSpot Ten NASes, ten administrators and ten thousand users that connect an average of 3 times a day for two hours every day. One-time passwords. After one month: 785 MB. 9.5 GB after one year.

Now, if we assume a small system with five operators, 100,000 prepaid cards distributed in 100 bundles and 1,000 HotSpots subscribed, after one month the database would require 30 GB of storage space, and after one year that requirement would grow to 150 GB. [1]

[1] A small system is assumed to have registered 250 small HotSpots, 500 medium HotSpots, 240 large and 10 very large

A more optimistic approach could assume a system with still five operators but 1,000,000 prepaid cards distributed in 1,000 bundles and 10,000 HotSpots subscribed. This would require a 301 GB database after one month and a 1.5 TB database after a year. [2]

The database for the prototype is hosted on a 160 GB hard drive which is suitable for a year of activity assuming a small system. When the system grows to a larger size or more time passes the storage space should be increased or the database should be distributed.

6.2 Further Developments

As we have seen, in order for a final product to evolve from this prototype a few changes need to be made. First of all the size of the system must be monitored in the terms expressed in the previous section. Second, in order to guarantee availability, redundancy needs to be added on all three servers. A detailed study on the capacity each individual server is not carried out in this work but is recommended. This would help approximate the time and conditions under which new servers would need to be added.

Another issue is performance. There are several algorithms which work in a prototype environment but would prove to be inefficient in the commercial product. One example is the algorithm for generating prepaid cards. It takes about 1 second to generate under 100 cards, but around 45 to generate 10,000. This algorithm won't be executed often, at least for the first year of the commercial product, but optimization is recommended. Other algorithms that deal with large amounts of data could also be tuned, like the algorithm for downloading accounting data. Another part of the system where performance is an issue is during authentication. The queries for authenticating a user, executed by the radius server can also be optimized to perform a smaller amount of select operations per query.

There are certain operations that are currently executed as database triggers. Actions like updating the monthly activity averages are executed upon each entry in the user activity table. This can prove to be too often on a system with high activity, as the commercial system is expected to be. These frequent updates could lead to eventual deadlocks or, in the best case, reduced database performance. A solution would be to update the averages with less frequency, without the use of triggers. The option of background processes, proposed by Mondru AB, which would take over this functionality, should be studied as a future replacement.

Finally we present a list of general system features that would make the system more robust.

- All the performance improvements mentioned in this chapter.
- Roaming between HotSpots for pre-paid users. This would require an additional HotSpot to HotSpot relation in the database, it would increase the usability of the prepaid card system, allowing users to use one card in several HotSpots.
- Self-upgrading of HotSpots. Administrators should be able to activate additional functionalities that are turned off by default upon subscription (i.e. one-time passwords). This would require developing an on-line payment system. This feature would attract users into upgrading to use all the system features.
- Providing with a feature to change passwords separate from the "modify" functionalities. This would add to the security of the system, as passwords wouldn't be requested from the database, but only updated.
- Sending users new passwords by SMS upon request. If the user loses a password, he can request the HotSpot administrator for a new one, which would be generated by the system and sent over SMS.

[2] This system of a larger scale is assumed to have 2500 small HotSpots, 5000 medium HotSpots, 2400 large and 100 very large

- When a HotSpot is registered, the password for the HotSpot administrator should be send by either SMS or email, to increase security.
- All the items mentioned in the “shoulds” section of the requirements.

Chapter 7

Conclusions

The final architecture design consists of one access point controller application, one radius server, a web server, one SMS server and a shared database. The main development effort was devoted to the web server. All the other components were configured to cooperate, share information, and together with the web server, now provide HotSpot owners with new, innovative functionalities. HotSpot owners can now have access to features like web-based HotSpot management, monitoring user activity, providing wireless internet access to pre-paid as well as post-paid users, allow for one-time passwords over SMS and many more.

Development was carried out in a modular fashion, where similar functionalities were bundled together in the implementation. This allows for the final prototype to be flexible when new requirements arise, as new modules can be incorporated with ease, requiring only minor changes on the existing modules.

One of the requirements for a commercial product is high availability, which the current architecture of this prototype does not guarantee. However it was designed in a way such that higher levels of availability could be obtained by adding redundancy in a few places. Additional radius servers, web servers and SMS servers can be brought in and integrated into the architecture increasing the availability of the system, requiring very little integration effort. Redundant databases can also be incorporated easily, but distributed databases would require careful planning on partitioning of the data.

FInally, load analyses indicated that the existing procedures for querying the database will continue to be efficient only for a maximum number of users. When this number is reached, the current procedures will have to be fine-tuned so that they execute at faster rates.

Bibliography

[1] J. Jakobsen *Mondru AB whitepaper on the HotSpot Air Interface*

[2] In-Stat/MDR *http://www.instat.com/press.asp?Sku=IN030831MU&ID=831*

[3] ******* *******

[4] ChilliSpot *http://www.chillispot.org/*

[5] FreeRadius *http://www.freeradius.org/*

[6] Kannel *http://www.kannel.org/*

[7] PostgreSQL *http://www.postgresql.org/*

[8] C. Larman. *UML y Patrones. Introduccion al Diseno Orientado a Objetos.*

[9] R. S. Pressman. *Ingenieria del Software. Un Enfoque Practico.*

[10] C. Rigney et. al. *RFC 2856 Remote Authentication Dial In User Service*

Appendix A

Appendix A: Use Cases

A.1 First Development Cycle

A.1.1 Login

- **Actors:** Administrator
- **Type:** Primary
- **Description:** An Administrator introduces his id, password and the name of his WISP to log into the system.
- **Referenced Requirements:** 1k

A.1.2 Logout

- **Actors:** Administrator
- **Type:** Primary
- **Description:** An Administrator chooses the option to logout at any point in time, exiting the system.
- **Referenced Requirements:** 1k

A.1.3 Authenticate User

- **Actors:** Administrator
- **Type:** Primary
- **Description:** A User types in his id and password. A NAS in the WISP he belogs to retreives that information, which is sent to the RS, together with any required NAS details. The RS authenticates that information against the repository and grants/denies authorization to the user.
- **Referenced Requirements:** 4a

A.1.4 Register User

- **Actors:** Administrator
- **Type:** Primary

- **Description:** An Administrator introduces the requested User information, registering him into the system.
- **Referenced Requirements:** 1f, 1g

A.1.5 View User List

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Administrator specifies a criteria to find a (group of) User(s). The system presents him with a list of all Users that match that criteria. This can range from a list of all Users in his WISP to a single User.
- **Referenced Requirements:** 1f

A.1.6 View User

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of Users, an Administrator choses the User whose information he wishes to view.
- **Referenced Requirements:** 1f

A.1.7 Modify User

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of Users, an Administrator choses the User whose information he wishes to modify. He will also be presented with the option to modify a User when viewing his information.
- **Referenced Requirements:** 1f, 1g

A.1.8 Change User Status

- **Actors:** Administrator, System
- **Type:** Primary
- **Description:** From a list of Users, an Administrator choses the User whose status he wishes to modify. He will also be presented with the option to change the status of a User when viewing his information. A User's status will also be changed if he reaches any limits (volume/time). The possible status are: Active, Suspended and Inactive. *See Use Case A.2.18.*
- **Referenced Requirements:** 1h, 1n

A.2 Second Development Cycle

A.2.1 View WISP List

- **Actors:** Operator
- **Type:** Secondary
- **Description:** An Operator specifies a criteria to find a (group of) WISP(s). The system presents him with a list of all WISPs that match that criteria. This can range from a list of all WISPs to a single WISP.
- **Referenced Requirements:** 1c

A.2.2 View WISP

- **Actors:** Operator
- **Type:** Secondary
- **Description:** From a list of WISPs, an Operator choses the WISP whose information he wishes to view.
- **Referenced Requirements:** 1c

A.2.3 Modify WISP's Privileges

- **Actors:** Operator
- **Type:** Primary
- **Description:** From a list of WISPs, an Operator choses the WISP whose privileges he wishes to modify. He will also be presented with the option to change the privileges of a WISP when viewing its information.
- **Referenced Requirements:** 1c, 1b

A.2.4 Delete WISP

- **Actors:** Operator
- **Type:** Primary
- **Description:** From a list of WISPs, an Operator choses the WISP he wishes to delete. Deleting a WISP causes all information related to him to be deleted, except for Users and User Activity Records.
- **Referenced Requirements:** 1c. 1b, 1q

A.2.5 Register WISP

- **Actors:** Administrator
- **Type:** Primary
- **Description:** An Administrator (wannabe) introduces the requested information about his WISP, NAS and himself, registering all three into the system.
- **Referenced Requirements:** 1a, 1b

A.2.6 View WISP

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Administrator choses the option to see his WISP's information
- **Referenced Requirements:** 1a, 1b

A.2.7 Modify WISP

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Administrator choses the option to change his WISP's information. He will also be presented with the option to modify his WISP when viewing its information.
- **Referenced Requirements:** 1a

A.2.8 Register NAS

- **Actors:** Administrator
- **Type:** Primary
- **Description:** An Administrator introduces the requested NAS information, registering it into the system.
- **Referenced Requirements:** 1o

A.2.9 View NAS List

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Operator specifies a criteria to find a (group of) NAS(s). The system presents him with a list of all the NASs that match that criteria. This can range from a list of all NASs to a single NAS.
- **Referenced Requirements:** 1o

A.2.10 View NAS

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of NASs an Administrator choses the NAS whose information he wishes to view.
- **Referenced Requirements:** 1n

A.2.11 Modify NAS

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of NASs, an Administrator choses the NAS whose information he wishes to modify. He will also be presented with the option to modify a NAS when viewing its information.
- **Referenced Requirements:** 1o, 1p

A.2.12 Delete NAS

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of NASs, an Administrator choses the NAS he wishes to delete.
- **Referenced Requirements:** 1o, 1p

A.2.13 Register Administrator

- **Actors:** Administrator
- **Type:** Primary
- **Description:** An administrator introduces the requested Administrator information, registering him into the system.
- **Referenced Requirements:** 1d, 1e

A.2.14 View Administrator List

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Administrator specifies a criteria to find a (group of) Administrator(s). The system presents him with a list of all the Administrators that match that criteria. This can range from a list of all Administrators to a single Administrator.
- **Referenced Requirements:** 1d

A.2.15 View Administrator

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of Administrators an Administrator choses the one whose information he wishes to view.
- **Referenced Requirements:** 1d

A.2.16 Modify Administrator

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of Administrators an Administrator choses the one whose information he wishes to modify. He will also be presented with the option to modify an Administrator when viewing his information.
- **Referenced Requirements:** 1d, 1e

A.2.17 Delete Administrator

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of Administrators, an Administrator choses the one he wishes to delete. All Administrators can be deleted except for the *root administrator*.
- **Referenced Requirements:** 1d, 1e

A.2.18 Update User Activity Records

- **Actors:** Administrator
- **Type:** Primary
- **Description:** While the User is connected, the system periodically updates his activity information in the repository allowing the Administrator to view it as described in *see Use Case A.2.19*. When the allowed volume/time for this user has expired the user will be disconnected accordingly. *See Use Case A.3.6* and *Use Case A.1.8*.
- **Referenced Requirements:** 4c

A.2.19 View User Activity Records

- **Actors:** Administrator
- **Type:** Primary
- **Description:** From a list of Users, an Administrator choses the User whose Activity Records he wishes to view.
- **Referenced Requirements:** 1g, 1i

A.2.20 Register Operator

- **Actors:** Operator
- **Type:** Secondary
- **Description:** An operator introduces the requested Operator information registering him into the system.
- **Referenced Requirements:** 1l, 1m

A.2.21 View Operator List

- **Actors:** Operator
- **Type:** Secondary
- **Description:** An Operator specifies a criteria to find a (group of) Operator(s). The system presents him with a list of all the Operators that match that criteria. This can range from a list of all Operators to a single Operator.
- **Referenced Requirements:** 1l

A.2.22 View Operator

- **Actors:** Operator
- **Type:** Secondary
- **Description:** From a list of Operators, an Operator choses the one whose information he wishes to view.
- **Referenced Requirements:** 1l

A.2.23 Modify Operator

- **Actors:** Operator
- **Type:** Secondary
- **Description:** From a list of Operators, an Operator choses the one whose information he wishes to modify. He will also be presented with the option to modify an Operator when viewing his information.
- **Referenced Requirements:** 1l, 1m

A.2.24 Delete Operator

- **Actors:** Operator
- **Type:** Secondary
- **Description:** From a list of Operators, an Operator choses the one he wishes to delete. All Operators can be deleted except for the *root operator*.
- **Referenced Requirements:** 1l

A.3 Third Development Cycle

A.3.1 Activate Prepaid Card Bundle

- **Actors:** Administrator
- **Type:** Primary
- **Description:** The Administrator introduces the code of the PPC bundle he wishes to activate
- **Referenced Requirements:** 2e

A.3.2 Generate Prepaid Card Bundle Information

- **Actors:** Operator
- **Type:** Primary
- **Description:** The Operator introduces the necessary parameters to generate the prepaid card bundle information. The system then generates a file with the desired information for all the cards in the bundle. The Operator takes this information to a printing facility.
- **Referenced Requirements:** 2a, 2b, 2c

A.3.3 View Prepaid Card Bundle List

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** An Operator specifies a criteria to find a (group of) PPC Bundle(s). The system presents him with a list of all the PPC Bundles that match that criteria. This can range from a list of all PPC Bundles to a single PPC Bundle.
- **Referenced Requirements:** 1d

A.3.4 View Prepaid Card Bundle Information

- **Actors:** Operator
- **Type:** Secondary
- **Description:** From a list of PPC Bundles, an Operator choses the Bundle whose information he wishes to view
- **Referenced Requirements:** 2d

A.3.5 Change Prepaid Card Bundle Status

- **Actors:** Operator, System
- **Type:** Secondary
- **Description:** From a list of PPC Bundles, on Operator choses the Bundle whose status he wishes to change (active/inactive). He will also be presented with the option to change the status of a Bundle when viewing its information.
- **Referenced Requirements:** 2f

A.3.6 Change Prepaid Card Status

- **Actors:** Ststem
- **Type:** Secondary
- **Description:** The status of a PPC is changed (to inactive) by the system when its allowed time/volume expires. *see Use Case A.2.18*
- **Referenced Requirements:** 2g, 4b

A.3.7 Change Language

- **Actors:** Administrator
- **Type:** Secondary
- **Description:** From a list of available languages, an Administrator choses the language in which he wishes to use the system.
- **Referenced Requirements:** 1j

A.3.8 Generate SMS one-time Password

- **Actors:** Administrator
- **Type:** Primary
- **Description:** The User sends an SMS to the system, receiving an SMS one-time password back.
- **Referenced Requirements:** 3a, 3b

A.3.9 Change current Login Method

- **Actors:** Administrator
- **Type:** Primary
- **Description:** The Administrator chooses the option to change the login method which is currently active
- **Referenced Requirements:** 3c