Team project
"Software for self-testing of the Telecommunication network of University of Freiburg"

Tri Atmoko
Refik Hadžialić

October 22, 2011

Albert-Ludwigs-Universität Freiburg
Lehrstuhl für Komunikationsysteme
Prof. Dr. Gerhard Schneider

Supervisors:
Konrad Maier
Dennis Wehrle

Sommersemester 2011

# Contents

1 Introduction and Motivation 3

2 Requirements 4
2.1 Logical and algorithmic requirements . . . . . 4
2.2 Software requirements . . . . . 5
2.3 Hardware requirements . . . . . 7

3 Database design 8

4 Software design 10
4.1 Database access . . . . . 11
4.2 Controlling the cell phones . . . . . 11
4.3 Client and Server class . . . . . 12
4.4 Ping class . . . . . 13
4.5 Data logging . . . . . 14
4.6 SSH Class . . . . . 14

5 Hardware design 15
5.1 BeagleBoard . . . . . 15
5.2 Cell phones . . . . . 16
5.3 Cables for the cell phones . . . . . 16
5.4 Server . . . . . 16

6 Communication protocol 17
6.1 Handler side . . . . . 17
6.2 Verification of the protocol . . . . . 17

7 Security and safety of the system 20
7.1 Encryption of the communication channels . . . . . 20
7.2 Security on the web site . . . . . 21

8 Web page 25
8.1 Communication between the web page and the test software . . . . . 25
8.2 Results on the web page . . . . . 25

9 How to use and start the system 27
9.1 Required libraries . . . . . 27
9.2 Configuring hardware . . . . . 27

10 Conclusion 28

# 1 Introduction and Motivation

In the following report, the authors will try to give you a brief insight into our team project. The goal of our project was to develop a mechanism for automatic testing of our University Telecommunication network. The Telecommunication network of University of Freiburg consists of: our own internal GSM and telephone network systems; GSM redirecting device (if one initiates a call to one of the four external GSM networks, it redirects the calls to: T-mobile, 02, Vodaphone or E-Plus); a SIP gateway for land-line calls inside of Germany (sipgate.de) and international calls. Since we did not have access to internal servers, our strategy was to exploit the existing systems from an external perspective and infer the results out of our findings. Before we had started working on our project, we had to analyze the overall network to come up with the test cases that contain the highest information content. The next step in our procedure was to implement our ideas into a working piece of software. Gradually we implemented a bit-by-bit of the final software. Every single step was accompanied by testing and validation procedures. At the end we connected all the “black-boxes” into one big piece of software. We have fulfilled our requirements, goals and made a fully operable test software. Despite developing a working software, all the way along we thought about the simplicity of the usage of the software. In the following chapters we will describe in more detail our approach to the problem and how each subsystem works. This particular report and our wiki page should be a sufficient guide and manual for understanding, running and continuing the development of our test software. Our team members were enthusiastic about the idea that our team project will contribute to a better perferomance and quality of the telecommunication network for the University staff and all their students.

# 2 Requirements

At the start of the project the requirements were not completely known but as the time had passed we redefined our requirements and goals. The first and the most important part at the start was to identify the key goals of our team project. The basic goal of our team project was to build a test software system which could tell an operator user what part of the system is not properly working in our University telecommunication network. Konrad and Dennis suggested us to analyze figure 1 and depending on it to build our test software. Our first attempt was to see what could we test without having access to

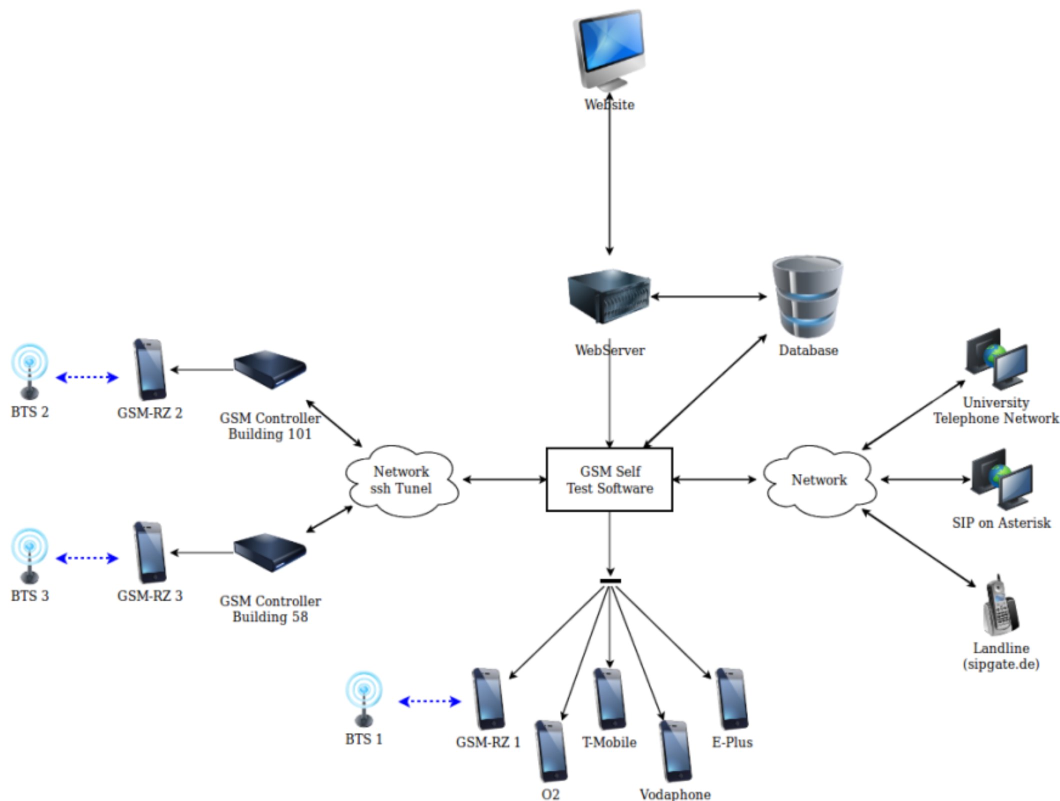


Figure 1: Overview of the Freiburg University telecommunication network [1]

the system. We installed numerous communication programs to see what others have done. After gaining access to the communication software, we had decided to build most of the test software ourselves. Libraries, which were used, were only the ones we could not develop ourselves because of the time-span of our team project.

## 2.1 Logical and algorithmic requirements

Despite the software and hardware requirements, the logic in our team project may be considered as the most important part. Controlling the software and hardware in a specific manner was one of the requirements in our team project. Moreover, we were required to draw a use case diagram and a simple test case diagram so that we could

better understand all the problems we had to deal with but also to easier follow the development of our test software.

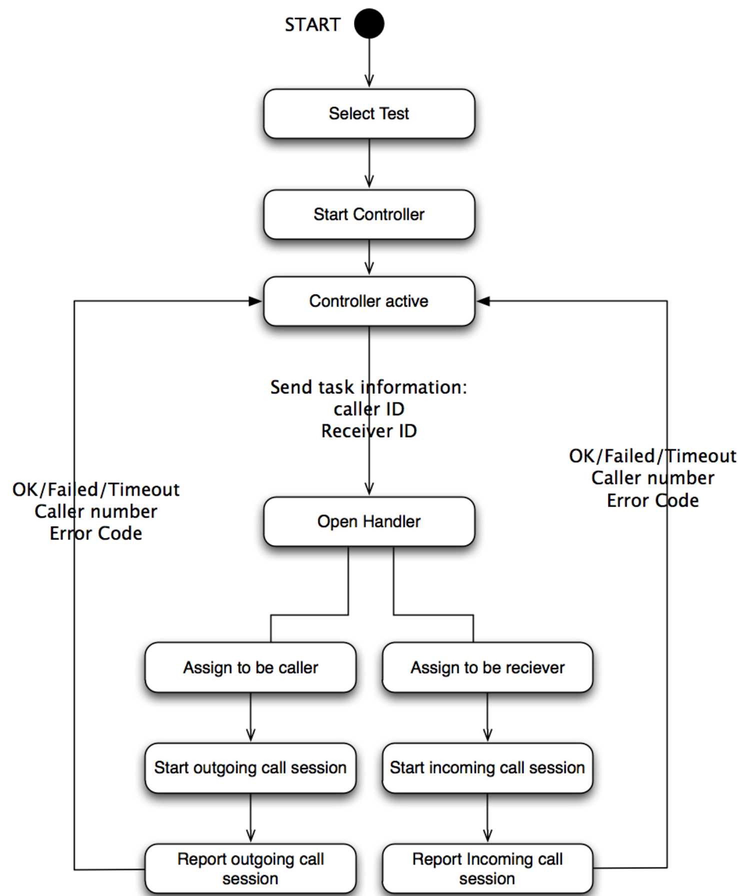


Figure 2: Simple algorithmic overview of a test case

## 2.2 Software requirements

Afterwards, as we had defined our logical approach to the problems, we had to choose the programming language to realize our ideas. Since we had the freedom of choice, between the three suggested programming languages Java, C++ and Python, we made a joint decision to use Python as the main programming language in our team project. One of the requirements was to finish the team project in time, therefore our decision to use Python is justified. Using Python we could work faster and integrate our subsystems more effectively [2]. Our programming language of choice is multi-platform, therefore our test software would be easy portable to other operating systems.

Likewise we had to decide how our test software will work. One of the requirements by Dennis and Konrad was to make the software capable of being run from the terminal. The

next requirement was to make an appealing GUI so that even an user without advanced Linux experience could handle the software and read out the results.

In addition it was required to log all the past tests. Later on a machine learning algorithm or some other intelligence could be applied to deduce some error behavior of the system (e.g. an intelligent algorithm could find that part of the system fail in a combined manner). To accomplish the logging of all the tests we had to use a database system. We decided to use MySQL since it is open source and well supported. However, one should keep in mind the test results are only stored in the database in case the test was started from the web site.

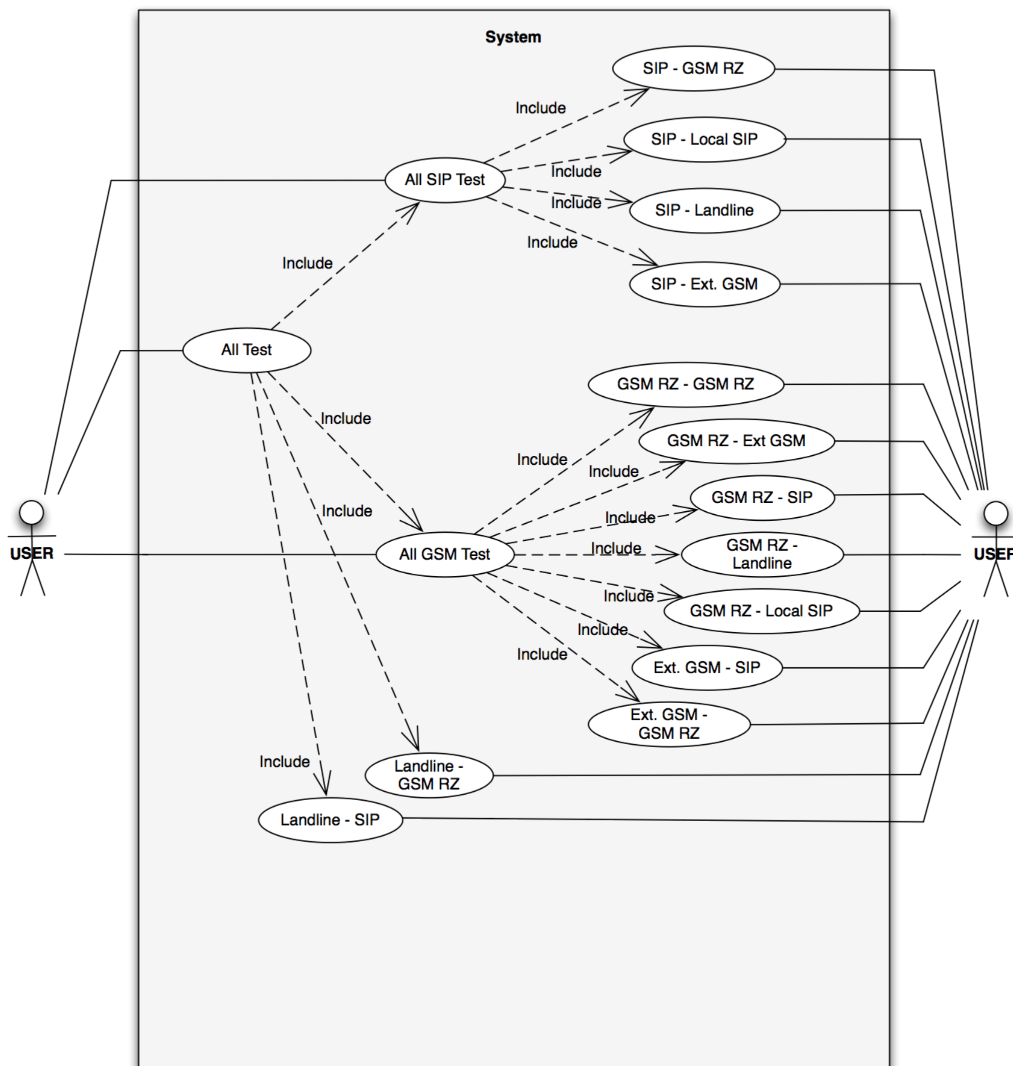


Figure 3: Test case diagram

## 2.3 Hardware requirements

Likewise the software requirements, we had hardware requirements as well. We were required to identify the hardware we will need to perform the tests. It was important to find old and cheap cell phones that could support *AT Modem* commands because our budget was limited.

A problem we had to deal with at the start was that the base stations are located at different geographical points which were not near to each other. No one should go everyday to the rooms where our cell phones are located only to change or charge the batteries. In the cable subsection we describe our approach to the charging battery problem. As we defined our requirements we continued with the process of developing the test software. During the development time we refined our requirements. In the next chapters we will explain our database, software and hardware design ideas.

# 3 Database design

As we mentioned in the software requirements section, we decided to use MySQL as our database system for storing the test information and results. It was not difficult to decide what database to use, since MySQL is one of the most supported database and one can find a library to use it with major programming languages. The key point in the design of our database was the simplicity and speed of accessing the data. We had decided to use seven tables. In the following paragraphs we will explain each table separately and its usage.

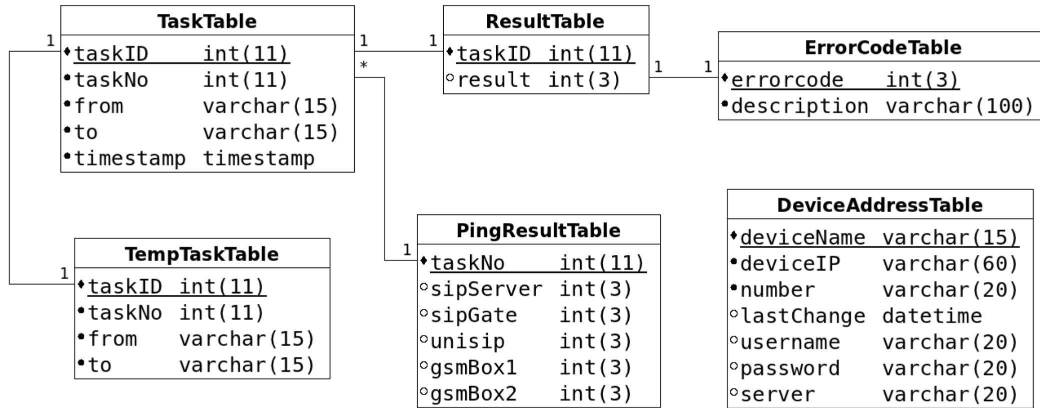


Figure 4: Database relationship diagram

The *PingResultTable* table has six attributes (*taskNo, sipServer, sipGate, unisip, gsmBox1, gsmBox2*), all of integer type. The *taskNo* attribute identifies the test number but not a single test (e.g. an operator user has selected three different tests to be executed, all of the three tests will have the same *taskNo* to identify them together as belonging to one test group and *taskId* identifies each single test and will be explained later). *sipServer* represents the Asterisk server ping result. *sipGate* is used to represent the SIP Gate server for the landline calls (`http://www.sipgate.de`). *uniSip* represents the ping results for our local University telephone network SIP server. *gsmBox1* and *gsmBox2* are the two single-chip Linux computers (BeagleBoard), that controll two cell phones each one (i.e. they are also known under the name of *nanoBTSx*). *taskNo* is the primary and unique key in the table *PingResultTable*. Rest of the attributes (i.e. *sipServer, sipGate, uniSip, gsmBox1, gsmBox2*) are used to insert the ping results, if the assigned servers are reachable or not. Before any test attempt is made, our test software first tries to ping the servers. These results are then stored in the *PingResultTable*.

The *ErrorCodeTable* table defines all the possible test results in the system, in other words it represents a list with error codes with their appropriate descriptions and meanings. It consists of two attributes (*errorcode* and *description*), the first is of integer type and the second of varchar type (the description message is allowed to be only 100 characters long). The *ErrorCodeTable* table is used by the main test software (i.e. controller) to report the operator user what kind of error had appeared in the system.

The *DeviceAddressTable* is the table containing the location and identification data for each server and device. The table consists of seven attributes, *deviceName, portName, number, lastChange, username, password, server*. *deviceName* is the attribute with the name of the device or server (e.g. GSMRZ1 or landline), it is of varchar type. *portName* is the attribute field with the location address for a cell phone (e.g. */dev/ttyUSB1*) or 'localhost' instead of NULL value for a server, it is of the varchar type. *number* represents the number of the used service (i.e. number of the cell phone, SIP, etc.) and is of varchar type. *lastChange* is a time value and represents the date and time the given entry was modified (we had plans in future versions of our test software that if an device gets a new IP address assigned it automatically changes it in the database). *username* is the field with the username stored in for a server/service, like SIP and landline. *password* attribute stores the password information for the given service. The *server* attribute stores information about the location of the server, IP or DNS address of the server. All three fields, *username*, *password* and *server* are of varchar type. The information stored in the given table is used by the test software to obtain usernames, passwords and addresses of the used services for the tests.

The *ResultTable* table is used by the test system to store final results for the performed tests. Our given table consists of two fields, *taskID* and *result* and both are of integer type. For each test entry with unique *taskID* an error code is assigned in the *result* field, depending on the test results. Error codes found in the *ErrorCodeTable* table can be only assigned to this field.

The *TempTaskTable* table represents the table with the tasks the system has to execute next time the test software is started. The given table gets new data every time an operator user submits one or more test cases from the website to be executed. *TempTaskTable* includes four attributes, *taskID, taskNo, from, to*. Former two are of integer type and later two of varchar type. *taskID* and *taskNo* identify the test task to be executed, *taskID* is the unique primary key. *from* and *to* fields have to match the names given in *DeviceAddressTable.deviceName*, these two attributes specify the caller and callee devices/services. Consequently, after the tasks get executed, the test tasks are removed and the given table is empty again until next tests are added to it. However, all the test tasks even after deleting them from *TempTaskTable* are kept in the *TaskTable*. The reason why the authors of this project divided it into two tables was because of the database row selection speed. We had made the assumption that with time the database size will grow and therefore the database speed will not be the same as during the development period.

The *TaskTable* table, as mentioned before contains all the tests ever performed from the web site. It is made out of five attributes, *taskID, taskNo, from, to, timestamp*. The first four fields are the same as in *TempTaskTable*, however the last one, *timestamp*, is used to record the exact time when the test was performed.

# 4 Software design

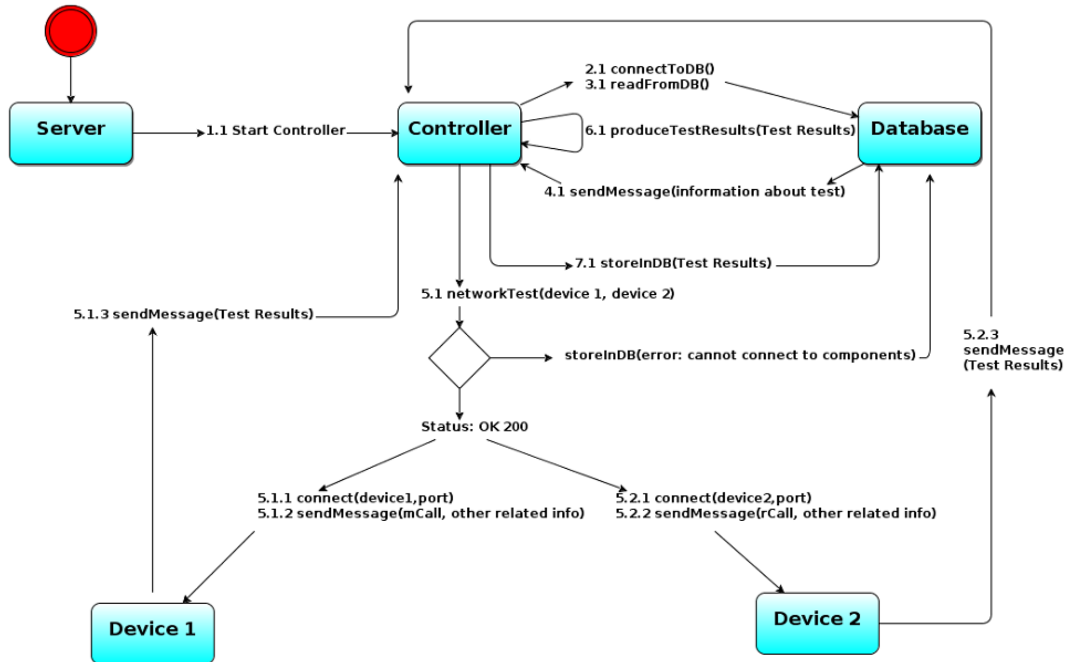


Figure 5: Class diagram for the dbClass

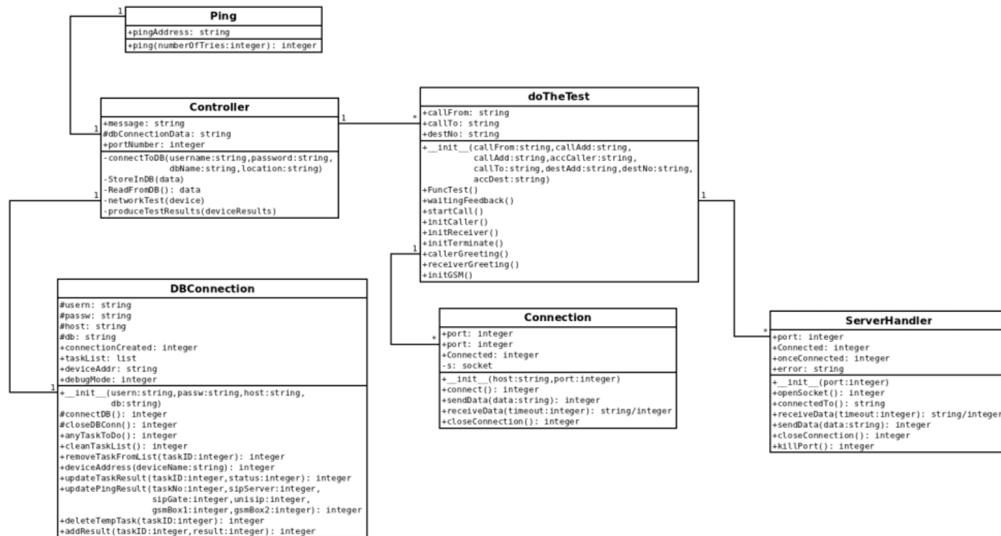


Figure 6: Class diagram (some classes were excluded)

## 4.1 Database access

Accessing the database is of critical value to our project, therefore we had developed our own class that limits the access to the database. In the process of developing our own class we used the MySQLdb library in Python [3]. The database class has two working modes, a normal working mode and a debugging mode. The difference between these two modes is in the output information. In case the error handling function raises an error and it is unknown, if the debug mode is set a complete back-trace of the error will be printed out. A developer can change the mode by setting the variable *debugMode=1*. The class diagram can be seen in the following figure. The method names are self-explanatory and

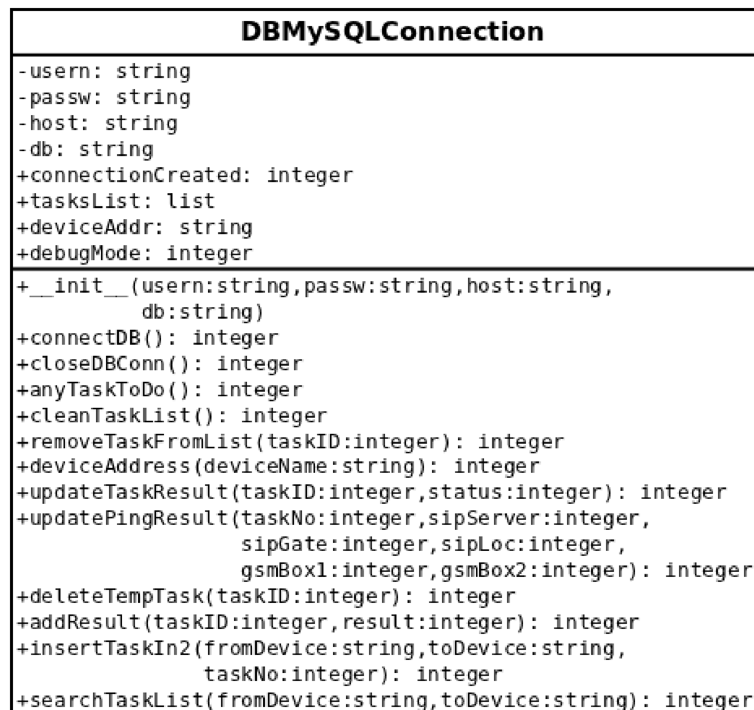


Figure 7: Class diagram for the dbClass

do not require extra explanations. All the outputs produced by the class can be found on the project wiki page [4].

## 4.2 Controlling the cell phones

Our first version of the developed program code for controlling the cell phones used predefined timed values to send commands instead of using a state controlled approach to confirm that every command was successfully received and executed by the cell phone. It meant we had to make an enormous number of assumptions. In comparison to our second approach, to build a state controlled cell phone control class, our first approach was inferior and slower. The state controlled method connected two cell phones, on the same base station, up to 15 times faster than the timed approach. One can easily apply the class just by correctly defining the parameters: port address, baud rate and timeout.

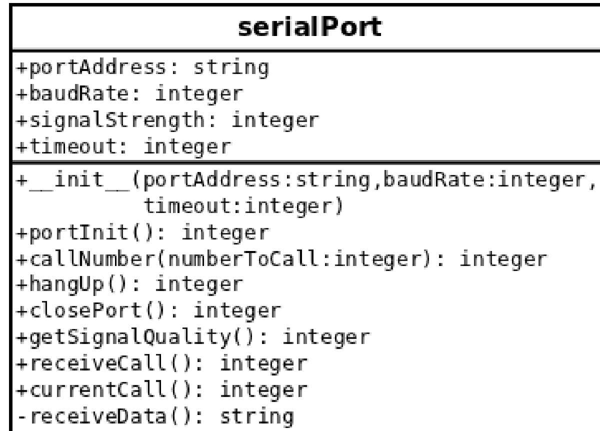


Figure 8: GSM class diagram for controlling the cell phones

The former two are self-explanatory and the timeout parameter is used to define when the alarm function should raise a timeout exception. A timeout exception gets raised when the cell phone does not respond (i.e. when the cell phone enters a deadlock or delayed state). We had used the serial port library inside of Python although we use USB cables to connect to our cell phones. One should be aware that our USB cables create a virtual serial port. More details on class design and an example can be found on our project wiki [4].

## 4.3 Client and Server class

Our socket communication code is based on the example given in the Python socket manual [5]. We extended it into two classes, a client and a server class. We had used the TCP protocol to base our two classes on[1]. The Server class can be seen in the following figure. The server class is implemented to accept only local connections[2]. First we determine our IP address and then create the socket to listen only for the same IP address (with a different IP address than the selected one a connection cannot be even established). One has to define the port on which the server object should listen. When receiving data one can easily define the timeout to be raised if data are not received in the timeout range or set it to *0* to infinitely wait for the buffer to be filled with received data. While testing the server class we had the problem to listen on the same port if the application was forcibly[3] restarted in less than 60 seconds. We got the error message: *"Address already in use"*. This is not known as error behavior but rather an option to help the server to catch lost live packets (i.e. packets that are still in the network looking for it is goal destination). We solved the problem by changing the socket options with the *SO_REUSEADDR* parameter. This enabled us to get around the error when we tried to restart our server application. Before solving the problem without using the socket

[1]TCP is reliable compared to UDP (i.e. transmitted packets get also delivered), packets are ordered when received and data are received in a stream (i.e. multiple packets can be read at once).
[2]More details are given in the section 7.1
[3]Manually closed using CTRL+C and run again.

parameter, we had another solution to get around this problem by killing the application running the port, this old method is obsolete now. In the process of testing the client

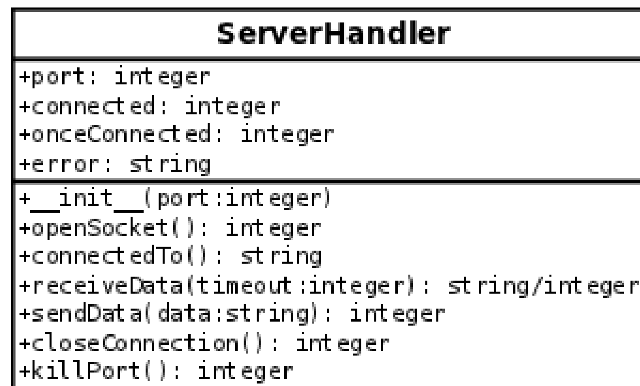


Figure 9: Server class, used by the server application

class we did not have any major problems. The only major flow we had to debug was when one of the sides disconnects that we get out of the waiting loop if the timeout variable was set to *0* (i.e. infinite waiting loop). The client class can be seen in the following figure. To initialize the client object one needs to define the IP address and the port of the server application listening on it. Once an instance of it is created and

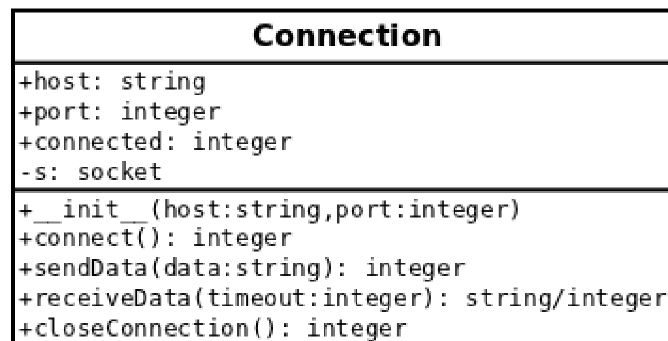


Figure 10: Client class, used by the client application

loaded with the IP address and the port, one needs to call the *connect()* method. The method will produce an integer based on its connection state. Output information and the programming code can be found on our project wiki page [4].

## 4.4 Ping class

Before making any test and establishing a connection we were required to ensure that the server is online. The best way to assess the liveness property was to ping the server computer running the required service. Once the class is properly defined, we could easily set the number of ping tries. A ping timeout response was set up to 2 seconds. For more details and insights, one can read more about it on our wiki page [4].

Figure 11: Ping class, used by test software

## 4.5 Data logging

If bugs appear it is important to reconstruct the case. One of the best ways to reconstruct the case was to log every single step part of code gets executed. We had used the logging class to follow our handler code run on the BeagleBoard. In case there is an error we could look inside of the log files and track the error. How the class works and what kind of outputs it produces can be found on our project wiki page [4].

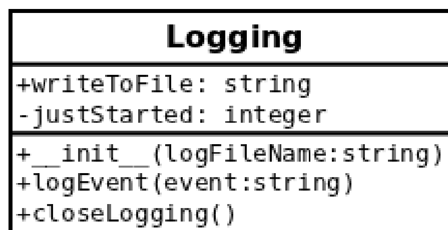


Figure 12: Logging class

## 4.6 SSH Class

# 5 Hardware design

In our team project we had the option to choose all the required hardware ourself beside the two BeagleBoards, which we were supplied by Konrad and Dennis. Since one of the project goals was to reduce the costs as much as it was possible, we had tried to use some of the leftovers found in our lab.

## 5.1 BeagleBoard

"The BeagleBoard is an OMAP3530 platform designed specifically to address the Open Source Community. It has been equipped with a minimum set of features to allow the

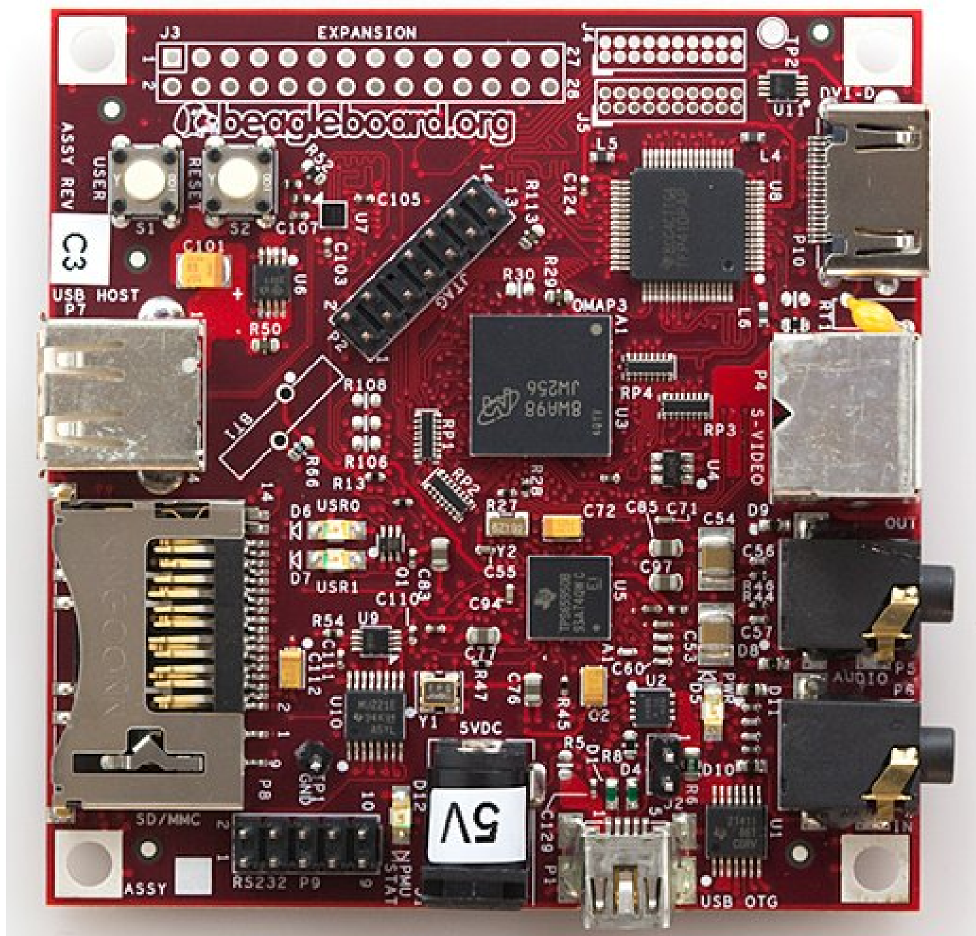


Figure 13: BeagleBoard, a Linux-on-chip board where our controller software runs the GSM device

user to experience the power of the OMAP3530 and is not intended as a full development platform as many of the features and interfaces supplied by the OMAP3530 are not accessible from the BeagleBoard" [10]. We run on it a special precompiled version of

Ubuntu for the ARM processor type. The Linux system boots up from an SD Card. The board has an USB hub and network port attached to it. In our project it is connected to our internal university LAN network and to a cell phone. We positioned the two BeagleBoards in rooms where we had LAN access and GSM signal coverage of our two local base stations.

## 5.2 Cell phones

Our first attempt was to control a Nokia cell phone 3310 with the supplied USB connection cable. The protocols used by old versions of Nokia cell phones, as the 3310, use the F-Bus protocol. It was not easy to work with. After performing various experiments we succeeded to send and to read SMS messages. Later on we found out that it was not possible to send commands for receiving and making the calls. In the meantime we found two Siemens phones, one M45 and S55. The first one, Siemens M45, had a cable supplied with it and it was not difficult to control it with the standard set of AT modem commands. At the start we did not have a cable supplied for the Siemens S55 phone. We controlled it over the Bluetooth port.

## 5.3 Cables for the cell phones

Due to the fact that we had used 5 cell phones on a single computer, the best solution was to order 5 USB cables. Konrad bought 5 cables for 5 Siemens S55 cell phones. All of the cables have an USB2Serial chip converter inside of them. Once they were plugged into the USB port, Ubuntu automatically recognized the cables and installed the drivers. The virtual serial ports were created and could be found on */dev/ttyUSBx*, where $x$ is the automatically assigned number for the port. Some of the cables had the capability to charge the Siemens S55 phones. Konrad had opened several cables to solder the power supplies to some contacts and the problem was solved for all of the cables.

## 5.4 Server

We were given an old Pentium 3 computer where we installed Ubuntu Linux. Configured the Apache web server and MySQL.

# 6 Communication protocol

A communication protocol represents a set of well defined rules by whose help two or more computing systems exchange information inbetween. When defining these rules, it is important to define a limited state space for every possible event, no matter did we get the appropriate response from the other side. Our approch to this problem was to build a simple synchronous protocol, where every expected message is confirmed or otherwise the connection between two sides is immediatelly terminated. Since designing protocols is a demanding and challenging topic which requires years of experience and verification, we do not expect that we had developed the best possible and an optimum protocol. In the following paragraphs we will try to give you a brief introduction of how our protocol works.

## 6.1 Handler side

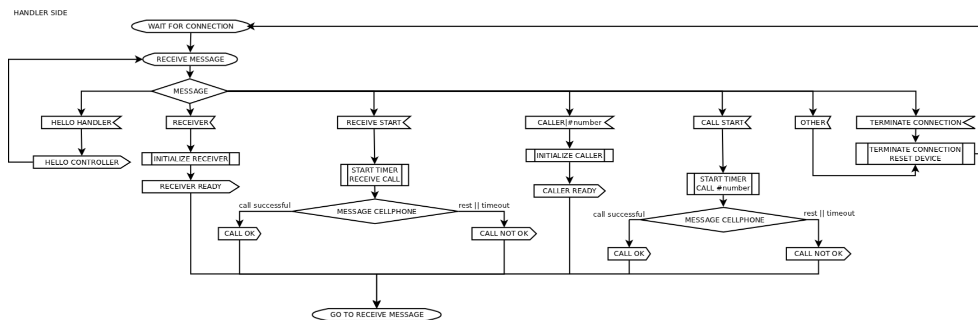


Figure 14: Flowchart of the protocol, on the handler side

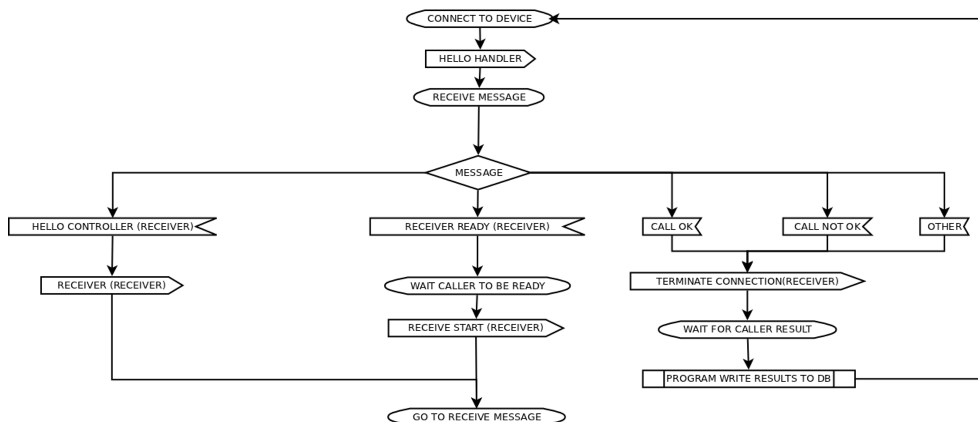


Figure 15: Flowchart of the protocol, on the controller side for the caller

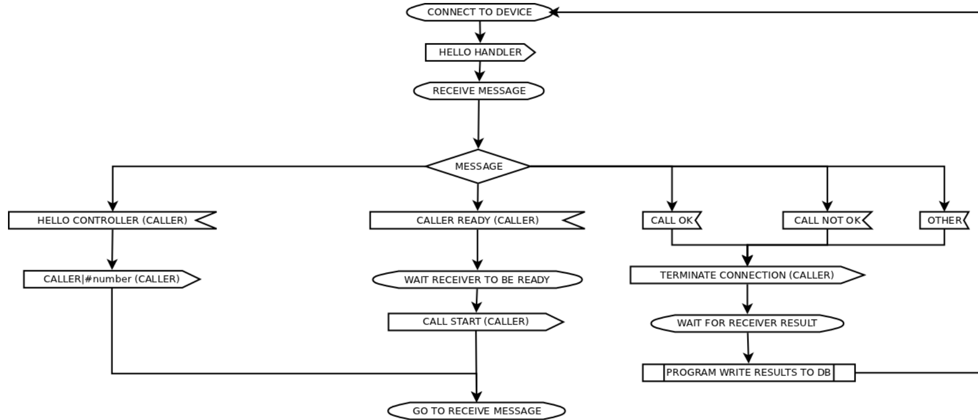


Figure 16: Flowchart of the protocol, on the controller side for the receiver

## 6.2 Verification of the protocol

"SPIN is a model checker - a software tool for verifying models of physical systems, in particular, computerized systems. First, a model is written that describes the behavior of the system; then, correctness properties that express requirements on the system's behavior are specified; finally, the model checker is run to check if the correctness properties hold for the model, and, if not, to provide a counterexample: a computation that does not satisfy a correctness property." [6]. We modeled our simple protocol in SPIN using the programming language PROMELA [6]. Since PROMELA is similar to C it was not possible to ensure 100% matching with Python but we had made the assumptions of it. We modeled both sides, server and client side. As well as the server side being a caller and a callee. It was important to find out if our protocol is deadlock or delayed state free. For more details our model can be found on our wiki project page with the PROMELA source code [4]. We had built in a 50% random probability that the call test will not be successful, to make the model even more realistic. Our protocol idea was deadlock free and the verification results prove it:

```
(Spin Version 6.1.0 -- 2 May 2011)
    + Partial Order Reduction
Full statespace search for:
    never claim - (none specified)
    assertion violations +
    cycle checks - (disabled by -DSAFETY)
    invalid end states +
State-vector 44 byte, depth reached 65, errors: 0
        40 states, stored
         3 states, matched
        43 transitions (= stored+matched)
        90 atomic steps
hash conflicts: 0 (resolved)
    2.195 memory usage (Mbyte)
unreached in proctype Server1
    (0 of 36 states)
unreached in proctype Server2
    (0 of 36 states)
```

```
unreached in proctype Client
    (0 of 67 states)
pan: elapsed time 0 seconds
```

After we had modeled the basic idea we had written the code that implements our idea. The Python code resembles some kind of a state machine which remembers the last state and what the next state should be in case of receiving corresponding message. Otherwise it enters the exit state and then the start state.

# 7 Security and safety of the system

Safety and security of the software plays a major role in our project. It is of vital importance that only as few as possible people have access to our test system since the resulting data could be exploited to plan an attack (e.g. assume the University alarm system uses the SIP gateway to connect to the outside world and to alarm the police, if one knows that the SIP gateway is not working properly, a burglar could plan to rob the University building just at that moment). Therefore the choice to go Open Source is justified due to the fact that one should know how every single detail of the system works. All the time, while we were working on the project, we were made aware of this issue by Denis and Konrad. We decided to use asymmetric key cryptography, where each side has two keys (private and public). In the next sections we will explain in more details how we applied the methods.

## 7.1 Encryption of the communication channels

At first we thought to encrypt the data before sending them but since none of us was an expert on encryption standards the idea was rejected. Alongside the fact that none of us had been an expert in the field of cryptography, we were neither experts in the field of Internet programming. One could find maybe a way to disable our server software with various hacking methods (e.g. trying to open the port until the system runs out of memory and in our case the system which we used on the handler side was a BeagleBoard with ARM architecture running on a single chip TI OMAP processor, refer to the picture in figure). We had to eliminate even the slightest possible threat in return for spending more time for debugging the test software system. Despite we were aware of all these facts, we had to choose one of the plenty implemented encryption standards on Linux. Denis and Konrad suggested using the SSH Tunneling method.

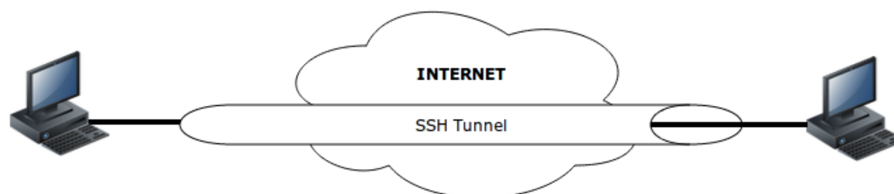


Figure 17: SSH Tunnel, all the communication inside the tunnel is encrypted

Using the SSH Tunnel port forwarding method we could hide the real port we had used for our socket connection. On the other hand we could force the socket to accept only local connections (i.e. from the machine where the handler software was running). The SSH Tunnel port forwarding method creates an encrypted tunnel between the two computers and then it creates two ports, one on the local and remote computer. All the data sent through the port on the local machine appear on the port at the remote machine.
The first problem we faced was that SSH required the username and password every time we tried to make an SSH connection. We could avoid this problem by copying the public key from our server (where our test software runs) to the BeagleBoard [7]. This can be

performed by executing the following commands in the terminal shell. One has to create first the private and public keys on the local machine(i.e. server computer, where the test software runs):

```
jsmith@local-host$ [Note: You are on local-host here]

jsmith@local-host$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jsmith/.ssh/id_rsa):[Enter key]
Enter passphrase (empty for no passphrase): [Press enter key]
Enter same passphrase again: [Press enter key]
Your identification has been saved in /home/jsmith/.ssh/id_rsa.
Your public key has been saved in /home/jsmith/.ssh/id_rsa.pub.
The key fingerprint is:
33:b3:fe:af:95:95:18:11:31:d5:de:96:2f:f2:35:f9 jsmith@local-host
```

Then one needs to copy the public key to the remote machine (BeagleBoard) using ssh-copy-id:

```
jsmith@local-host$ ssh-copy-id -i ~/.ssh/id_rsa.pub remote-host
jsmith@remote-host's password:
Now try logging into the machine, with "ssh 'remote-host'", and check in:

.ssh/authorized_keys

to make sure we haven't added extra keys that you weren't expecting.
```

After we have created the public and private keys, and coppied the public key on the machine to which we want to connect, we can test if we can make an SSH connection to the remote machine:

```
jsmith@local-host$ ssh remote-host
Last login: Sun Nov 16 17:22:33 2008 from 192.168.1.2
[Note: SSH did not ask for password.]

jsmith@remote-host$ [Note: You are on remote-host here]
```

The test was successful. We tested it with our SSH Tunnel port forwarding class and it worked perfectly.

## 7.2 Security on the web site

Securing the communication channels without making certain the web site is safe would be worthless. We decided to use the *https* protocol instead of the *http* since a person in the middle could sniff our data (e.g. a person is connected with his/her smart-phone over an unprotected wireless network) [8]. At the same time the web site should be accessible only by the authorized personel. Our first approach to this problem was to build an PHP page with *MD5* hashed passwords, however we got a suggestion by Konrad and Denis to use a safer encryption method implemented in the Apache web server software, *.htaccess*. By using these two techniques we protected the web site of some vulnerabilities known to us. If the web site will be only accessed from our local university network, we can additionally add an IP filter mask as well. In the following paragraph we will explain our procedure how to generate the keys and to enable the https protocol.

First we want to generate a server key by typing the following command:

```
openssl genrsa -des3 -out server.key 4096
```

This will generate a 4096 bit long private server key, one is asked to enter two times a password for the *server.key*. Using the generated private server key, we will create a certificate signing request, *server.csr*. We were prompted with a series of questions like country, state, organization name and etc which we had to enter to resume.

```
openssl req -new -key server.key -out server.csr
```

In the next step we had to sign the certificate signing request and enter the amount of days for how long it should be valid. In our case we entered the duration of one year, one can make it for longer periods as well (i.e. the amount of 365 has to be changed).

```
openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

We were asked to enter the password again for *server.key*. After we have completed this step we had to make a version of the *server.key* which did not require a password, *server.key.insecure* and we will rename the files appropriately.

```
openssl rsa -in server.key -out server.key.insecure
mv server.key server.key.secure
mv server.key.insecure server.key
```

The generated files are very sensitive, since they are our keys. After these steps were completed, we had generated 4 files: *server.crt*, *server.csr*, *server.key* and *server.key.secure*. Now we need to enable the SSL engine on the Apache web server. We coppied *server.key* and *server.crt* into */etc/appache2/ssl*.

```
refik@ubuntu:/etc/apache2$ sudo mkdir ssl
cp server.key /etc/apache2/ssl
cp server.crt /etc/apache2/ssl
```

Then we enabled SSL by typing in *a2enmod ssl*, "it is simply a general purpose utility to establish a symlink between a module in */etc/apache2/mods-available* to */etc/apache2/mods-enabled* (or give a message to the effect that a given module does not exist or that it is already symlinked for loading)" [8].

```
refik@ubuntu:/etc/apache2/ssl$ sudo a2enmod ssl
Enabling module ssl.
See /usr/share/doc/apache2.2-common/README.Debian.gz on how to configure SSL and create self
    -signed certificates.
Run '/etc/init.d/apache2 restart' to activate new configuration!
```

In the next procedure we had to establish a symlink from the 'available' default-ssl file to the 'enabled' file [8]. Then we created a folder where our secured PHP files will be located (e.g. https://some-domain-name.com/test-software).

```
refik@ubuntu:/etc/apache2/ssl$ sudo ln -s /etc/apache2/sites-available/default-ssl /etc/apache2/
    sites-enabled/000-default-ssl
refik@ubuntu:/etc/apache2/ssl$ cd /var/
refik@ubuntu:/var$ sudo mkdir www-ssl
```

We had backed up our old configuration files for the virtual hosts, for the case that the damage the Apache configuration files. Then we edited the *default-ssl* file.

```
refik@ubuntu:/var$ cd /etc/apache2/sites−available
refik@ubuntu:/etc/apache2/sites−available$ sudo cp default default_original
refik@ubuntu:/etc/apache2/sites−available$ sudo cp default−ssl default−ssl_original
refik@ubuntu:/etc/apache2/sites−available$ sudo vim default−ssl
```

Only the beginning of the file is listed here and we have modified the line starting with *DocumentRoot* from *DocumentRoot /var/www* to *DocumentRoot /var/www-ssl* (i.e. we had to redefine the location of our SSL directory).

```
<IfModule mod_ssl.c>
<VirtualHost _default_:443>
        ServerAdmin webmaster@localhost

        DocumentRoot /var/www−ssl
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>
```

One should keep in mind that the port 443 should be free for Apache to use it. In the proceeding step we had to ensure that Apache listens on the given port for a *https* connection. One could test that by going into the */etc/apache2/ports.conf*.

```
<IfModule mod_ssl.c>
    # If you add NameVirtualHost ∗:443 here, you will also have to change
    # the VirtualHost statement in /etc/apache2/sites−available/default−ssl
    # to <VirtualHost ∗:443>
    # Server Name Indication for SSL named virtual hosts is currently not
    # supported by MSIE on Windows XP.
    Listen 443
</IfModule>
```

In our case it was set up correctly, since the command: *Listen 443* was present. In our last configuration step we had to edit *default-ssl* file to define the correct locations of our keys and to ensure the SSL engine was turned on.

```
refik@ubuntu:/etc/apache2/sites−available$ sudo vim default−ssl
```

The following part of the file had to be found and modified according to our locations:

```
SSLEngine on

        # A self-signed (snakeoil) certificate can be created by installing
        # the ssl-cert package. See
        # /usr/share/doc/apache2.2-common/README.Debian.gz for more info.
        # If both key and certificate are stored in the same file, only the
        # SSLCertificateFile directive is needed.
        SSLCertificateFile /etc/apache2/ssl/server.crt
        SSLCertificateKeyFile /etc/apache2/ssl/server.key

        # Server Certificate Chain:
        # Point SSLCertificateChainFile at a file containing the
```

Finally we had configured our server and can proceed with the restart of the apache web server. We created a test web site */var/www-ssl/index.php* and navigated our browser to *https://localhost*. The test was successful!

```
refik@ubuntu:/etc/apache2/sites-available$ sudo /etc/init.d/apache2 restart
 * Restarting web server apache2 [Sat Oct 08 21:52:51 2011] [warn] _default_ VirtualHost overlap on
      port 443, the first has precedence
 ... waiting [Sat Oct 08 21:52:52 2011] [warn] _default_ VirtualHost overlap on port 443, the first has
      precedence [ OK ]
refik@ubuntu:/etc/apache2/sites-available$
```

# 8 Web page

One of the requests of our team project was to build a test system that could be started from the web site. Since we used the Open Source platform to base our project on, it was certain we will use it for the web site as well. The dynamic parts of the web site were programmed using PHP and JavaScript. The GUI was done using CSS. The web site opens TCP/IP sessions between itself and the Python test software. Due reasons explained in the section above, a test user needs first to enter his username and password to acccess the web site. Then a test user can manually select what type of tests he wants to perform or he can select already defined test, like the simple, smart or full test. (Describe here these three type of tests). Data about the performing tests are inserted into the database only in the case if the mutex lock for the web site can be obtained[4]. This way we can avoid inserting data about the test in case there is already a test user on the website performing some tests on the system.

## 8.1 Communication between the web page and the test software

Our first idea was that the PHP file starts the test software. However, parts of our test software open new terminal windows and since PHP has restrictions for starting GUI applications our approach was condemned for a failure at the start. We had to deal with this problem and our solution to it was to write a little Python script that will run in background and start our test software when required. Once a person starts the test over the web site, it automatically connects to the Python script over an TCP/IP socket. Before being able to start the test software one needs first to obtain the mutex lock on the web site and to check if there is a mutex lock for the test software running. Using this approach we can ensure that only one user at the time can be on the web site and run only one instance of the test software. In the next step we send the Python script a message to start the test software. The test software obtains a mutex lock as well. When the test software is started the web page checks if a software lock is obtained. Once it is obtained we can proceed with creating a new socket connection between the web site and the test software. Our TCP/IP communication between the web site and the test software is not encrypted since both the web page and the test software run on the same server computer. The mutex locks are freed after the tests are performed. Our test software has a timeout timer in case that the web site hangs or somehow the socket connection breaks where it automatically shuts down.

## 8.2 Results on the web page

All the performed test results are displayed on the web site. The results are displayed in real time after each selected test case is performed. After all the test cases have been performed a topological picture is generated which represents the current state of the system, this can bee seen in the following figure. Afterwards, when the result picture is generated, the test user can easily see what is wrong in the system. Various icons represent different subsystems. Reading the test results is simple as looking at the icons

[4]The mutex lock will be explained in the next subsection.

and identifying if they have: a green plus signs (i.e. working properly), a red minus sign (i.e. not working properly) and a yellow exclamation mark (i.e. it was not tested).

- Triangles represent BTS stations
- Cellphones represent the external networks (E-Plus, Vodaphone, T-Mobile and O2)
- Telephone represents the landline and a telephone with a mortarboard the University telephone network
- Servers represent the OpenBSC and LsfKs-Asterisk
- Two monitors represent the SIP system

The inference mechanism works as following: if a test case works, we can conclude that the subsystems connected inbetween the two ends are working properly as well. We use the pChart library[5] to generate the topological picture of our telecommunication system [9].

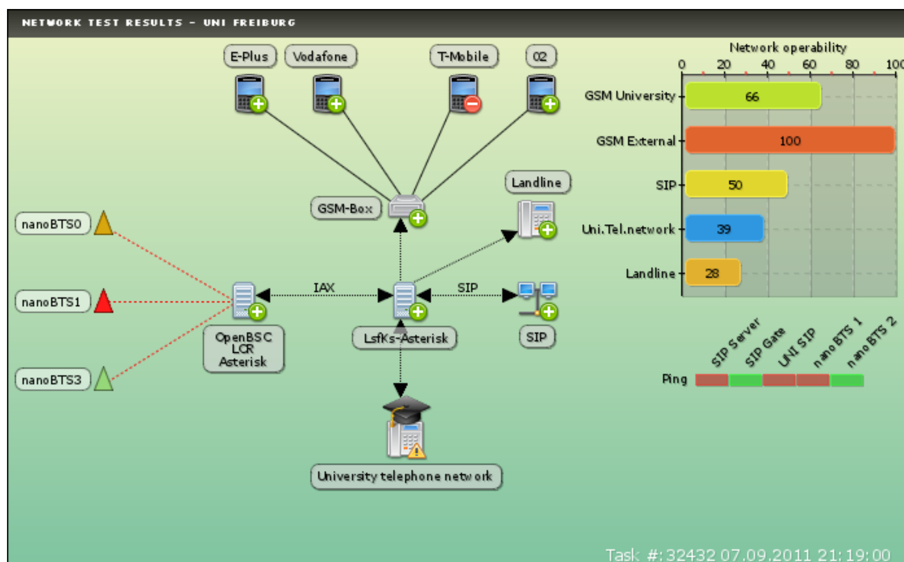


Figure 18: Result image showing working, defected and not tested subsystems

On the right side of the result picture the test user can immediatelly identify the network operability in percentage[6]. Bellow the network operability statistics are the ping results statistics located. If one of the fields is red it means the subsystem is not online or cannot be seen by our server computer where the test software is located.

[5]It is under the GNU GPLv3 license and our project is nonprofit!

[6]The test user has to take into account that this percantage is only valid if a full test is performed.

# 9 How to use and start the system

## 9.1 Required libraries

## 9.2 Configuring hardware

# 10 Conclusion

## References


[1] *Projects based on RZ-GSM*, accessed on 10.06.2011, available at http://lab.ks.uni-freiburg.de/projects/gsm/wiki.

[2] *Python Programming Language - Official Website*, accessed on 10.06.2011, available at http://www.python.org/.

[3] *MySQLdb User's Guide*, accessed on 05.06.2011, available at http://mysql-python.sourceforge.net/MySQLdb.html.

[4] *[2011] GSM Selftest - Wiki - Lehrstuhl für Kommunikationssysteme*, accessed on 20.09.2011, available at http://lab.ks.uni-freiburg.de/projects/gsm-selftest/wiki.

[5] *17.2. socket - Low-level networking interface*, accessed on 20.06.2011, available at http://docs.python.org/library/socket.html.

[6] M. Ben-Ari *Principles of the Spin Model Checker*, Springer Verlag, Weizmann Institute of Science, Israel, ISBN: 978-1-84628-769-5, 2008.

[7] R. Natarajan, *3 Steps to perform SSH login without password using ssh-keygen & ssh-copy-id*, accessed on 18.08.2011, available at http://goo.gl/fX68N.

[8] P. Bramscher, *Creating Certificate Authorities and self-signed SSL certificates*, accessed on 05.09.2011, available at http://www.tc.umn.edu/~brams006/selfsign.html.

[9] *pChart*, accessed on 15.08.2011, available at http://www.pchart.net/.

[10] *BeagleBoard System Reference Manual*, accessed on 20.06.2011, available at http://beagleboard.org/static/BBSRM_latest.pdf.