

Supervisors:

Stefanoiu, Dan ("Politehnica" University of Bucharest)
Balan, Radu (Siemens Corporate Research, Princeton, NJ)

WLAN-based Wireless Medium Surveillance System for Location Estimation

Vasile, Matei-Eugen

<skuphundaku@yahoo.com>

*To Letiția,
without whom it would have been much harder*

Contents

1	Introduction	7
2	Wireless location systems	8
3	Location system	11
3.1	Location system architecture	11
3.1.1	Overview	11
3.1.2	Hardware architecture	11
3.1.3	Software architecture	13
3.2	Location system hardware	13
3.2.1	Server(s)	13
3.2.2	Wireless sensors	13
3.2.3	Wireless access point	14
3.2.4	Wireless capable device	14
3.2.5	Additional hardware	14
3.3	Location system software architecture	14
3.3.1	Functionality	14
3.3.2	Components	14
3.3.3	Data flow	15
3.3.4	Location server	16
3.3.5	Location database	24
3.3.6	Traffic generator server	26
3.3.7	Wireless medium sniffer	26
3.3.8	Location database user interface	27
3.3.9	Calibration device user interface	33
3.3.10	Fingerprint generator tool	38
3.3.11	Perl Location libraries	39
3.3.12	Building the software	40
4	Experimental results	45
5	Conclusions and future work	51
	Bibliography	53

Appendix	55
A Semaphores	55
B Signals	57
C Sockets	59
D SQL	61
E Threads	64
F XML	67

List of Figures

2.1	Location system hierarchy	8
3.1	Location system architecture	12
3.2	Location system software architecture	16
3.3	Location database architecture	25
3.4	Admin GUI: Admin panel	29
3.5	Admin GUI: Admin panel's available options	29
3.6	Admin GUI: Floors panel	30
3.7	Admin GUI: APs panel	31
3.8	Admin GUI: APs panel's available actions	32
3.9	The Calibration GUI	33
3.10	Choosing the floor	33
3.11	The selected floor plan. It shows the positions present in the database	34
3.12	Adding a new point	35
3.13	CLI: the database schema was selected	36
3.14	CLI: the calibration command was sent to the Location server	37
3.15	CLI: the calibration device received a reply from the server	37
4.1	The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= -0.02	46
4.2	The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= -1.55	46
4.3	The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= 3.15	47
4.4	Distribution of probability for the accuracy of the location estimation when using a Gaussian model	47
4.5	Distribution of probability for the accuracy of the location estimation when using the naive Bayes method	50
1	Client/server communication when using TCP sockets	60

List of Tables

2.1	Parameter comparison of indoor positioning systems	10
2.2	Performance comparison of indoor positioning systems	10
4.1	Distribution of probability for the accuracy of the location estimation when using a Gaussian model	48
4.2	Distribution of probability for the accuracy of the location estimation when using the naive Bayes method	49
4.3	Comparison of correctness of location estimation for a given distance error	50

Chapter 1

Introduction

The problem of location determination has become a very interesting topic during the last few years with the emergence of location based services. These services range from asset tracking and location based information to emergency response. Asset tracking is a very important aspect for most companies but that is especially true for those in the logistics business. Being able to track what is where at any time is very important. Another notable type of location based service are the services that are based on the location of the user. Applications like these can be envisaged in hospitals, where the response time of the medics could be increased by being able to guide them from their location to the location they are required to be in, in the shortest amount of time possible. Also a type of location information based service could be used in the tourism industry for easily guiding visitors. Emergency response is another type of location based service that is gaining more and more ground. For example, in the U.S., Enhanced 911 compliance is a requirement for all the wireless telephony carriers.

The approach used for solving this problem can vary depending on which technology is used. For outdoor locations GPS is a good choice. The accuracy of the system is within a few meters. GPS starts showing its limitations when trying to use it for indoor positioning though. This is why, for indoor location, other techniques have to be developed. Because the wide availability of wireless LAN equipment, the idea of trying to use WLAN equipment for implementing location estimation came naturally.

In this paper a solution based on wireless LANs' signal strength measurements will be presented. The approach relies on using a network infrastructure of wireless sniffers to collect information about the wireless traffic in a particular area, store it and use it to determine the location of wireless devices within that specific area.

Chapter 2

Wireless location systems

Overview

A basic functional block description of a positioning system can be found in [8]. The structure can be seen as a hierarchical system made up of a number of location sensing devices, at the bottom of the hierarchy, a location estimation engine, in the middle, and a display system, at the top of the hierarchy, as in figure 2.1.

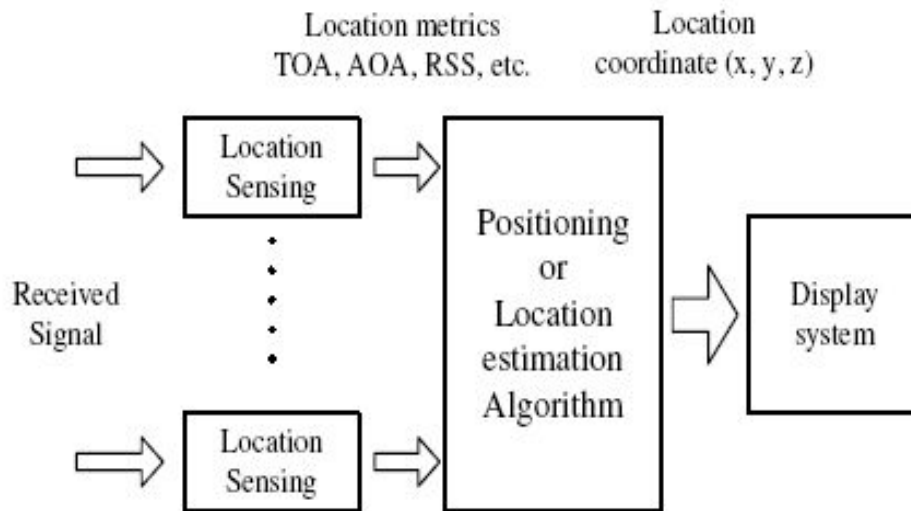


Figure 2.1: Location system hierarchy

The location sensing devices can use technologies like radio frequency (RF), infrared or ultrasound to gather location relevant data. In this paper, RF location technology will be investigated. Then, after settling for a location technology of choice, a location sensing technique has to be chosen. In the case of RF technologies, the location sensing techniques can be based on time, direction (angle) or signal strength level. The sensing technique converts the sensed signal into location metrics that are time of arrival (TOA), angle of arrival (AOA) or received signal strength indication (RSSI). The location system presented in this paper is using a RSSI based approach.

The RSSI based approach requires the measurement of received signal strength at a series of positions in order to create a database of location fingerprints. To estimate the mobile location, the system needs to first measure the received signal strength at particular locations and then search

for the pattern or fingerprint with the closest match in the database. This technique does not require the mobile station to see at least three base stations or access points in order to determine the location. The disadvantage of this technique is that it is very time-consuming to perform an exhaustive data collection for a wide area network, such as in outdoor positioning systems. The advantage is that it can be used in situations where the determination of a RF propagation model is difficult or even impossible, such as in indoor positioning systems.

At the present moment, the vast majority of the indoor positioning systems are based on signal strength approaches. This is the case because in indoor environments it is very difficult to determine a propagation model for the radio signal. This makes TOA and AOA approaches impractical in indoor environments.

Classification

Having already delimited a frame of reference in regard to how an indoors location estimation system could be implemented, the implementations can be clasified by a number of characteristics of the location system:

absolute or relative referencing: Absolute referencing systems share single or unified reference grid. Relative referencing systems have their own frame of reference grid for each locator.

network- or mobile-based: Network-based systems estimate the location at a central location in the system, e.g. a location estimation server. Mobile-based location systems estimate their own location, without the need of any dedicated location server.

network- or mobile-assisted: Network-assisted systems estimate the location of a mobile station by using measurements done by devices that are part of the network infrastructure. Mobile-assisted location systems estimate the location of a mobile station by using measurements done by the mobile station itself.

These are the basic location system clasification tools. Starting from here, any number of other classifications could be extracted. For example, the location systems could be clasified from a privacy standpoint. It is obvious that a mobile-based and mobile-assisted location system has far greater privacy than a network-based and nework-assisted location system.

The implementation presented in this paper is a mobile-based, network-assisted, absolute referencing location system. The mobile-based aspect can be easily changed to network-based. On the other hand, the network-assisted aspect of the implementation is central to the entire design, and, as it will be shown later on, this is where this location system improves on existing designs.

State-of-the-art

At the present moment, the designs that are predominately presented in the speciality literature are mobile-based and mobile-assisted location systems. Their operation is based on the probe request/response mechanism that is specified by the IEEE 802.11 standard and that is implemented in all the 802.11 compliant wireless LAN equipment. The algorithm they base their operation on is:

```
/* in an infinite loop... */
while (1) {
    for every available channel {
```

```

    send probe request;
    wait for probe response;
    if (received probe response) {
        compute RSSI of probe response frame;
    }
}
}

```

This algorithm is used to get signal strength measurements for all the available wireless access points in the vicinity on the mobile device. Using these measurements and a previously generated fingerprint of the signal strength in the respective area, the mobile station can estimate its own position. The flaw of this approach is that the strength of the signal from different access points is measured sequentially, not in parallel.

One of the most used methods for estimating the location, after the measurements were done, is using Bayes’ theorem. It is more precise than using a “Nearest neighbor” or “Weighted k nearest neighbor” method. Using just a set of measurements (one measurement for each available wireless access point), though, doesn’t usually yield satisfactory results. That is why different types of optimizations are made to the basic location determination algorithm. By implementing different optimizations, different researchers have obtained the results listed in table 2.2 while using the systems described in table 2.1. **NOTE:** All the mentioned teams have used a bayesian inference method for estimating the location.

Some of the improvements made to the basic algorithm include:

- using more than one set of measurements when estimating the location.
- assigning a weight to each estimated position and computing a weighted average of the results in order to get a more precise location estimation (see [13]).
- compensating for small-scale variations in received signal strength (see [12]).

System	Spacing	Positions	Samples/Pos.	APs	Orient.	Env.
Roos et al [9]	2m	155	40	10	N/A	1-floor
Ladd et al [10]	3m	11	1307	5	2	Hallway
Youssef et al [11]	1.5m	110	300	4	N/A	Hallway
Xiang et al [14]	N/A	100	300	5	4	1-floor

Table 2.1: Parameter comparison of indoor positioning systems

System	Precision
Roos et al [9]	within 2.5m, 90%
Ladd et al [10]	within 1.5m, 77%
Youssef et al [11]	within 2.1m, over 90%
Xiang et al [14]	within 1.8m, 90% (static device)

Table 2.2: Performance comparison of indoor positioning systems

A more detailed general presentation of location estimation systems can be found in [16].

Chapter 3

Location system

3.1 Location system architecture

3.1.1 Overview

The Location system is made up of components that communicate with each other in order to provide the system's functionality. The architecture of the whole system can be structured on two layers: the hardware layer and the software layer. The deployment of the system can be done by either keeping a strict correspondence between the hardware and software components, or by running several software components on the same hardware component. The choice can be done by taking into account factors like performance and/or cost.

3.1.2 Hardware architecture

Hardware-wise, the system's architecture has the following components:

- the server machine/machines
- the Wireless LAN access point
- the Wireless LAN sensors
- the mobile device
- the network infrastructure connecting the server(s), AP and sensors

The server machine(s) host the Location server, the Location database server and the traffic generator server. The Location server accepts two types of incoming connections: calibration commands coming from the mobile device and sensed data coming from the Wireless LAN sensors. In order to do that, the machine hosting the server has to be reachable over the network. The Location Database server holds all the data about the area in which the location estimation is done. The information is written in the database either from the Location server or from the database administration GUI, over the network. The component that reads data directly from the database server is the calibration GUI. The Location server and the database administration interface could be hosted on the same machine as the database server but, in most cases, the calibration client GUI will be running on a different machine. This is the reason why the calibration

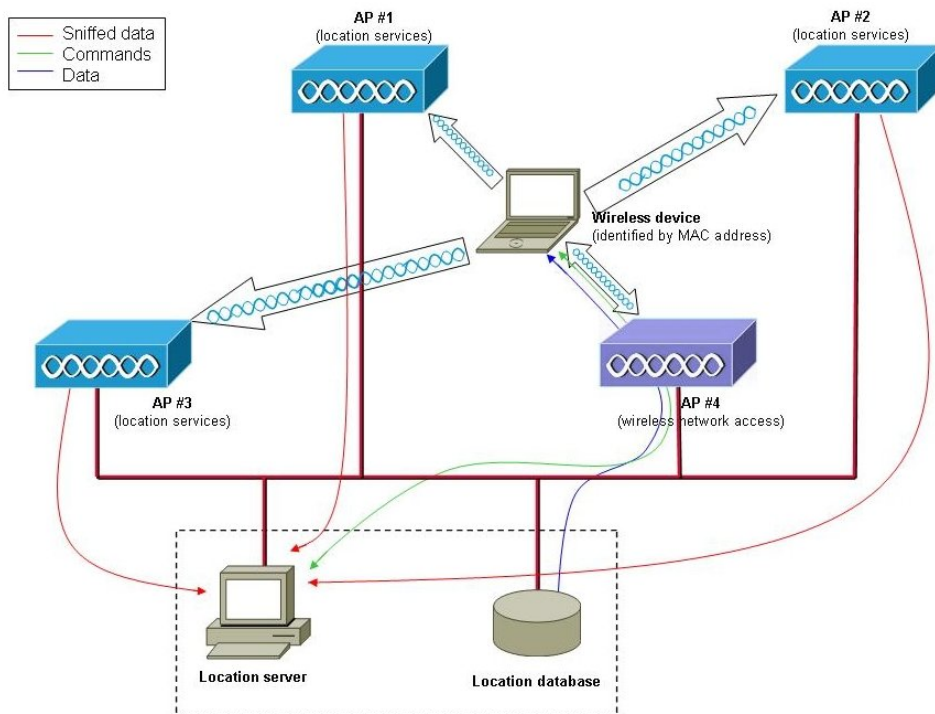


Figure 3.1: Location system architecture

database has to have network access as well. Last but not least, the traffic generator server needs network access as well in order to perform its duties. It's task is to provide the mobile device a way of generating network traffic over the wireless medium.

The Wireless LAN access point has the task of providing access to the network infrastructure to the mobile device. This is important due to several reasons: First, the calibration client which is running on the mobile device needs access to the Location database server in order to get all the necessary information about the area in which the calibration is done. Secondly, the calibration commands are sent over the network and they have to reach the Location server, which is connected to the wired part of the network. It also needs a way of receiving the replies to these commands. Thirdly, in order to generate wireless network traffic, the calibration client running on the mobile device has to connect to the traffic generator server.

The Wireless LAN sensors are connected to the same network as the servers and the Wireless LAN access point. They listen to all the Wireless LAN traffic and gathers information about it. They can, if needed, filter the wireless traffic by listening only to the frames that have a specified destination.

The mobile device is used to calibrate the system. It uses the network access provided by the Wireless LAN access point in order to connect to the system's servers. As said earlier, it connects to the Location database server in order to fetch from the database the necessary information about the area that is being calibrated, to the Location server to send calibration commands and to wait for the replies coming from the server and to the traffic generator server in order to

generate wireless network traffic that is going to be sensed by the Wireless sensors and used in the calibration process.

The network interconnecting all the system's components has two major components: the wired infrastructure, to which all the system's hardware connect with the notable exception of the mobile device and the wireless part which is used by the mobile device to calibrate the location system.

This is the bare minimum of hardware components for a system configuration in which all the servers are hosted on the same machine. Different configurations can be adopted. For instance, if the hardware is not a problem, each server software could be run on a different machine. On the other hand, the Location server and the Location database server could run on the mobile device machine. In this configuration, only the traffic generator server would be run on a separate machine. In the current implementation of the Location system, the traffic generator server *must* be run on a different machine in order to correctly generate wireless traffic.

3.1.3 Software architecture

The software architecture of the system has the following components:

- the Location server
- the Location database
- the traffic generator server
- the wireless sensor software
- the Location database user interface
- the calibration device user interface
- the fingerprint generator tool
- the Perl Location libraries

The Location system's software architecture will be presented extensively in section [3.3.2](#).

3.2 Location system hardware

3.2.1 Server(s)

As said earlier, in section [3.1.2](#), the servers can all run on the same machine or they can each run on individual machines. The minimum requirements are 1 PC with an Ethernet network interface.

3.2.2 Wireless sensors

The Wireless LAN sensors are based on the *Siemens Wireless AP2610* hardware platform. This platform is built around the *Atheros AR5312* chipset.

3.2.3 Wireless access point

Any Wireless LAN access point can be used.

3.2.4 Wireless capable device

At the present moment, the mobile device can be any kind of laptop computer. The only requirement is that it must have a Wireless LAN network interface.

3.2.5 Additional hardware

To create the wired Ethernet segment to which the Wireless access point and all the servers and sensors are connected, an Ethernet switch is necessary. The only requirement for the switch is that it has to have enough ports for all the devices that make up the Location system.

3.3 Location system software architecture

3.3.1 Functionality

This software was designed and developed to effectively gather and process wireless data in order to determine the location of wireless enabled devices. To reach this goal, it has to be able to accomplish a number of tasks. These tasks are:

1. to create and administer the database in which location data will be stored
2. to sniff the wireless transmission medium in order to obtain relevant location data
3. to store the obtained data in the database mentioned before
4. to use the received and stored data in order to determine the wireless devices' location

The way these tasks are accomplished will be detailed during the description of the individual components of the Location software architecture.

3.3.2 Components

The architecture of the Location software consists of a number of components:

- the Location server
- the Location database
- the traffic generator server
- the wireless sensor software
- the Location database user interface
- the calibration device user interface
- the fingerprint generator tool
- the Perl Location libraries

The Location server acts as a hub between most of the other components. It receives sniffed data from the sensors and stores it in the database, and it interacts with the calibration device user interface. This interaction consists in receiving calibration commands and sending replies to the calibration device. More information about the Location server can be found in section [3.3.4](#).

The Location database is where the sniffed wireless data is stored after it is received from the wireless sensors. For this, a relational database management system (DBMS) is used. More information about the choice of the DBMS and about the database architecture can be found in section [3.3.5](#).

The traffic generator server is used by the calibration device for establishing a connection and using this connection to generate traffic over the wireless network. The point in this is to generate wireless traffic that can be sniffed by the wireless sensors. More information about the traffic generator server can be found in section [3.3.6](#).

The wireless sensor software runs on wireless access points. Its purpose is to continuously scan all the wireless channels in order to capture wireless frames and, then, to send the sniffed information to the Location server. More information about the wireless sensor can be found in section [3.3.7](#).

The Location database user interface is a stand-alone program that interacts just with the Location database component. It is used to create the database structure necessary for the system's operation, to initialize it and, after data is stored in the database, to view the data. More information about the Location database user interface can be found in section [3.3.8](#).

The calibration device user interface, like the database user interface, is a stand-alone program. It is used for performing the location system calibration. It reads data from the database, it sends calibration commands to the Location server and it waits for replies from the Location server. More information about the calibration device user interface can be found in section [3.3.9](#).

The fingerprint generator tool It is used to extract data from the *_calibration* table in the database, filter out the frames that were not captured by all the access points and from the remaining frames to generate a fingerprint that is then stored in the *_fingerprint* table. More information about the fingerprint generator tool can be found in section [3.3.10](#).

The Perl Location libraries are used to provide lower level services to the executable Perl scripts of the Location system. They provide backends to the PostgreSQL database, to the network and to the XML parser. More information about the Perl Location libraries can be found in section [3.3.11](#).

3.3.3 Data flow

The information flows on a number of different pathways in the system:

1. between the database and database user interface: the user can create and/or delete the table structures necessary for storing the location data and can add, modify or delete data from the existing table structures in the database

2. from the wireless transmission medium, to the wireless sensors and then to the server, where it is inserted into the database
3. from the database to the calibration device user interface and then to the server: the calibration device fetches floor data from the database and using it, it creates calibration commands for the server and then it waits for the server's reply
4. from the calibration device user interface to the traffic generator server, over the wireless transmission medium: after sending the calibration command to the server, the calibration device establishes a connection to the traffic generator server. The data sent through this connection is sent using the calibration device's wireless LAN interface, over the wireless transmission medium. These frames will be captured by the wireless sensors and the acquired information will be sent to the Location server

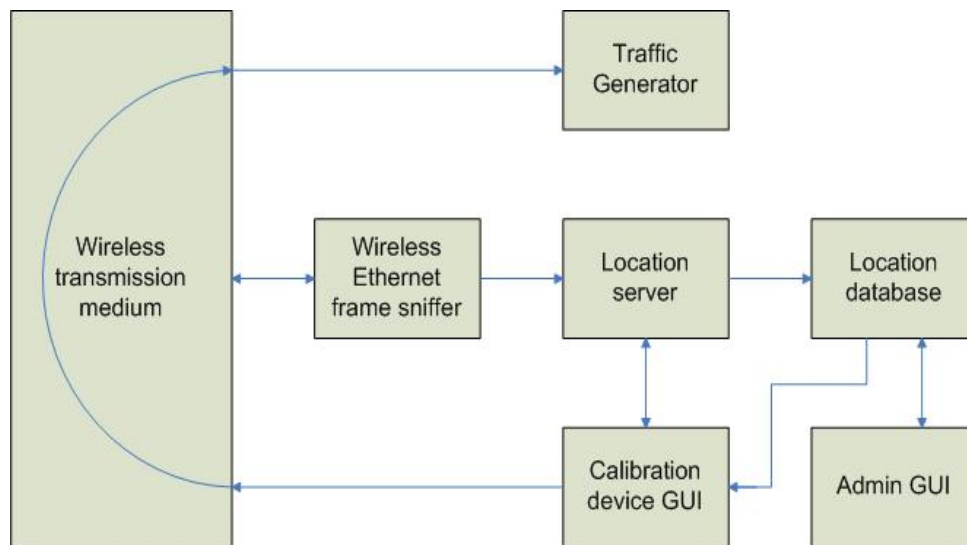


Figure 3.2: Location system software architecture

3.3.4 Location server

Description

The Location server is the central point of the Location system. It acts as a hub between the other components of the system. Software-wise, it is a daemon that runs two processes in the background. The two processes communicate with each other in order to provide the server's functionality. When the server starts, it spawns two child processes and the parent process exists, thus daemonizing the processes.

Before entering normal operating mode, the server initializes all the necessary components, namely:

- the server gets its configuration parameters. For more information about the server configuration phase, see [3.3.4](#).

- before the server processes are spawned, the necessary mutexes are initialized. The mutex descriptors are, later, inherited by the server processes.
- signal handlers are defined for all the signals of interest for each of the server processes.
- the TCP/IP server sockets of both server processes are initialized and the servers begin listening on the ports specified during the server configuration phase.
- the “Sniffing server” (one of the server processes) establishes its connection to the Location server.

All the initializations are done using the configuration options received by the server during the server configuration phase.

The other server process is the “Calibration server”, which receives calibration requests from the calibration device software and, then, parses the calibration command message. After parsing the calibration command, it lets sends it to the other process, which is the “Sniffing server”. Both processes keep track of the current operation mode or task. From here on, the term *task* will be used when talking about the operation more. Just before the “Calibration server” sends the calibration command to the “Sniffing server”, it changes its own current task in order to reflect the action that is performed as a result of receiving the calibration command. The “Sniffing server”, after it receives the command from the “Calibration server”, it changes the current task as well. There are three tasks implemented:

CALIBRATE: the server stores calibration information. It accepts messages coming from the wireless sensors and stores the received information in the *_calibration* table of the database. The task ends either when the requested number of frames were captured or when the calibration task timer expires.

SNIFF: the server stores sniffed wireless information. It accepts messages coming from the wireless sensors and stores the received information in the *_measurements* table of the database. The task never ends.

NONE: no action is performed by the server. The server ignores all messages coming from the wireless sensors.

After the sniffing server finishes performing the current task, it replies to the calibration server and the calibration server replies to the calibration device software. In the current implementation, this happens just for the **CALIBRATE** task because when the current task is **SNIFF**, the current task changes only when a calibration request or a stop command is received and when the current task is **NONE**, the current task changes only when a calibration request or a sniffing command is received.

The “Calibration server” accepts only one connection at a time. The **CALIBRATE** tasks are non-interruptible. That means that if the current task is **CALIBRATE**, no other commands will be accepted until the **CALIBRATE** task ends. If the task is not **CALIBRATE**, new commands are accepted, every new command, modifying the current task accordingly.

The “Calibration server” receives the commands as XML formatted command strings which it parses and from which it extracts the command data. After extracting the command data, it sends the data to the “Sniffing server” by means of a pipe and then it raises a signal according to the task the server processes must switch to. The raising of the signal triggers the execution of the signal handler attached to that respective signal. The signal handler is the one that was

configured during the initialization phase. In this implementation of the Location system, the signal sent by the “Calibration server” is sent only to the “Sniffing server”. The signal handler from the “Sniffing server” extracts the command data from the pipe and begins executing the task it was commanded to perform. The expiration of the calibration interval timer leads to the interruption of the calibration task by means of a signal as well. In this case, the signal is raised by the “Sniffing server” and is handled by both server processes. The two processes have two different signal handlers attached to that signal though.

The “Sniffing server” accepts incoming connections from the system’s wireless sensors. For each sensor, it spawns a new thread that runs as long as the wireless sensor is active if the current designated task is to do calibration for a point or to store sniffed information to be used in the location determination. Before spawning a new thread for the newly connected wireless sensor, the “Sniffing server” sends the server a MAC address to be used in filtering the captured traffic. All the communication with the wireless devices is done in these connection specific threads. If the task is to calibrate a location point, the data is inserted in the *_calibration* table of the currently used schema. If the task is to gather sniffed data, the data is inserted in the *_measurements* table. If the task is to do nothing, then no connection from the wireless sensors will be accepted. For more information regarding the Location database tables see [3.3.5](#).

The communication with the components outside the Location server is done via TCP/IP communication. The communication between the two processes of the Location server is done by using Unix IPC (inter-process communication) mechanisms. The asynchronous events are signaled using Posix signals, the command/reply data is communicated using pipes and the critical sections are isolated using mutexes. For more information on these topics, see [\[2\]](#).

The two servers use the same IP address because they are running on the same machine but they use different ports for receiving incoming connections. The next section will explain how the Location server can be configured and started.

The Location server software makes use of a number of modules that handle different aspects of the server’s functionality. These modules are:

The database module offers an interface to the database server. For more information see section [3.3.4](#).

The mutex module offers an interface to the SVID (*System V Interface Definition*) semaphore implementation. For more information see section [3.3.4](#).

The networking module handles the networking part of the server. For more information see section [3.3.4](#).

The XML parser module offers a wrapper for the Expat XML parser that is designed to parse incoming XML formatted messages. For more information see section [3.3.4](#).

Usage

The Location server uses a configuration file to get its configuration parameters. The configuration file is not hardcoded. It can be specified as a parameter on the command line, when the Location server is started:

```
> ./server -f <CONFIG_FILE_PATH>
```

The file contains the following information:

[General]

```
server_address=<LOCATION_SERVER_IP_ADDRESS>
calibration_port=<CALIBRATION_SERVER_PORT_NUMBER>
sniffing_port=<SNIFFING_SERVER_PORT_NUMBER>
connection_count=<SNIFFING_SERVER_CONNECTION_LIMIT>
database_address=<DATABASE_SERVER_IP>
database_port=<DATABASE_SERVER_PORT>
database_name=<DATABASE_NAME>
database_username=<DATABASE_USERNAME>
database_password=<DATABASE_PASSWORD>
log_file=<SEVER_LOGFILE_PATH>
```

or, the parameters could be entered individually from the command line:

```
> ./server -a <LOCATION_SERVER_IP_ADDRESS>
           -c <CALIBRATION_SERVER_PORT_NUMBER>
           -s <SNIFFING_SERVER_PORT_NUMBER>
           -t <SNIFFING_SERVER_CONNECTION_LIMIT>
           -d <DATABASE_SERVER_IP>
           -r <DATABASE_SERVER_PORT>
           -n <DATABASE_NAME>
           -u <DATABASE_USERNAME>
           -p <DATABASE_PASSWORD>
           -l <SEVER_LOGFILE_PATH>
```

If the -f and some of the other flags are used at the same time, the configurations entered as parameters on the command line take precedence.

If the server is started without specifying some, or any, of the parameters, it will display a warning for each missing parameter and use the hardcoded default value for that parameter.

To stop the Location server use the `killall` command:

```
> killall server
```

For more information about the `killall` command, see the `killall` man page^[4].

Database module

The Location server's database module provides the server with two functions:

1. a function for connecting to a database. It receives as parameters all the connection parameters needed to connect to the database.
2. a function for sending a command to the database, after having established the connection to the respective database. The command can be any correct SQL command. It receives as a parameter the SQL command string that is, then, sent to the database server.

The database module uses a native C interface for connecting to the database server. For more information about the choice of the database management system and about the database programming interfaces, see [3.3.5](#).

For more information regarding SQL, see the SQL appendix.

Mutex module

The Location server's mutex module provides the server with a wrapper to the UNIX SVID (System V Interface Description) semaphore library. It provides four functions:

1. a function for creating and initializing a mutex.
2. a function for getting a mutex (decrementing a mutex). If the mutex is already 0, the thread will have to wait until the mutex value is greater than 0 in order to gain access to the critical section.
3. a function for freeing a mutex (incrementing a mutex).
4. a function for destroying a mutex.

For more information regarding semaphores, see the Semaphores appendix.

Networking module

The Location server's networking module provides the server with all the functions that are needed for interacting with other components of the Location system over TCP/IP networks. It isolates the networking details from the server code by providing the following functions:

1. a function for creating and initializing a server socket. It receives as parameters the IP address of the server, the TCP port number of the server and the maximum number of simultaneous incoming connections allowed by the server.
2. a function for receiving sniffed data from the wireless sensors. It is also used to detect a connection timeout: if the received doesn't receive any packet for some specified time interval, the function will return a value that signifies a connection timeout. The amount of time before timing out is, currently, hardcoded to 10 seconds.
3. a function for sending the calibration device software a reply after a calibration command was executed by the server.
4. a function for sending a wireless sensor the MAC address used in filtering the sniffed wireless frames.

For more information regarding sockets, see the Sockets appendix.

XML module

The Location server's XML module provides the server with one function. This function receives an XML formatted string and, using the data extracted from the string, it fills a structure that is returned to the caller function. It recognizes the following tags:

MAC1 contains the destination MAC address of the captured frame. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the "Sniffing server".

MAC2 contains the source MAC address of the captured frame if it is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the ‘Sniffing server’, or, in case of a calibration command, it contains the filter MAC address to be used when capturing frames.

AP contains the MAC address of the wireless sensor that captured the reported frame. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the ‘Sniffing server’.

BYTE1 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the first byte of the respective MAC address.

BYTE2 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the second byte of the respective MAC address.

BYTE3 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the third byte of the respective MAC address.

BYTE4 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the fourth byte of the respective MAC address.

BYTE5 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the fifth byte of the respective MAC address.

BYTE6 must be inside **MAC1**, **MAC2** or **AP** tags. It contains the sixth byte of the respective MAC address.

X contains the x coordinate of the point that is going to be calibrated. It is used in calibration commands.

Y contains the y coordinate of the point that is going to be calibrated. It is used in calibration commands.

Z contains the z coordinate of the point that is going to be calibrated. It is used in calibration commands.

HEADING contains the orientation of the calibration device at the point that is going to be calibrated. It is used in calibration commands.

COUNT contains the number of frames that are going to be captured in order to calibrate the current calibration point. It is used in calibration commands.

TIMEOUT contains the length, in seconds, of the time interval in which **COUNT** frames have to be captured in order to complete the calibration. It is used in calibration commands.

TASK is used for all commands. It specifies the task that the Location server must switch to in order to follow the respective command. The values it can take are:

0. NONE
1. CALIBRATE
2. SNIFF

DB is used for calibration and sniffing initialization commands. It specifies the database schema to be used for the current task.

RSSI contains the RSSI measured by the wireless sensor for the current frame. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the “Sniffing server”.

SEQ1 contains the first 32b word used in sequencing the captured wireless data. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the “Sniffing server”.

SEQ2 contains the second 32b word used in sequencing the captured wireless data. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the “Sniffing server”.

SEQ3 contains the third 32b word used in sequencing the captured wireless data. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the “Sniffing server”.

SEQ4 contains the fourth 32b word used in sequencing the captured wireless data. It is used in the messages containing sniffed wireless information that are sent by the wireless sensors to the “Sniffing server”.

For more information regarding the sequencing fields, see [3.3.7](#).

Here there are samples of all the XML formatted message types that the Location server receives. The first one is the command which sets the current task to **NONE**:

```
<COMMAND>
  <TASK>0</TASK>
</COMMAND>
```

The second type of command sets the task to **SNIFF** and it specifies what database schema to use for storing the received wireless data:

```
<COMMAND>
  <TASK>2</TASK>
  <DB>tst</DB>
</COMMAND>
```

The third, and most complex one, is the calibration command. It sets the current task to **CALIBRATE**, specifies what database schema to use for storing the received wireless data, specifies which frames are of interest by sending the MAC address of the calibration device in order to use it to filter the wireless traffic and specifies the calibration point coordinates, orientation and end condition:

```
<COMMAND>
  <MAC2>
    <BYTE1>00</BYTE1>
    <BYTE2>14</BYTE2>
    <BYTE3>A4</BYTE3>
    <BYTE4>F2</BYTE4>
```

```

    <BYTE5>A6</BYTE5>
    <BYTE6>CF</BYTE6>
</MAC2>
<X>20</X>
<Y>21</Y>
<Z>1</Z>
<HEADING>3.17152150586994</HEADING>
<COUNT>9000</COUNT>
<TASK>1</TASK>
<TIMEOUT>120</TIMEOUT>
<DB>tst</DB>
</COMMAND>

```

All these commands are received by the “Calibration server”. They can be sent by any device, not necessarily by the calibration device. The fact that the calibration device should start generating wireless traffic after the third type of command is sent should be taken into consideration though!

The last type of message that the Location server receives in its current implementation is the following:

```

<SNIFF>
  <MAC1>
    <BYTE1>00</BYTE1>
    <BYTE2>0C</BYTE2>
    <BYTE3>41</BYTE3>
    <BYTE4>18</BYTE4>
    <BYTE5>A6</BYTE5>
    <BYTE6>F5</BYTE6>
  </MAC1>
  <MAC2>
    <BYTE1>00</BYTE1>
    <BYTE2>14</BYTE2>
    <BYTE3>A4</BYTE3>
    <BYTE4>F2</BYTE4>
    <BYTE5>A6</BYTE5>
    <BYTE6>CF</BYTE6>
  </MAC2>
  <AP>
    <BYTE1>00</BYTE1>
    <BYTE2>0F</BYTE2>
    <BYTE3>BB</BYTE3>
    <BYTE4>0A</BYTE4>
    <BYTE5>D6</BYTE5>
    <BYTE6>38</BYTE6>
  </AP>
  <RSSI>1C</RSSI>
  <SEQ1>20979023</SEQ1>
  <SEQ2>DA143631</SEQ2>

```

```
<SEQ3>C785A3EF</SEQ3>
<SEQ4>00000000</SEQ4>
</SNIFF>
```

This type of messages are received by the “Sniffing server” from the wireless sensors. It contains data obtained from the captured wireless frames.

For more information regarding XML, see the XML appendix.

3.3.5 Location database

DBMS choice

The reasons for choosing PostgreSQL was made, first, because it is an Open-Source DBMS (database management system) and, secondly, because, among the Open-Source DBMSs, it’s the most powerful choice, being able to offer features that enterprise-grade DBMSs like Oracle are offering.

Database architecture

The database can hold more database schemas. Each schema is defined by a name and each of them contain five tables:

<SCHEMA_NAME>_floors: is a table that contains a record for each floor store in the respective schema.

<SCHEMA_NAME>_aps: is a table that contains a record for each access point in the respective schema. Each access point uses the number of the floor it’s on as a foreign key.

<SCHEMA_NAME>_calibration: is a table that contains a record for each received message containing captured frame data and positioning information. The data in this table is used in populating the **<SCHEMA_NAME>_fingerprint** table.

<SCHEMA_NAME>_fingerprint: is a table that contains a record for each calibration point. The record stores a model which it associates with the respective position informaton.

<SCHEMA_NAME>_measurements: is a table that contains a record for each received message containing captured frame data.

Programming interfaces

One of the strong points of PostgreSQL is that it there are client libraries for a wide variety of programming languages.

The PostgreSQL server comes with a C library included. Its name is *libpq*. To use it from C, *libpq-fe.h* must be included and, then, at build time, the *-lpq* flag has to be give to the linker, in order to be able to link the *libpq* library.

In Perl, the DBD-Pg module can be used. It is based on the DBI Perl module and specification. To use it, `use DBI;` must be added at the beginning of the script and, then, when connecting to the database, the DBD-Pg driver must be specified.

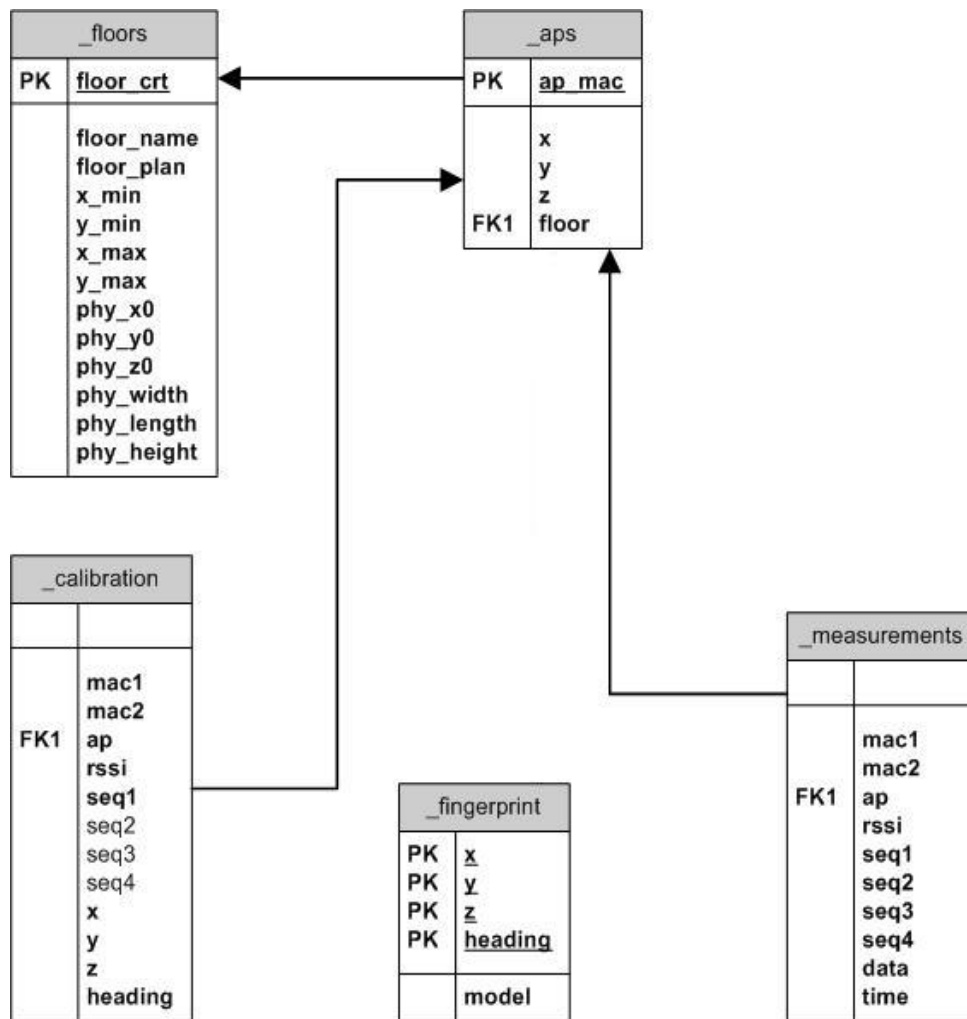


Figure 3.3: Location database architecture

3.3.6 Traffic generator server

The traffic generator server is used by the calibration device's software to generate wireless network traffic. It creates a server socket for the IP address and port number sent as parameters from the command line and it accepts only one connection at a time. The connection is closed when the client decides to close it. While the connection is up and running, the server reads the messages coming from the calibration client and discards them.

Usage

The traffic generator server has to be started using the following command line command:

```
> ./trafgen <IP_ADDRESS> <PORT NUMBER>
```

3.3.7 Wireless medium sniffer

The sniffer software is based on the Atheros reference design for Wireless LAN access points. The modifications to the Atheros reference design were done in order to change the operation mode of a hardware device which is, by design, a Wireless LAN access point. To reach this goal, a series of modifications were done. Some are just configuration changes, other are modifications done in the source code.

The modifications done in the code consist in making the device receive and interpret all 802.11 frames that are detected on the wireless medium. The only type of filtering that is implemented is a mechanism that can filter the received frames by their source MAC address. The modification done to allow the capture of all incoming frames is done in the *vportEnd.c* file, which controls the behaviour of the Wireless LAN interfaces.

All the received packets are inspected in order to extract sequencing information from the frame. There are 4 4-byte words that are used for sequencing:

1. the first word is composed as following: the higher part is a 2-byte word that is the 802.11 sequence field and the lower part is a 2-byte word that is extracted from the TCP header, in the case of frames carrying IP packets with TCP segments inside. The word consist of the TCP header checksum.
2. the second 4-byte word is the TCP sequence number in the case of frames carrying IP packets with TCP segments inside.
3. the third 4-byte word is the first 4 data bytes in the case of frames carrying IP packets with TCP segments inside.
4. the fourth 4-byte word is the 802.11 frame's FCS (Frame Check Sequence).

This information is collected in the function which is queued as a job by the interrupt service routine that handles incoming frames. This function is the entry point of all received frames.

Other modifications done in the source code are the removal of the 802.11a capabilities, by removing the initialization of the 5Ghz radio and of the support for web-based firmware update, web server and telnet server.

A new component is created and added to the OS build. The component is made up of two files: a source file and a header. This component provides communication capabilities with the Location server, using TCP/IP. It exports two variables:

1. a variable which is a VxWorks message queue descriptor. It is used to send captured data from the function that processes the incoming frames to the task that handles the communication with the Location server.
2. a variable which stores a MAC address. If this MAC address is different from FF:FF:FF:FF:FF:FF, only the frames that have the source MAC address identical to the one stored in this variable will be processed.

The component has two functions:

1. one that initializes the component at boot time. It creates a mutex that is going to be used to isolate critical sections during the task's operation, initializes the message queue and spawns the task that handles the communication with the Location server.
2. one that contains the code that is run by the task that handles the communication with the Location server. It first configures a socket that is to be used to connect to the Location server and, after that, it enters two imbedded cycles. In the outer cycle, the sensor tries to successfully open a socket and establish a connection to the Location server. If and when the connection is established, the code enters the inner cycle, where it uses the established connection to send the data that it finds in the message queue. The code exits the inner queue after it encounters 10 transmission error. When that happens, it closes the connection, exits to the outer cycle and tries to establish a new connection to the Location server

The rest of the modification are configuration file modification or configuration file-related modifications. The configuration file-related modifications is adding support for handling two new settings in the configuration file:

- the Location server IP address: contains the IP address of the Location server to which the wireless sensor is supposed to connect to
- the Location server port number: contains the port number of the Location server to which the wireless sensor is supposed to connect to

Apart from adding these new settings, some other configurations have been made:

- the transmission rate have been set to 1Mbps
- the SSID broadcast has been disabled
- the power control has been disabled
- the channel has been set to channel 6

3.3.8 Location database user interface

The Location database user interface is, as the name implies, designed for administering the Location database. The interface is made up of 6 panels, each purposefully designed for administering different aspects of the database.

- The **Admin** panel is designed for creating new database schemas in the database, deleting database schemas from the database and for selecting a schema to work with. The schemas aren't actually using PostgreSQL's schema mechanism. Instead, when it creates the tables of the database, it appends the database name before the table name. By doing this, the tables belonging to the same schema can be easily identified and multiple groups of tables can be created in the same database. An advantage of doing this, instead of using PostgreSQL's schema mechanism is that it's independent from the underlying database management system. For example, MySQL doesn't have a schema mechanism implemented, and porting the application to MySQL would be more difficult. Nevertheless, this mechanism provides the ability to use more table groupings in the same database but it doesn't provide the rest of the facilities that schemas provide. This is not really a problem, taking into account the fact that only table grouping is required for this application.

All the other panels operate only on the currently selected schema.

- The **Floors** panel is designed to manage the floor data in the database. It can be used to inspect the floors that are already stored in the database, to add new floors, to update existing floors or to delete floors from the database.
- The **APs** panel is designed to manage the access point data in the database. It can be used to view the placement of the access points, to view lists of access points that match certain criteria, to add new access points or to delete access points that match certain criteria from the database. The access point placement can be viewed on a per floor basis but the other operations are independent of the floor the access point is on.
- The **Calibration** panel is designed to manage the calibration point data in the database. It can be used to view or to delete calibration points that match certain criteria from the database.
- The **Fingerprint** panel is designed to manage the fingerprint point data in the database. It can be used to view or to delete fingerprint points that match certain criteria from the database.
- The **Measurements** panel is designed to manage the measurements stored in the database. It can be used to view or to delete measurements that match certain criteria from the database.

Switching between panels is done using the View menu from the menu toolbar. Initially, the **View** menu is disabled. This is done because the user must first choose a schema and only after having selected a schema, he can begin working with the database. The **File** menu can be used to exit the program.

The Admin panel is designed to create database schemas, to delete existing schemas and to select one of the existing schemas for use.

The Floors panel can be used to view the information stored in the database regarding the floors that have been already added, to add new floors, to delete floors and to modify existing floors. Selecting existing floors can be done from the radio button group on the upper left side of the panel. The command buttons can be found on the lower left side of the panel.



Figure 3.4: Admin GUI: Admin panel

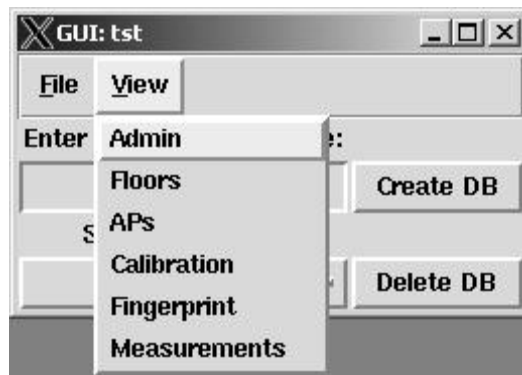


Figure 3.5: Admin GUI: Admin panel's available options



Figure 3.6: Admin GUI: Floors panel

The **APs panel** can be used, after there is already floor information stored in the database, to place access points across the stored floors. The x and y coordinates can be inputted by clicking the floor plan of the currently selected floor. The z coordinate has to be inputted manually, taking into account the correct z values for the current floor. Another very important piece of information that has to be specified for adding a new access point is the MAC address of the access point.

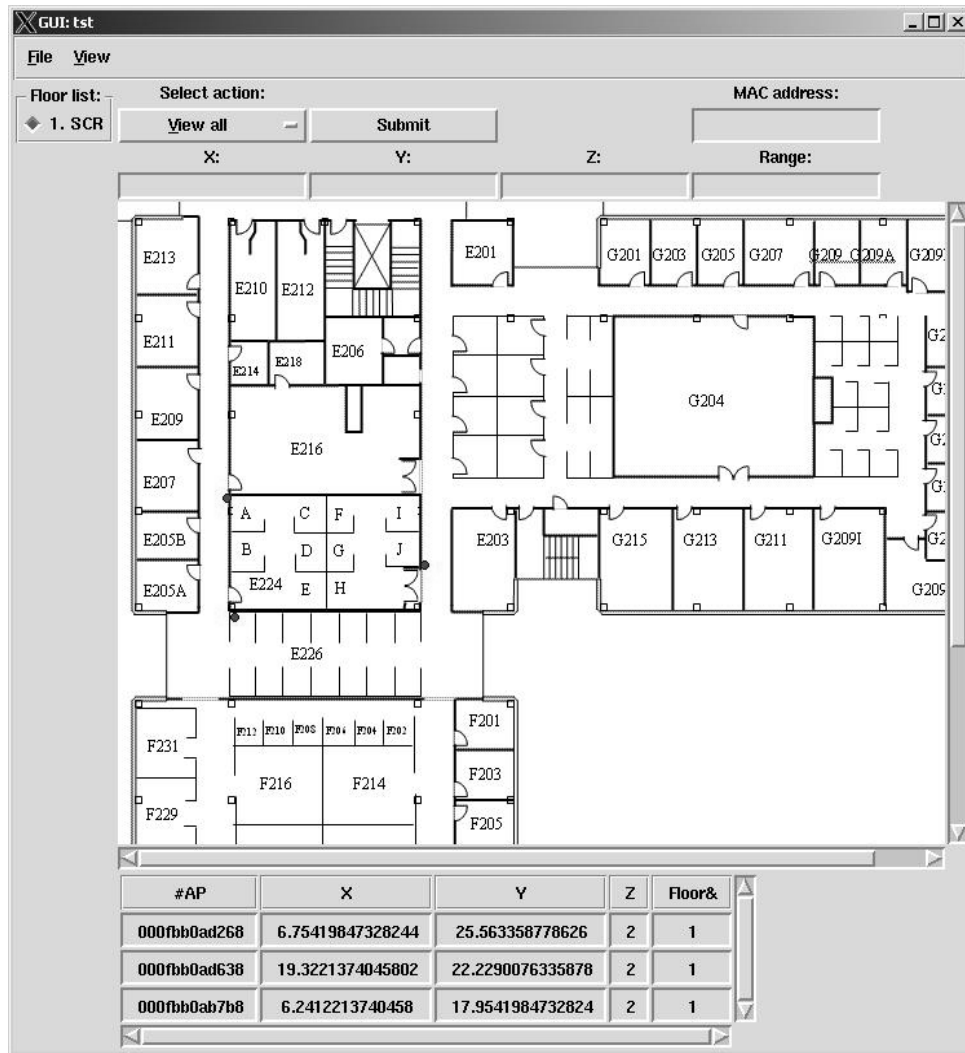


Figure 3.7: Admin GUI: APs panel

Adding access points isn't the only thing that can be accomplished via the APs panel. Using this panel, the positions of the APs placed on the current floor and a list containing all the access point information in the database can be displayed. The display can show all access points or just a subset of all the access points. The filtering can be done by filling in any of the data entry boxes. Only the access points matching those values will be shown.

Another action that can be performed from this panel is to delete access points from the database. The delete can be global (delete all access points) or filtered, just like the display.

The **Calibration, Fingerprint** and **Measurements** panel can just display lists or filtered lists of records from the database's *_calibration*, *_fingerprint* and *_measurements* respectively. Due to the fact that a large number of records will be usually present in these tables, these three panels

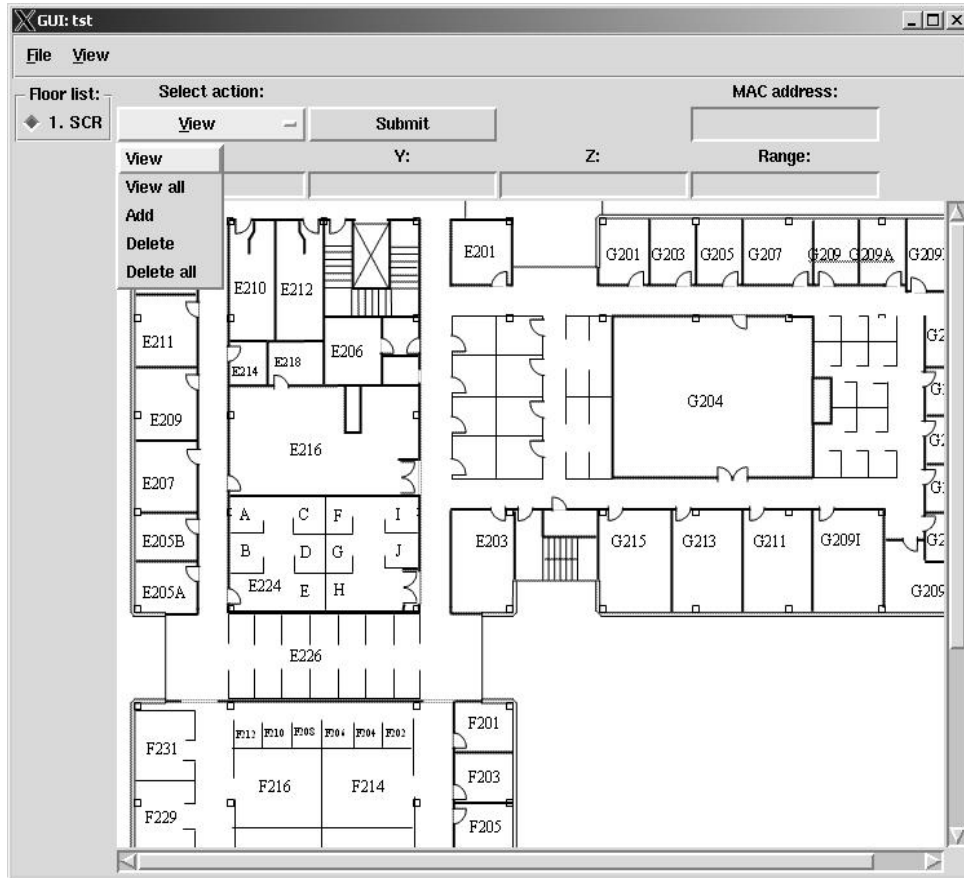


Figure 3.8: Admin GUI: APs panel's available actions

have't been updated lately and because the response time would be too long. For development, it is much faster to use the command line database interface.

The Admin GUI uses a configuration file to get its configuration parameters. The configuration file is hardcoded to `../admin.conf`. The file contains the following information:

```
[General]
database_address=<DATABASE_SERVER_IP>
database_port=543<DATABASE_SERVER_PORT>
database_name=<DATABASE_NAME>
database_username=<DATABASE_USERNAME>
database_password=<DATABASE_PASSWORD>
work_dir=.
```

The `work_dir` parameter specifies the path to the directory where the images containing the floor plans extracted from the database will be stored.

The Admin GUI is developed in Perl/Tk. For more information about Perl/Tk, see [7].

3.3.9 Calibration device user interface

The calibration device user interface is used on the calibration device to perform the Location system calibration. It allows the user to select the desired database schema:



Figure 3.9: The Calibration GUI



Figure 3.10: Choosing the floor

and, after that, to select the desired floor:

Having made these choices, the user can begin start adding calibration points. To add a calibration point, the user can specify all the coordinates by hand, using the entry fields. Alternatively, the user can click the floor plan in order to input the x, y and heading coordinates. The x and the y are determined by the closest grid point to the place where the user clicked. The heading can be specified by clicking and, instead of releasing the button, dragging in the desired direction.

After selecting the desired coordinates, the user has to fill in three more parameters:

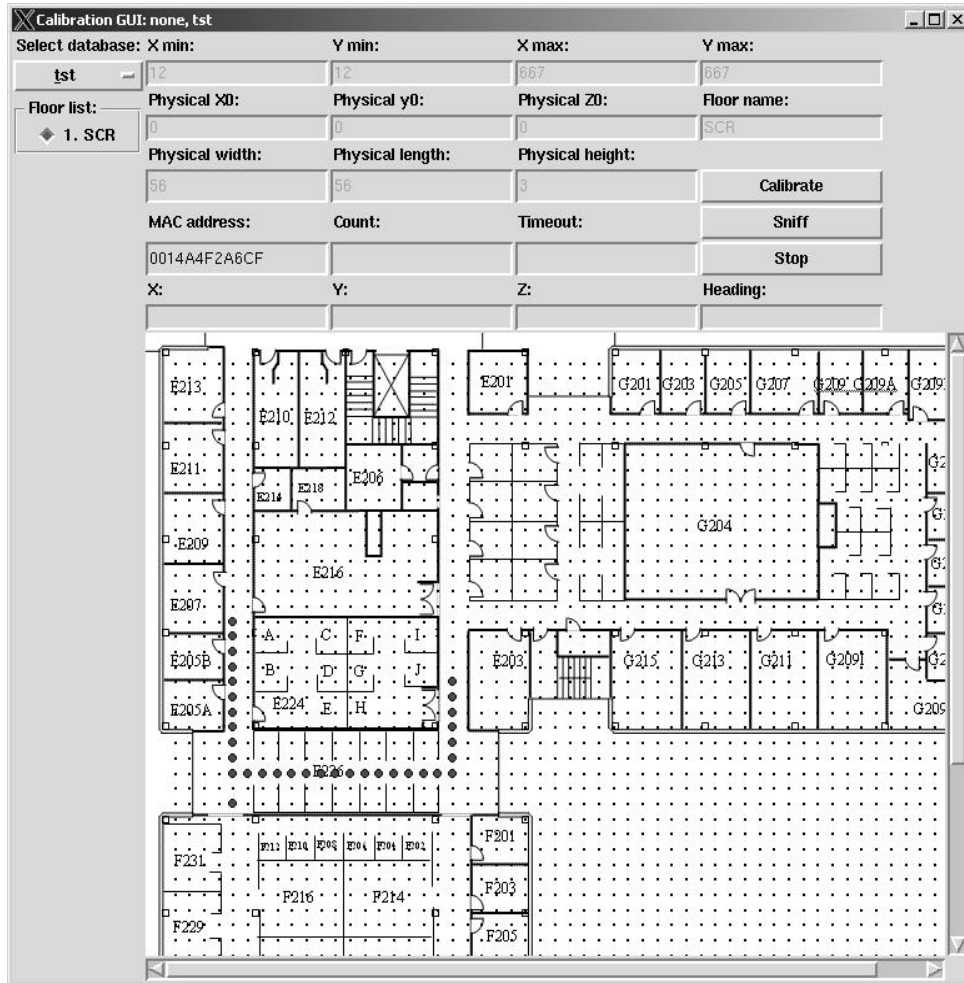


Figure 3.11: The selected floor plan. It shows the positions present in the database

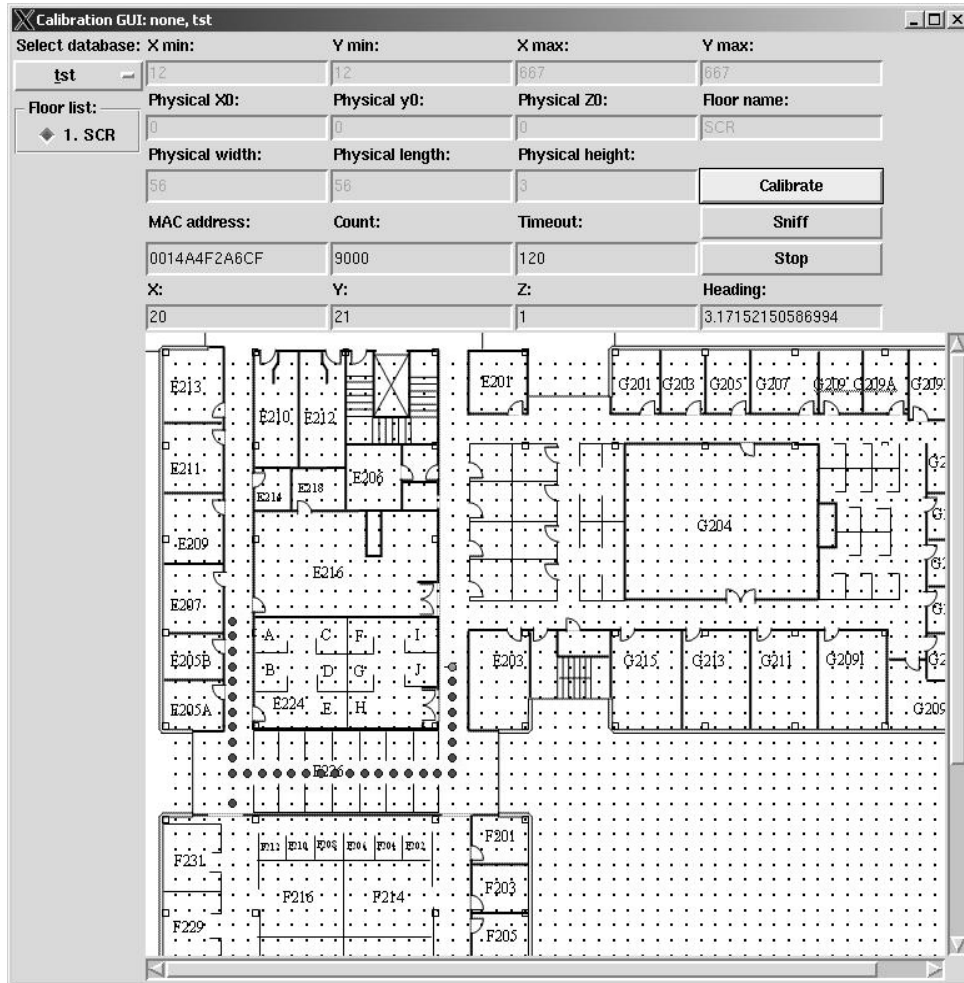


Figure 3.12: Adding a new point

1. the MAC address of the calibration device
2. the number representing, on one hand, the number of packets to be sent, and on the other hand, the number of reported sniffed frames by all the wireless sniffers in the system
3. the timeout for the calibration process: if the server doesn't receive the indicated number of packets before the timer expires, the calibration process will stop and report the number of received frame information messages up to that moment, if the number of expected received frame information messages is received prior to the expiration of the timer, the calibration process stops yet again and the time remaining until the expiration of the timer is reported

The actions done in the calibration GUI have an echo in the command line session that spawned the GUI. First, the name of the selected database schema is displayed. Then, the pixel coordinates of the selected calibration point on the floor plan is displayed:

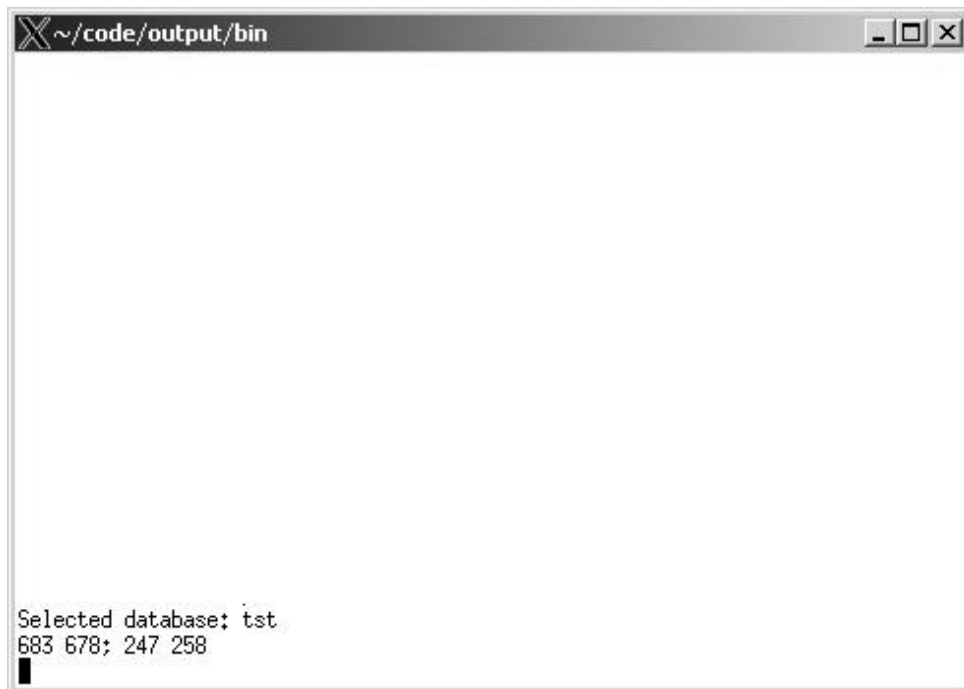


Figure 3.13: CLI: the database schema was selected

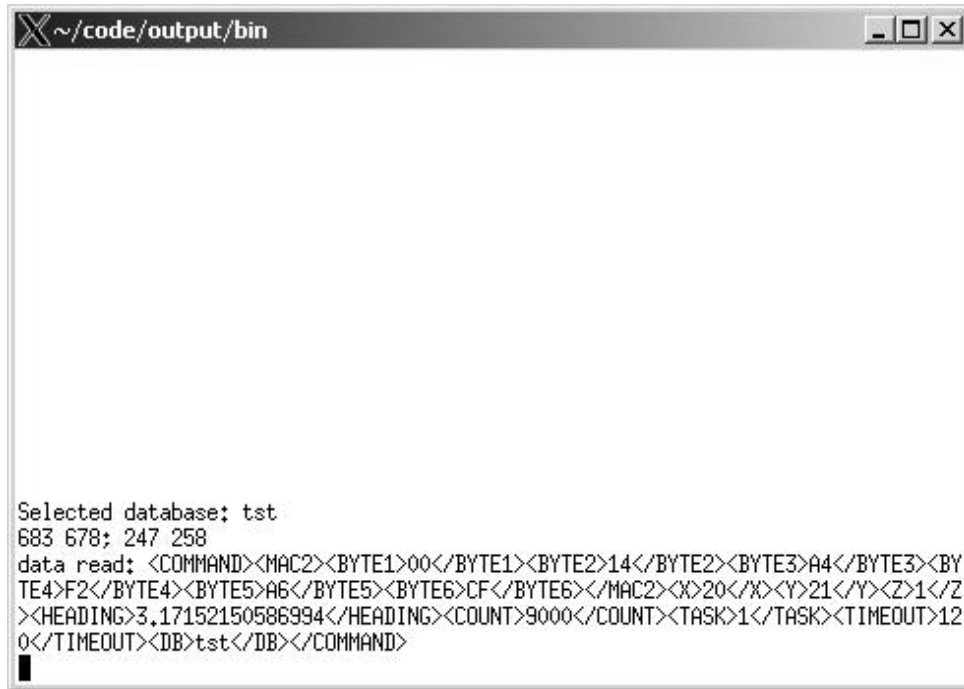
The calibration command is sent to the Location server:

The reply is received from the server:

The calibration device GUI works on two threads: one is the GUI which receives input from the user and the other is a thread that handles just the network communication with the Location server. The communication between the two threads is done using pipes. For more information on Perl networking and IPC (inter-process communication) see [5].

The calibration device GUI uses a configuration file to get its configuration parameters. The configuration file is hardcoded to `../calibrate.conf`. The file contains the following information:

```
[General]
server_address=<LOCATION_SERVER_IP_ADDRESS>
calibration_port=<LOCATION_SERVER_PORT_NUMBER>
```



```
~/code/output/bin

Selected database: tst
683 678; 247 258
data read: <COMMAND><MAC2><BYTE1>00</BYTE1><BYTE2>14</BYTE2><BYTE3>A4</BYTE3><BY
TE4>F2</BYTE4><BYTE5>A6</BYTE5><BYTE6>CF</BYTE6></MAC2><X>20</X><Y>21</Y><Z>1</Z
><HEADING>3.17152150586994</HEADING><COUNT>9000</COUNT><TASK>1</TASK><TIMEOUT>12
0</TIMEOUT><DB>tst</DB></COMMAND>
```

Figure 3.14: CLI: the calibration command was sent to the Location server



```
~/code/output/bin

Selected database: tst
683 678; 247 258
data read: <COMMAND><MAC2><BYTE1>00</BYTE1><BYTE2>14</BYTE2><BYTE3>A4</BYTE3><BY
TE4>F2</BYTE4><BYTE5>A6</BYTE5><BYTE6>CF</BYTE6></MAC2><X>20</X><Y>21</Y><Z>1</Z
><HEADING>3.17152150586994</HEADING><COUNT>9000</COUNT><TASK>1</TASK><TIMEOUT>12
0</TIMEOUT><DB>tst</DB></COMMAND>
received reply
syswriting: 6338
0.000000done!
```

Figure 3.15: CLI: the calibration device received a reply from the server

```
trafgen_address=<TRAFFIC_GENERATOR_SERVER_IP_ADDRESS>
trafgen_port=<TRAFFIC_GENERATOR_PORT_NUMBER>
database_address=<DATABASE_SERVER_IP>
database_port=543<DATABASE_SERVER_PORT>
database_name=<DATABASE_NAME>
database_username=<DATABASE_USERNAME>
database_password=<DATABASE_PASSWORD>
work_dir=.
```

The `work_dir` parameter specifies the path to the directory where the images containing the floor plans extracted from the database will be stored.

The calibration device GUI is developed in Perl/Tk. For more information about Perl/Tk, see [7].

3.3.10 Fingerprint generator tool

The `extract.pl` script extracts the data stored in the `_calibration` table and filters it so that only the information regarding the packets captured by all wireless sensors is kept. Using this information, the script creates a number of files, among which, the most important are:

fingerprint.data: here is written the fingerprint data that is going to be stored in the `_fingerprint` table. The data is already formatted as “SQL INSERT INTO” commands that need only to be run

test.data: this is a file containing a line for each frame that has been captured by all wireless sensors in the system. The first columns of the file contain the RSSI of the respective frame as detected at each wireless sensor. The last four columns contain the calibration coordinates from where the respective frame was sent

I_<x>_<y>_<z>_<heading>_<ap>: files containing the RSSI of all packets received from the $(x,y,z,heading)$ position by the wireless sensor `ap`

The script needs Matlab in order to successfully run. It uses Matlab to compute the fingerprint data.

The `extract.pl` script uses a configuration file to get its configuration parameters. The configuration file is hardcoded to `../extract.conf`. The file contains the following information:

```
[General]
database_address=<DATABASE_SERVER_IP>
database_port=543<DATABASE_SERVER_PORT>
database_name=<DATABASE_NAME>
database_username=<DATABASE_USERNAME>
database_password=<DATABASE_PASSWORD>
```

The script receives another set of parameters from the command line. The first and the only non-optional parameter is the name of the used database schema. The user can specify, at the command line if the script should extract the data from all positions in the `_calibration` table or just from one individual position. If only one position is required, the first 4 parameters are the position’s coordinates. If the heading is not important, all headings from the same (x,y,z)

coordinates can be “flattened” and the placeholder value of 255 will be assigned to the heading in the results. To accomplish this, the last parameter (replacing the heading argument) on the command line has to be 'noh' (no heading), without the quotes. This argument can be specified either when extracting all data or when extracting data from only one position. When extracting all data, the argument doesn't replace any other parameter but it is added immediately after the database schema's name. When extracting data from only one set of (x,y,z) coordinates, another parameter can replace the heading: 'all'. By using this parameter, all the positions with different headings but with the specified (x,y,z) coordinates will be extracted.

3.3.11 Perl Location libraries

Perl database module

The Perl database module provides the Perl based components with access to the information stored in the Location database. In order for it to function correctly, the module requires two extra Perl modules to be installed on the machine that uses the `Location::DB` Perl module. These modules are:

DBI is needed if any Perl database interface is to be used.

DBD::Pg is the module implementing the Perl interface for PostgreSQL databases.

The `Location::DB` module provides a large number of functions for interacting with the Location database, among which there are:

- a function for connecting to a database from a database server, using a specified user and password.
- a function for adding a new database schema.
- a function for deleting a database schema.
- functions for getting the whole content or the filtered content of any of the tables of the selected schema.
- functions for adding records to any of the tables of the selected schema.
- functions for deleting records from any of the tables of the selected schema.
- a function for executing a user specified “SELECT” command (it returns an array containing the result of the “SELECT”).
- a function for executing a user specified command of any kind (it returns 0 if the operation was successful or -1 if it wasn't).

For more information regarding SQL, see the SQL appendix.

Perl networking module

The Perl networking module provides the Perl based components the necessary functions for network communications. It implements TCP/IP communications by using sockets. To accomplish that, it uses the `Socket` module. The `Socket` module is included in any modern Perl distribution. It also needs the `Location::XML` module. The `Location::Net` module is necessary for parsing the replies to the calibration commands that are sent by the “Calibration server”.

The `Location::Net` module implements two functions:

1. a function for connecting to the specified “Calibration server”, sending a calibration command containing an XML formatted string, waiting for the reply, parsing the reply and returning the reply data to the caller function. The connection parameters for connecting to the “Calibration server” and the calibration command string are sent as parameters to this function. The parsing of the reply is done with the help of the `Location::XML` module.
2. a function for connecting to the specified traffic generator server and sending a specified number of TCP/IP packets to this server. The number of packets and the timeout between the sending of two consecutive packets are sent as parameters to this function.

For more information regarding sockets, see the Sockets appendix.

Perl XML module

The Perl XML module provides the Perl based components with a XML parsing interface. The `Location::XML` module is, actually, a wrapper for the `XML::Parser::Expat` module that is included in any modern Perl distribution. The `Location::XML` module exports one function. This function receives an XML formatted string and, using the data extracted from the string, it fills a structure that is returned to the caller function. It recognizes the following tags:

COUNT contains the number of captured frame data messages that the “Sniffing server” received before the calibration timer expired.

TIME contains the remaining time (in seconds) to the expiration of the calibration timer from the moment when the required number of captured frame data messages were received.

Here is a sample of XML formatted reply received from the “Calibration server”:

```
<REPLY>
  <COUNT>6338</COUNT>
  <TIME>0.000000</TIME>
</REPLY>
```

For more information regarding XML, see the XML appendix.

3.3.12 Building the software

Building the servers and tools

Building all the Location system software, with the exception of the sensor software can be as simple as:


```
> cd <LOCATION_HOME>
> cd src
> make all
```

This is the case because the whole build process is handled by a series of makefiles. The build environment of the Location system (minus the sensor software) is the following:

```
<LOCATION_HOME>/
  output/
    bin/
    doc/
    lib/
  src/
    admin/
    calibrate/
    doc/
    extract/
    lib/
    scripts/
    server/
    trafgen/
```

All the sources of the Location system can be found in the `<LOCATION_HOME>/src` directory. In this directory there is a subdirectory for each component, one for the documentation (that is more or less unused at the present moment) and one for a series of auxiliary scripts by other parts of the Location tools.

Apart from the components' directories there is one more thing in the `<LOCATION_HOME>/src` directory: the makefile for the entire build environment. From this file, the build of every component is started and all the results are gathered and sent to the `<LOCATION_HOME>/output` directory. The `<LOCATION_HOME>/output` directory is the place where the binary programs and runnable scripts reside.

The `make` program uses the makefiles in order to automatize the build process, the cleaning process if the user wants to revert to the raw sources and the creation of the documentation. At the present moment the build environment can handle the documentation process but writing the documentation is in the *Work in progress* phase.

Each makefile contains a number of targets. Each target has two parts: the first is the specification of the files that have to exist in order for the target to be run, or the targets that have to run before running the respective target. The second part of a target is a series of commands that have to be run for the respective target. These commands are regular command line commands or program calls. If a target doesn't have any prerequisites, the target is always run. If the prerequisites contain only files and the respective files haven't been modified since the last make run, the target won't be run. This is very helpful when working with large projects because it allows to recompile just the necessary modules when the modifications are local to just some of the source code files. In this case the build time can be reduced drastically. A target can be the filename of the file that will be created after running the respective target or it can be a phony target. Phony targets aren't files to be created. They are more like a name assigned to a batch of commands that are to be run when the execution of the respective target is requested. To execute a specific target, the command is:

```
> make <TARGET_NAME>
```

If no target is specified when running make:

```
> make
```

then the first target in the makefile will be run.

The make program uses makefiles to know what commands to execute. If make is called without parameters, like in the snippets presented earlier, it will look for a file called `Makefile` in the current directory. If it won't find the default makefile, it will display an error message and exit. If another makefile needs to be specified, the `-f` flag and the file path of the requested makefile has to be specified on the command line.

If no modifications to the build environment are required, the only two necessary targets are:

all: this can be written explicitly or omitted because it is always the first target in all the makefiles. It is used to compile/recompile the whole Location system and system tools.

clean: this target is used to remove all the files and directories created by the build system, practically reverting to a *clean* state of the sources.

A complete reference of *make* can be found at [3].

Building the wireless sensor software

The wireless sensor software is a modified version of the *Atheros AP reference code*. The Atheros code, itself, is based on Wind River's *VxWorks 5.4* real-time operating system. In order to build the wireless sensor software, the tools to building a VxWorks OS image are required. In this case, the tool used for this job was Wind River's *Tornado* integrated development environment.

To create the wireless sensor software, the *ap-ar5312* project was used as a starting point. This is the project configured for Atheros AR5312 chipsets.

The first modification was to add a new definition to be sent to the compiler. The newly defined label is *LOCATION* and it is used to delimit all the code that is used for building the wireless sensor software. Every block of code that was added, is delimited using preprocessor commands and this new definition:

```
#ifdef LOCATION
/*
 * some new code
 */
#endif
```

or, if the code needed to be omitted from the build:

```
#ifndef LOCATION
/*
 * some old code
 */
#endif
```

or, if some code had to be replaced:

```

#ifdef LOCATION
/*
 * some new code
 */
#else
/*
 * some old code
 */
#endif

```

The modifications of the source files were done in order to:

- add functions for displaying and for setting the IP address and port number of the Location server to which the wireless sensor will try to connect
- add function for writing in and reading from the configuration file the Location server connection parameters (IP address and port number)
- add code in the function that processes all the received frames. It extracts relevant information from the frame and puts that information in a message queue, to be sent to the Location server
- modify the filtering of the incoming frames so that the wireless interface will function in promiscuous mode
- remove the telnet server, the web server and the ability to do a web-based firmware update
- remove the support for the 802.11a radio
- add a function that extracts sequencing data from received packets
- define a new symbol for the priority of the Location service task

On top of those, an entirely new component was added to the OS image build. More details about the new component can be found in [3.3.7](#). For more information on how to add a new component to the VxWorks build see [\[6\]](#).

Deploying the wireless sensor software

The deployment of the wireless sensor software on the hardware devices is a 5 step process. Exhaustive information about configuring the device's bootloader and the connection to the serial port can be found in [\[6\]](#). Here is an overview of the entire process:

1. FTP server: the OS image file will be downloaded from a FTP server, that's why an FTP server has to be configured, and the OS image has to be accessible using this server. If you use for the OS image deployment the same machine that was used for building the OS image, it is good to know that when the Tornado IDE is installed, a FTP server is installed as well. The best course of action would be configuring the home directory of the FTP server to coincide with the directory where the OS image is built. The OS image file name can be changed, but our case, the used OS image file is *vxWorksCompressed*

2. Network connection: the hardware device has to have a working network connection to the machine running the FTP server
3. Serial port connection: using a null modem cable, connect to the serial port of the hardware device. The configuration parameters for the serial link are:
 - baud rate: 9600
 - parity: no parity bit
 - stop bit(s): 1 stop bit
 - flow control: no flow control
4. Bootloader configuration: after making the connection to the device's serial port and configuring your terminal emulator software with the specified parameters, power up the device. At the beginning of the boot process, a counter will show up. Press any key before the countdown expires to interrupt the boot process and enter the bootloader configuration mode. Here, you will have to configure the bootloader to boot from the FTP server. After the configuration is done, you can boot using the selected method by entering @ from the console.
5. Configure the wireless sensor: this step can be done either before or after starting to use a new OS image because the configuration is non-volatile. All the configuration data is stored in a file in flash. The following commands are available to the user for configuring the connection to a Location server

get locaddr - it can be used to see what is the IP address of the Location server currently used

get locport - it can be used to see what is the port number of the Location server currently used

set locaddr IPAddress - it can be used to set the IP address of the Location server to be used

set locport PortNo - it can be used to set the port number of the Location server to be used

An example of configuring the Location connection settings is the following (The command are entered using the sensor device's console connection):

```
> set locaddr 192.168.1.18
> set locaddr 5430
```

Chapter 4

Experimental results

The calibration of the system was done in the area shown in figure 3.11. The area where the system was tested is the first floor of the Siemens Corporate Research building, in Princeton, New Jersey. The test area was covered by 3 wireless sensors. In this area there are 32 positions 1m apart and a 33rd position 2m apart from the rest. At each position the number of orientations may vary. With only one exception, at all calibration points, 4 orthogonal orientations were used during the calibration phase. The exception mentioned earlier consists in the fact that at coordinates $(5,30,1)$ (the 33rd position) the calibration was done for 8 orientations instead of just 4. The orientations at those coordinates are 45 degrees apart.

At each calibration point $((x,y,z,heading)$ coordinate set) 9000 frames were captured by the 3 wireless sensors covering the area. That means that each sensor captured around 3000 frames. As mentioned earlier, in 3.3.10, only the frames that were captured by all wireless sensors were kept. The rest were filtered out.

All the data that was used in computing the fingerprint is then used to test the performance of the location estimation. The test consisted in running each set of values from the *fingerprint.data* through the location estimation algorithm and compare the estimated position with the position stated in the file (the real position).

Before doing the calibration, the assumption that the RSSI distributions were following a Gaussian model was made. The hypothesis was infirmed by the results: many of the obtained RSSI distributions are not Gaussian. They are a Gaussian mix at best, because they are multimodal, having 2 modes most of the time:

A hypothesis that was confirmed by the experiment was that the mobile device's orientation influences the RSSI of the frames sent by the device. The three groups of histograms from above were obtained at the same (x,y,z) coordinates but using orientations that were 90 degrees apart.

2 different methods of storing the fingerprint and estimating the locations were used. The first was the method that hypothesised that the RSSIs of the frames have a Gaussian distribution. The results can be found in table 4.1.

The plot of the values from table 4.1 generates the figure 4.4.

The second method used the whole RSSIs' histogram at each $(x,y,z,heading)$ coordinate as fingerprint. Using the fingerprint, a Bayesian inference mechanism was used to estimate the location. The results can be found in table 4.2.

The plot of the values from table 4.2 generates the figure 4.5.

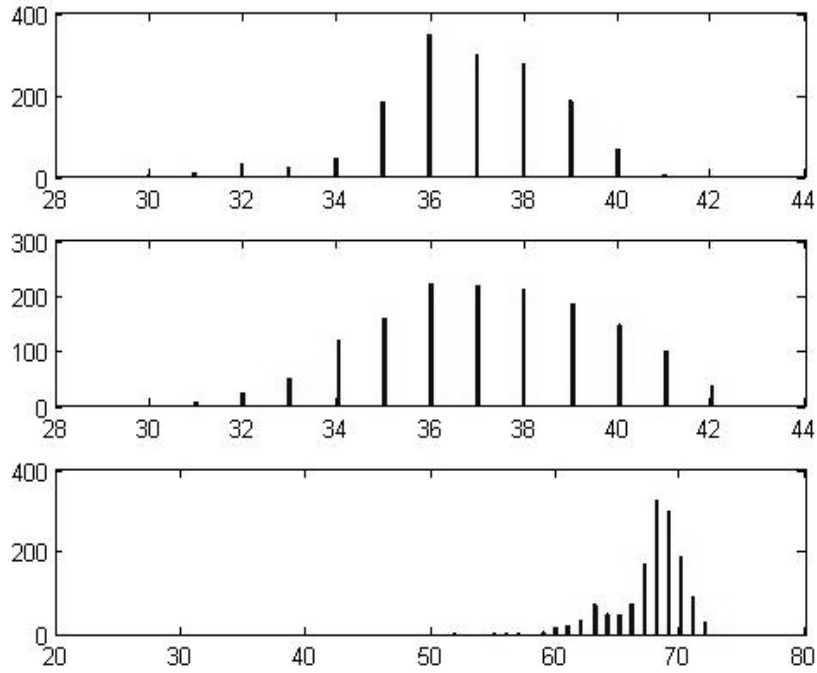


Figure 4.1: The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= -0.02

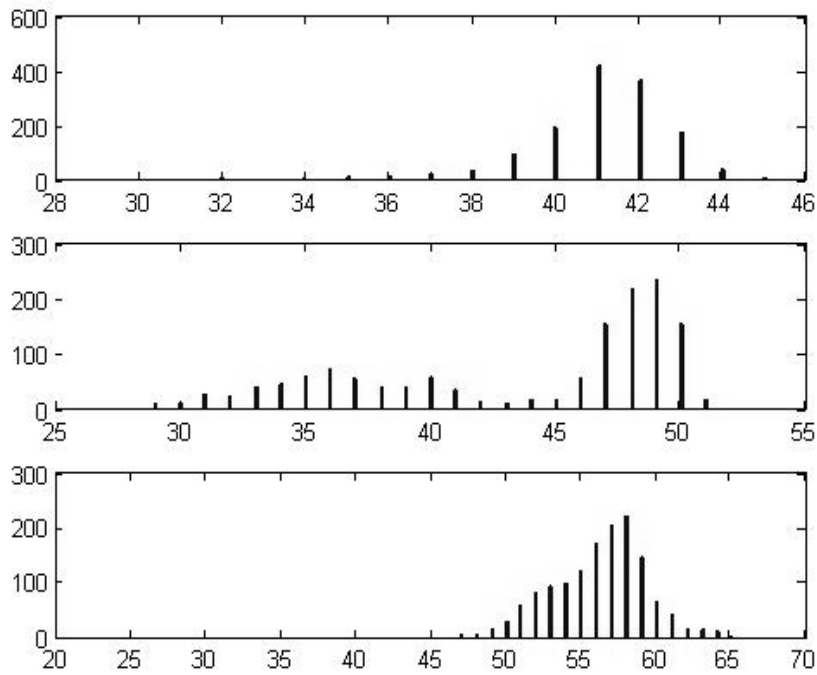


Figure 4.2: The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= -1.55

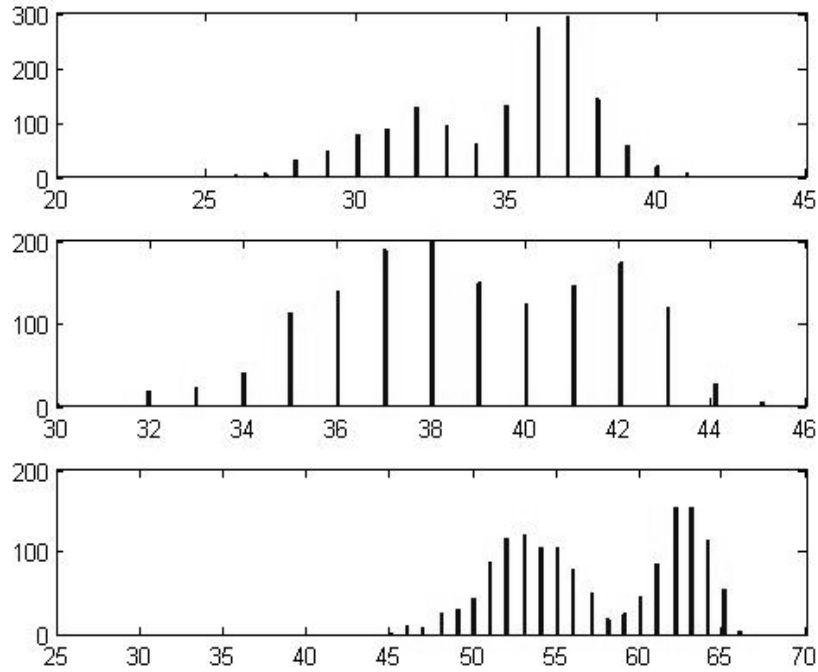


Figure 4.3: The histograms for the position at coordinates $x=20$, $y=22$, $z=1$ and heading= 3.15

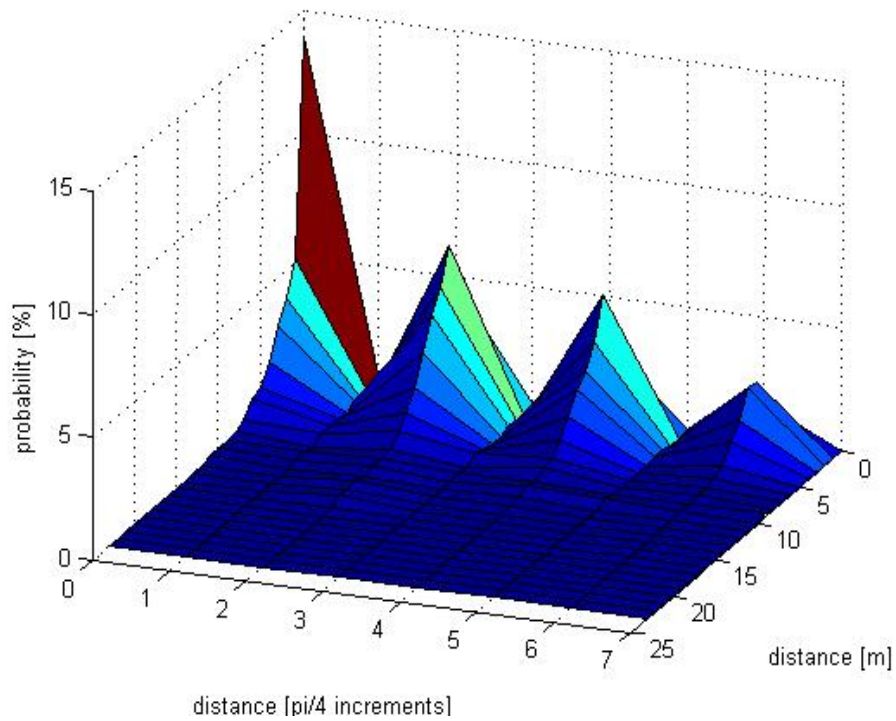


Figure 4.4: Distribution of probability for the accuracy of the location estimation when using a Gaussian model

Distance error[m]	Total distance error probability[%]	0 angle error probability[%]	$\frac{\pi}{4}$ angle error probability[%]	$\frac{\pi}{2}$ angle error probability[%]	$\frac{3\pi}{4}$ angle error probability[%]	π angle error probability[%]	$\frac{5\pi}{4}$ angle error probability[%]	$\frac{3\pi}{2}$ angle error probability[%]	$\frac{7\pi}{4}$ angle error probability[%]
0	23.62	58.94	1.53	19.64	0.87	12.30	0.38	6.30	0.03
1	19.91	26.37	0.00	32.94	0.00	27.09	0.00	13.59	0.00
2	16.06	24.25	1.12	32.27	1.11	23.69	0.13	17.43	0.00
3	13.02	23.71	3.53	33.33	2.31	22.45	1.20	13.18	0.28
4	9.21	20.62	2.83	33.34	1.75	26.62	1.87	12.28	0.68
5	5.92	20.35	5.48	34.32	2.35	27.75	0.60	8.67	0.48
6	4.10	24.67	8.29	30.25	4.65	23.82	1.96	6.18	0.18
7	2.73	16.47	10.67	30.76	13.10	19.84	1.89	7.28	0.00
8	1.30	19.71	2.46	28.91	2.87	32.68	2.07	11.20	0.10
9	1.29	14.89	13.99	23.05	10.79	12.46	6.42	8.05	10.34
10	1.12	14.82	5.42	25.06	0.80	24.42	17.39	11.81	0.28
11	0.46	35.69	1.45	22.82	0.48	30.56	2.22	3.68	3.09
12	0.43	5.51	6.34	67.98	0.00	15.28	0.62	4.16	0.10
13	0.30	5.61	0.30	78.91	0.46	11.99	0.00	2.73	0.00
14	0.30	1.52	0.15	94.83	0.00	2.43	0.30	0.76	0.00
15	0.13	3.81	7.96	70.93	2.77	13.84	0.00	0.69	0.00
16	0.04	11.70	0.00	37.23	0.00	41.49	0.00	9.57	0.00
17	0.01	6.06	6.06	39.39	0.00	42.42	0.00	6.06	0.00
18	0.01	0.00	0.00	91.67	0.00	8.33	0.00	0.00	0.00
19	0.01	9.09	0.00	77.27	0.00	13.64	0.00	0.00	0.00
20	0.00	22.22	0.00	55.56	0.00	22.22	0.00	0.00	0.00
21	0.00	25.00	0.00	50.00	0.00	0.00	0.00	25.00	0.00
22	0.00	33.33	0.00	0.00	33.33	33.33	0.00	0.00	0.00
23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00	0.00

Table 4.1: Distribution of probability for the accuracy of the location estimation when using a Gaussian model

Distance error[m]	Total distance error probability[%]	0 angle error probability[%]	$\frac{\pi}{4}$ angle error probability[%]	$\frac{\pi}{2}$ angle error probability[%]	$\frac{3\pi}{4}$ angle error probability[%]	π angle error probability[%]	$\frac{5\pi}{4}$ angle error probability[%]	$\frac{3\pi}{2}$ angle error probability[%]	$\frac{7\pi}{4}$ angle error probability[%]
0	34.04	73.27	1.38	12.02	0.80	8.89	0.38	3.15	0.11
1	17.12	23.66	0.00	35.37	0.00	28.81	0.00	12.16	0.00
2	13.43	25.11	1.56	30.58	1.04	23.69	0.28	17.74	0.00
3	11.11	26.84	3.65	30.17	2.36	19.41	1.67	14.85	1.06
4	8.32	18.10	1.94	36.09	1.73	27.73	2.01	12.30	0.11
5	5.33	23.33	5.04	36.96	1.50	22.37	0.51	9.72	0.57
6	3.12	21.25	4.89	34.41	1.51	24.55	3.85	9.38	0.16
7	2.78	21.06	6.68	31.02	12.24	22.21	0.81	5.94	0.05
8	1.43	15.26	1.29	30.80	0.78	39.62	1.19	10.80	0.25
9	1.50	23.29	11.05	28.72	3.02	14.39	6.63	12.87	0.03
10	0.73	7.22	5.55	31.34	5.06	13.63	19.68	15.48	2.04
11	0.39	39.02	4.81	23.91	1.03	16.59	1.14	13.39	0.11
12	0.26	12.41	14.80	55.61	0.34	14.80	0.68	1.02	0.34
13	0.12	15.83	0.39	45.56	12.74	22.39	0.00	3.09	0.00
14	0.14	2.88	2.88	73.08	0.64	10.58	8.33	1.60	0.00
15	0.14	2.89	55.63	37.94	0.32	2.57	0.64	0.00	0.00
16	0.01	10.00	6.67	30.00	16.67	30.00	0.00	6.67	0.00
17	0.00	63.64	27.27	9.09	0.00	0.00	0.00	0.00	0.00
18	0.00	20.00	0.00	40.00	0.00	20.00	0.00	20.00	0.00
19	0.00	25.00	0.00	50.00	0.00	25.00	0.00	0.00	0.00
20	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
21	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00
22	0.00	0.00	60.00	20.00	0.00	20.00	0.00	0.00	0.00
23	0.00	42.86	0.00	57.14	0.00	0.00	0.00	0.00	0.00
24	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00

Table 4.2: Distribution of probability for the accuracy of the location estimation when using the naive Bayes method

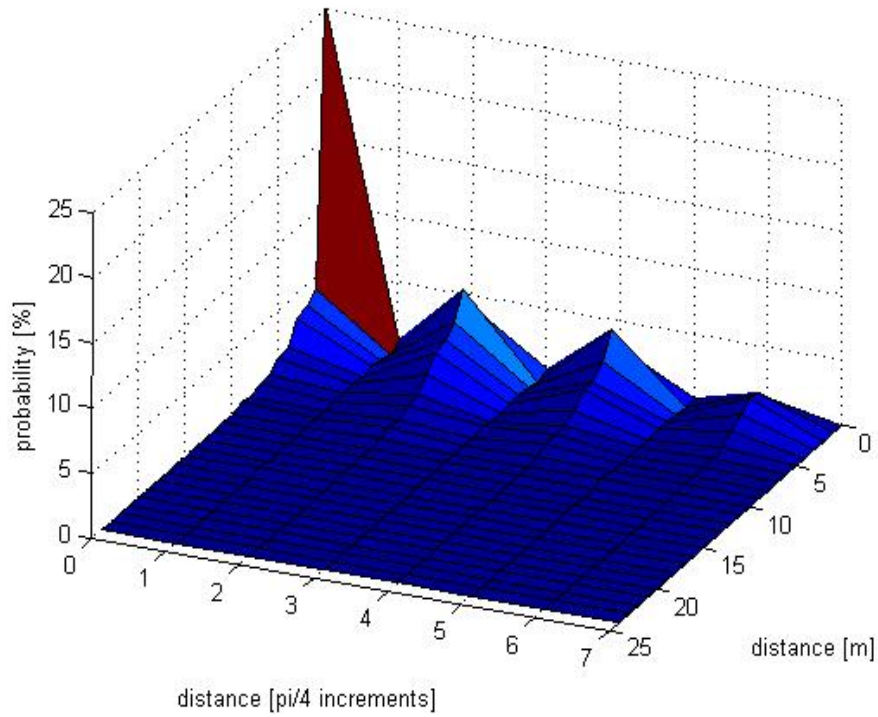


Figure 4.5: Distribution of probability for the accuracy of the location estimation when using the naive Bayes method

Distance error[m]	Gauss total distance error probability[%]	Bayes total distance error probability[%]
0	23.62	34.04
1	43.53	51.16
2	59.59	64.59
3	72.61	75.70
4	81.82	84.02
5	87.74	89.35

Table 4.3: Comparison of correctness of location estimation for a given distance error

Chapter 5

Conclusions and future work

The main goal of this implementation of a location estimation system was to gather, simultaneously, wireless traffic information from several points. The advantage of this implementation is that the signal strength of the same wireless frame can be measured in more than one location at the same time. This is impossible when using a mobile-based and mobile-assisted location system. By using this feature, the correlation between the measured strength of the wireless signal in several positions can be studied. An goal of this direction of research could be to develop a location estimation system that could function correctly with any type of wireless interface and even while using the power control features of the wireless interfaces in use.

The results that were shown are obtained by using just the basic location estimation algorithm and by using just a set of measurements (one signal strength measurement from each available wireless access point). At this point the results are not as precise as those of the best location systems' implementations. It should be taken into consideration that implementing various improvements for this location estimation system can increase the system's performance by a large margin but, on the other hand, can slow down its response time.

At the present moment, while using 3 wireless sensors, the system can capture around 100 frames/second. That means that each wireless sensor can generate around 30 records/second in the *_measurements* table of the system's database. Taking into account that, currently, only one channel is used, if a production deployment is intended, all the channels will have to be scanned. Depending on the region, there are either 11 (in the US) or 13 (in Europe) channels. Factoring in the time necessary for changing the channels, roughly, 2, maybe 3, frames from the same wireless device could be captured every second. That would be enough for implementing a real-time location estimation system. If this direction of improvement is to be pursued, the issue of how fast the location can be estimated once the measurements are obtained has to be investigated. The difficulty lies in the fact that the larger the area, the larger the fingerprint and the larger the fingerprint, the longer the time necessary for testing all the possibilities. Some type of clustering technique, like the one presented in [11] could be useful for solving this problem. Another way of speeding the process is searching for possible positions only in the vicinity of the last estimated position, assuming that the mobile device can't travel further than a certain distance in a given interval of time.

The choice of precision over speed, or of speed over precision, has to be made depending on the intended application of the system.

Other way the system could be improved would be to use Kalman filtering in order to implement some type of trajectory estimation and movement tracking for mobile wireless devices. This would greatly improve the tracking performance of the system because this system, as most of the other

location estimation systems currently available, is better suited for estimating the location of static devices.

Another aspect of the system that can be further investigated is how it handles the addition of more wireless sensors. This could have at least two effects on the system: first, the precision of the location estimation could increase. The fact that using a larger number of wireless sensors improves the performance of the system was already noticed by other teams of researchers (see [15]). Secondly, the throughput of captured data from each individual wireless sensor could decrease. A bottleneck could appear either at the “Sniffing server” or at the Location database.

Bibliography

- [1] [VxWorks and Tornado II FAQ](#)
- [2] [SVID Semaphores](#)
- [3] [GNU ‘make’](#)
- [4] [‘killall’ manual page](#)
- [5] [perldoc.perl.org](#)
- [6] Atheros AR5001 AP User’s Guide
- [7] Stephen O. Lidie and Nancy Walsh, “Mastering Perl/Tk”, O’Reilly, 2002
- [8] K. Pahlavan, X. Li and J. P. Makela, “Indoor geolocation science and technology”, IEEE Commun. Mag., vol. 40, no. 2, pp. 112-118, Feb. 2002
- [9] T. Roos, P. Myllymaki, H. Tirri, P. Misikangas, and J. Sievanen, “A probabilistic approach to wlan user location estimation”, International Journal of Wireless Information Networks, vol. 9, no. 3, pp. 155-164, July 2002
- [10] A. M. Ladd, K. E. Bekris, G. Marceau, A. Rudys, L. E. Kavraki, and D. S. Wallach, “Robotics-based location sensing using wireless ethernet”, in Proc. ACM International Conference on Mobile Computing and Networking (MOBICOM’02), 2002, pp. 227-238
- [11] M. Youssef, A. Agrawala, and A. U. Shankar, “WLAN location determination via clustering and probability distributions”, in Proc. IEEE International Conference on Pervasive Computing and Communications (PerCom’03), Dallas-Fort Worth, TX, Mar. 23-26, 2003, pp. 23-26
- [12] M. Youssef and A. Agrawala, “Small-Scale Compensation for WLAN Location Determination Systems”, in IEEE WCNC 2003, Mar. 2003
- [13] M. Youssef and A. Agrawala, “Continuous space estimation for WLAN location determination systems”, in Proc. IEEE International Conference on Computer Communications and Networks, 2004, pp. 161-166
- [14] Z. Xiang, S. Song, J. Chen, H. Wang, J. Huang, and X. Gao, “A wireless LAN-based indoor positioning technology”, IBM Journal of Research and Development, vol. 48, no. 5/6, pp. 617-626, Sept./Nov. 2004

- [15] K. Kaemarungsi and P. Krishnarmurthy, “Modeling of indoor positioning system based on location Fingerprinting”, in Proc. IEEE Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM’04), Hong Kong, China, Mar. 2004, pp. 1012-1022
- [16] K. Kaemarungsi, “Design of indoor positioning systems based on location fingerprinting technique”, University of Pittsburgh, 2005
- [17] [http://cs.pub.ro/so/index.php?section=Laboratoare&file=04.%20IPC%20\(1\)](http://cs.pub.ro/so/index.php?section=Laboratoare&file=04.%20IPC%20(1))
- [18] <http://www.tldp.org/LDP/lpg/node46.html#SECTION00743000000000000000>
- [19] <http://cs.pub.ro/so/index.php?section=Laboratoare&file=08.%20Semnale>
- [20] <http://www.coding-zone.co.uk/cpp/articles/140101networkprogramming.shtml>
- [21] <http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html>
- [22] <http://www.iana.org/assignments/port-numbers>
- [23] <http://www.postgresql.org/docs/8.1/static/tutorial-sql.html>
- [24] <http://www.llnl.gov/computing/tutorials/pthreads/>
- [25] <http://www.w3schools.com/xml/default.asp>
- [26] <http://www.xml.com/pub/a/98/10/guide0.html>

Appendix A: Semaphores

Semaphores are used for controlling the access to critical sections. Critical sections are those segments of code that must be accessed by only one thread at a time. A semaphore is blocking access to the critical section when its value is 0 and it allows access to the critical section when its value is greater than 0. The operations that can be performed on a semaphore are to set its value and test its value.

A semaphore ID permits the access to `nsems` semaphores. A `semget` call offers access to a set of semaphores. The set can contain a single semaphore. The read, increment and decrement semaphore operations of a semaphore set are executed by using a `semop` call, which processes `nsops` operations at one time. The operations are applied only if all of them are successful. Each operation is specified by using a `sembuf` structure. The semaphore operations can have a `SEM_UNDO` flag, which signals the fact that at the end of the process which performed the semaphore operation, the semaphore operation will be undone.

If a semaphore decrement operation is tried and it can't be executed, the process will be sent to a waiting queue until the semaphore's value will increase above 0 again (unless the `IPC_NOWAIT` flag was specified). A semaphore read operation can lead to the process being sent to the waiting queue as well.

Structures

A semaphore set is described by the following structure:

```
struct semid_ds
{
    struct ipc_perm sem_perm;
    time_t sem_otime;          /* last semop time */
    time_t sem_ctime;          /* last change time */
    struct wait_queue *eventn; /* wait for a semval to increase */
    struct wait_queue *eventz; /* wait for a semval to become 0 */
    struct sem_undo *undo;     /* undo entries */
    ushort sem_nsems;          /* no. of semaphores in array */
}
```

Each semaphore is described by the following structure:

```
struct sem
{
    short sempid;             /* pid of last semop() */
    ushort semval;           /* current value */
    ushort semncnt;          /* num procs awaiting increase in semval */
}
```

```
    ushort semzcnt; /* num procs awaiting semval = 0 */  
}
```

Limitations

The structure sizes:

semid_ds: 44 bytes; a structure for a semaphore set;

sem: 8 bytes; a structure for each semaphore in the system;

sembuf: 6 bytes; user allocated;

sem_undo: 20 bytes; a structure for each undo request.

Limits:

SEMVMX: 32767; the maximum value of a semaphore (short);

SEMMNI: the maximum number of semaphore identifiers in the system;

SEMMSL: the maximum number of semaphores for an semaphore ID;

SEMMNS: the number of semaphores in the system;

SEMOPM: the number of semaphore operations in a **semop** call.

For more information regarding semaphores see [\[17, 18\]](#).

Appendix B: Signals

Signals are a concept specific to UNIX operating systems. A signal is a software interrupt that interrupts the normal execution of the currently running process. The operating system uses them for signaling the running process the fact that some exceptional situation has been encountered and allowing the process to handle the situation. Each signal is associated with a class of events that can arise and that fulfill some criteria. The processes can handle, ignore or allow the operating system to execute the default action when receiving a signal. Usually, the default action is to terminate the receiving process.

The signal type set is a finite set. The operating system maintains a table containing the signal types that will be taken into consideration for each process. At process initialization, this table contains the default signal actions. The way a signal is handled is decided by the action stated in the table.

A signal received by a process can be generated by the operating system or by another process. Even if the signal is sent by another process, the signal is sent through the operating system. If a process wants to block a type of signal, the operating system won't send that type of signal to the respective process and it will save the first signal of that type. The rest of the signals of the blocked signal type that are sent to the process that is blocking that signal type will be lost. When the process removes the signal block, if there is any signal waiting to be sent, it will be sent to the process. If two signals come in a quick enough succession, the two could be received as only one signal. There is no delivery confirmation mechanism when using signals.

Signal generation

The events that generate signals could be classified in three categories:

errors indicate that a program has encountered an error and that it cannot continue its execution.

Not all errors generate signals though. Division by zero and trying to access invalid memory addresses are types of error that generate signals.

external events are, usually, generated by I/O or other processes. These events include the availability of new input data, timer expirations or the termination of child processes.

explicit requests indicate the call of a library function, like `kill`, in order to generate a signal.

The signals can be generated synchronously or asynchronously. A synchronous signal is generated while a specific action is performed and it is set during that action. Most errors generate synchronous signals. Asynchronous signals are generated by events that are uncontrollable by the receiving process. External signals and explicit requests for other processes generate asynchronous signals.

Signal transmission and reception

When a signal is generated, it enters a pending state. Normally, it stays in this state for a very short time interval and, then, it is sent to the destination process. If the respective signal type, the signal could remain in the pending state indefinitely. The signal will be sent to the destination process as soon as the respective signal is unblocked.

When the signal is received, the specified action is executed. For some signals, like **SIGKILL** or **SIGSTOP**, the action is not modifiable but the majority of signals can choose to ignore the signal, to specify a signal handler or to use the implicit action. While a signal's handler is running, the respective signal type is blocked.

The signals which, normally, represent errors have a special feature: when one of these signals terminate the process, it writes a core dump in which the state of the process at the time of termination is written.

Standard signal types

To find out which are the available signal types on the running UNIX distribution, the following command can be used:

```
> kill -l
```

The output looks like:

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR	31) SIGSYS	33) SIGRTMIN	34) SIGRTMIN+1
35) SIGRTMIN+2	36) SIGRTMIN+3	37) SIGRTMIN+4	38) SIGRTMIN+5
39) SIGRTMIN+6	40) SIGRTMIN+7	41) SIGRTMIN+8	42) SIGRTMIN+9
43) SIGRTMIN+10	44) SIGRTMIN+11	45) SIGRTMIN+12	46) SIGRTMIN+13
47) SIGRTMIN+14	48) SIGRTMIN+15	49) SIGRTMAX-15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

For more information regarding signals see [\[19\]](#).

Appendix C: Sockets

A socket represents a communication channel between processes. In UNIX, it is represented by a file descriptor. It allows communication between processes running on different machines connected to the same network.

Socket types

When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of the same type and in the same domain. Each address domain has its own address format. The address of a socket in the Internet domain consists of the IP address of the host machine (a 32 bit address). In addition, each socket needs a port number on that host. Port numbers are 16 bit unsigned integers. The port numbers from 0 to 1023 are the *well known port numbers*. These port numbers are assigned by the IANA and on most systems can only be used by system (or root) processes or by programs executed by privileged users. The port numbers from 1024 to 49151 are the *registered port numbers*. The registered ports are listed by the IANA and on most systems can be used by ordinary user processes or programs executed by ordinary users. All the port numbers above 49151 can be used freely.

There are two widely used socket types, stream sockets, and datagram sockets. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communications protocol. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (User Datagram Protocol), which is unreliable and message oriented.

Client/server architectures

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Notice also that once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

1. create a socket with the `socket` system call;

2. connect the socket to the address of the server using the `connect` system call;
3. send and receive data.

The steps involved in establishing a socket on the server side are as follows:

1. create a socket with the `socket` system call;
2. bind the socket to an address using the `bind` system call. For a server socket using TCP/IP communications, an address consists of an IP address and a port number on the host machine;
3. listen for connections with the `listen` system call;
4. accept a connection with the `accept` system call. This call typically blocks until a client connects with the server;
5. send and receive data.

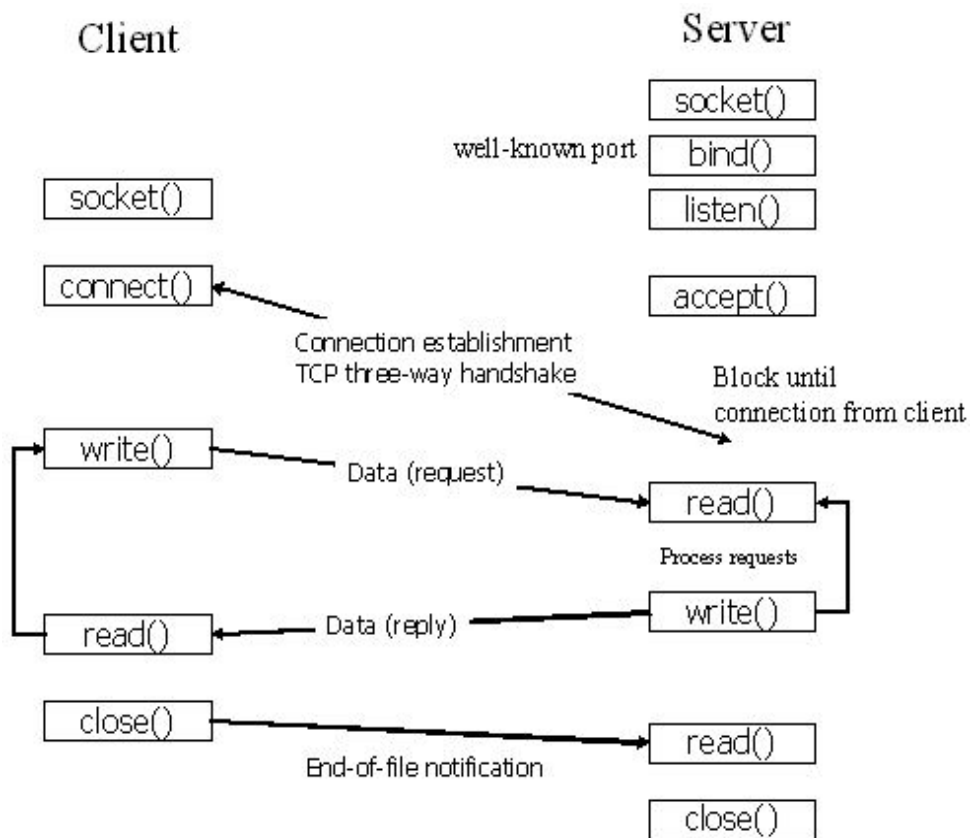


Figure 1: Client/server communication when using TCP sockets

For more information regarding sockets see [20, 21, 22].

Appendix D: SQL

PostgreSQL, like many of today's database management systems is a relational database management system. That means it is a system for managing data stored in relations. *Relation* is essentially a mathematical term for *table*. Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Creating tables

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (  
    city          varchar(80),  
    temp_lo      int,          -- low temperature  
    temp_hi      int,          -- high temperature  
    prcp         real,        -- precipitation  
    date         date  
);
```

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("–") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

PostgreSQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical key words, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (  
    name          varchar(80),  
    location      point  
);
```

The `point` type is an example of a PostgreSQL-specific data type.

Deleting tables

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

Adding records to a table

The INSERT statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
  VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
  VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Querying a table

To retrieve data from a table, the table is queried. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here * is a shorthand for "all columns". So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 rows)

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

city	temp_avg	date
San Francisco	48	1994-11-27
San Francisco	50	1994-11-29
Hayward	45	1994-11-29

(3 rows)

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27

(1 row)

For more information regarding SQL see [\[23\]](#).

Appendix E: Threads

In shared memory multiprocessor architectures, such as SMPs, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

What is a thread?

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread. To go one step further, imagine a main program that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.

Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:

- process ID, process group ID, user ID, and group ID
- environment
- working directory
- program instructions
- registers
- stack
- heap
- file descriptors
- signal actions
- shared libraries
- inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code. This independent flow of control is accomplished because a thread maintains its own:

- registers
- stack pointer
- scheduling properties (such as policy or priority)
- set of pending and blocked signals
- thread specific data.

In the UNIX environment a thread:

- exists within a process and uses the process resources
- has its own independent flow of control as long as its parent process exists and the OS supports it
- duplicates only the essential resources it needs to be independently schedulable
- may share the process resources with other threads that act equally independently (and dependently)
- dies if the parent process dies - or something similar
- is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

Because threads within the same process share resources:

- changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- two pointers having the same value point to the same data.
- reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

What are Pthreads

In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library.

The Pthreads API

The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. this standard is not freely available on the Web - it must be purchased from IEEE. The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.).

Mutexes: The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

The Pthreads API contains over 60 subroutines. The pthread.h header file must be included in each source file using the Pthreads library. For some implementations, such as IBM's AIX, it may need to be the first include file. The current POSIX standard is defined only for the C language.

For more information regarding threads see [\[24\]](#).

Appendix F: XML

XML is a markup language for documents containing structured information. Structured information contains both content and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure.

A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents. XML documents are composed of markup and content. XML was designed to describe, exchange, share and store data. It is extensible because the XML tags are not predefined. This allows the programmer/author to define his own tags and his own document structure.

XML is used to exchange and share data

With XML, data can be exchanged between incompatible systems. In the real world, computer systems and databases contain data in incompatible formats. One of the most time-consuming challenges for developers has been to exchange data between such systems over the Internet. Converting the data to XML can greatly reduce this complexity and create data that can be read by many different types of applications. By using XML, plain text files can be used to share data. Since XML data is stored in plain text format, XML provides a software- and hardware-independent way of sharing data. This makes it much easier to create data that different applications can work with. It also makes it easier to expand or upgrade a system to new operating systems, servers, applications, and new browsers.

XML is used to store data

With XML, plain text files can be used to store data. XML can also be used to store data in files or in databases. Applications can be written to store and retrieve information from the store, and generic applications can be used to display the data. Since XML is independent of hardware, software and application, you can make your data available to any type of interpreter.

An example XML document

XML documents use a self-describing and simple syntax.

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
```

```
</note>
```

The first line in the document describes the root element of the document (like it was saying: "this document is a note"). The next 4 lines describe 4 child elements of the root (**to**, **from**, **heading**, and **body**):

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
```

And finally the last line defines the end of the root element.

XML syntax

A properly formed XML document must follow several rules, like:

- all XML elements must have a closing tag. With XML, it is illegal to omit the closing tag.
- XML tags are case sensitive. With XML, the tag `<Letter>` is different from the tag `<letter>`. Opening and closing tags must therefore be written with the same case.
- XML elements must be properly nested.
- XML documents must have a single root element. All other elements must be within this root element.

XML elements must follow these naming rules:

- names can contain letters, numbers, and other characters
- names must not start with a number or punctuation character
- names must not start with the letters xml (or XML, or Xml, etc)
- names cannot contain spaces

All elements can have sub elements (child elements). Sub elements must be correctly nested within their parent element:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML elements are extensible. XML documents can be extended to carry more information. Take the following example:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

Imagine that we created an application that extracted the `<to>`, `<from>`, and `<body>` elements from the XML document to produce some output. Imagine, then, that the author of the XML document added some extra information to it:

```
<note>
  <date>2002-08-01</date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The application should still be able to find the `<to>`, `<from>`, and `<body>` elements in the XML document and produce the same output.

For more information regarding XML see [\[25, 26\]](#).