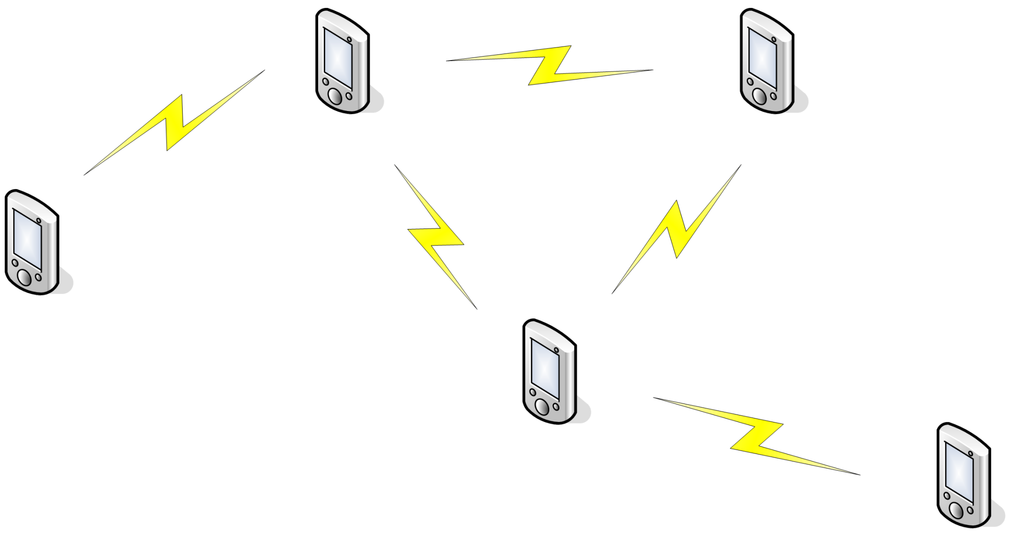
Institut für
Technische Informatik und
Kommunikationsnetze

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Ivo Trajkovic, Clemens Wacha

Ad Hoc Communication with Handhelds



Semester Thesis TIK-2006-02
October 2005 to February 2006

Advisors: Vincent Lenders,
Dr. Martin May
Supervisor: Prof. Dr. Bernhard Plattner

Abstract

Links and routes in a wireless ad hoc network can break due to many reasons including moving users, fading, collisions, interference, or noise. Not much analysis has been done on this subject. The goal of this thesis is to conduct a series of experiments with real data transfers between network nodes. The test network consists of 10 HP iPAQs running Windows Mobile 2003 Second Edition and communicate through Wireless Local Area Network (WLAN) interfaces. To achieve this goal a modular and flexible measurement application was written that allows us to measure a broad range of metrics. Furthermore it provides a simple plugin-like mechanism so that other aspects can be analysed by extending the program accordingly with minimal effort. The thesis also covers detailed documentation on the program internals, the application design and the usage of the program itself. The collected data is analysed for packet loss under optimal conditions (static positions, no other interference), packet loss under real life conditions (real user mobility, interference with other WLANs), lost packet series and connection lifetimes.

Abstract

Links und Routen in einem drahtlosen ad hoc Netzwerk können durch verschiedene Ursachen unterbrochen werden wie z.B. Knoten die sich bewegen, Fading, Kollisionen, Interferenz oder Rauschen. Zum aktuellen Zeitpunkt gibt es nur wenige Untersuchungen auf diesem Gebiet. Das Ziel dieser Arbeit ist es, eine Reihe von Experimenten mit echten Datentransfers zwischen Knoten im Netzwerk durchzuführen. Das Test-Netzwerk besteht aus 10 HP iPAQs welche unter Windows Mobile 2003 Second Edition laufen und über WLAN kommunizieren. Für die Analyse wurde ein modulares und flexibles Computerprogramm geschrieben welches uns erlaubt viele unterschiedliche Metriken zu messen. Desweiteren enthält es einen einfachen Plugin-ähnlichen Mechanismus welcher die Untersuchung weiterer Aspekte ermöglicht. Dazu kann das Programm mit minimalem Aufwand entsprechend erweitert werden. Die Arbeit beinhaltet auch detaillierte Dokumentation über die Interna des Programms, das Programm Design und die Verwendung des Programms. Die gesammelten Messdaten wurden nach Paketverlust unter Optimalbedingungen (statische Positionen, keine Interferenz), Paketverlust unter Realbedingugen (Benutzer Beweglichkeit, Interferenz mit anderen WLANs), verlorene Paketserien und Lebensdauer von Verbindungen untersucht.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Scope

Small and mobile devices such as handhelds are becoming more and more ubiquitous. Nowadays the use of WLAN with such devices is mostly limited to downloading messages or surfing the Web via a fixed access point. Ad hoc networking is a promising alternative communication paradigm where devices form a network in a dynamic and self-organizing manner without any fixed infrastructure support such as access points or dedicated routers. Therefore communication is no more limited to fixed places controlled by network operators but becomes in principle possible anywhere as long as cooperative nodes exist in the neighborhood. However multi-hop communication and routing in ad hoc networks is quite challenging since users can be mobile causing links and routes to break frequently.

1.2 Motivation

Links and routes in a wireless ad hoc network can break due to many reasons including moving users, fading, collisions, interference, or noise. In a preliminary study conducted at the CSG [8], we discovered that mobility failures and transmission failures (fading, collisions, interference, noise) tend to occur at different moments during the lifetime of a link. To illustrate this effect, we plotted the CDF of the link residual lifetime in Fig. 1 (see original assignment for this plot) from an experiment conducted at ETH with twenty mobile users. We see that transmission failures tend to happen very quickly and are quite rare after long periods. In contrast link failures due to mobility tend to occur only after a few minutes but after 3500 seconds, almost all links are affected by this type of failure.
The goal of this thesis is to extend the experimental results from [8] by conducting a series of experiments with real data transfers between network nodes instead of reconstructing a connectivity graph from periodic beacons.

1.3 Methodology

To assess the route and link lifetime in mobile ad hoc networks, a WLAN test network should be used consisting of HP iPAQs hx2400 running Windows Mobile Edition 2003. The tests should include real user mobility by distributing the iPAQs to researchers and students at the CSG lab.

1.4 Assignment

1.4.1 Tasks

- Study the semester thesis report of Jörg Wagner [8].
- Reproduce a set of experiments from the thesis to get familiar with the developed tools.
- Identify or propose a mechanism to send traffic between nodes in the network. An existing routing protocol or a simple flooding scheme can be used.

- Conduct the necessary experiments to assess the link and route lifetime distribution with real data transfers. In addition to the total lifetime, the interval distribution should be measured until a broken link or route reappears. Conduct the experiments with a mobility scenario (distributing iPAQs to people) and also with a static scenario.
- Analyze the collected data and make final conclusions.

1.4.2 Deliverables

- At the end of the second week, a detailed time schedule of the semester thesis must be given and discussed with the advisors.
- At half time of the semester thesis, a short discussion of 15 minutes with the professor and the advisors will take place. The students have to talk about the major aspects of the ongoing work. At this point, the students should already have a preliminary version of the written report, including a table of contents. This preliminary version should be brought along to the short discussion.
- At the end of the semester thesis, a presentation of 20 minutes must be given during the TIK or the communication systems group meeting. It should give an overview as well as the most important details of the work.
- The final report may be written in English or German. It must contain an abstract written in both English and German, the assignment and the time schedule. Its structure should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Three copies of the final report must be delivered to TIK.

1.5 State at Project Kickoff

In a preliminary thesis [8] Jörg Wagner has created a measurement application for Windows Mobile Edition that is able to exchange ping packets with peer devices over a single hop. No routing was implemented. A connectivity graph was reconstructed from the measured data. A second application periodically flooded a message into the network. The device running this application had a static position and represented a gateway.
Messages from the gateway were forwarded by peer devices (running the measurment application) to other peer devices. The applications were written in C/C++ using the eMbedded Visual C++ 4.0 Integrated Development Environment (IDE) from Microsoft. The network packets were sent via UDP sockets using the *Winsock API*.
During the measurements Jörg Wagner encountered a problem known as *BSSID partitioning*. Although he was not able to find a proper solution he managed to detect partitioning from within the program and let the device reboot if it chose the ”wrong” (smaller) BSSID. This patch helped keeping *BSSID partitioning* at a minimum.

The collected data from the real WLAN ad hoc network was analysed with respect to

- node degrees
- path lengths
- clustering
- connectivity to gateway
- route stability under different routing metrics

and compared with two mobility models. The random waypoint model and the random reference point group mobility model. Both of them yielded good match with the empirical data from the measurements with respect to route stability. A comparison between three different routing metrics showed that a metric aiming to minimize the packet error rate yielded the most stable routes to the particular test network.

All these results are based on a connectivity graph which was reconstructed from measurement data of single hop ping packets. The overall traffic in the test network was therefore very low which leads to the question whether the results still remain the same if the traffic is increased drastically.

1.6 Overview

This work is divided into three parts:

- Chapter 2 explains the setup of the test environment and the hardware used. It also describes common problems in wireless ad hoc networks and contains a short discussion on routing algorithms.
- Chapter 3 deals with the measurement software that was created specifically for analysis. It contains a birds view of the program as well as detailed information on the program internals.
- Chapter 4 discusses the results of the conducted measurements.

Chapter 2

Testbed

2.1 Ad Hoc WLAN Setup

Our ad hoc network consisted of 10 HP iPAQ devices which communicated over the WLAN interface. The devices run Windows Mobile 2003 Second Edition. Our analyser software was written in C/C++ natively for Windows CE. Since it is not possible to access RAW sockets in Windows CE we were forced to create User Datagram Protocol (UDP) packets and send them to the broadcast address. The devices were configured to use ad hoc mode with the SSID *sa_ahcomm* and an IP address in the private range 192.168.0.X/24. To facilitate analysis we synchronized the time on the devices before every measurement since the clocks in some iPAQs have a huge drift around +/- 1 second per minute!
Most measurements were done on channel 11 because the devices automatically choose that channel and it was not possible to change it directly on the device [1]. Some of the measurements were recorded in ETZ A-floor where there are no other WLANs interfering and others in ETZ G-Floor to test the behavior under real life conditions.

2.2 Near-Far Problem

The near-far problem is a common situation in wireless communication systems (in particular, Carrier Sense Multiple Access (CSMA)). The near-far problem can also be called the hearability problem. In some signal jamming techniques the near-far problem is exploited to disrupt communication.
The following example can show a possible problem that can occur. Consider a receiver and two transmitters (one close to the receiver; the other far away). If both transmitters transmit simultaneously and at equal power then the receiver will receive more power from the closer transmitter. This makes the farther transmitter more difficult if not impossible to "understand". Since one transmission is the other's noise the Signal-to-noise ratio (SNR) for the farther transmitter is much lower. If the closer transmitter transmits a signal that has a higher order of magnitude than the farther transmitter then the SNR for the farther transmitter may be below detectability and the farther transmitter may just as well not transmit. This effectively jams the communication channel.
In CSMA systems like WLAN this is commonly solved by dynamic output power adjustment of the transmitters. The closer transmitters use less power than the further ones so that the SNR for all transmitters at the receiver is roughly the same. This sometimes can have a noticeable impact on battery life, which can be dramatically different depending on the distance to the base station.

2.3 Near-Far Problem in Ad Hoc WLAN Networks

Dynamic output power adjustment works well if you have one base station and several clients. In such a setup clients only communicate with the base station and not among each other. In

[1] To make the devices use another channel you have to use a notebook that creates the ad hoc network on a different channel and then let the devices join one after the other.

ad hoc networks there is no base station. Every client may communicate with any other client. This makes the dynamic output power adjustment a difficult task.
Consider a situation where you have three clients in a row. The distance between the first client and the client in the middle is only half the distance of the middle client and the last client. Both outer clients can adjust their output power as usual where as the one in the middle would have to adjust its output power for every packet it sends depending on the destining device. Since adjusting output power is a task that needs some time to settle, it cannot be done on a per packet basis.

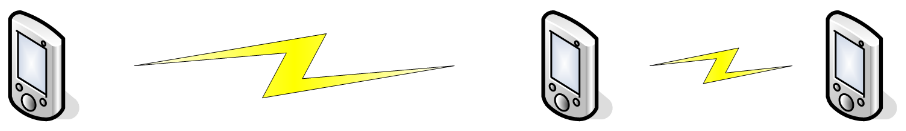


Figure 2.1: Dynamic power adjustment: The device in the middle has to adjust its power to either the left or the right device

Another problem in ad hoc wireless networks caused by the near-far problem are collisions. Collisions happen if two devices transmit a packet at the same time. WLAN uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) to avoid collisions since it is not possible to detect collisions directly in WLANs. A transmitted signal from a device can be decoded in a range of about 100 meters but it jams all other transmissions in a range of about 500 meters! A receiver (4) that is 300 meters away from a sender (1) cannot receive a sent packet since it is too far away but the distance might be to small to eliminate the senders (1) noise. It is therefore possible that data sent by another sender (5) closer to the receiver (4) cannot be received correctly at the same time.

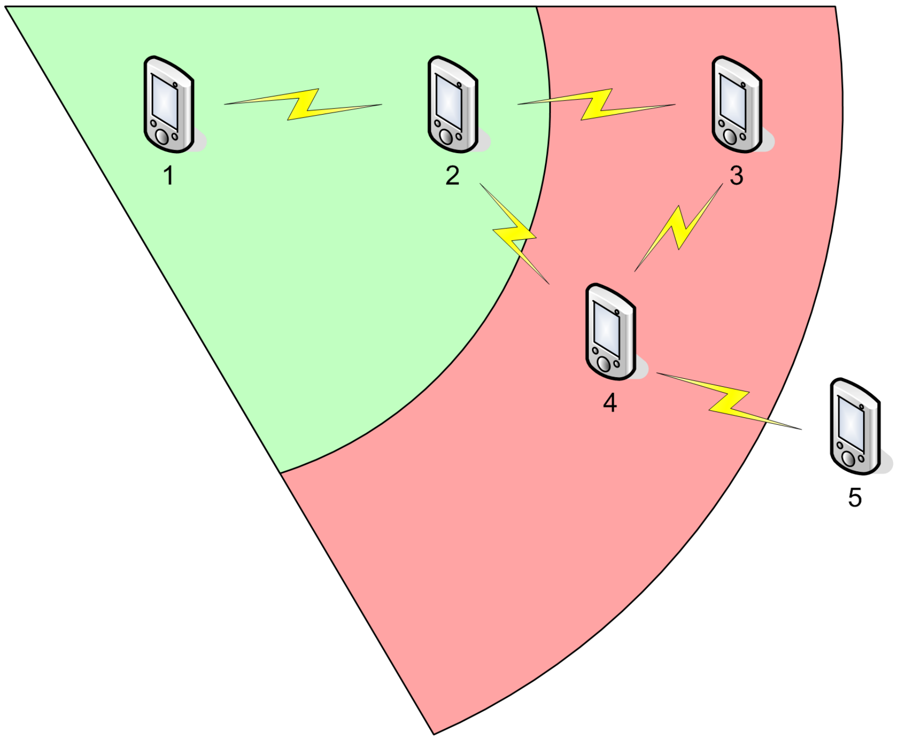


Figure 2.2: Hearability: The signal sent by device 1 can only be decoded by device 2. Devices 3 and 4 are jammed by device 1

To prevent this behaviour there is a further mechanism to avoid collisions in WLANs: Carrier Sense Multiple Access / Collision Avoidance Ready to Send / Clear to Send (CSMA/CA RT-S/CTS). Unfortunately this mechanism is disabled for packets sent to the broadcast address. Since we have no access to RAW sockets in Windows CE (and therefore cannot create our own packets) we have to use a higher level of communication. Our program sends UDP packets to the broadcast address and we can therefore not use this mechanism.

2.4 BSSID Partitioning

In WLAN computer networking, a Service Set IDentifier (SSID) is a code attached to all packets on a wireless network to identify each packet as part of that network. The code consists of a maximum of 32 alphanumeric characters. All wireless devices attempting to communicate with each other must share the same SSID. Apart from identifying each packet, the SSID also serves to uniquely identify a group of wireless network devices used in a given *Service Set*.

There are two major variants of the SSID:

- Ad hoc wireless networks that consist of client machines without an access point use the Independent Basic Service Set IDentifier (IBSSID)
- whereas on an infrastructure network which includes an access point, the Basic Service Set IDentifier (BSSID) is used instead.

The naming is for convention only as the IEEE 802.11 standard dictates that an ad hoc networks and infrastructure networks are each defined by an SSID, otherwise known as a *Network Name*. A Network Name is commonly set to the name of the network operator, such as a company name. Equipment manufacturers have liberally used all of the above SSID naming conventions to essentially describe the same thing. In some instances, the convention is wrong, as in the case of BSSID.
In real life a WLAN network is uniquely identified by its BSSID. The BSSID is defined as the Media Access Control (MAC) address of the Access Point (AP) in infrastructure mode. Every AP needs a unique BSSID where as the SSID defines the network name it is in.
In ad hoc mode the SSID and BSSID both define the network the device should connect to. The SSID is a network name defined by the user. The BSSID is a locally administered IEEE MAC address generated from a 46-bit random number. Every device in an ad hoc network is able to advertise a BSSID to other devices. Whenever two devices meet they agree on one common BSSID which they advertise to newly arriving devices. If two separate networks having the same SSID join they usually do not have the same BSSID in which case they need to agree on one BSSID.
There are no general guidelines on how to solve this problem so every manufacturer does it differently. The HP iPAQ Wi-Fi driver on Windows CE unfortunately uses a very simple scheme. Every iPAQ advertises its BSSID in ad hoc mode. If it receives a beacon of another device sharing the same SSID but having a different BSSID it switches to that BSSID namely the one of the last received beacon. This works well as long as only one device enters the network. However if two separate networks with the same SSID and different BSSIDs merge you end up having a network with lots of devices sending competing beacons.
Since the devices need some time for this switch, this leads to a network where devices randomly start hopping back and forth between the two (or even more) networks. Unfortunately this situation does not need to stabilize because the probability of switching to the other network is the same in both directions. Jörg Wagner [8] also has described this problem in his work. He has implemented a workaround in his program that should speed up the stabilisation process drastically. We did not implement this feature into our program. A visit on the homepage of HP showed that there was a software update for our iPAQs. Our first hope was that they may have solved the problem already. The problem didn't occur during previous sessions with up to seven concurrently running devices. The first time we encountered BSSID partitioning for real was already fairly at the end of our work. We tried to integrate the original code written by Jörg but after some investigation we found this task to be non-trivial and too time consuming since his solution is very tightly integrated into his program. Have a look at `/sa_ahcomm/software/iPAQ/routing_analyser/ndishelper.cpp` for a similar approach. The Windows CE version of the function GetBSSID() shows how to access the NDIS interface on Windows.

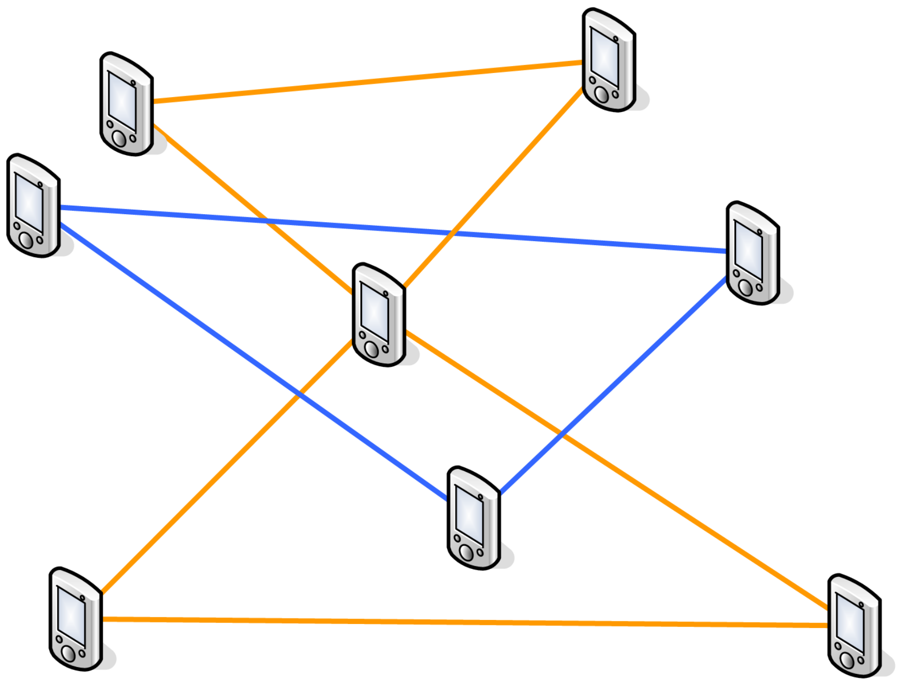


Figure 2.3: Illustration of BSSID partitioning

2.5 Routing Algorithms

Our goal was to verify some of the measurements from Jörg Wagners work [8]. We wanted to measure the data flow over more than one hop with real data and see if the results of the models also represent the real world behavior. His testing application was not suitable for this task since it could only send ”ping” packets to other devices and there was no routing implemented. The first idea was to extend it with a routing algorithm.
The decision for a specific routing algorithm was not clear at the beginning and we started to search the Internet for ad hoc routing algorithms. We found tons of algorithms in no time. Wikipedia can give you a good insight about this topic at http://en.wikipedia.org/wiki/Ad_hoc_routing_protocol_list. This list also shows that ad hoc routing is a very popular subject and that a lot of research is still going on.
We have concentrated ourselves on only a few typical algorithms since many of them are very similar.

2.5.1 TORA

The algorithm builds a unidirected graph. Each participating node i has an assigned height defined by an ordered quintuple. Information can flow only downhill, so loops are impossible. The algorithm claims to be scalable.

Conclusion

Given only few information about the implementation and the algorithm would lead to a time-consuming implementation. More information about this algorithm can be found at http://www.isr.umd.edu/ISR/accomplishments/037_Routing/

2.5.2 AODV

The Ad hoc On Demand Distance Vector (AODV) routing algorithm is a routing protocol designed for ad hoc mobile networks. AODV is capable of both unicast and multicast routing. It is an on-demand algorithm, meaning that it builds routes between nodes only as desired by the source nodes. It maintains these routes as long as they are needed by the sources. In addition, AODV forms trees which connect multicast group members. The trees are composed of the group members and the nodes needed to connect the members. AODV uses sequence numbers to ensure the freshness of routes. It is loop-free, self-starting, and scales to large numbers of mobile nodes.

Conclusion

You can find a good explanation on the algorithm at `http://moment.cs.ucsb.edu/AODV/aodv.html` and `http://moment.cs.ucsb.edu/pub/wwan_chakeres_i.pdf`. Source code is available at `http://www.docs.uu.se/docs/research/projects/scanet/aodv/aodvuu.shtml`. Due to the complexity of the algorithm an implementation from scratch would be too time consuming. An adaption of the existing source code onto the iPAQs could be possible but the outcome is unclear.

2.5.3 DSR

The Dynamic Source Routing (DSR) protocol was designed especially for Mobile Ad Hoc Network (MANET) applications. Its main feature is that every data packet follows the source route stored in its header. This route gives the address of each node through which the packet should be forwarded in order to reach its final destination. Each node on the path has a routing role and must transmit the packet to the next hop identified in the source route. Each node maintains a *Route Cache* in which it stores every source route it has learned. When a node needs to send a data packet, it first checks its route cache for a source route to the destination. If no route is found, it attempts to find one using the route discovery mechanism. A monitoring mechanism, called route maintenance, is used in each operation along a route. This mechanism checks the validity of each route used.

Conclusion

Documentation can be found at `http://www.antd.nist.gov/wctg/DSRreadme.pdf`. An implementation in JAVA can be downloaded from `http://www.cs.cmu.edu/~dmaltz/dsr.html`. The algorithm seems difficult and would have to be ported from Java to C/C++.

2.6 Routing Algorithm Decision

Analysis of the existing programs showed that it would become difficult to extend them with a routing algorithm. We decided to create a completely new program und use a simple flooding scheme for simplicity. [2]
One device should act as a *streamer* and create packets and send them to the UDP broadcast address. Every device receiving a packet should send the packet to the broadcast address as well. That way a packet would traverse the network from the *streamer* to the border of the network. Since such a simple scheme would create a lot of traffic due to echoes, every device should *remember* a packet and only forward it once. Since this is not easy to implement we decided to create a backup solution in case we would not be able to handle the more sophisticated solution by adding a Time To Tive (TTL) to every packet. Every hop decrements the TTL of a received packet by one. If it reaches zero the packed is discarded. This mechanism works identical to the implementation found in TCP/IP. For a detailed discussion on the routing algorithm see Sect. 3.1.4.

[2]For a detailed discussion about the old program and our decision to begin with a new program see Sect. 3.1.

Chapter 3

Implementation

3.1 Routing Analyser Application

The Routing Analyser application is one of the main parts of this semester thesis. The primary idea was to take the existing measurement applications and extend them with a routing algorithm so that we could make real data transfers. A problem of the old application was the Graphical User Interface (GUI). We were told that people taking part in the real life measurements tended to forget the device at their desks which would render the measurements unusable. Carrying the device around all the time was a boring job since the measurement program did not display its actions. Therefore the second task was to brush up the GUI a little bit.
After detailed analysis of the existing work we decided to start from scratch due to the following reasons:

- Every thread was realised as a huge function that did all the work.
- The documentation only covered the big picture. A lot of details were missing.
- The source code had very few comments.
- Therefore, adding a routing algorithm seemed to become very difficult.
- The GUI was not prepared to handle more than one view.
- The program crashed for an unknown reasonon devices where we have installed the software update . We did not investigate the problem any further.

Aside the program being written all in C making no use of C++ classes or the Standard Template Library (STL). Of course it is not necessary to use C++ extensively but our experience in writing programs in C/C++ has shown that writing *FIFOs*, *LISTs*, and *QUEUEs* in C is very error prone and fixing bugs takes a lot of time. Since we were not familiar with the former applications we also did not want to take the risk of any unknown limitations or hidden bugs.
We decided to start with a new application and take over all the useful code from the old programs. We also learned a lot about Windows programming from the former applications.
The communication between the devices still works over WLAN using UDP packets sent to the broadcast address.
At the current state our program implements a simple flooding algorithm and can perform the following tasks:

- Send and forward discovery packets that tell that a device is in range
- stream raw audio data
- receive, forward and play audio data
- display a list of all devices in range including their name, distance and offered services
- display a lot of statistic data of program internals that greatly assist developers in debugging and help make the application more interesting

- display debug messages
- log all network activities to the storage card for offline analysis
- log debug messages to the storage card to make debugging easier
- The program is highly customisable. Most options that can be changed are stored as constants in the file `config.h`.

3.1.1 Application Design

Separation

A major goal in the design of the new application was the possibility to exchange the routing algorithm. This can only be done if the algorithm is separated from the transport layer below and the application layer above. During the design phase we also heard that other people at the institute are running Linux on the iPAQs and trying to get the internal WLAN interface to work. Implementing such a routing algorithm in Linux gives you much more flexibility than with Windows CE since you have full access to all necessary layers and functions.

Platform Independence

The idea came up that we could write the program to work on other platforms as well. If this decision is made at the very beginning it is much easier to achieve this goal whereas porting a finished application to another platform usually means to rewrite many parts. All platform-specific functions have to be separated or summarized into modules. We decided to design the application for platform independence but focus on Windows CE. Where it is possible, wrapper functions should be used so that a later port to another platform would be as comfortable as possible.

The following system calls are platform specific:

- accessing system time and date
- threads in general
- semaphores and mutexes to protect critical sections
- network access
- filesystem access other than open, read, write and close

We could have used Simple Directmedia Layer (SDL) for this job but as mentioned before we did not want to make the program actually work on other platforms but only facilitate the porting process. Another aspect was that there was not an official port of SDL for Windows CE yet and SDL does not provide access to the network directly. To make use of the network interface you need the SDL_net extension as well which is yet another external dependency.
We have collected all these calls in a library we called the PI (Platform Independence) library. In fact a lot of parts are very similar to SDL. We got to know a lot from the code in the SDL implementation. For a detailed discussion about the PI library see App. D.2. In the current state the PI library is completely working on Windows CE, Windows 2000/XP, Linux, Mac OSX. The necessary adjustments were minimal so we have setup everything inside the file `config.h`. The program compiles and runs on all these platforms though without GUI and therefore with limited interactivity. Streaming, forwarding and logging is completely working. Doing measurements only with notebooks should work out of the box. This could be interesting since notebooks usually have better WLAN hardware and software impementations than our iPAQs and you also have a broader range of packet sniffers and other analysis tools on these platforms.

Threads

Since Jörg Wagner used threads for all his tasks we decided to use them as well since we have never programmed Windows (or even Windows CE) before and this solution has already worked for him. When using threads you need to find a way to enable communication among each other. This usually leads to race conditions and you have to protect the shared memory areas with semaphores or mutexes. We tried to solve this problem in a generic way using *messages* and *message queues*. If you want to send a message to a thread do the following:

- Create the message
- Add your data to the message
- Get/Find a pointer to the class instance that should receive the message
- Send the message to the instance using the pointer and the `PushMessage()` member.
- Tell the message that you don't need it anymore by calling `Discard()` on the message.

The Routing Analyser uses this system for all inter thread communication. For an in-depth discussion on the message system see App. D.1.

Model View Controller

To make the GUI optional we had to separate it from the main application core in a clean way. This was done by using the Model-View-Controller (MVC) approach [1].
Although the GUI also uses our message queue implementation the communication works a little bit different. Since we did not want to make the application core depend on the GUI we have created a *subscription service*. The GUI acts as observer and subscribes to the main application core in a generic way and then gets the messages sent automatically. See App. D.1.2 for more information on this subject.

Energy Management

Another important subject is energy management. The Li-Ion Battery used in iPAQs don't last longer than four hours with the WLAN interface turned on. Therefore we have optimized the application for low CPU usage. As an example the message queue model does not send complete messages to other objects but only a pointer to it. This helps keep CPU usage low. A typical value is under 10% when the application is idle and around $40 - 50\%$ when we are forwarding packets. In eMbedded Visual C++ development environment you can use the *Remote Performance Monitor* from the *Tools* menu to measure these values.

3.1.2 Main Application Core

The main application core can be found in `main.cpp` in function `MainApplication::Main()`. This function sets up everything and is responsible for communication in the application layer. Windows CE programs do not have a user accessible `main()` function. Instead the first called function is `WinMain()`. In our program this function creates a new instance of *MainApplication* and then creates and launches a thread from `MainApplication::Main()`.
Remark that every GUI dialog has its own message handler. The functions have names like `xxxxxxxProc()`. The GUI functions can all be found in the file `winmain.cpp`. The menubar you see when you start the program is part of the main window of the application. The corresponding message handler is called `MainWndProc()`. However the list view and the three labels on the first screen are actually in a separate dialog whose message handler is called `MainDlgProc()`. All dialog windows are created at program startup in the function `MainWndProc()` and destroyed at the end of the program in the same function. If you need more information about Windows programming see the source code in file `winmain.cpp` which contains a lot of comments or check the example programs that come with eMbedded Visual C++ and can be found at `C:\Program Files\Windows CE Tools\wce420\POCKET PC 2003\Samples\Win32`.

[1] See `http://en.wikipedia.org/wiki/Model-view-controller` for a good explanation.

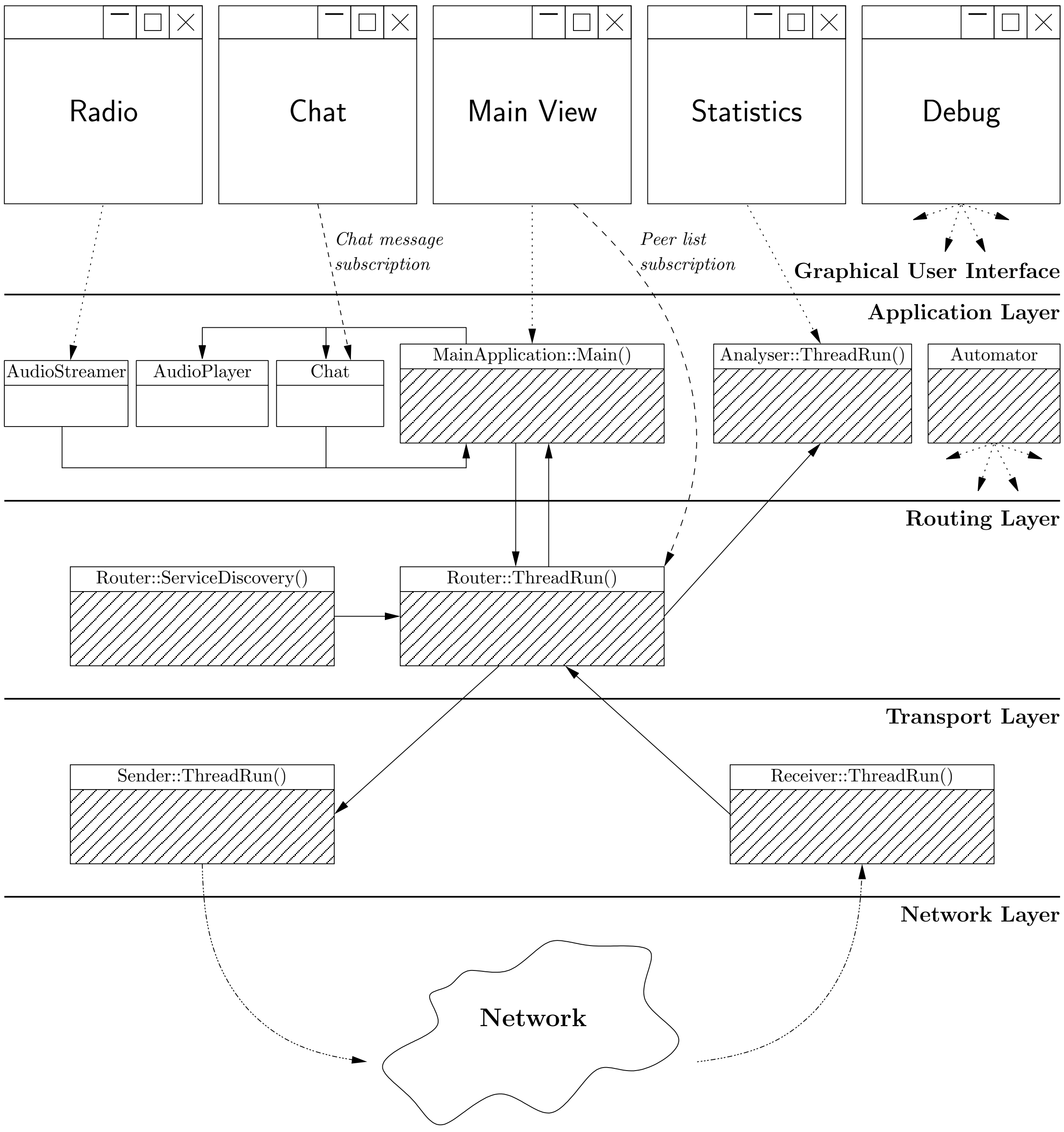


Figure 3.1: Routing Analyser application design

The *common controls*-example is a very good place to start. It can be found at `C:\Program Files\Windows CE Tools\wce420\POCKET PC 2003\Samples\Win32\Comctls`.

In detail the `MainApplication::Main()` function works as follows:

- Sets up and launches all threads.
- Contains the main loop.
- Receives messages from the routing layer and forwards them to the respective modules
- Receives messages from the chat and audio streamer modules and hands them down to the routing layer.
- Periodically polls for the IP address of the iPAQ.

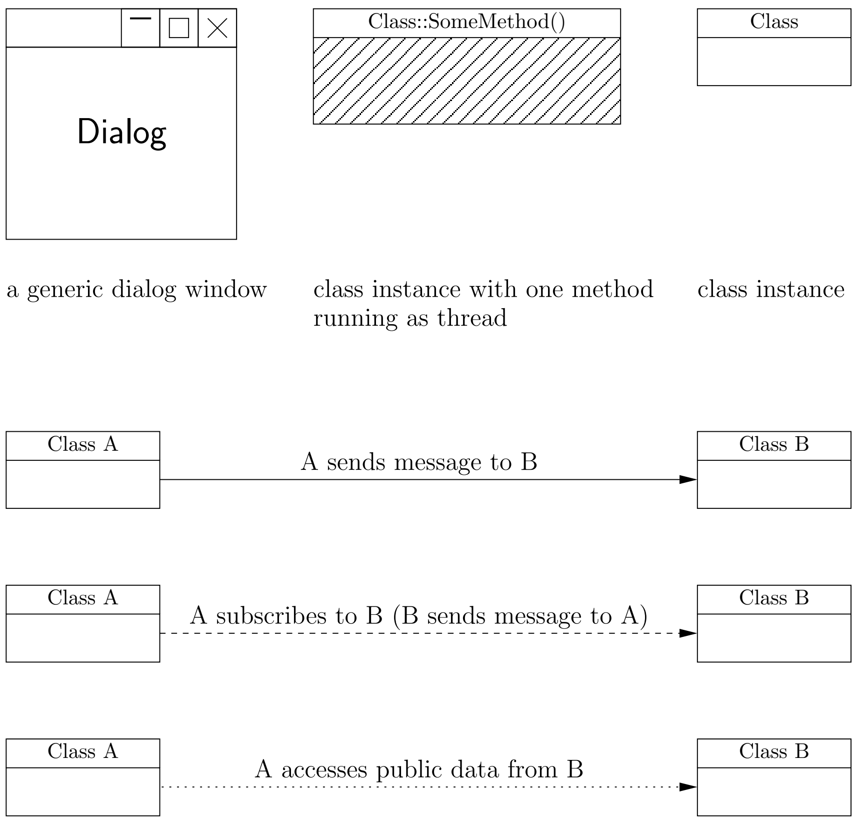


Figure 3.2: Explanation of the design relations

- Periodically calls chat, audio streamer and audio player module

Since the chat, audio player and audio streamer modules are not threads, they need to be called periodically so they can update their internal state. In our application every class that needs periodic attention has some sort of `Update()` member that takes care of this. Remark that neither the main application core nor any of the other objects does ever speak to the GUI directly. The GUI collects the information using public members of the objects or if this is not possible uses the subscriber service to be informed of changes.

3.1.3 Threads

As you can see from Fig. 3.1 the hatched objects are threads. Every class that runs as a thread contains a `ThreadRun()` member. The only exception is the Router class which provides another function `ServiceDiscovery()` that runs as a separate thread too.

Router::ServiceDiscovery()

The ServiceDiscovery()-function creates and sends a so-called heartbeat packet to the router thread. These packets tell other iPAQs that a device is in range. It contains the following information:

- The senders IP address.
- The service mask which gives a hint on what services a device offers. See Sect. A.1 for more information
- The name of the device (which is taken from the *Owner information* of the iPAQ).

The program maintains a list of all peers in range using a class called *PeerList*. The router object contains an instance of PeerList called *plist*. The information about visible peers of a service discovery packet is extracted and added to the peer list. If a device already has an entry in the peer list the entry is updated. If an entry in the peer list is not updated within 15 seconds (adjustable in `config.h`) the entry is removed. The list view in the applications main screen displays the current content of the peer list. Remark that a peer that is sending audio packets but no service discovery packets will not show up in the peer list. You are however still able to listen to the audio stream.

Router::ThreadRun()

The router function implements the routing algorithm. It is the most important function of the application. Every packet that is received from the network or created by a module sooner or later will traverse this function (as a packet message). Depending on where the packet came from this function will forward or drop it. In the current implementation this function is also responsible to forward the packets to the Analyser. For a more detailed discussion about this function see Sect. 3.1.4.
If you want to change the routing algorithm you will have to exchange or adapt at least this function. If you want to implement a completely different routing algorithm you may have to rewrite the Router class and probably the PacketMessage class that is responsible for the packet layout at byte level. If you want to change the protocol from UDP to RAW sockets you have to adjust the PacketMessage class and the network layer which is an easy task thanks to its modularity. Everything else can be left alone.

Sender::ThreadRun()

The sender thread receives packet messages from the router thread and sends them over the network socket. This action is repeated as long as there are messages in the queue. When an error occurs the packet will be discarded and a log entry generated. To prevent packet send bursts we have implemented a simple low pass filter that slows down sending according to how many packets are waiting. The delay gets smaller the more packets are waiting. The first packet in the queue is always sent immediately. The longest delay is 500 ms.

Receiver::ThreadRun()

The receiver thread waits until a packet has arrived from the network socket and hands it over to the router thread.

Analyser::ThreadRun()

The analyser thread creates log entries for every packet it receives. After fetching the next message from the queue, it updates some network statistics. Most of the simpler network statistics are realised as public member variables and changed directly by other threads or objects. The analyser thread does not care where a packet came from. In our current implementation only the router thread sends messages to the analyser. For a description of the log file format see App. B. For a detailed discussion about this function see Sect. 3.1.6.

AutomatorThread()

The Automator is a special thread implemented in `main.cpp`. It adds simple script support to the application. Whenever you need to automate the application you can put your code into this function. The Automator has an enable-flag which is set to false by default. As soon as the flag is set to true the Automator starts executing your code once and stops when it is done. If you like you can restart the Automator after its job is done. We have used the Automator to make measurements with different streaming rates when we did not want to take manual actions.

Here is an example how a scripted measurement could look like:

```
LogMessage("Automator starting");

app->streamer->StreamStop();

/////////////////////////////////////////////
```

```
LogMessage("Measuring at 5.5 Kb/s");
app->streamer->SamplingRate(5500);
app->streamer->StreamStart();

// wait 5 min
PI_Sleep( 300000 );
app->streamer->StreamStop();
app->analyser->LogRotate();

////////////////////////////////////////////
LogMessage("Measuring at 11 Kb/s");
app->streamer->SamplingRate(11025);
app->streamer->StreamStart();

// wait 5 min
PI_Sleep( 300000 );
app->streamer->StreamStop();
app->analyser->LogRotate();

////////////////////////////////////////////
LogMessage("Measuring at 22 Kb/s");
app->streamer->SamplingRate(22050);
app->streamer->StreamStart();

// wait 5 min
PI_Sleep( 300000 );
app->streamer->StreamStop();
app->analyser->LogRotate();

LogMessage("Automator stopped");
```

3.1.4 Routing

The router class consists of a set of member functions that implement the flooding algorithm and a simple service discovery mechanism.
The class contains two functions which run as thread: `Router::ServiceDiscovery()` and `Router::ThreadRun()`.
A message arriving at the routers message queue can have three different origins as you can see in Fig. 3.1:

1. `Router::ServiceDiscovery()` (ID_DISC)
2. `Receiver::ThreadRun()` (ID_RECV)
3. `MainApplication::Main()` (ID_MAIN)

Messages received from the service discovery or from the main application loop are instantly forwarded to the sender thread.
Messages received from the receiver thread are subject for routing. The decisions that need to be done for such packet messages are best described in the flowchart in Fig. 3.3.
As you can see in the flowchart the router class checks whether a received packet is an echo meaning that the packet has already been forwarded once. To be able to detect this we have to remember the packet. This is done using the *route list*. The route list is just an STL vector of the *RouteInfo* structs. A route therefore consists of the source IP the packet originally came from and the service id of the packet (whether it is a discovery, chat or audio packet). The RouteInfo struct also contains a list of sequence numbers to let us know if a packet is an echo or not. The list is configured to remember no more than 500 sequence numbers (adjustable in `config.h`). If the number is too big the list takes more memory, if it is too small it cannot remember all packets. If the list is full the packet with the smallest sequence number is replaced. If this packet is received again it is no more recognized as echo and forwarded again.
Since it is possible for packets to outrun other packets it is not enough if we only remember the highest sequence number or the number of the last packet received. If a route does not get updated for 5 seconds (adjustable in `config.h`) it is removed from the route list. Note that the

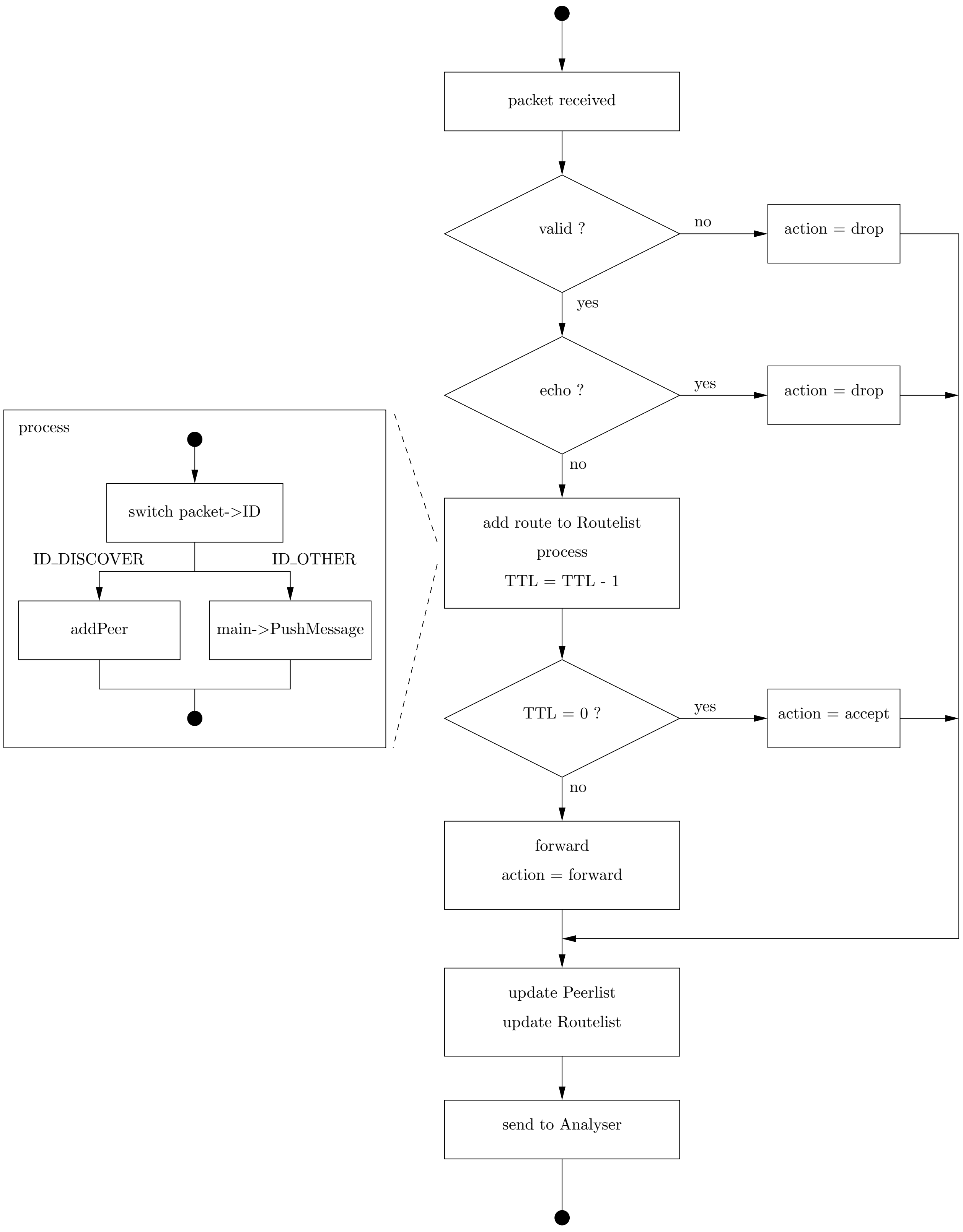


Figure 3.3: Router::ThreadRun() flowchart for packets received from the network

only purpose of the route list is to prevent ping pong flooding of messages. It is absolutely no problem if a route dies and revives during a data transfer.

Another mechanism to prevent endless forwarding of a packet is the TTL of a packet. The implementation should behave identical to the one used in TCP/IP. Upon receiving the packet the TTL is decreased by one. If it reaches zero the packet will not be forwarded again.

3.1.5 Network Packet Format

The network packet format is implemented as a message and hence it is defined in `message.h`. If you want to create a new network packet you have to create an object of type *PacketMessage*. This class provides you with all functions that are needed to fill in your data or (if you have received a packet from the network) to get the data out of the packet. We used two different packet types that are very similar.

Discovery Packet is the packet that tells you that a device is in range, what services the device offers and the device name.

Payload Packet is the most used packet type. It contains a payload field that can carry arbitrary data. We have used this packet type to send audio data and chat messages.

Both packet types are defined in class *PacketMessage*. If the current implementation does not fit your needs you can easily add more types and extend class PacketMessage accordingly. Every packet contains an *ID* field that tells you what type it has. The data itself is stored in a *ByteStream* which is actually just a `typedef` to an STL vector of UInt8. If you have never worked with the STL before a vector behaves just like an array in C. Since every field in our packet has a fixed length the implementation is really trivial. We work directly on the ByteStream meaning that i.e. the `setTTL()` function does nothing more than just converting your number to a UInt8 and then putting it at index 3 in the ByteStream.
The class does not check whether you have filled out all mandatory fields or if the data you have filled in actually makes sense. The only check that is done is that i.e. a timestamp you set is put at the correct place and does not overwrite succeeding fields. The same applies for "get" functions. They return the data stored at that position to you. There is a function that tells you if a packet `isValid()`. Since we store the packet length at the beginning of the packet we have a redundancy here. For a received packet the propagated length must match the actual received length and the actual length must be at least 20 bytes. Aside of that, every "get" function returns sane values even if the packet is shorter than the position of the field you want to access.

The following fields are common in all packets:

Field	Size (bytes)	Description
ID	1	The type of the packet. See enum for Service IDs in `message.h`.
Length	2	The length of the packet including all header fields.
TTL	1	The TTL of the packet.
Sequence Number	4	The sequence number of the packet
Source IP	4	The source IP of the host that created the packet originally
Destination IP	4	The destination IP of the host that should ultimately receive the packet
Timestamp	4	The timestamp of creation. See PL_Timestamp() in App. D.2.3

Discovery packets have two additional fields. For a graphical representation see Fig. 3.4.

Field	Size (bytes)	Description
Service Mask	1	The bitmask that carries all offered services. See enum for Service IDs in `message.h`.
Device Name	xx	The name of the device.

Payload packets only have one additional field for the payload. For a graphical representation see Fig. 3.5.

Field	Size (bytes)	Description
Payload	xx	The payload of the packet.

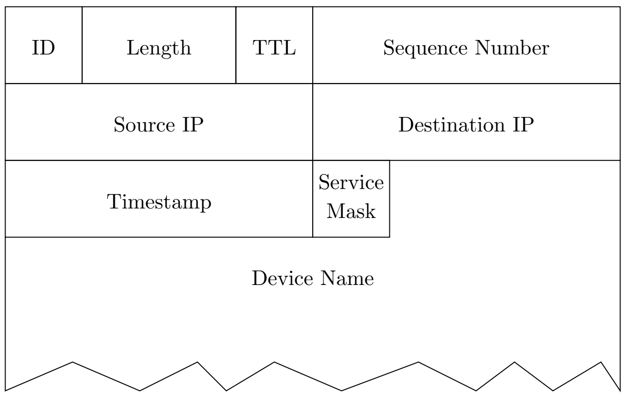


Figure 3.4: Network packet format of a discovery packet

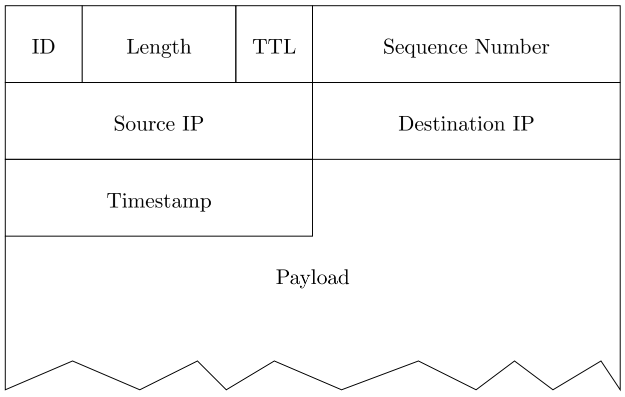


Figure 3.5: Network packet format of a payload packet

3.1.6 Analyser

The analyser class takes care of our log files. Every packet received is converted to a semicolon separated log line and appended to the logfile. Upon program start a new directory with the current date is created. All log files from this session are saved in that directory.
See Fig. 3.6 for a flowchart that describes the most important part of the analyser.
If a file gets too big the analyser automatically closes this file and continues logging to another file with a new index.
In any case where logging is not possible (i.e. because there is no storage card or the storage card is full) the analyser stops logging and sends a message to the debug log.
To prevent data loss when the application crashes or the battery is low the analyser ensures that the data is flushed to the storage card. Although there is a function in C called `fflush()` that should force the operating system to write the data to the storage card, it did nothing in Windows CE. We had to manually close and re-open the file to make sure everything is saved properly.
You can adjust many options of the analyser in `config.h`.

3.1.7 Modules

This section describes the functionality and the implementation of the modules. They are unimportant for the routing algorithm. You can easily add more modules or remove the existing ones.

Chat Module

The chat module is the simplest one. If you intend to create a new module have a look at the source code in file `chat.cpp`. Basically the module implements a subscription server to make the

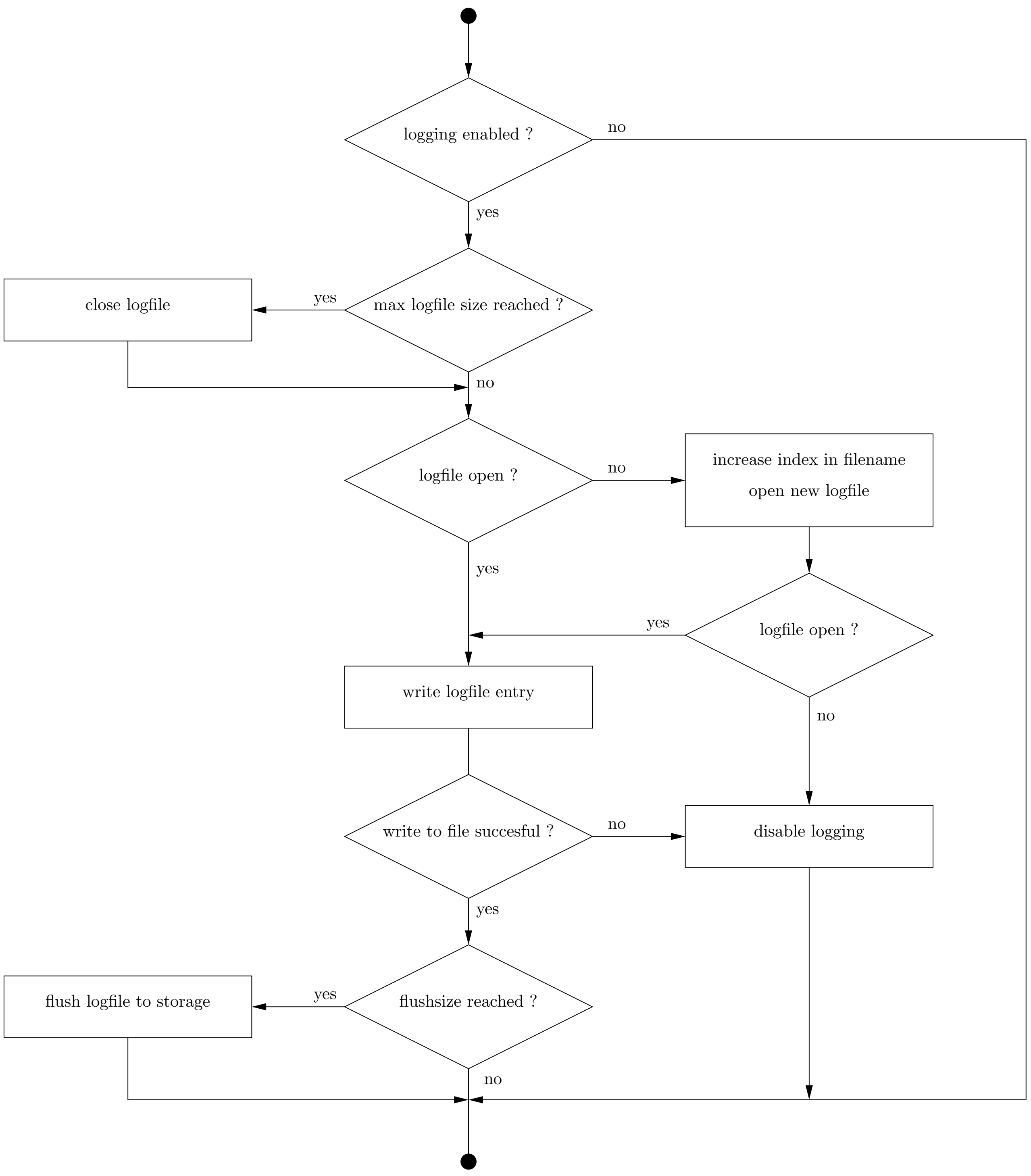


Figure 3.6: Analyser::Log() function flowchart

chat messages available to the GUI and a `Send()` function that creates chat messages and hands them over to the router thread.

Remark that the design-chart in Fig. 3.1 indicates that the messages are first sent to the main loop and forwarded to the router thread by the main loop. In reality this is not true. The chat module assigns the message the originator ID of the main loop and then sends the message directly to the router thread. This has no effect on the routing. It just is a little bit faster and makes the application use less CPU.

Audio Streamer

The streamer module is an example how to do real data transfers. It opens a file on the storage card and starts reading it. The content is filled into packets of fixed size and then forwarded to the router thread. As for the chat module the data is sent directly to the router thread without indirection.
Since we are streaming audio we have to make sure that we are not streaming too fast or too slow. This is done by measuring the time between the calls to `AudioStreamer::Update()`. The streamer then calculates how many packets it is behind. This works great on notebooks however the problem with iPAQs is that the internal clock is very imprecise. They are either too slow or too fast. Since the buffer in the audio player is 3 – 4 seconds and the clock is off by one second per minute it only takes a few minutes until we have a buffer overflow or underflow.

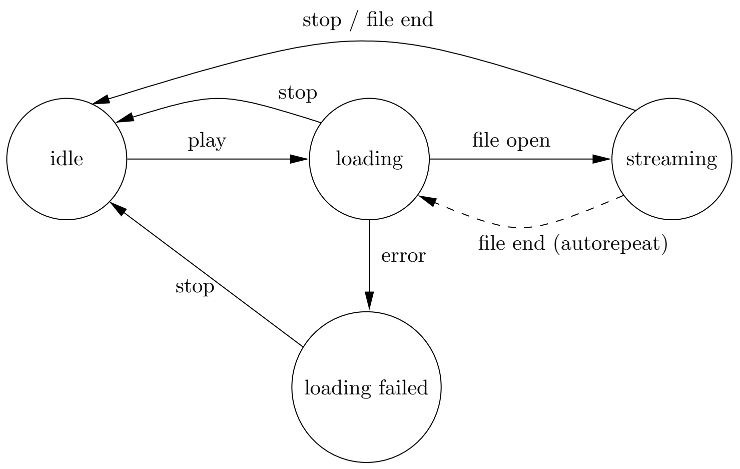


Figure 3.7: AudioStreamer state diagram

Figure 3.7 shows the state diagram of the streamer. Eveything else is a straight forward implementation and can be best examined directly in the source code (see `sound.cpp`).

Audio Player

The audio player receives audio packets and starts playing the data through the internal sound card. This is done using the very easy to use FMOD library[2]. We have used version 3.75 for Windows CE. The original file can be found on the cdrom at `sa_ahcomm/software/iPAQ/fmodapi375ce.zip`. It is platform independent and should work in Windows 2000/XP as well as in Linux. This was not tested by us.
Lets start with a simple state diagram. The only thing to mention here is that the player will only play music from one source at the same time. By using the tap'n'hold feature you can tune in manually to a specific peer (if it provides an audio stream). If you are not yet tuned to a channel the player automatically tunes in to the first audio packet it receives. This makes it possible to just hit the play button in the radio dialog without choosing a channel first. Remark that the player once tuned stays tuned to that IP address. If you lose the stream the player waits until it receives data again from that IP.
An annoying property of our flooding algorithm is that packets may outrun other packets which means that the audio packets don't need to arrive in the correct order. As long as we receive packets in order we can start playing them. But as soon as one sequence number is missing it could be for two different reasons:

- The packet is ultimately lost because of collisions or because we were not in range when it was sent.
- The packet is late.

To deal with this problem we first have to buffer all incoming packets in a self sorting queue called *msg_cache*. If we don't lose any packets they end up in the correct order in the msg_cache. If

[2]see `http://www.fmod.org/`

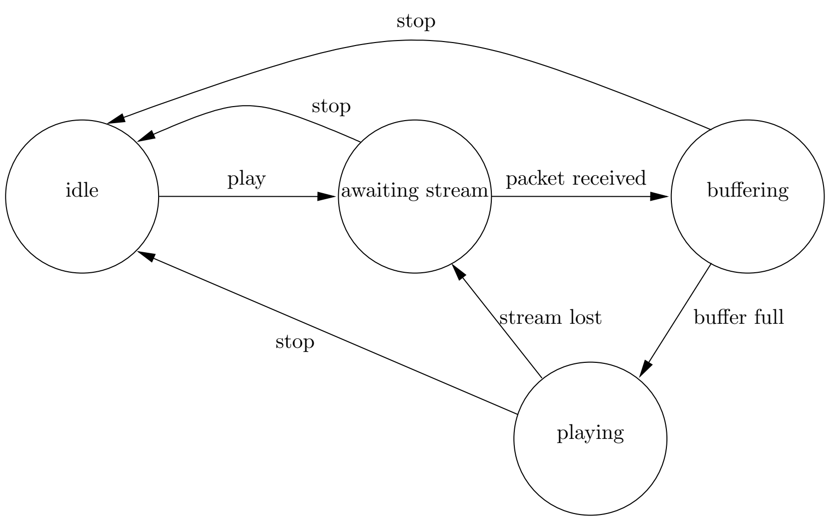


Figure 3.8: AudioPlayer state diagram

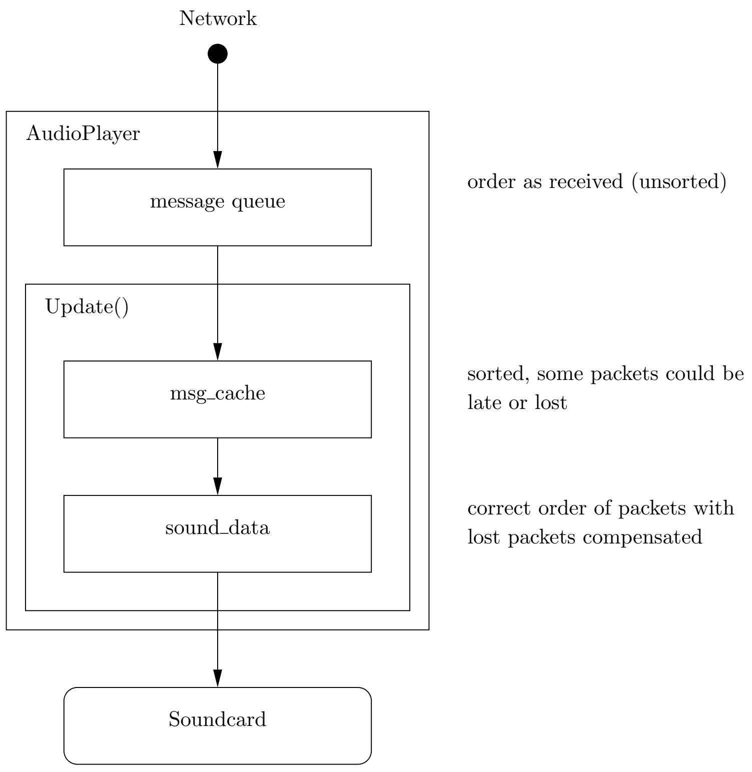

Network
AudioPlayer
message queue
order as received (unsorted)
Update()
msg_cache
sorted, some packets could be late or lost
sound_data
correct order of packets with lost packets compensated
Soundcard

Figure 3.9: AudioPlayer::Update() packet flow

packets are missing in the msg_cache we have to make sure that they are not late. If this happens we just wait some time and if they are still missing we consider them lost and compensate them with silence or white noise so we don't lose sync.
The flowchart in Fig. 3.9 shows exactly what we are doing. Since we have a timeout for late packets it could be that we consider a packet lost and jump over it but it arrives later. In this case the packet is dropped and a message is sent to the debug log.

Command Line Interface

The Command Line Interface (CLI) is only active if you compile the program under Windows, Linux or Mac OSX. You can find its implementation in `main.cpp`. The CLI runs in a separate thread and does not show up in the application design (see Fig. 3.1). Think of it as a replacement for the GUI.

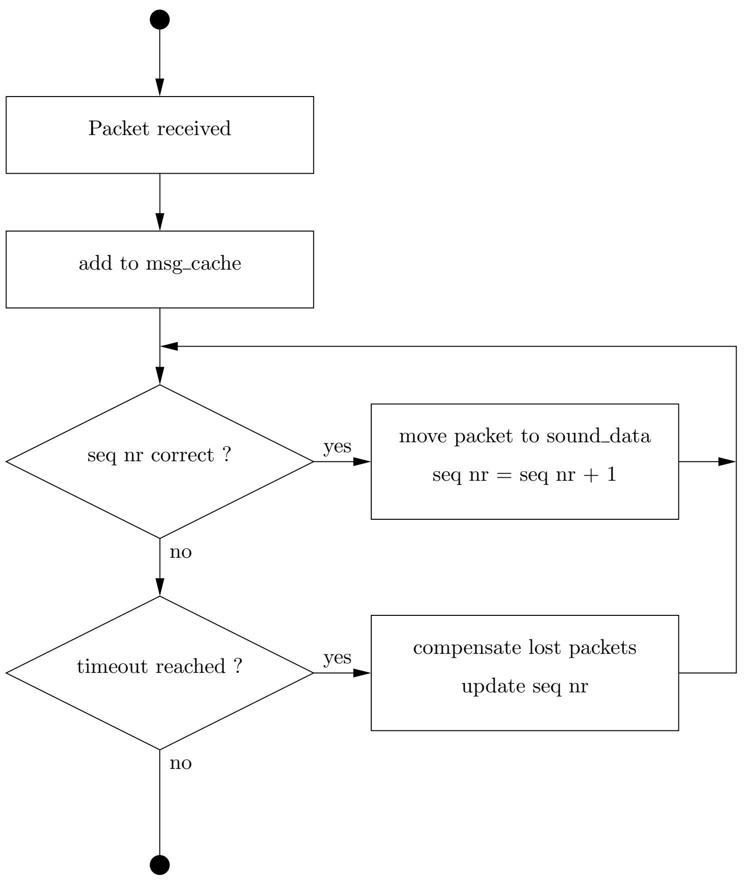


Figure 3.10: AudioPlayer::Update() flowchart

Currently it supports four commands as you can see from the following table.

Command	Description
exit	Exits the program
play	Starts playing the first audio stream found (requires fmod)
stream	Starts streaming an audio file
stop	Stops streaming the audio file

Adding new commands is very simple. Just have a look at the source code.

3.1.8 Debugging

Debugging is done using debug messages. There are two functions that let you create debug messages:

- LogMessage(const char* message, ...)
- LogMessage(std::string& message)

The first one behaves exactly like `printf()`. The second one just takes a C++ string. Both functions create a *LogTicket* with a timestamp and your message and add the ticket to the internal debugging facility.
The current implementation of the debugging facility is rather trivial:

- The messages are appended to a debug file which is stored on the storage card. You can change it as usual in the file `config.h`.
- When compiled under Windows CE the messages are stored in a vector as well, so that the debugging window can access the messages.

- If compiled under Windows, Linux or Mac OSX the messages are also sent to the standard output.

To make the functions available just `#include "debug.h"` in your file.

Chapter 4

Measurements

4.1 MTU Test

In our very first measurement we wanted to find out what the ratio between lost and received packets is depending on a variable Maximum Transfer Unit (MTU) size. Here we define the MTU size as the maximum amount of data in bytes which the program packs into a UDP packet of size 1500 bytes. The network topology we have chosen for this test was simply two iPAQs side by side. The streaming datarate was constantly 22 kB/s. The plot of the results shows that the smaller the difference between 1500 bytes and the chosen MTU size the smaller the packetloss which seems straightforward. In other words, if we pack more bytes into a packet, fewer packets will be sent in both directions, therefore fewer collisions will happen.

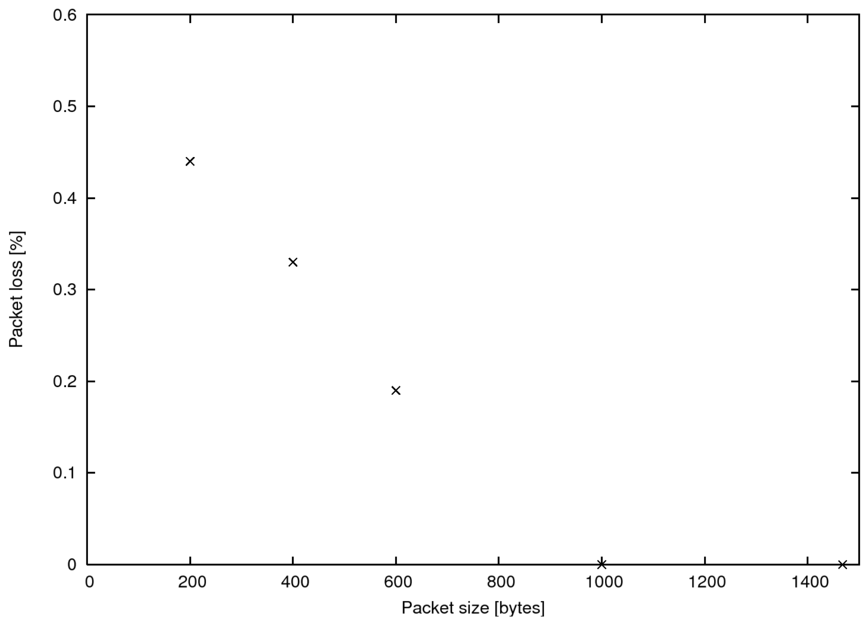


Figure 4.1: MTU Plot 1

4.2 Packetloss Under Optimal Conditions (L-Setup)

In this measurement series we tried to make an ideal L-setup, which is shown in Fig. 4.2. Node 1 is streaming an audio file and therefore called ”Streamer”, node 3 is at the end of the L-chain and therefore called ”Endpeer” and node 2 is called ”Router”. Endpeer and Router just play and flood what they receive.
The first goal of such a setup was to minimise the noise level produced by the Streamer at the Endpeers position. Therefore we accomplished this measurement in the underground garage of

the ETZ building where the concrete walls would do this job and no other WLAN devices (like access points and laptops) would interfere.
The second goal was to measure the Routers and the Endpeers packetloss of all audio packets sent by the Streamer depending on the streaming information rate and mode. In ”Single continous packet stream” mode there is a constant time gap between two successive packets. ”Double continous” mode is the same except that the Streamer sends every packet twice. Note that for this mode the information rate on the x-axis (Fig. 4.3 and 4.4) must be doubled to get the effective datarate. In contrast, in the ”Single burst” mode a burst of seven packets is sent. After that the Streamer is silent for a certain time before it sends another burst.
Considering Fig. 4.3 and 4.4 we see, that there is practically no difference between the ”Single continous” and the ”Single burst” mode for high datarates. The reason is that the ”Single burst” mode converges to the ”Single continous” mode by increasing the datarate, because then the length of the Streamers silence between two bursts is decreasing. In contrary, for low datarates the continous mode is better than the burst mode, because the probability of a collision of two packets decreases. Furthermore we can recognise that sending every packet twice results in a great improvement concerning an audio streaming application.

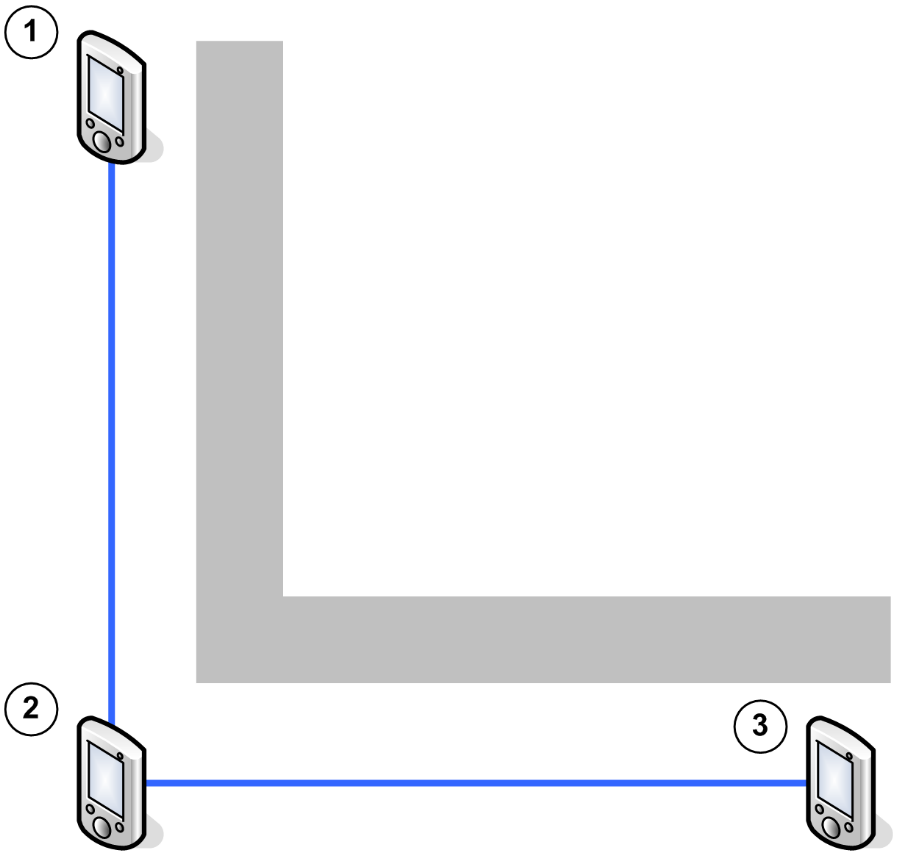


Figure 4.2: L-Setup

4.3 Walking Group Measurement

In this measurement we distributed 5 iPAQs incl. the Streamer to 5 people. The whole group took a walk for a couple of minutes during which we measured the packetloss. We wanted to find out, if there is a difference concerning the connection quality between the static and dynamic (walking) case. Figures 4.5 and 4.6 show, that there is practically no difference between those 2 cases. The reason for the peak of node 2 in figure 4.5 was a low battery status which shows how sensitive the used technology is.
However in this special case, where all receiving nodes are next to each other, we have a large factor of redundancy, meaning that every node receives the same packet several times.

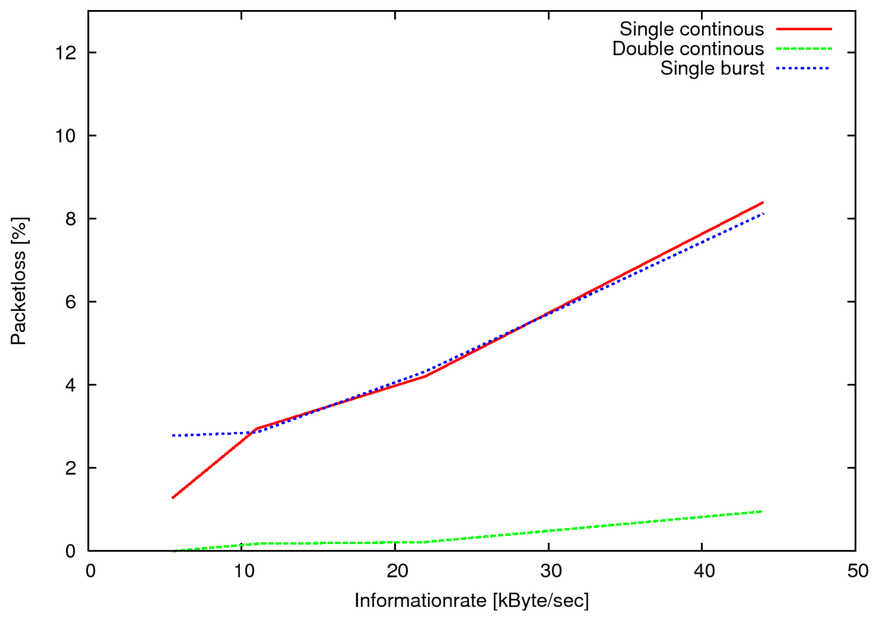


Figure 4.3: Packetloss at Router

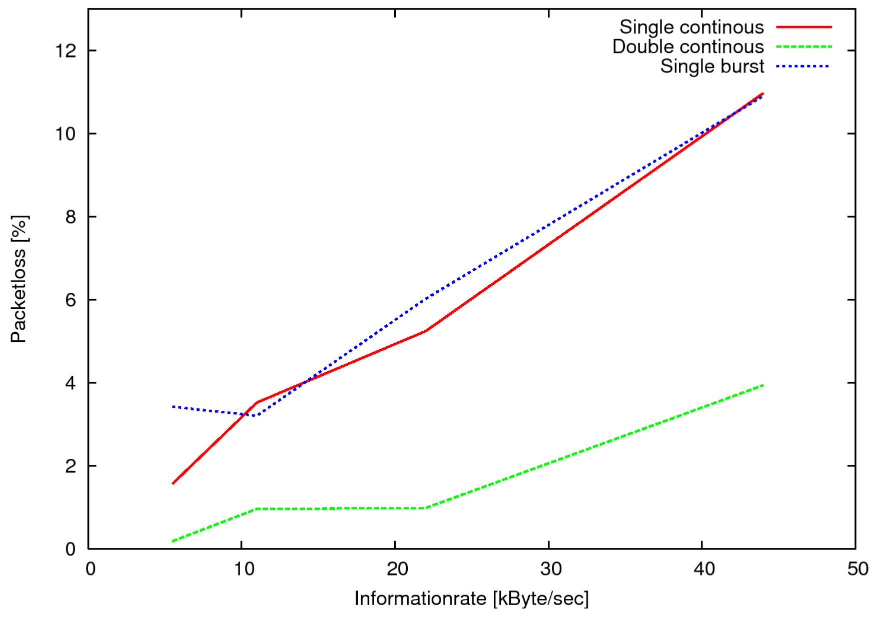


Figure 4.4: Packetloss at Endpeer

Furthermore, during some other measurements we have experienced a so-called ”slow tuning” problem, which makes the packetloss increase dramatically for a couple of seconds, if the position of a node has been changed. After a tuning phase the packetloss drops to the level it was before changing the position. We know from experience, that this slow tuning problem disappears by reducing the distance between the nodes. These two circumstances have caused a low packetloss during this measurement.

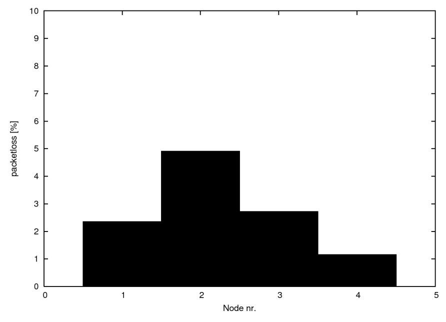


Figure 4.5: Not moving group measurement: packetlosses of the 4 receiving nodes

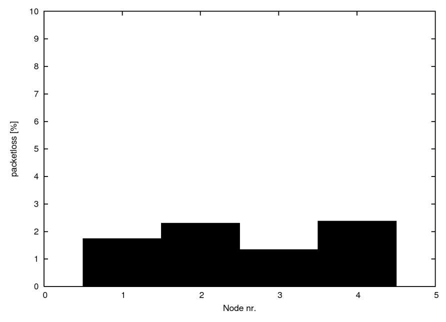


Figure 4.6: Walking group measurement: packetlosses of the 4 receiving nodes

4.4 Connection Length

Another interesting quality which roughly gives information about the networks topology is the avarage connection length. Figures 4.7 and 4.8 have been extracted from two different measurements. The x-axis on each plot represents the number of hops to the Streamer (i.e. connection length to the Streamer) and the y-axis is its occurence during the whole measurement given as a PMF.
The measurement setup in both cases was a dynamic network of 10 nodes (incl. Streamer), realised by distributing them to some people in the G-Floor who have their workplaces near the location where we placed the Streamer. Those people were told to carry the iPAQs with them, no matter where they go.
From the figures we can conclude that nodes 1, 3, 5 and 8 from measurement one and nodes 4, 7 and 10 from measurement two were stationary positioned somewhere near the Streamer most of the time. The other nodes had a connection mostly over two or three hops. Furthermore a node hardly had a connection for more than four hops to the Streamer, which means that the topology of both measurements was an accumulation around the Streamer.

4.5 Lost Packet Series

A high number of successive lost packets, referred to as ”lost packet series”, relating to our application can happen for two reasons:

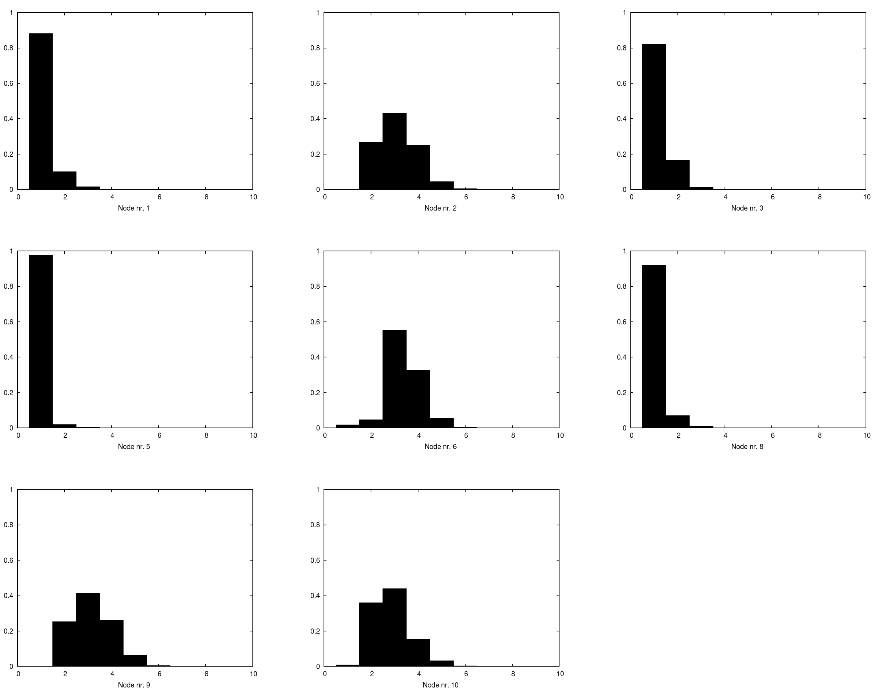


Figure 4.7: Measurement 1: PMF of Number of hops to the Streamer for each node

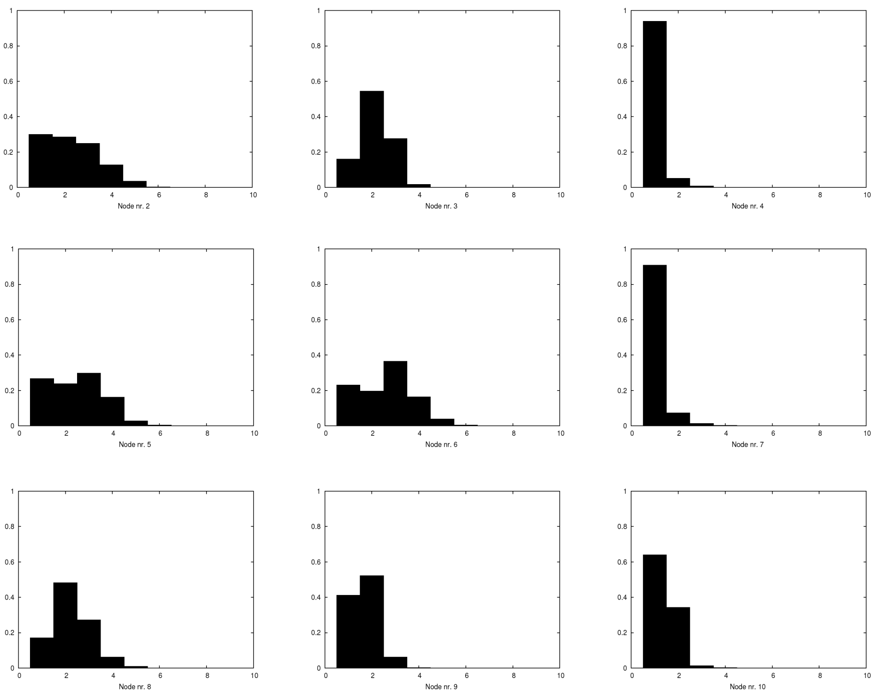


Figure 4.8: Measurement 2: PMF of number of hops to the Streamer for each node

- On one hand because of a change of position of nodes (slow tuning problem)
- On the other hand a bad connection to the Streamer caused i.e. by a big distance.

Figures 4.9 and 4.10 are a summary of lost packet series over all nodes from two different measurements with the same setups as described in Sect. 4.4. Both plots show, that the size of the most frequently lost packet series is 1, which means that either the network was rather stationary or the avarage connection to Streamer was good.

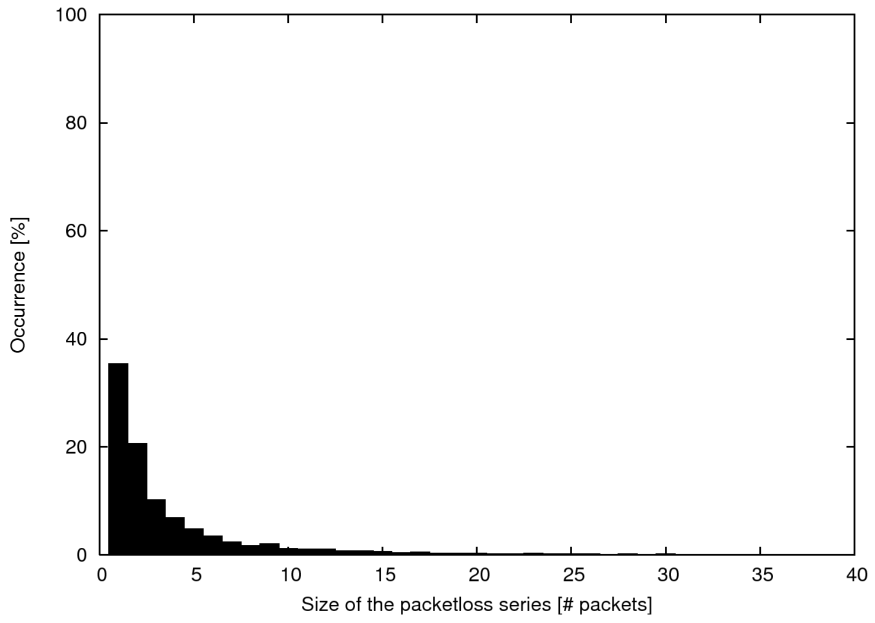


Figure 4.9: Measurement 1: Size of lost packet series and how often they occured

4.6 Delays

Measuring delays (i.e. how long does a packet need from one node to another) in our network were a basic problem for two reasons:

- On one hand it is not possible to get an absolute timestamp in milliseconds since some fixed point in time,
- on the other hand the time discrepance between iPAQs is very high, meaning that a general solution to this problem (for example Network Time Protocol (NTP)) might have to be applied. We did not analyse such delays for this would have gone beyond the scope of our time frame.

4.7 Connection Lifetime

During all our measurements we have used the connection-less UDP protocol. Now the question arises: how does the life time of a connection have to be defined? The way we did it is the following:
If a node doesn't receive n or more successive packets, then it has lost its connection to the Streamer. Usually a connection loss occurs when there is no route to the Streamer. Based on this definition we can compute the life time of a connection as follows:

$$\frac{packet\ size}{streaming\ datarate\ /\ 8} \cdot (seq.\ number\ last\ packet\ -\ seq.\ number\ first\ packet)$$

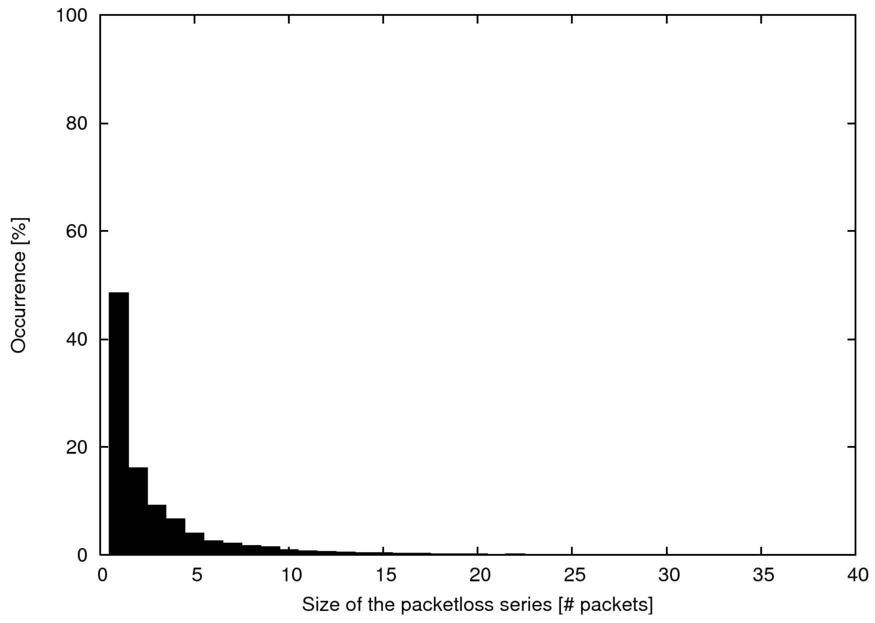


Figure 4.10: Measurement 2: Size of lost packet series and how often they occured

Figure 4.11 shows the CDF for all occured connection life times, totalised over all nodes (except the Streamer) involved in the measurement, depending of the definition of the term ”connection loss”, i.e. $n = 2$, 4 and 10. The measurement setup was the same as in Sect. 4.4
On one hand we can see, that for increasing n the mean life time increases too, therefore the CDF curve gets flatter. On the other hand we can say, that totalised over all three n's the connection to the Streamer have short life times, which is typical for dynamic networks.

4.8 Routing Overhead

The Routing Overhead consists of all received heartbeat packets (service id 1) and the header of all the other packets received by a node related to the payload data, which is audio in our case. For quantitative determination of the overhead one can either use the stats window (described in Sect. A.5) or use the information given in Sect. 3.1.5. The Routing Overhead in our implementation lies around 2% after a few minutes of audio streaming.

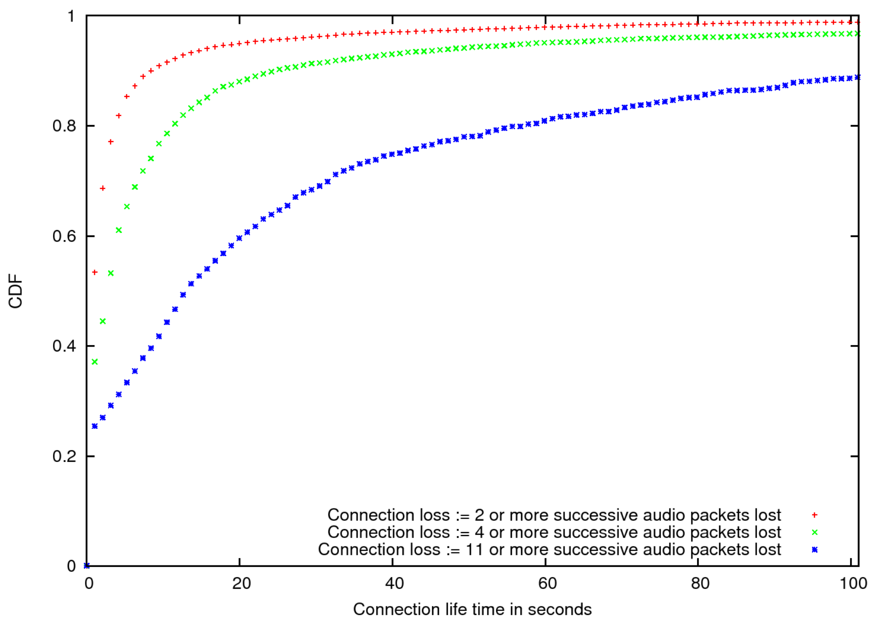


Figure 4.11: CDF

Chapter 5

Conclusions

5.1 Conclusion & Prospect

This semester thesis is mainly divided into two parts:

- Implementation of a flexible and modular measurement application
- Measurements and analysis of various static and mobile scenarios

We have created a completely new application that runs on the HP iPAQs as well as on other platforms. We have compiled and tested the application under Windows 2000, Linux and Mac OSX. The program is based on a clean design which makes it concise, easy to maintain and extensible. The routing algorithm currently implemented is a simple flooding scheme. It can be replaced easily since there are almost no dependencies to the surrounding application layers. All network traffic that is received by the device is sent to a customisable logger which stores the data in a file in Comma Separated Value (CSV) format making it easy to analyse the measured data in a spreadsheet application. We have learned that a good software design can speed up the development process heavily and help trace down bugs quickly. As a result our application has no known bugs left and a stable memory footprint even after several hours running time.
The conducted measurements have shown, that mobile ad hoc networks are very sensitive to external influences (cars, movement of the nodes, etc.) depending on the topology. Nevertheless such networks are suited for an audio application as shown in Figures 4.9 and 4.10, where mostly single packets get lost. We have observed, that with simple means already big improvements can be obtained (Double continuous mode) and certainly even bigger ones with elaborated methods (forward error correction, etc.). The clearly shortened life time of the iPAQs Li-Ion rechargeable battery due to the use of WLAN shows, that the energy budget plays a major role during the development and selection of those methods. Concerning the audio application it must be added, that we have always sent just raw samples without any compression. By the use of a suitable compression algorithm the quality of the service can be once again improved drastically.

5.2 Problems

During our work we encountered several problems we should mention here for they could affect future development in a negative way.

5.2.1 Software

- The C++ implementation on Windows CE is not complete causing difficulties and restrictions (i.e. there is no support for *Exceptions* or for *StringStreams*). There is a complete implementation of the STL for Windows CE provided by a third party. Since we didn't have knowlegde of this problem at the beginning of our work we did not used it and we had no time to integrate it into our program.
- Debugging gets very difficult if the program crashes at some point. In eMbedded Visual C++ development environment there is no way to display a stack trace after the program

has crashed. One way to work around this problem is either to add only a bit of new code and test the functionlity very often. Another way is to develop on another platform such as Linux or Mac OSX and use *GDB* as a debugger.

- The WLAN driver does not support all IOCTL commands as specified in the NDIS specification by Microsoft. We were not able to set a specific BSSID although it is a requirement in the specification. Additionally a BSSID scan should return a list of all BSSIDs in range but only the first entry in this array contains sane values.

5.2.2 Hardware

- The position of the devices can have a huge influence on packet loss. If you move a device around a few centimeters the overall packet loss can easily increase about 30% and more but stabilises as soon as you stop moving the device. This effect only appears if the sending device has a distance of at least 20 m to the receiving device. We did not see this effect in the walking group measurement in Sect. refresults:walking.
- The hardware clock in some of the iPAQs has a huge drift of about 1 second per minute! This makes measuring of transmission delays difficult. It also makes the analysis difficult if you want to compare log files of one device with those of another device.

5.3 Outlook

There are three different directions that can be taken from here.

1. Enhance the Audio Streamer and do more measurements
2. Use the application as a generic analysis tool
3. Change the routing algorithm and add additional modules accordingly.

5.3.1 Audio Streamer Enhancements

- We were able to show that the *double send*-solution decreases packet loss enormously. This has to be confirmed in a real life group measurement.
- More measurements should be taken with other devices such as notebooks. That way it could be possible to eliminate potential effects caused by the iPAQ hardware.
- Instead of sending every packet twice the protocol could be enhanced so that packets contain redundant information that could enable forward error correction.

5.3.2 Using the Application for New Tasks

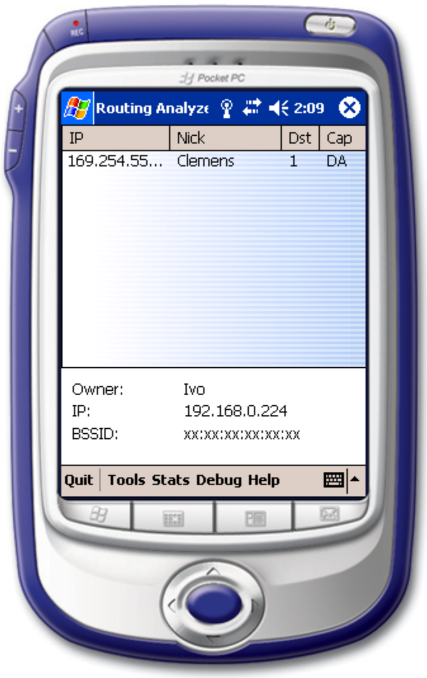
- Exchange the flooding scheme with a real routing algorithm.
- Add more modules

Appendix A

Routing Analyser Program Manual

A.1 Main Window

Lists all peer devices in range. Dst stands for Distance in hops (a direct connection denotes as 1 hop). Cap stands for Capabilities (D - Discoverable, A - AudioStreaming, C - ChatWindow open). You can tap'n'hold on the listbox to chat with a peer or to tune in to a specific audio stream. The rest should be self explanatory.



IP	Nick	Dst	Cap
169.254.55...	Clemens	1	DA

Owner:	Ivo
IP:	192.168.0.224
BSSID:	xx:xx:xx:xx:xx:xx

Figure A.1: Routing Analyser main view

A.2 Tools - Radio

The upper part of the window controls the audio player. If there is no preference of a specific audio streamer the device auto tunes in to the first received audio packet. If the audio stream is lost it remains tuned in, unless you click stop and play again.

The lower part controls the audio streamer and should be self explanatory.

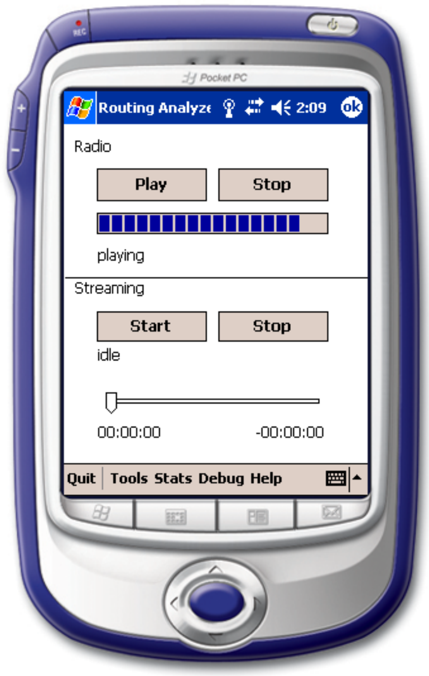


Figure A.2: Routing Analyser radio screen

A.3 Tools - Chat

The chat window behaves IRC like. Just enter text into the edit box on the top of the screen and press the send to all button to chat with all other peers currently in the network. Alternatively you can use the quickchat buttons below. The are customizable through the `/CF Card/quickchat.txt`: The first 6 lines of this file replace the built in quickchat messages.
If the chat window is opened via the menu bar messages are sent to all devices You can send a message to a specific peer device by using the tap'n'hold feature in the list box from the main window. Remark that personal messages are visible for everyone (just like it is in IRC chat)
Also remark that there is no retransmission for lost packets. If your chat message gets lost on its way to the destination you will not be informed of it. Send it more than one to increase the probability of success.

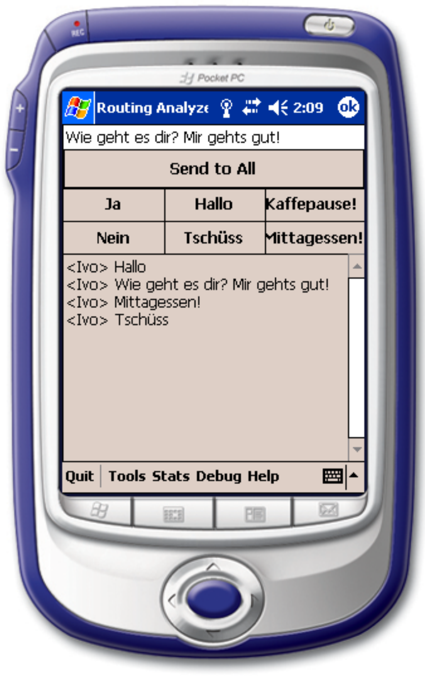
A.4 Debug Window

Lists all log messages created by the application. The messages are also stored on the CF Card in file debug.txt. Remark that the window itself can only store around 64KB of data. If the window is full no data is appended until you click the clear button.

A.5 Stats Window

A.5.1 Section Network (packets)

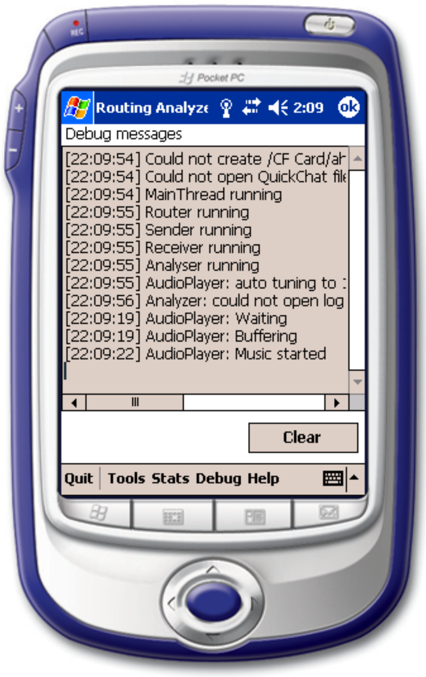
- sent: num packets sent by this device
- recv: num packet received by this device
- forw: num packet forwarded by this device
- IN rate: rate in kB/s of all received packets
- dropped: num of all dropped packets



Wie geht es dir? Mir gehts gut!

```
<Ivo> Hallo
<Ivo> Wie geht es dir? Mir gehts gut!
<Ivo> Mittagessen!
<Ivo> Tschüss
```

Figure A.3: Routing Analyser chat screen



```
[22:09:54] Could not create /CF Card/ar
[22:09:54] Could not open QuickChat fil
[22:09:54] MainThread running
[22:09:55] Router running
[22:09:55] Sender running
[22:09:55] Receiver running
[22:09:55] Analyser running
[22:09:55] AudioPlayer: auto tuning to
[22:09:56] Analyzer: could not open log
[22:09:19] AudioPlayer: Waiting
[22:09:19] AudioPlayer: Buffering
[22:09:22] AudioPlayer: Music started
```

Figure A.4: Routing Analyser debug screen

- echo: packets dropped because they were echoes
- ttl=0: packets dropped because ttl was 0
- !valid: packets dropped because they were invalid

A.5.2 Section Routing (bytes)

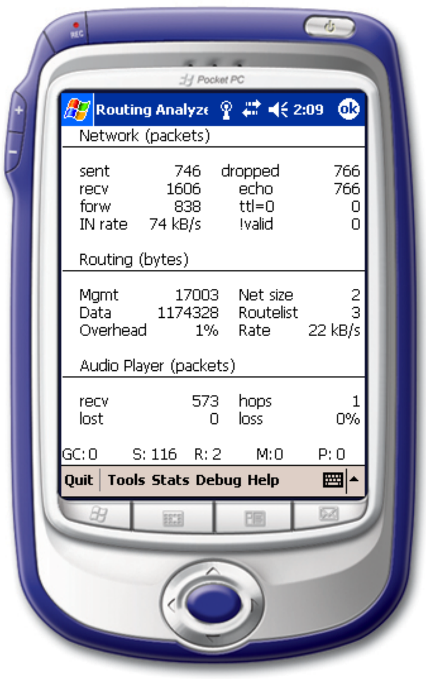
- Mgmt: num received bytes of management data
- Data: num received bytes of (audio/chat) payload data
- Overhead: Mgmt / (Data + Mgmt) * 100
- Net size: num of devices currently in range (also over more than one hop)
- Routelist: num of active routes (every service on a device creates a separate route)
- Rate: rate in kB/s of all forwarded packets

A.5.3 Section Audio Player (packets)

- recv: num of received audio packets
- lost: num of lost audio packets
- hops: hop distance of last packet processed
- loss: current packet loss in

A.5.4 Section Last Line

- GC: num of messages scheduled for deletion by garbage collector
- S: num of messages waiting to be sent
- R: num of messages waiting at router thread
- M: num of messages waiting at main thread
- P: num of messages waiting at audio player



```
Network (packets)

sent          746   dropped      766
recv         1606     echo       766
forw          838     ttl=0        0
IN rate   74 kB/s     !valid       0

Routing (bytes)

Mgmt        17003   Net size       2
Data      1174328   Routelist      3
Overhead       1%   Rate     22 kB/s

Audio Player (packets)

recv          573   hops           1
lost            0   loss          0%

GC: 0    S: 116   R: 2    M: 0    P: 0
```

Figure A.5: Routing Analyser statistics screen

Appendix B

Logfile Format and Analysis

The log files are saved in CSV format. This makes it easy to analyse the files in a spreadsheet application like Microsoft Excel[1]
The values are separated by semicolons (;). The first line of every file contains labels that describe the contents of every column. A typical excerpt from a file looks like this:

```
ISO Time;Timestamp;Source IP;Predecessor hop IP;Service ID;Seq Nr;TTL;BSSID;Action
2006-02-10T18:33:08;49875032;192.168.0.69;192.168.0.52;2;2265;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:08;3523427;192.168.0.66;192.168.0.66;1;0;20;c2:39:ee:1d:0b:b7;5
2006-02-10T18:33:08;49875104;192.168.0.69;192.168.0.52;2;2268;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:08;49875032;192.168.0.69;192.168.0.66;2;2265;18;c2:39:ee:1d:0b:b7;5
2006-02-10T18:33:08;49875155;192.168.0.69;192.168.0.52;2;2270;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:08;49875297;192.168.0.69;192.168.0.52;2;2274;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875748;192.168.0.69;192.168.0.52;2;2285;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875774;192.168.0.69;192.168.0.52;2;2286;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875830;192.168.0.69;192.168.0.52;2;2287;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875104;192.168.0.69;192.168.0.66;2;2268;18;c2:39:ee:1d:0b:b7;5
2006-02-10T18:33:09;49875859;192.168.0.69;192.168.0.52;2;2288;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875890;192.168.0.69;192.168.0.52;2;2289;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875926;192.168.0.69;192.168.0.52;2;2290;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875956;192.168.0.69;192.168.0.52;2;2291;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49875984;192.168.0.69;192.168.0.52;2;2292;18;c2:39:ee:1d:0b:b7;2
2006-02-10T18:33:09;49876011;192.168.0.69;192.168.0.52;2;2293;18;c2:39:ee:1d:0b:b7;2
```

Column	**Content**
ISO Time	Time when packet was logged.
Timestamp	Time when the packet was sent from the originating device. See PL_Timestamp() in App. D.2.2 for more information.
Source IP	The IP address of the device that created the packet.
Predecessor Hop IP	The IP address of the device we have received this packet from.
Service ID	The service id of the packet (discover(1),audio(2), chat(4)).
Seq Nr	The sequence number of the packet (every service has its own sequence numbers).
TTL	The time-to-live of a packet. Exact value depends on action taken by the routing. See Fig. 3.3.
BSSID	The BSSID this device was connected to when receiving the packet.
Action	The action, that was taken by the routing thread. See *enum for Packet actions* in file `message.h`

If you want to change the logfile format have a look at the file `analyser.cpp`. You can change the layout and the content in function `Analyser::BuildLogLine()`.
You can analyse the files using Microsoft Excel or if the files are too big use a perl script and/or grep et al.

[1]Older versions of Excel do not recognize CSV files properly if you double click them. In that case change the files extension to `.xls` and double click the file.

Appendix C

Development Environments

C.1 Windows CE

C.1.1 Overview

This section describes the software tools used in development and how to develop your program using eMbedded Visual C++ and download it to the device or the emulator.

C.1.2 Software Installation

The software installation is a little bit tricky since the order in which the programs have to be installed is important. The software needed can be found on the CD-ROM at `/sa_ahcomm/software/installation/devel_kit/`

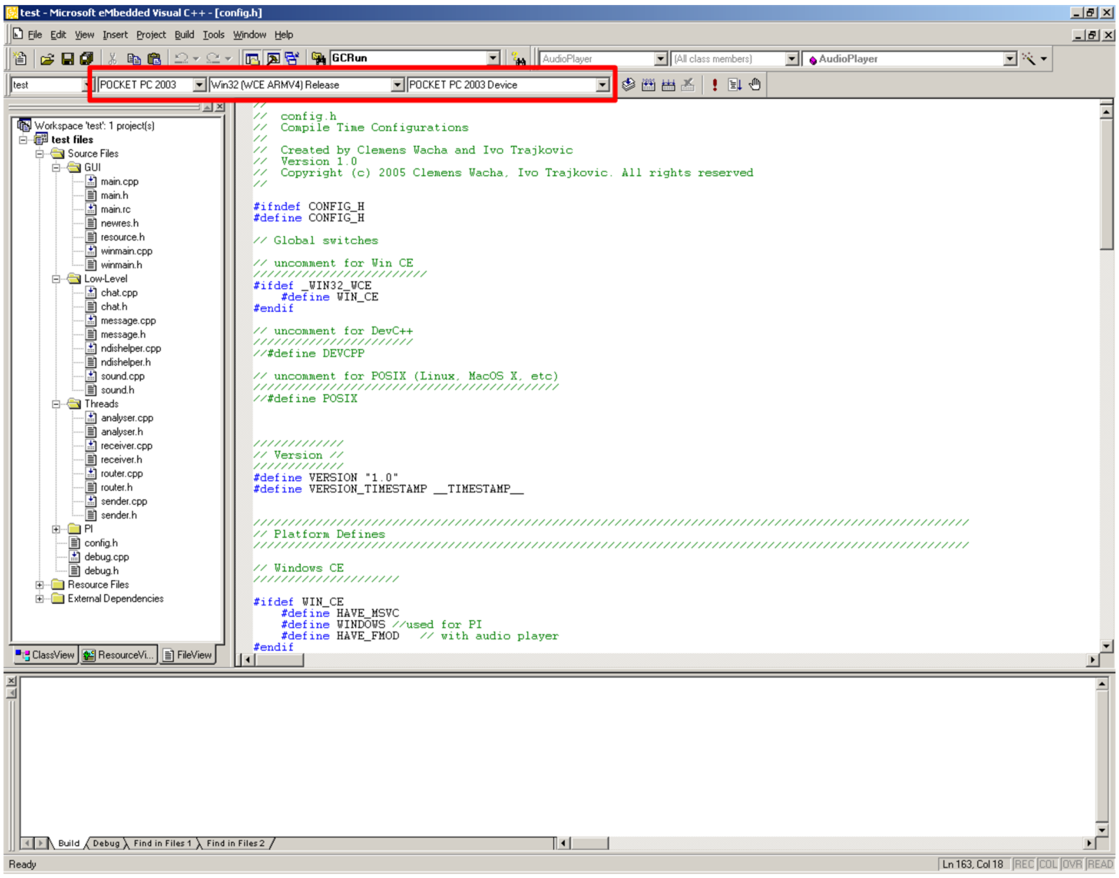
1. Install ActiveSync (MSASYNC.exe)
2. Install embeddedVisual C++ 4 (embeddedVisualC++4.exe)
3. Install embeddedVisual C++ 4 Service Pack (evc4sp4.exe)
4. Install Microsoft Pocket PC 2003 SDK (Microsoft Pocket PC 2003 SDK.msi)
5. You might have to install the *Microsoft Loopback Network Adapter* to make automatic downloading from within eVC++ work.

C.1.3 eMbedded Visual C++ 4.0

Microsoft eMbedded Visual C++ 4.0 is a stand-alone integrated development environment. It allows you to create applications and system components for Windows CE 4.2 based devices. In order to generate code for a specific device it requires a Software Development Kit (SDK) depending on the platform to target. eMbedded Visual C++ 4.0 includes the native code C and C++ compilers. The code is based on the Win32 API, optional on the Microsoft Foundation Classes and/or ATL APIs.

The Application

First of all you have to make sure that everything is setup correctly. In the *WCE Configuration Toolbar* (see Fig. C.1) make sure that you have selected *Pocket PC 2003*, *Win32 (WCE ARMV4) Release* and *POCKET PC 2003 Device*.
Choosing release mode has the effect that there are no debugging symbols added to your application. This makes the application faster and we could not find a debugger that could tell us a backtrace of the program execution if an error occurred. This was quite problematic since the only debugging facility left is our own programmed *printf* replacement called LogMessage().
To build the application click *Build -> Rebuild All*. The application is compiled and automatically transferred to the device if it is in the cradle. If you want to change the path the file is downloaded to on the device you can do so by opening *Project -> Settings*. Switch to the *Debug* register and set the path in the edit box *Download Directory*.


test - Microsoft eMbedded Visual C++ - [config.h]

test | POCKET PC 2003 | Win32 (WCE ARMV4) Release | POCKET PC 2003 Device

```
//
//  config.h
//  Compile Time Configurations
//
//  Created by Clemens Wacha and Ivo Trajkovic
//  Version 1.0
//  Copyright (c) 2005 Clemens Wacha, Ivo Trajkovic. All rights reserved
//

#ifndef CONFIG_H
#define CONFIG_H

// Global switches

// uncomment for Win CE
/////////////////////////
#ifdef _WIN32_WCE
    #define WIN_CE
#endif

// uncomment for DevC++
///////////////////////
//#define DEVCPP

// uncomment for POSIX (Linux, MacOS X, etc)
////////////////////////////////////////////
//#define POSIX



/////////////
// Version //
/////////////
#define VERSION "1.0"
#define VERSION_TIMESTAMP __TIMESTAMP__


//////////////////////////////////////////////////////////////////////////////////////////////////////
// Platform Defines
//////////////////////////////////////////////////////////////////////////////////////////////////////

// Windows CE
/////////////////////

#ifdef WIN_CE
    #define HAVE_MSVC
    #define WINDOWS //used for PI
    #define HAVE_FMOD   // with audio player
#endif
```

Figure C.1: eMbedded Visual C++ main view

The program also needs the fmodce.dll library for FMOD support. This file is not automatically copied to the device so you will have to do this manually. From `routing_analyser/api/wce4/armv4` copy the file fmodce.dll to the same folder on the device where you installed the routing_analyser.exe otherwise you will encounter an error message:

```
Cannot find 'routing_analyser' (or one of its components). Make sure the path
and filename are correct and all the required libraries are available.
```

Restrictions

There are some restrictions regarding this development environment and Windows CE:

- Microsofts implementation of C++ and the STL is not complete.
- iostreams are missing completely
- C++ exceptions are not supported
- The integrated debugger is useless (does not even display a backtrace of the current execution)
- On Windows CE it is not possible to access/create RAW sockets
- BSSID scanning does not work as it should (only the first found BSSID is returned)
- Setting the BSSID seems not to be supported by the driver

Emulator

The program can also be run inside the emulator. To do this you have to switch some settings in the *WCE Configuration Toolbar*. Change the options to *Pocket PC 2003*, *Win32 (WCE emulator) Release* and *POCKET PC 2003 Emulator*.
After compiling the program you will have to start the emulator by hand because in its default configuration the emulator does not allow incoming network traffic. You have to start the emulator with the command line option `/Ethernet virtualswitch`. If you launch the emulator from the command line you will have to provide a lot of other options such as the pocket pc ROM image and the skin to use. We have created a batch file that contains all options needed. Depending on your windows version and language you might have to make some adjustments. The file can be found on the CD-ROM at `/sa_ahcomm/software/computer/Emulator_start.bat`.
After launching the emulator with all options you have to install the program by hand:

1. Setup a shared folder that points to `/sa_ahcomm/software/iPAQ/routing_analyser` (if not already done by the batch script).
2. Start the *File Explorer* application inside the emulator.
3. copy the routing_analyser.exe from the `Storage Card/emulatorRel/` folder to `/` (the base folder of the emulator).
4. Do a tap'n'hold on the background and select *View All Files*.
5. Copy the fmodce.dll from `Storage Card/api/wce4/emuv4/` to the same folder where you copied the routing_analyser.exe.
6. Now you can start the program by clicking on it.

Remark that we have used the emulator only for screenshots.

C.2 Windows using DevC++

The Routing Analyser also compiles on Windows using DevC++ [1]. Open the DevC++ project file located at `/sa_ahcomm/software/iPAQ/routing_analyser/routing_analyser.dev` and hit CTRL-F11 to rebuild all files. The adjustments needed to make the program compile under DevC++ is done automatically in the project settings. You can change them if you click *Project -> Project Options* and then switch to register *Parameters*.
Remark that the program when compiled under Windows does not provide a GUI. However there is a simple command line interface built-in that activates automatically if there is no GUI. See Sect. 3.1.7 for a list of supported commands.

C.3 Linux / Mac OS X

To compile the program under Linux or Mac OSX we have supplied a makefile. No changes to the file `config.h` are needed. Make sure you have installed the GNU C++ compiler (g++) and type *make* in the project directory.
Remark that the program when compiled under Linux or Mac OSX does not provide a GUI. However there is a simple command line interface built-in that activates automatically if there is no GUI. See Sect. 3.1.7 for a list of supported commands.

[1] You can get DevC++ at `http://www.bloodshed.net/dev/devcpp.html`

Appendix D

API Documentation

D.1 Message Passing

Message passing enables two objects to exchange messages of various kind. Sometimes one object needs to inform the other object that some state has changed or it is necessary to send arbitrary data from one object to the other.
Message passing can be done in two ways:

active Object1 sends a message to Object2. Object1 can decide if it wants to send a certain message to another object only or to a range of objects. This scenario provides full flexibility on what to send where. The drawback is that Object1 has to ”know” at compile time that there exists an object called Object2 and that it can send messages to it. Section D.1.1 describes an implementation for active message passing.

passive Object2 subscribes to Object1 and gets messages sent if updates are pending. This is similar to polling for messages. The main difference compared to active message passing is that Object1 does not need to ”know” at compile time that there is an Object2 or maybe even Object3 and Object4 since they all subscribe at Object1 in a generic way at runtime. The drawback compared to active message passing is that Object1 will always send its messages to all subscribers. It is not possible for Object1 to decide when it is sufficient to send a message to a specific subscriber only and when to inform all of them. For an implementation of passive message passing see Sec. D.1.2.

D.1.1 A C++ Message Queue

This is an implementation of a generic message queue that enables a C++ class to receive and process messages sent by other classes of functions. It is thread safe and therefore suitable for inter thread communication. The message system consists of two classes:

- Class <MessageQ>, the queue class itself and
- Class <Message>, a basic message object that can be received by the MessageQ.

To make use of the message queue just extend a class of your choice with MessageQ. A message can then be added to the queue using `PushMessage()`. Since a Message object itself does not provide any useful members to store your data you will have to create your own message class and extend Message from it. See the example on how to use it.

```
#include "message.h"

// Enum for sender_id (see below for details)
enum {
    ID_THREAD1,
    ID_THREAD2,

    ID_ETC
}
```

```
// Enum for your messages (see constructor implementation for details)
enum {
    MSG_MYMESSAGE1,
    MSG_MYMESSAGE2
}


// First of all we define our own message class and derive it from Message
//////////////////////////////////////////////////////////////////////////
class MyMessage : public Message {
    public:

    // sender_id is a simple integer that tells the receiver of the message
    // where it comes from (you have to give the value in the constructor since
    // you cannot change it afterwards)
    //
    // we will use the enum from the beginning for simplicity
    MyMessage(int sender_id);
    ~MyMessage();

    // Our message content
    std::string content;

};

// The implementation is straight forward
//////////////////////////////////////////
MyMessage::MyMessage(int sender_id) : Message(sender_id, MSG_MYMESSAGE1) {
    content = "";
}

MyMessage::~MyMessage() {}

// Ok, now lets create a class that can receive our messages
// Remark that we derive it from MessageQ
//////////////////////////////////////////////////////////////
class MyReceiver : public MessageQ {
public:
    MyReceiver();
    ~MyReceiver();

    void ShowMessages();

};

// The implementation is straight forward
//////////////////////////////////////////
MyReceiver::MyReceiver() {}
MyReceiver::~MyReceiver() {}

void MyReceiver::ShowMessages() {
    Message* msg;

    // Poll for messages
    msg = PollMessage();

    if(msg != NULL) {
        cout << "got message" << endl;

        // now see what type of message we have received
        if(msg->Type() ==  MSG_MYMESSAGE1) {
            cout << "message content: " << (MyMessage*)msg->content << endl;
        }

        // now we are done using the message. We have to discard it manually,
        // otherwise the garbage collector cannot destroy the message
        // NEVER FORGET THIS STEP!
        msg->Discard();
    }
}

// finally we can start using our classes
```

```
/////////////////////////////////////
int main(int argc, char* argv[]) {

    // Create an object of MyReceiver
    MyReceiver m;

    // Create a message. Remark that this MUST be
    // a freely allocated object using the new operator
    MyMessage* msg = new MyMessage(ID_THREAD1);
    msg->content = "Hello MessageQ!";

    // Push the message into the queue:
    m.PushMessage(msg);

    m.ShowMessages();

    // Since we don't need the message anymore, we have to discard it manually
    msg->Discard();

    return 0;
}
```

In the example above you can see that we call `msg->Discard()` two times although it looks like we only have one object. Since we only send a pointer from one thread/class to the other we have to make sure that the message is not destroyed before the receiver has read the message. Consider the following setup. Class A creates a message and sends it to Class B. Class B reads the message and decides to forward it to Class C and Class D. Class A cannot know this. It is clear that Class A cannot delete the message since it never knows when Class B has processed the message. However since Class B also forwards the message it cannot delete it either since Class B cannot know when Class C and Class D have processed the message. Neither Class C nor Class D can safely delete the message since they both don't know when the other has processed the message. For an illustration of this setup see Fig.

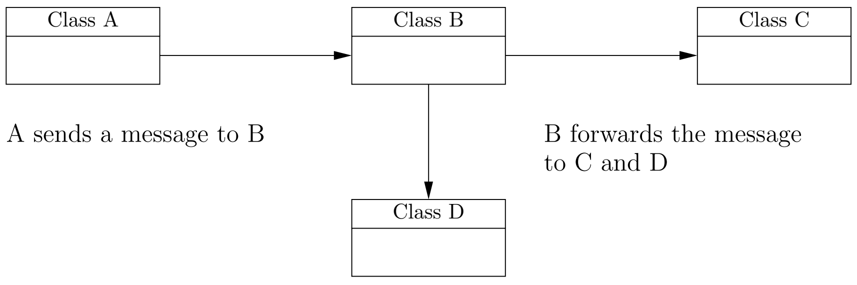


Figure D.1: Message flow

To solve this problem a message implements a simple reference counter and the message queue implements a simple garbage collector. If a message is created its reference count is set to one. `PushMessage()` increments the reference counter by one, whereas `Discard()` decreases it by one and kicks off the garbage collector. The garbage collector checks whether the reference count has reached zero and if true deletes the message. There are debugging facilities that tell you if you are accessing a message whose reference count is zero or smaller to help you trace down bugs.

Message

The Routing Analyser uses the following enums for its message passing:

```
// Enum for Threads (sender_id)
enum {
    ID_RECV,
    ID_SEND,
    ID_DISC,
    ID_CHECK,
    ID_MAIN,
    ID_SOUND,
    ID_CHAT
};


// Enum for Message types (only MSG_PACKET and MSG_PEER are used)
enum {
    MSG_NONE = 0,
    MSG_PACKET,
    MSG_PEER,
    MSG_LAST
};
```

Functions

Functions marked **internal** are intended for internal use only. There is no need to call them if you just want to use the framework.

Message::Message

Constructor for messages

```
Message(
    int sender_id,
    UInt8 type);
```

Parameter Description

`sender_id`
A number that represents the creator of the message.

`type`
A number that identifies the message. Useful if you have several messages that extend Message.

Message::NewReference

internal. Increases the internal reference count (automatically called by MessageQ::PushMessage).

```
void NewReference();
```

Message::RefCount

internal Returns current reference count. If zero message is scheduled for deletion.

```
void RefCount();
```

function result returns current reference count.

Message::Discard

Tells the message that you don't need it anymore by decreasing the internal reference count. You are not allowed to use the message anymore after this call. It could be already destroyed by the garbage collector.

```
void Discard();
```

Message::MsgType

Returns the message type. This function is useful if you have different messages in your program and want to find out what type of message you have received. The message type is usually an enum you provide in some header file. The value is usually set in the constructor.

```
UInt8 MsgType();
```

function result return message type.

Message::SetType

Sets the message type. This function is useful if you have different messages in your program and want to set the type of the message before you send it to another object. Remark that this is usually not needed since the message type should be set in the constructor of your own message class!

```
void SetType(
    UInt8 type,);
```

Parameter Description
`type`
    The message type.

Message::MsgSenderID

Returns the senders id. This is usually an enum that provided an integer number for every thread in your program. Remark that the sender id does not change if the message is forwarded. In the example from Fig. D.1 Class C and Class D both read a sender id of Class A (and not of class B!).

```
int MsgSenderID();
```

function result Returns id of creator of the message.

MessageQ

Functions

Functions marked **internal** are intended for internal use only. There is no need to call them if you just want to use the framework.

MessageQ::PushMessage

Pushes a message pointer into the message queue.

```
int PushMessage(
    Message* message);
```

Parameter Description
`message`
    The message to be pushed.

function result returns 0 on success or -1 if the message couldn't be pushed.

MessageQ::PollMessage

Polls for currently pending messages. Returns instantly if there are no messages pending.

```
Message* PollMessage();
```

function result returns a pointer to the first pending message or NULL if there are no messages pending.

MessageQ::WaitMessage

Waits indefinitely for the next available message.

```
Message* WaitMessage();
```

function result returns pointer to pending message.

MessageQ::MessageQSize

Returns number of messages pending.

```
unsigned int MessageQSize();
```

function result returns number of messages waiting in the queue.

MessageQ::FlushMessages

Deletes all waiting messages.

```
void FlushMessages();
```

MessageQ::GCRegister

internal Registers message at garbage collector. This function is automatically called by `Message::Discard()` if message reference count reaches zero. Also checks for duplicate insertion. Remark that a message will only be deleted if its reference count reaches zero.

```
static void GCRegister(
    Message* message);
```

Parameter Description

`message`
The message to be registered.

MessageQ::GCRun

internal Runs the garbage collector that removes unused messages (with reference count = zero). This function is automatically called by `Message::Discard()`.

```
static void GCRun();
```

MessageQ::GCStatus

Creates a LogMessage how many objects the garbage collector is currently tracking

```
static void GCStatus();
```

MessageQ::GCSize

Returns the number of objects currently tracked by the garbage collector.

```
static unsigned int GCSize();
```

function result returns number objects currently tracked by the garbage collector.

D.1.2 SubscriptionServer

The subscription server is a class that implements a subscriber service. An object implementing this service is able to ”inform” other objects (called subscribers) of its activities using messages. The subscribing object must implement a MessageQ to receive the notification messages. If a subscriber wants to become up to date it may call `DeliverAll()` to receive all notifications needed to get a complete picture of what happened. Remark that for a complete picture one does not need to have all notifications. I.e. an ADD and a later REMOVE notification would cancel each other!

Functions

SubscriptionServer::Register

Registers the message queue of a subscriber. The subscriber queue is not added if it is already subscribed.

```
void Register(
    MessageQ* subscriber_q);
```

Parameter Description
`subscriber_q`
The MessageQ of the subscriber.

SubscriptionServer::Deregister

Deregisters the message queue of a subscriber. The subscriber queue is not removed if it is not subscribed.

```
void Deregister(
    MessageQ* subscriber_q);
```

Parameter Description
`subscriber_q`
The MessageQ of the subscriber.

SubscriptionServer::Deliver

Delivers a message to all subscribers.

```
void Deliver(
    Message* message);
```

Parameter Description

`message`
    The message to deliver.

SubscriptionServer::DeliverAll

Virtual function that has to be implemented. If a subscriber has "lost" all notification messages or just wants to be up to date, this function sends as many messages as are needed to become up to date. A bad but working implementation would save all messages ever received. A call to this function would then send all those messages in the correct order to the subscriber.

```
virtual void DeliverAll(
    MessageQ* subscriber_q);
```

Parameter Description

`subscriber_q`
    The MessageQ of the subscriber the messages should be sent to.

D.2 PI Library - platform independent helper functions

The PI library is a set of functions that deal with platform dependent problems. The main idea is to provide a defined interface to unify access to problematic functions across platforms. It is very similar to SDL[1]. In fact some function are heavily inspired by SDL. The reason why we "reinvented" the wheel again is that SDL did not cover all our needs directly. If you need platform independent network support you also need SDL_net in addition to SDL. That's a huge overhead if you only need network support! If you want to use SDL or any other cross platform library you can easily tune the PI functions to use the library of your choice. Remark that the Routing Analyser separates all platform specific access using the PI functions. This should make it very easy to port it to another platform.
The current release supports the following platforms:

- Microsoft Windows
- Microsoft Windows CE
- Linux
- Mac OSX

There are minor compilation problems left on Solaris which should be easy to fix.
The library itself provides interfaces for the following operating system facilities:

- Threads and mutexes
- Time, date and delays
- BSD like sockets (partial)
- Filesystem (partial)

None of the implementations is actually a complete implementation. However if you need threads or access to the system time the implementation should be sufficient in 95% of all cases. Apart from that if you need sockets of filesystem access you will much probably have to adjust or add other functions since we have only implemented what we have really needed. See Sect. D.2.1 to Sect. D.2.4 for more details.

[1]see http://www.libsdl.org/

D.2.1 PI Threads

To make use of these functions you have to include the following header file
`#include "pi_threads.h"`

PI_Thread::PI_Thread

This is the constructor of the PI_Thread class. Just provide a function callback and an optional void pointer. The thread is launched immediately. The function callback should have the following definition:

`int my_thread(void* data);`

The data pointer provided to PI_Thread is handed over to your callback.

```
PI_Thread(
    int (*fn)(void *),
    void* data);
```

Parameter Description

`fn`
: The function you want to convert to a thread.

`data`
: A pointer that is handed over to the thread on creation.

PI_Thread::ThreadID

Get the 32-bit thread identifier for the current thread. Use this function if you are ”outside” of a given thread:

`value = my_thread->ThreadID();`

```
UInt32 ThreadID();
```

function result returns the thread identifier.

PI_Thread::GetThreadID

Get the 32-bit thread identifier for the current thread. Use this function if you are ”inside” the thread:

`value = GetThreadID();`

```
static UInt32 GetThreadID();
```

function result returns the thread identifier.

PI_Thread::Wait

Wait for a thread to finish (timeouts are not supported).

```
void Wait(
    int* status = NULL);
```

Parameter Description

`status`
: The return code for the thread function is placed in the area pointed to by status, if status is not NULL.

D.2.2 PI Time

To make use of these functions you have to include the following header file
`#include "pi_time.h"`

Typedefs

```
typedef struct {
  unsigned int year;    // current year
  unsigned int month;   // current month (1 - 12)
  unsigned int day;     // current day (1 - 31)
  unsigned int hour;    // current hour
  unsigned int minute;  // current minute
  unsigned int second;  // current second
} PI_Time;
```

Functions

PI_GetTime

Returns the current system time. The system time is expressed in Coordinated Universal Time (UTC).

```
void PI_GetTime(
    PI_Time& time);
```

Parameter Description
`time`
The returned time.

PI_GetMilliseconds

Gets the number of milliseconds since this function was first called.

```
unsigned int PI_GetMilliseconds();
```

function result returns milliseconds.

PI_Timestamp

Returns (number of seconds since 1970-01-01 00:00:00 * 1000) + milliseconds of current second. Remark that this function will most probably wrap the real number of milliseconds due to limited range of an unsigned integer!

```
unsigned int PI_Timestamp();
```

function result returns number of milliseconds since 1970.

PI_Sleep

Wait a specified number of milliseconds before returning.

```
void PI_Sleep(
    unsigned int ms);
```

Parameter Description
`ms`
    Number of milliseconds to sleep.

function result return value description.

D.2.3 PI Sockets

To make use of these functions you have to include the following header file
`#include "pi_socket.h"`

Typedefs

```
typedef std::vector<UInt8> UDPByteStream;
```

Functions

ip2string

Converts the integer representation of an ip (in Network Byte Order) to a C-string in dotted quad format (e.g. 192.168.0.2). Remark that the pointer returned points to a static array inside ip2string. Subsequent calls change the array!

```
char* ip2string(
    UInt32 ip);
```

Parameter Description
`ip`
    IP address to convert.

function result returns C-string representation of the IP.

string2ip

Converts the C-string representation of an IP (from dotted quad format) to an integer in Network Byte Order.

```
UInt32 string2ip(
    char* ip_string);
```

Parameter Description
`ip_string`
    C-string with the IP address.

function result returns IP in integer representation.

PI_MyIP

Get the current IP for your host. If you have more than one IP assigned returns the first IP found.

```
UInt32 PI_MyIP();
```

function result returns your IP in integer representation.

PI_UDPSocket::open

Opens a UDP socket

```
int open(
    const int port);
```

Parameter Description

`port`
: The port to open.

function result returns 1 if successful, 0 else.

PI_UDPSocket::close

Closes the socket.

```
int close();
```

function result returns 1 if successful, 0 else.

PI_UDPSocket::bind

Binds to the socket. Mandatory if you want to read from it.

```
int bind();
```

function result returns 1 if successful, 0 else.

PI_UDPSocket::read

Reads from socket and stores data in UDPByteStream, from_ip is set to the senders IP address in Network Byte Order.

```
int read(
    UDPByteStream& stream,
    UInt32* from_ip);
```

Parameter Description

`stream`
: UDPByteStream to store data in.

`from_ip`
: Pointer where to store the senders ip.

function result returns number of bytes read or 0 on error.

PI_UDPSocket::write

Writes to socket. Data is read from UDPByteStream. The data is sent currently sent to the <BROADCAST> address.

```
int write(
    const UDPByteStream& stream);
```

Parameter Description

`stream`
    Data to send.

function result returns number of bytes written or 0 on error.

PI_UDPSocket::is_open

Get current socket state.

```
int is_open();
```

function result returns 1 if socket is open, 0 else.

D.2.4 PI Filesystem

PI_Mkdir

Creates a directory in the filesystem. Remark that you can only create one directory at a time and not a whole hierarchy.

```
int PI_Mkdir(
    const char* directory);
```

Parameter Description

`directory`
    The directory to create.

function result returns 0 if successful, 1 on error.

Appendix E

Tips on using the iPAQs and ActiveSync

During our work with the iPAQs and ActiveSync we have developed a best practice guide and have found some tricks that simplified our daily work.

E.1 iPAQ Initialization

1. Hard-Reset the Devices by pressing and holding Button 1 (Calendar) and 4 (iTask) simultaneously and then pressing the Reset button (using the Pen) for more than 2 seconds. Press the Reset-Button again to powerup the device
2. After the iPAQ has booted for the first time, complete the introduction and wait until the device has rebooted.
3. Use the iPAQ Backup application to restore Base-xx.pbf (the file can be found in `/sa_ahcomm/software/iPAQ/`).
4. Under *Settings->System->About->Device ID* set the Device Name to ahcomm-XX where as XX represents the last two digits of the TIK-Inv. number. If you forget this step ActiveSync will not be able to distinguish the devices.
5. Activate WLAN and configure the HP iPAQ Wi-Fi Adapter with a valid IP Address. Restart WLAN to make the changes take effect.

E.2 ActiveSync with Computer

The first time you connect a device to your computer you are asked whether you want to use *Guest Mode* or create a *Standard Partnership*. If you only want to develop your software and test it on the device *Guest Mode* is sufficient.
If you need the devices to use valid time and date you will have to synchronize them with your computer. But first of all you have to click yourself through an annoying Assistant. The main problem is that the batteries often lose energy so fast that the devices hard reset and you have to associate them again using the wizard.

To setup a standard Partnership do the following:

1. Place the device into the cradle and wait until the *New Partnership* dialog opens
2. Choose *Standard Partnership*
3. Choose *Synchronize with this computer*
4. Choose *No, I want to synchronize with two computers* (You can only sync with a max. of 2 computers)

5. Deselect all conduits
6. Click *Finish*

Remark that you may have to sync twice so that the time gets updated correctly! If you want to connect a device to a third computer, use *Guest Mode* on the third computer.

In Guest Mode you can just access the device normally. Remember that there is no time synchronization done in this mode.

E.3 Creating a Backup of a device

The Backup was created directly on the iPAQ using the iPAQ Backup Tool.

1. Start iPAQ Backup (you can find it under Start->Programs)
2. Switch to advanced mode (Options->Switch to advanced mode)
3. Unselect all
4. Select System Data Folder
5. Click the Save as button (...)
6. Name: base-1_10_09 (the version number represents the installed WindowsCE version!)
7. Location: CF Card
8. Folder: None
9. Click Ok
10. Click Backup
11. Click Start
12. Click Ok to restart the device

E.4 Settings used for Base Backup

(for instructions on how to use the backup refer to Chap. E)

Settings:

- Settings->Sound off
- Settings->About->Device ID->Device Name = ahcomm-NEW
- Owner->Name = ahcomm-NEW
- Settings->Menus: everything deactivated except File Explorer
- Settings->Menus: New Menu: everything deactivated except Note
- Settings->Today: Theme Windows Default
- Settings->Today: Items: Date, Owner Info, TodayPanel? activated - others deactivated
- Settings->Backlight: Brightness On Battery set to minimum value
- Settings->Clock & Alarms: Timezone: Berlin, Rome (GMT+1)
- Settings->Power: Advanced: Deactivate auto poweroff
- Settings->Power: USB Charging deactivated to save battery charge cycles

- Settings->Beam: Deactivate Receive all incoming infrared beams
- Settings->Regional Settings: Time: Time Style: HH:mm:ss (24h)
- Settings->Regional Settings: Date: Short date: dd-MMM-yy
- Settings->Buttons: Assign all buttons to <None>
- Settings->Buttons: Assign Button 4 to <OK/Close>
- Settings->iPAQ Wireless: (discard assistant, power on Wi-Fi goto Settings) Network Adapters: HP iPAQ Wi-Fi Adapter: IP set to 192.168.0.0/24
- Settings->iPAQ Wireless: Settings: Networks to access: Only computer-to-computer
- Settings->iPAQ Wireless: Restart Wi-Fi to make changes take effect
- Restart the device (as usual with Windows...)
- After copying your application to the device, create a shortcut of it in /Windows/Start Menu/Programs
- You may now Assin in Settings->Buttons: Button 1 to <your application>

Appendix F

Schedule

F.1 Milestones

Nr	Due	Description
1	Week 47	Field test completed using the old software
2	Week 47	Routing protocol chosen for implementation
3	Week 51	Routing protocol implemented Software ready for deployment
4	Week 5	Measurements completed
5	Week 6	Analysis completed
6	Week 7	Documentation completed

F.2 Week Tasks

Week 43, 2005	Start of semester	Orientation
Week 44, 2005		Orientation
Week 45, 2005		Work of J. Wagner read Schedule created All administrative work done
Week 46, 2005		Reconstruct J. Wagners experiments in a field test Compare measurements with old results
Week 47, 2005		Creation of an overview of routing protocols One protocol chosen for implementation
Week 48, 2005		Design of implementation, API, debug facilities
Week 49, 2005		Implementation
Week 50, 2005		Implementation
Week 51, 2005	Presentation	Implementation
Week 52, 2005		*** Christmas holidays ***
Week 1, 2006		*** Christmas holidays ***
Week 2, 2006		*** Christmas holidays ***
Week 3, 2006		Deploy iPAQs to users and start with measurements
Week 4, 2006		Measurements / analysis
Week 5, 2006		Measurements / analysis
Week 6, 2006		(Measurements) / analysis
Week 7, 2006	End of semester Final presentation	Analysis / completing documentation
Week 8, 2006		Completing documentation
Week 9, 2006		Completing documentation
Week 10, 2006	Start of exams	

Appendix G

Acronyms

AP Access Point

API Application Programming Interface

ASCII American Standard Code of Information Interchange

AODV Ad hoc On Demand Distance Vector

BER Bit Error Rate

bps bits per second

BIOS Basic Input/Output System

BSSID Basic Service Set IDentifier

CDF Cumulative Density Function

CDROM Compact Disk - Read Only Memory

CLI Command Line Interface

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

CSMA/CD Carrier Sense Multiple Access with Collision Detection

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

CSMA/CA RTS/CTS Carrier Sense Multiple Access / Collision Avoidance Ready to Send / Clear to Send

CSV Comma Separated Value

DSR Dynamic Source Routing

FEC Forward Error Correction

FIFO First In First Out

FSM Finite State Machine

GNU GNU's Not Unix http://www.gnu.org

GPL GNU General Public License

GUI Graphical User Interface

IBSSID Independent Basic Service Set IDentifier

I/O Input/Output

IDE Integrated Development Environment

ID IDentification

IEEE Institute of Electrical and Electronic Engineers

IP Internet Protocol

IRQ Interrupt ReQuest

KB Kilo Byte(s)

MAC Media Access Control

MANET Mobile Ad Hoc Network

MB MegaByte

MTU Maximum Transfer Unit

MVC Model-View-Controller

NTP Network Time Protocol

OS Operating System

PC Personal Computer

PDF Probability Density Function

PMF Probability Mass Function

POSIX Portable Operating System for unIX

SDL Simple Directmedia Layer

SDK Software Development Kit

SIG Special Interest Group

SNR Signal-to-noise ratio

SSID Service Set IDentifier

STL Standard Template Library

TCP Transmission Control Protocol

TTL Time To Tive

UDP User Datagram Protocol

USB Universal Serial Bus

UTC Coordinated Universal Time

VM Virtual Machine

WLAN Wireless Local Area Network

Bibliography

[1] Lukas Ruf, *Latex Essentials – HowTo Create Your LaTeX-based Documentation*, TIK, ETH Zuerich, 2002.

[2] *LEO - Link Everything Online*, http://dict.leo.org/.

[3] *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/.

[4] Tobias Oetiker, *The Not So Short Introduction to $\LaTeX 2_\varepsilon$*, Version 4.14, 04 April, 2004 http://people.ee.ethz.ch/~oetiker/lshort/.

[5] Prata S., *C++ Primer Plus*, Sams, Third Edition, 1998

[6] Stroustrup B., *The C++ Programmng Language*, Addison Wesley, Third Edition, 2002

[7] Collins-Sussman B., Fritzpatrick B., Pilato M. *Version Control with Subersion*, for Subversion 1.1, Revision 1337

[8] Jörg Wagner, *Analysis of Dynamics in Mobile Ad Hoc Networks. Technical Report SA-2005-18,* TIK, ETH Zurich, September 2005

[9] M. Scott Corson, Vincent Park, Matthew Impett, *TORA*, Temporally-Ordered Routing Algorithm http://www.isr.umd.edu/ISR/accomplishments/037_Routing/

[10] *AODV*, Ad Hoc On Demand Distance Vector Routing Algorithm http://moment.cs.ucsb.edu/AODV/aodv.html

[11] *DSR*, Dynamic Source Routing Algorithm http://www.antd.nist.gov/wctg/DSRreadme.pdf