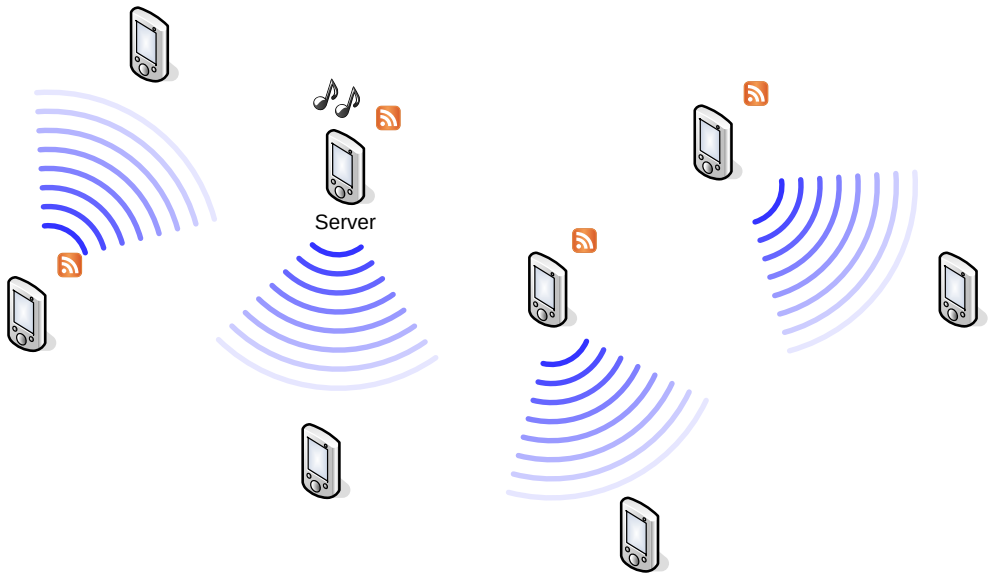
Institut für
Technische Informatik und
Kommunikationsnetze

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

Clemens Wacha

Wireless Ad Hoc Podcasting with Handhelds



Master Thesis MA-2007-05
October 2006 to April 2007

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

Abstract

Podcasting has become popular for dissemination of streaming content over the Internet. It is based on subscriptions where software clients query servers for updates of subscribed content feeds. The main limitation with the current system is the inflexible separation of downloading to a docked media player and expending of the data when on the move. The solution proposed here uses peer-to-peer synchronization to exchange content directly between neighboring devices. We have written a podcasting application in C++ that uses WLAN for peer-to-peer content synchronization. The program sends UDP broadcast messages to discover devices. Among those in range, one peer is randomly selected and a TCP connection is created for data exchange. To speed up the synchronization process we employ a bloom filter. Although the program was primarily designed for HP iPAQ handhelds, it is completely platform-independent and runs successfully on Windows 2000/XP, Mac OS X and Linux.

We have conducted measurements to determine the actual performance of the proposed wireless podcasting solution. The test network consists of 19 HP iPAQs running Windows Mobile 2003 Second Edition as well as 5 ThinkPads T60 running Windows XP. All devices communicate through their integrated WLAN interfaces. Specifically, we examined the distribution speed of content in the ad hoc podcasting mode, i.e. we measured the time necessary to distribute content from one device to all other devices in range. Furthermore, we have examined the effect of different distribution strategies on the performance and fairness.

Results point out that the most fair solution to all devices is also the most efficient one. Finally, we have shown that the use of a bloom filter significantly speeds up synchronization of a large number of channels. To summarize, we have designed, implemented and examined an innovative solution which allows to extend existing podcasting technology to mobile users.

Abstract

Podcasting wurde populär durch die Verbreitung von Mediendateien über das Internet. Basierend auf Abonnements können entsprechende Programme automatisch nach neuen Episoden eines abonnierten *Feeds* suchen. Ein Hauptnachteil des bestehenden Systems ist die fehlende Möglichkeit, unterwegs neue Inhalte zu empfangen oder zu senden.
Die hier vorgeschlagene Lösung verwendet peer-to-peer Synchronisation, um Inhalte direkt zwischen benachbarten Geräten auszutauschen. Wir haben ein Podcasting-Programm in C++ geschrieben, welches WLAN für die Synchronisation verwendet. Das Programm sendet UDP-Pakete an die Broadcast-Adresse, um andere Geräte zu finden. Unter allen benachbarten Geräten wird eines zufällig ausgewählt und eine TCP-Verbindung für den Datenaustausch aufgebaut. Um den Abgleich der Inhalte zu beschleunigen, haben wir einen Bloom-Filter verwendet. Obwohl unser Programm ursprünglich für HP iPAQ Handhelds entwickelt wurde, ist es dennoch komplett plattform-unabhängig und läuft erfolgreich auf Windows 2000/XP, Mac OS X und Linux.
Wir haben Messungen durchgeführt, um die tatsächliche Leistung der vorgeschlagenen Wireless Podcasting Lösung zu ermitteln. Das verwendete Test-Netzwerk bestand aus 19 HP iPAQs, welche unter Windows Mobile 2003 Second Edition laufen als auch aus 5 ThinkPads T60 mit Windows XP. Alle Geräte kommunizieren über die integrierten WLAN Schnittstellen. Wir haben im besonderen die Verteilungsgeschwindigkeit von Inhalten im ad hoc Podcasting-Modus untersucht. Dazu haben wir die Zeit gemessen, welche benötigt wird, um Daten von einem Gerät auf alle anderen Geräte in Reichweite zu verteilen. Des weiteren haben wir den Einfluss von verschiedenen Verteilungs-Strategien auf den Datendurchsatz und die Fairness untersucht.
Die Resultate zeigen, dass die fairste Lösung für alle Geräte auch die Effizienteste ist. Schliesslich konnten wir nachweisen, dass der Einsatz eines Bloom-Filters die Synchronisation einer grossen Anzahl an Feeds erheblich beschleunigt. Zusammenfassend haben wir eine Lösung entwickelt und untersucht, die es uns ermöglicht, die existierende Podcasting-Technologie auch in mobilen ad hoc Netzen einzusetzen.

Acknowledgements

During the work on this thesis a number of people supported me in their own ways and I would like to express my gratitude to them.

I would like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to write my master thesis at the Communication Systems Group and for the supervision of my work.
I would like to thank my two advisors Dr. Vincent Lenders and Dr. Martin May for their efforts they put into this work. Martin always had time for interesting discussions and provided me with new ideas. Vincent gave me great phone support despite spending the second half of this work at the Princeton University in New Jersey. He always challenged my decisions and guided me into the right direction. It was a pleasure to collaborate with them.

Thanks also go to all my friends who successfully distracted me with extensive coffee breaks and helped me escape from work to make room for other topics.

I would like to thank my parents for always providing me with everything I needed and particularly for the possibility to obtain this excellent education. Finally I want to express my special gratitude to them and my two sisters for their unconditional support.

Zurich, April 20th 2007

Clemens Wacha

Table of Contents

List of Figures

Chapter 1

Introduction

Wireless broadcasting is not openly available to anyone who would like to provide programs for a general audience. Luckily, the Internet enables alternative broadcasting modes, such as web logs and podcasts, which quickly have become popular among both content producers and end users. Podcasting is used for downloading of contents to a mobile media player when it is docked to a high-speed Internet connection. The playback is then often done when the player is off line because of the user's movements. This separation of downloading and playback limits the usefulness of the service since there could be hours passing between the downloading opportunities.

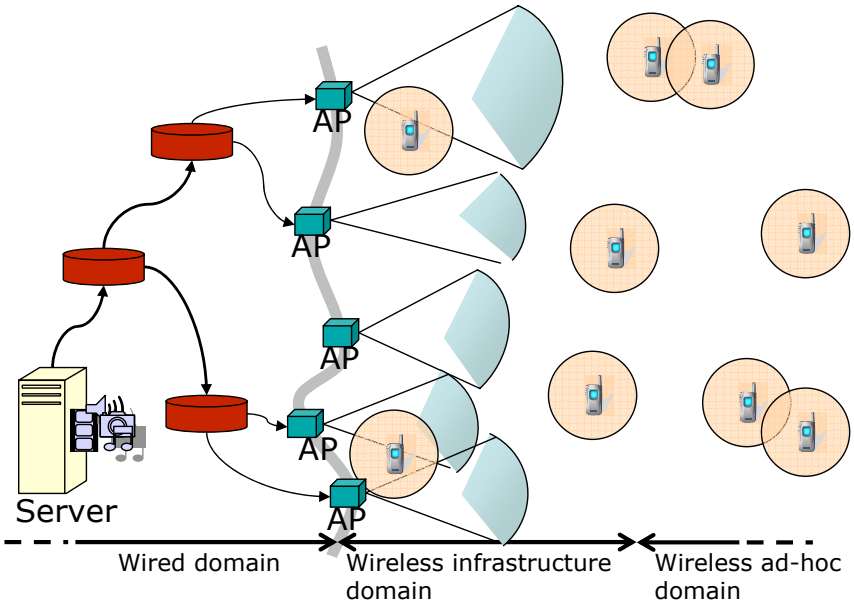


Figure 1.1: Contents are provided by means of podcast via access points and they are re-distributed in the ad hoc domain.

This work describes how the podcasting concept can be expanded for public (i.e., open and unrestricted) peer-to-peer delivery of contents amongst mobile nodes. Contents are provided in a wireless broadcasting area either by access points or by mobile nodes which have been filled with contents off line (Figure 1.1). In the first case, each access point fetches contents from podcast servers across the Internet and forwards them to mobile nodes within its range; this is done in the regular podcast mode. In the second case, a mobile node that has contents to share might provide data to another mobile node when they pass within radio range of one another. This is the new content distribution mode that we add to the existing one. Our ad hoc podcasting mode brings the following advantages. First, it provides nodes with contents when they are not connected to the Internet; second, it provides a new ad hoc broadcasting domain when also the sources of the data are mobile nodes (could be pictures or voice recordings from a mobile phone for instance). We will also refer to this wireless ad hoc mode of broadcasting as podcast in the

hope of broadening the current concept (motivated by the use of the feet of a pod of users for content distribution).
Wireless ad hoc podcasting, as we present in this work, is an application based on the *delay-tolerant broadcasting* concept we have proposed in an earlier paper [8]. The sharing of contents is based on a solicitation protocol by which a node asks a peer node for content. Hence, there is no flooding of contents in the broadcasting area. Contents are organized into *channels*, and nodes solicit *episodes* for one or more channels (the emphasized words denote the terms we use in this work).
The concept of channels allows for a higher hit rate of the queries than if they were for individual episodes of contents. The episodes of a particular channel will however reach a node in arbitrary order, and not all of them will be received; it is therefore important that all contents are provided in atomic units, which are short enough to be downloaded in a contact and which may be replayed without relation to other units. We believe that the podcasting application fulfills these requirements, since a channel could be composed of a mixture of entries of music, news items, weather updates and commentaries, such as the mix broadcast by many radio stations.
Our wireless ad hoc podcasting system is simple in its protocols, and lightweight in the computational requirements.
This work reports new research results regarding podcasting. The contribution is a proposal for a wireless ad hoc mode where mobile nodes exchange contents. We show how this mode may be implemented based on the Atom syndication protocol [1], and describe the protocols for soliciting and serving contents between nodes.

The work is structured as follows:

- Traditional podcasting, as existing in the Internet and other concepts used in this work are described in the next chapter.
- Chapter 3 introduces our new concept of wireless ad hoc podcasting and the necessary protocols.
- Chapter 4 describes our current implementation of this concept as a C++ application for HP iPAQ handhelds.
- Chapter 5 lists the hardware and software used for the measurements.
- Chapter 6 contains the evaluations and measurements done with the application.
- And Chapter 7 concludes the work.

Chapter 2

Related Work

2.1 Podcast

From Wikipedia [7]:

> A podcast is a media file that is distributed over the Internet using syndication feeds, for playback on portable media players and personal computers. Like *radio*, it can mean both the content and the method of syndication. The latter may also be termed podcasting. The host or author of a podcast is often called a podcaster. The term *podcast* is a portmanteau of the name of Apple's portable music player, the iPod, and broadcast; a pod refers to a container of some sort and the idea of broadcasting to a container or pod correctly describes the process of podcasting.
>
> Though podcasters' web sites may also offer direct download or streaming of their content, a podcast is distinguished from other digital audio formats by its ability to be downloaded automatically, using software capable of reading feed formats such as RSS or Atom.

From a technical point of view a podcast is very similar to a regular news feed in RSS or Atom format. Both news feed formats divide content into channels (i.e. podcasts or feeds) and episodes (i.e. news items, entries). An episode may contain an enclosure (i.e. attachment). When talking about podcasts the enclosure is usually an MP3-file or another sort of audio file.
A podcast with one episode therefore consists of two files. The MP3-file and the XML file containing the Atom or RSS meta information. This concept gets very interesting if we use a podcasting application that is able to check the meta file automatically if new episodes have arrived. iTunes is one of those applications and it is also able to automatically download new episodes.
As podcasts as well as news feeds have become very popular the idea arises to extend their usage to wearable devices such as HP iPAQ handhelds. Unfortunately these devices are not always connected to the internet which prevents them from periodically updating their content. This in turn leads to our goal of creating an application that is able to fetch new episodes not only from the internet but also from other handhelds that happen to be in WLAN range.

2.2 BitTorrent

From the introduction of the developer documentation [3]:

> BitTorrent is a protocol for distributing files. It identifies content by Uniform Resource Locator (URL) and is designed to integrate seamlessly with the web. Its advantage over plain Hyper Text Transfer Protocol (HTTP) is that when multiple downloads of the same file happen concurrently, the downloaders upload to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load.

A BitTorrent file distribution consists of these entities:

- An ordinary web server

- A static 'metainfo' file
- A BitTorrent tracker
- An 'original' downloader
- The end user web browsers
- The end user downloaders

In the big picture, BitTorrent's design consists of three different modes.

Search/Publish Connect to a tracker and download a list of other clients that are downloading the same file. Also tell the tracker that we are downloading the file as well.

Download Connect to other downloaders, exchange data and try to complete the file.

Seed When the download is complete seed the file to other peers. This mode basically works like the *download* mode where data get transferred in only one direction.

The term file here is not quite correct. BitTorrent also allows the exchange of several files across an arbitrary number of directories. When downloading several files they are treated as one big chunk (as if they were all concatenated).
A downloader first creates a connection to the tracker to get a list of other peers and then connects to some or all of them and starts downloading the file simultaneously. In the downloading stage, a client has several TCP connections open to various other clients throughout the Internet (also over multiple hops) and may download from all of them at once.
The general design of BitTorrent has some commonalities to the problem of wireless ad hoc podcasting. We will now describe the differences and discuss which parts of BitTorrent are also useful to us.

- A tracker makes no sense in a wireless ad hoc network as we can not guarantee its reachability at all times. Instead, every device has to explicitly ask all peers if they have the wanted file. The *Search/Publish* stage has to be designed completely different for our system.
- Podcasts consist of channels and episodes. Furthermore, two devices can only exchange data if they both share at least one common channel. The objects to be synchronized here are episodes and containing files. In BitTorrent the file to be downloaded must be known by URL. A device requesting missing episodes from a peer does not know their URLs or IDs in advance. It is not even clear if there are any missing episodes. Therefore we need a separate synchronization stage where we identify missing episodes and their files.
- After we know which files we can download from a peer device we may start with the actual download process. BitTorrent separates the files into chunks and pieces upon download. Because our requirements are identical we can use exactly the same design here as in BitTorrent.
- BitTorrent uses TCP connections which allow connections over several hops. This is not needed for wireless ad hoc networks as we only synchronize over one hop.

2.3 Bloom Filters

From Wikipedia [6]:

> The Bloom filter, conceived by Burton H. Bloom in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed [...]. The more elements that are added to the set, the larger the probability of false positives.

Bloom filters [5] work like dictionaries. A bloom filter defines two different operations:

1. ADD *word*

2. QUERY *word*

The *ADD* operation adds a word or text string to the dictionary whereas the *QUERY* operation tells us if a certain word can be found in the dictionary. As we could easily achieve this behavior by using a list of all added words, a bloom filter uses a bitfield and hash functions to speed up the lookup process and also effectively reduce memory usage. The drawback of this approach are false positives (i.e. the bloom filter tells us that a certain word consists in the dictionary although it was never inserted). Fortunately the probability of false positives can be calculated easily if we assume that the hash function distributes values with equal probability. If we know the number of bits of the bloom filter and the number of words added to the filter we can calculate the probability of false positives. Or more convenient, given a certain false positives probability and the number of elements we would like to add to the filter we can calculate the minimal size of the filter to achieve this probability.
When two devices both have a large number of channels it may take a long time to find channels that are common on both devices. We have used bloom filters to check for existing channels during a synchronisation. That way, we hope to speed up the synchronization process significantly. Of course, bloom filters can also be used to find missing episodes. The difference here is that when using a bloom filter we need to know what we are looking for. When we are searching for new episodes we don't know their IDs in advance. It can still be solved but the solution is a little bit more complicated. In a first attempt we are content with the implementation of the bloom filter in the channel negotiation.

Chapter 3

Design

3.1 Overview

Peer-to-peer synchronization is a task, that consists of several different problems. The devices have to find each other, then one of them has to establish a connection to the other. After that the two devices have to exchange meta information to find out if they actually have common data and need to synchronize. If they have, the actual synchronization can begin. To facilitate this complex process the problem is separated into two different and independent modes.

Discovery Mode deals with the problem of finding other devices and checks if we have to make a connection to the other device

Transfer Mode handles the whole data transfer. This mode is separated into several sub-modes for the different states of synchronization

3.2 Terminology

Before starting with the actual description of the design some explanations have to be made about the terms applied here. Where possible the terms have been taken from the RSS 2.0 [2] and Atom 1.0 [1] specifications. As some of the terms used by the specifications are very general, more specific terms have been selected to eliminate confusion with common terms and names. The following table provides an overview of the elements used here and the corresponding ones from the specifications.

This Work	RSS 2.0	Atom 1.0
Channel	channel	feed
ChannelID	-	id
Episode	item	entry
EpisodeID	guid (optional)	id
Enclosure	enclosure	link

As can be seen from the table above in RSS 2.0 there is no ID for channels and the ID for episodes (guid) is optional. This is indeed a major problem during synchronization as we must have a way to unambiguously identify the elements we want to synchronize (i.e. channels and episodes). Atom 1.0 provides mandatory IDs for both channels (in atom wording: feeds) and episodes (in atom wording: entries). In Atom, the content of the ID field must be an Internationalized Resource Identifiers (IRI) [4][1]. The ID must be permanent and universally unique.
Atom feeds found on the internet usually define a unique URL for the channel and for every episode and use that as ID. As a URL is unique in the namespace of the internet this works quite well.

[1] An IRI is like a Uniform Resource Identifier (URI) where the characters outsize A-Z are allowed as well. See RFC 3986 and RFC 3987 for more information.

Channel A *channel* can be a podcast or a news feed. It does not contain a lot of data and acts more like a container for *episodes*. It may contain zero or more *episodes*. Among its elements are a *link* with a URL of its origin and an *ID* - a permanent, universally unique identifier as defined in Atom.

Episode An *episode* is the actual data object of interest. *Episodes* are always part of a *channel*. An *episode* contains among other things a release date and a description field which carries arbitrary text (i.e. the news text in case of news feeds or a more specific description of the podcast). An *episode* also contains a unique *ID* analog to that of a *channel*. An *episode* may contain zero or more *enclosures*.

Enclosure An *enclosure* is an attachment that consists of one file. In a podcast the *enclosure* is the MP3-file that contains the music or speech.

Chunk For synchronization all *enclosures* of an *episode* are combined into one big data object. This data object is then separated into several *chunks* of identical size. The chunk size is a property of the *episode* and may vary from *episode* to *episode*. As with in BitTorrent the chunk size is usually 256 KB. Remark that the last *chunk* may be smaller than the chunk size due to data alignment.

Piece *Chunks* are usually too big to fit into one communication packet. They are separated again into *pieces* which are sent (one *piece* per packet) over the communication channel (i.e. WLAN). There is no limitation for the piece size. *Pieces* may have arbitrary sizes even within one chunk. To keep track of the correct order of *pieces* they are always accompanied by an *offset*. See below for more information on *offsets*.

Index The *index* is a number which refers to a specific *chunk*. The first chunk begins at index zero.

Offset The *offset* is a number which refers to a position inside a *chunk*. The first offset is zero.

Bitfield The idea of the *bitfield* was taken from BitTorrent. It works identical to it and is a property of an *episode*. It contains one bit for every *chunk*. If we have a *chunk* the bit at its *index* is set to 1, otherwise 0. An *episode* is complete if all bits in the *bitfield* are set to 1.

Bloom Filter A *bloom filter* is used to speed up synchronization. When two devices try to synchronize they have to send a query for every *channel* they have. This can take a long time if one of the devices has a huge number of *channels*. To circumvent this problem a *bloom filter* is used. The *bloom filter* on every device is initialized with the *channel ids* of every *channel* on the device. During the first steps of synchronization the *bloom filter* is sent to the remote device which then tests all its *channel ids* against the *bloom filter*. This is a local process and does not involve the exchange of messages over the communication channel. False negatives are not possible which means that a *channel* not found in the *bloom filter* is sure not to exist on the other device. False positives however are possible. A *channel* found in the *bloom filter* might still not exist on the other device. Although we still might query *channels* that do not exist on the other device the number of queries to make is reduced drastically and thus makes the synchronization faster. We would like to have a constant false positive probability regardless of the number of *channel ids* in the filter. Because we always know how many *channel ids* we have to maintain in our bloom filter we can automatically adjust its size if the number of channels changes. Every time the number changes we re-calculate the optimal size and re-create the bloom filter. As the number of channels does not change very often (especially not during a synchronization) we don't have to worry that re-creating the bloom filter is not very efficient. More details on the function of a bloom filter can be found in Sect. 2.3.

3.3 Discovery Mode

The discovery mode deals with the problem of finding other peers. There are several steps necessary to make a connection between two devices possible. First of all the two devices must share a common communication channel. We chose Wireless Local Area Network (WLAN) as transport

media. It was also tested with wired networks and the main concept also works with Bluetooth or any other communication channel that is able to connect two devices.
The problem of handshaking is already well known and has been addressed several times before [14] [15] [16]. We are not going to provide another solution but stick with a setup that works for us. Because it was not possible to change and configure important WLAN properties such as BSSID we have made the following assumptions.
Every device that takes part in the peer-to-peer network has to fulfill these prerequisites:

- WLAN is switched on
- The device is configured for ad hoc mode and the SSID is set to *da_podcast*
- The devices IP address is in a private range (i.e. 192.168.0/24) and does not conflict with the IP address of another device.

The actual *search problem* is now reduced to detecting special IP packets.
Every device sends *Service Discovery* messages over UDP to the broadcast address of the network (i.e. 192.168.0.255) and that way tells other peers about its state and peer_id. The messages are sent periodically.
Service Discovery packets contain amongst other things

- device name (Arbitrary name, freely chosen by the user - may be duplicate)
- peer id (Unique identifier that has a length of 20 random bytes as in BitTorrent. This identifier is used to distinguish devices. The IP address of the devices could have been used as well. The problem here is that the IP address may change (i.e. DHCP) or the communication channel does not have an IP address (i.e. Bluetooth))
- current sync status (*ready* or *choked*. If the device is *ready*, it accepts incoming connections by other devices. If it is *choked* it rejects incoming connections with an error message.)
- the server port of any running transfer server
- a *new content* date (Helps peers to determine if a re-synchronization is necessary. After a synchronization connection gets terminated the device checks if it has received new content. If it has it updates its internal *new content* date. As this date is advertised with every discovery packet other devices automatically know that they might have to start a re-synchronization with this device in order to download the new data. There is one drawback. The other device knows that new data has arrived at this peer but it does not know if the new content is also useful for itself. This mechanism can therefore still lead to a useless re-synchronization where no data can be exchanged.)

All devices are able to send and receive Service Discovery packets.
The uniqueness of the peer_id is ensured exactly as in BitTorrent. It is very unlikely that 20 randomly chosen bytes are identical to those of another peer. This is the only security measure taken.

3.3.1 Device Status and SyncList

Every device has an internal status. It can be either *ready* or *choked*. The concept was taken from BitTorrent and behaves identical to it. The device starts in *choked* mode. After some internal checks the status switches to *ready* and the device accepts incoming connections.
This concept helps to minimize the number of packets sent over the communication channel. If we see that a remote peer is not able to accept our synchronization attempt we don't have to try and get rejected.
The current implementation only accepts one concurrent connection to another device.
To keep track of all neighbor peers every devices maintains two identical lists. The *SyncList* and the *SyncHistory*.

The SyncList contains all peers in range and their current status. If a peer moves away and gets out of range it is automatically removed from the SyncList. That way the SyncList acts like a to do list. The device should try to synchronize with all peers that are in the SyncList. After successful synchronization with a peer its entry is removed from the SyncList and added to the SyncHistory.

The SyncHistory contains all peers that we were able to synchronize successfully. The entry of a peer is not removed if the peer gets out of range. Instead the entry is removed after a configurable time delay or when we detect that the peer's content has changed (via new-content-date field of Service Discovery packets),

If a device discovers a peer which can be found in the SyncHistory the device does not try to synchronize with this peer. Remark that this does not mean that there won't be any synchronization between those two devices. A device does always accept incoming connections if its own status allows it to.

Although a peer device can only announce a status of *ready* or *choked* there are more values which are possible for remote peers. First of all we differentiate between stable and unstable peers. When using WLAN we have to deal with a connectivity problem. If a peer is far away from the local device we have very bad connectivity. The good thing is that most of the packets sent by the remote peer will not reach the local device. The bad thing is that very few do. It is not very smart to start a synchronization with a peer with bad connectivity. We need to find a way to distinguish those peers from peers with good connectivity. Or in other words, divide all peers in range into unstable peers and stable peers.

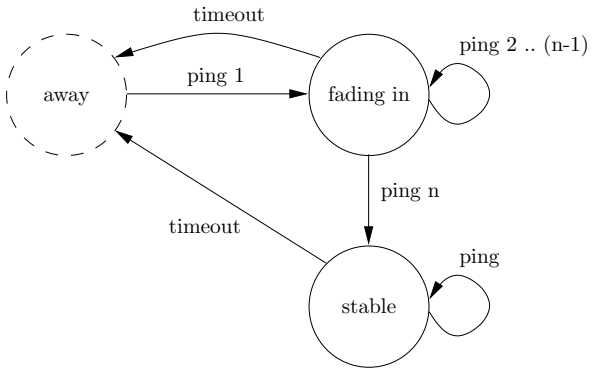


Figure 3.1: State diagram of neighbor peers

The mechanism used here is quite simple. We count the number of Service Discovery packets. The first packet received marks the peer as unstable. If we have received a certain number of Service Discovery packets (currently 3) the peer is marked as stable. If we don't receive Service Discovery packets for more than six seconds the peer times out and its entry is deleted (i.e. moves to state *away*). A peer which is not existent starts in the virtual state *away* as visible in Figure 3.1. After receiving one Service Discovery packet (here: ping) the peer starts to "fade in". There is no way for a stable peer to move back to the *fading in* state nor is there a *fading out* state. This is a three step concept. There is a hurdle to get into the stable state (three consecutive Service Discovery packets) and there is another hurdle to fall out of it (no Service Discovery packet for six seconds). The hurdle to get into the stable state should prevent peers with bad connectivity to interfere peers with good connectivity. The hurdle to fall out of the stable state should prevent peers which are nearby to get marked as *away* spuriously just because some of their Service Discovery packets got lost. Remember that Service Discovery packets are sent over UDP and tend to get lost especially when there is a lot of other traffic on the communication channel.

Every remote peer that we keep a record of in the SyncList is in one of the following states. Peers in the SyncHistory do not have a state.

FADE IN A remote peer enters this state if it gets in range. As long as the peer is marked unstable it remains in this state. After receiving enough Service Discovery packets the status of the remote peer changes to *READY* or *CHOKED*.

READY A peer device with this status is ready to accept incoming connections. If the device has never synchronized with this peer before it will launch a transfer client immediately and start synchronization.

CHOKED The remote peer is stable but does not accept incoming connections.

CONNECTING We are trying to establish a connection to this peer. If we get rejected, the status switches to *choked* otherwise the status switches to *CONNECTED*.

CONNECTED A synchronization with this peer is currently taking place.

3.4 Ad Hoc Transfer Mode

The goal of the transfer mode is to synchronize shared channels between two devices. This is done by exchanging episodes and their enclosures. Every device tries to keep its channels and the containing episodes up to date as good as possible. A synchronization between two devices basically looks as follows:

1. After two devices have made a connection the first step for both devices is to find out if some of the local channels also exist on the other device. After that both devices know which channels they have in common.
2. Now every device tries to update its list of episodes. So the next step is to query every shared channel for a list of episodes. New episodes are added to the channel.
3. Since episodes may contain enclosures they have to be downloaded as well. This is the last step which involves data exchange and the one that takes longest. Especially if both devices have incomplete enclosures they try to complement one another as good as possible.
4. Usually one device is done faster than the other. So the last step is just waiting for the other device to finish downloading. If they are both done the connection is terminated. The connection may also be terminated prematurely if a ratio was hit on one of the devices. We explain the concept of ratios in the following paragraphs.

The Transfer Mode works with TCP sockets and is based on a client-server model. After the local device has decided to synchronize with a remote peer it launches a *TransferClient* which tries to connect to a *TransferServer* at the remote peer. The IP address and port needed for this operation are already known from the Service Discovery packet. After successful connection both devices start to exchange messages. The protocol works like a Request-Response system. It is not a strict system though. Some requests do not generate a response whereas other requests can generate several responses! The goal was to send as few packets as possible and as many as needed.
To make the synchronization fair for both devices they can both download from each other at the same time. Though, in most cases only one device is downloading which creates an unfair situation for the serving device. It has to serve the requests from the connected device although the other device cannot provide new content. A mechanism to prevent larger imbalances are ratios. We can measure ratios in two different manners:

- Upload/Download Ratio. Every device counts the number of bytes transferred and calculates the relation of incoming to outgoing bytes. If this number exceeds a certain threshold we say that the *ratio was hit* and the connection is terminated.
- Upload/Download Limit. Every device counts the number of bytes transferred. If either the number of incoming bytes or the number of outgoing bytes exceeds a certain threshold we say that the *ratio was hit* and the connection is terminated.

There are four different values we can control for ratios: incoming ratio, outgoing ratio, incoming limit and outgoing limit. A serving device should only enforce ratios if it has scheduled synchronizations with other peers. I.e. if there is only one device in range it does not make sense to terminate the connection because of an exceeded download limit!

The *TransferServer* and *TransferClient* functions are almost identical. The synchronization process is separated into eight stages:

CONNECTING This is the first stage where all sockets are opened and the TransferServer and TransferClient are initialized.

HELLO This stage is used to send a hello message to the other end and exchange peer_ids. After the client has connected to the server the server either sends the first hello message or an error message (i.e. *too many connections*). After that the client responds with his hello message. After the messages are exchanged both devices switch to the next stage.

NEGOTIATE The device wants to keep its *channels* most up to date. First of all, we need to find out if any local *channel* also exists on the remote device. Instead of querying every single *channel* the devices exchange a *bloom filter* that contains all *channel ids* a device offers. Both devices then start to test their own *channels* against the *bloom filter* and create a list of possible shared *channels*. For more information on *bloom filters* refer to Sect. 2.3.

QUERY This is a symmetric stage which means that both devices may send as many requests as they like and answer to requests from the other peer. The goal in this stage is to confirm the *channels* that were selected in the previous stage and then download a list of *episodes* offered by the remote device. There are currently two different approaches for downloading *episodes*. The first approach requests all *episodes* that the remote peer has. That way it is possible to complete a probably incomplete list of *episodes* on the local device. This is useful if the *episodes* are treated like a set of objects rather than consequent issues of a podcast. The second approach requests all *episodes* which are newer than a given date. This is very useful for news feeds especially if there are a lot of news items in the *channel*. The amount of data to be transmitted can be reduced drastically. In both approaches all *episodes* received are then matched against the local list of *episodes* and added to it if necessary. After all *channels* have been queried, both devices are in sync with their episode infos. But since *episodes* can contain *enclosures* the device may have to download them from the remote device. The final step in this stage is to create a list of *enclosures* to download from the remote peer. Remark that either device switches to the next stage on its own. It is perfectly legal that one device is already in SERVING stage whereas the other is still in QUERY stage.

DOWNLOAD A device entering this stage starts to process the list of enclosures that was created in the previous stage. Missing *chunks* which are available at the remote peer are now downloaded. Remark that a *chunk* is separated into several *pieces* which are requested one-by-one. If a device cannot find any more data to download it switches to SERVING stage.

SERVING This stage is used to wait for the other device to query and download. If both devices reach the SERVING stage the devices close the connection.

CLOSING Both devices have closed the connection and the internal SyncList and SyncHistory are updated.

FINISHED The TransferClient and TransferServer quits.

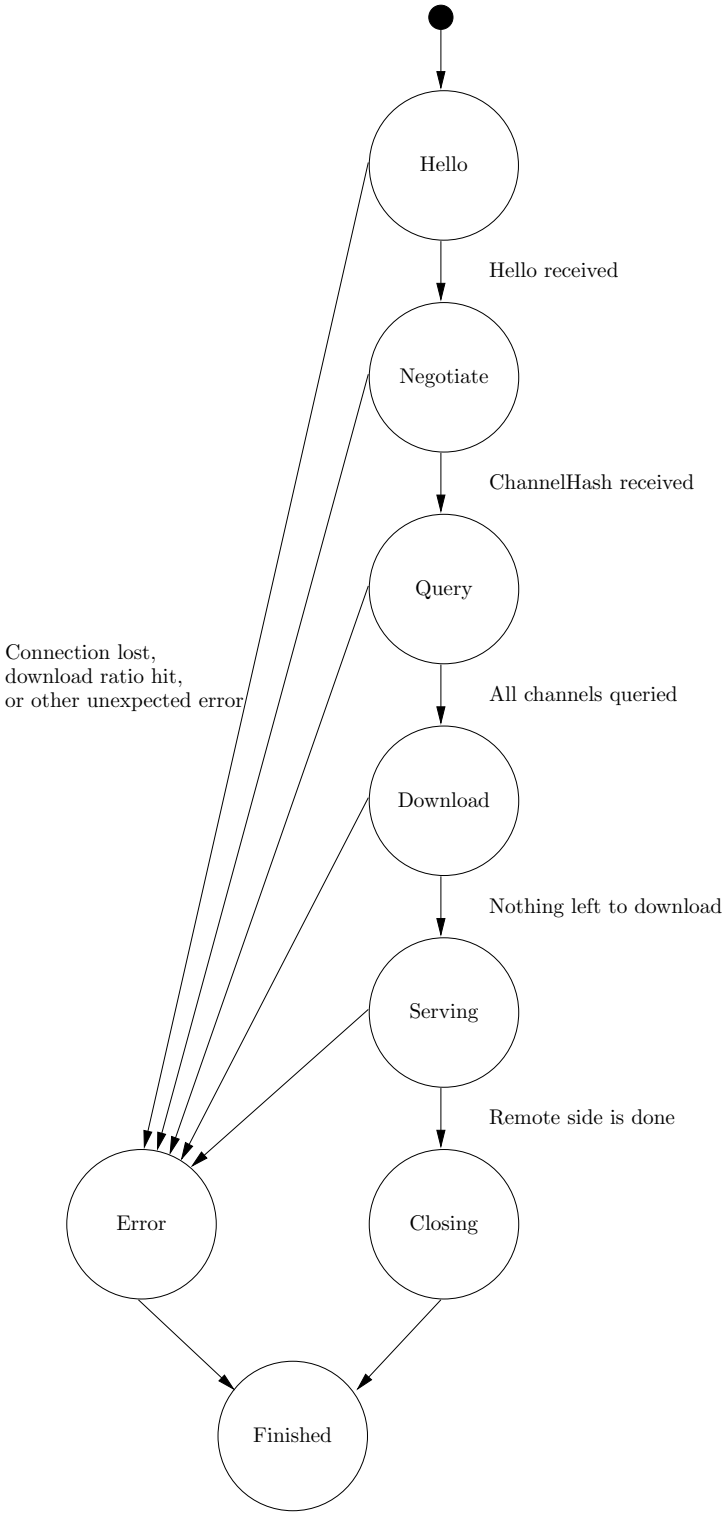


Figure 3.2: TransferServer / TransferClient state diagram

3.4.1 Message Types

ID	Content	Description
Hello	peer_id	Used in Hello stage to exchange peer_ids
ChannelHash	bloom key	Used in Negotiation Stage to exchange channel ids
QueryEpisodes	channel id, date, ref_id	Requests an EpisodeInfo for this channel id. The answer should be marked with the ref_id given
EpisodeInfo	ref_id, xml data	Contains episode encoded as XML snippet. This is the answer to a prior QueryEpisode, the ref_id matches that of the request.
RequestEpisode	channel id, episode id, ref_id	Informs the peer that all further requests for the episode (given by channel id and episode id) are referred to by ref_id from now on.
RequestByRef	ref_id, index, offset, length	Requests the piece at index, offset and length for the episode given by ref_id. Ref_id must have been registered with RequestEpisode before this message might be sent.
Piece	ref_id, index, offset, data	Answer to RequestByRef. Returns the data from piece at index with offset
Done		Sent when the peer enters SERVING stage to inform the remote device that it is done with downloading.
Error	error id, error text	General Error message. Contains an error number (see below) and a human readable text of the error.

3.4.2 Error Codes

Code	Text	Description	Action
400	abort	General error	Disconnect
401	full	Too many connections	Disconnect
402	no such channel	Requested channel could not be found	Go ahead
403	no such episode	Requested episode could not be found	Go ahead
404	piece not found	Requested piece could not be found	Go ahead (implementation broken)
405	pre-condition failed	Remote peer has sent RequestByRef prior to RequestEpisode	Disconnect

3.4.3 Example Sync Session

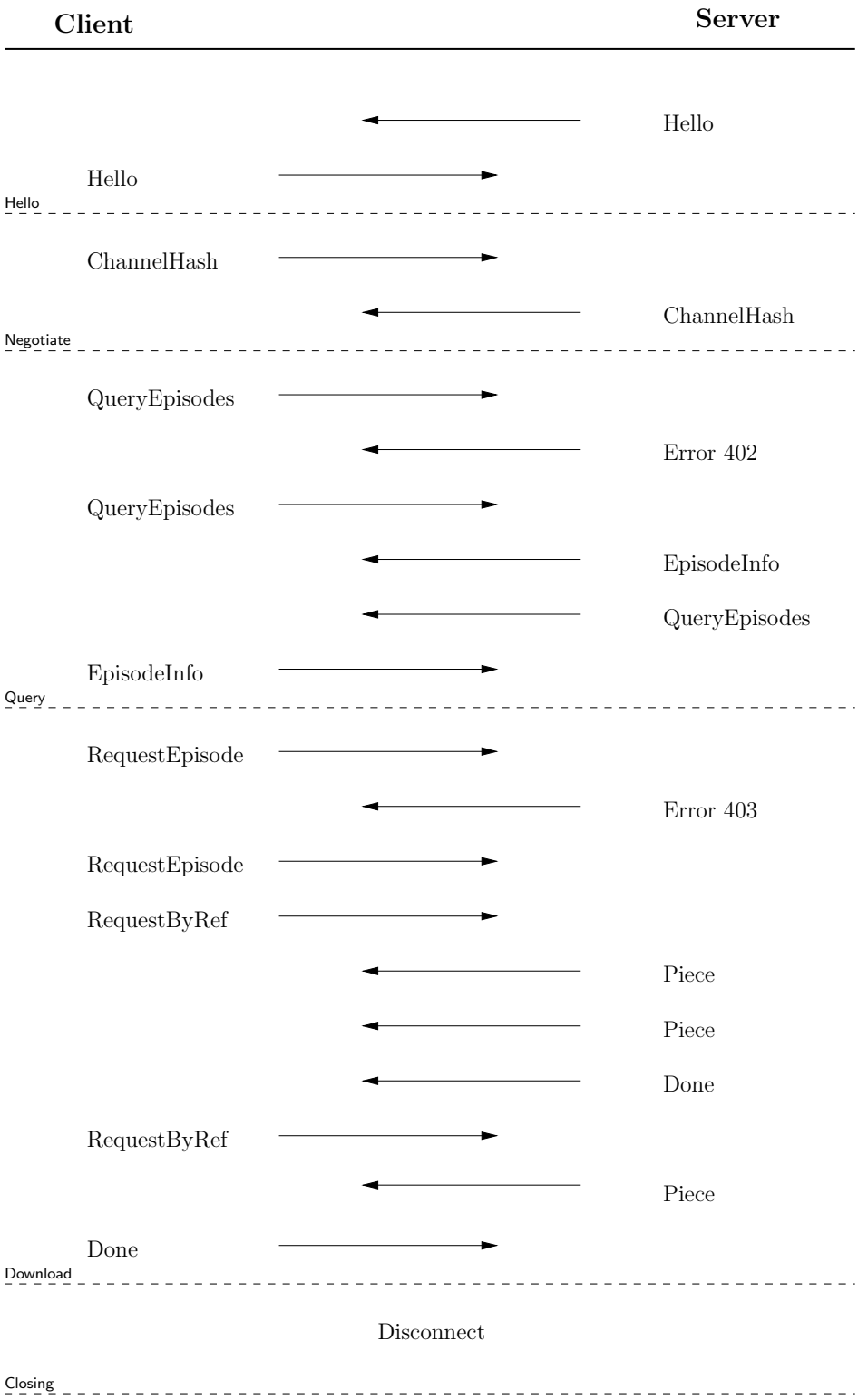


Figure 3.3: Example of network packet flow during a synchronization

Chapter 4

Implementation

This chapter explains all important details of the current implementation. The program is written in C++. Although it was thoroughly tested on Windows CE it also compiles and works on Windows XP/2000, Linux and Mac OS X. A GUI interface is only provided when compiled under Windows CE. An interactive command line replaces the GUI on other platforms. Additionally an integrated RPC-like server gives access to the command line over the network. The command line gives us control over every module and every parameter. Thus, we can completely change the behavior of the program during runtime.
The podcast application is based on the measurement application we developed earlier in [9]. The former application was heavily extended and adjusted. The podcasting application now consists of roughly 18'000 lines of code. This chapter makes re-use of the original documentation where appropriate to provide a complete picture.

4.1 Application Design

Platform Independence

The podcast application makes a lot of use of low-level system calls which introduce problems when porting the application to another platform. The original measurement application used a simple wrapper to overcome this problem.

The following system calls are platform specific:

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

All these functions are collected in the PI (Platform Independence) library which we have originally created in [9] and extended in this work to facilitate platform independent programming. For a detailed discussion about the PI library see App. D.2. A lot of parts are very similar to Simple Directmedia Layer (SDL)[1]. In the current state the PI library is completely working on Windows CE, Windows 2000/XP, Linux and Mac OSX. The necessary adjustments were minimal so we have setup everything inside the file `config.h`.

Threads

The podcast application uses threads for different parts of the program:

1. The main loop itself

[1] See http://www.libsdl.org/ for more information

2. Running data transfers (*TransferServer* class and *TransferClient* class)
3. GUI handling or command line
4. *Automator* class
5. RPC module

Inter-Thread communication is done using *messages* and *message queues*. If a message should be sent to a thread the following steps are necessary:

- Create the message
- Add 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.
- Finally, mark the message for deletion by calling `Discard()` on it.

For an in-depth discussion on the message system see App. D.1.

Model View Controller

To make the Graphical User Interface (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 [2].
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.

4.2 Routing and Service Discovery

The router class consists of a set of member functions that originally implement a flooding algorithm and a simple Service Discovery mechanism. Both are remnants from the former application. The flooding algorithm has been disabled in the podcasting program. The Service Discovery has been extended slightly.
The class contains two functions which are called from the main loop: `Router::ServiceDiscovery()` and `Router::Update()`.

A message arriving at the routers message queue can have three different origins:

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

Messages received from the Service Discovery module are instantly sent over the network. Messages received from the receiver function are subject for routing. The decisions that need to be done for such packet messages are best described in the flowchart in Fig. 4.1. Remark that most of its functionality is no longer used. We have disabled the packet forwarding functionality. This automatically leads to the fact that the Time To Tive (TTL) is not used as well.
It is vital to understand that the router module only sends and receives UDP packets (i.e. Service Discovery packets). UDP packet types other than Service Discovery packets have been disabled. TCP packets used for synchronization are sent and received from within the TransferServer and TransferClient threads and are in no way related to the router functions. The routing code itself has been left alone and works in the traditional way. The flowchart in Fig. 4.1 was added for convenience. It should give an estimate on the big picture.
For more details refer to [9].

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

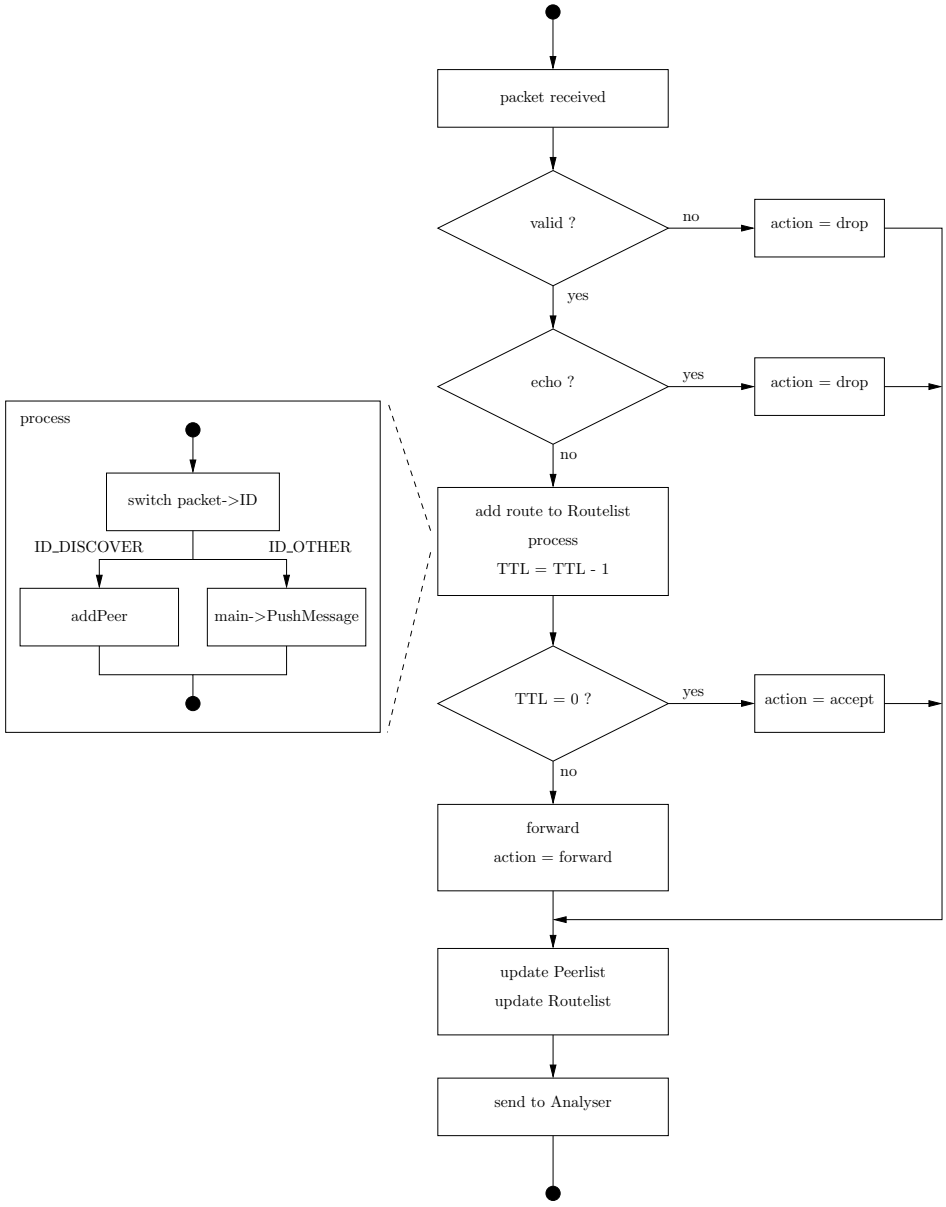


Figure 4.1: Router::Update() flowchart for UDP packets received from the network

4.3 PeerList

The PeerList implements the idea of stable and unstable peers from Sect. 3.3. Its implementation can be found in `router.cpp`. The router class contains one instance of a PeerList. The contents of the PeerList is displayed in the GUI on Windows CE in the Peers dialog. The contents of the PeerList also gets forwarded automatically to the SyncManager (using the *subscription service*).

4.4 SyncManager

The Discovery Mode as well as the SyncList and the SyncHistory are handled by the SyncManager. The module has a subscription to the routers PeerList. Whenever a new message from the PeerList arrives the SyncList gets updated. After that the SyncManager randomly selects one peer from the SyncList that is in state *ready* and tries to establish a connection to it by launching a `TransferClient` thread. The SyncManager is also responsible for accepting incoming TCP connections. If the internal state is set to *ready* it launches a `TransferServer` that handles the connection. Otherwise and error message is sent and the connection is terminated. The bookkeeping of running `TransferServer` and `TransferClient` threads tells the SyncManager which

state we are currently in. The list of running threads is updated when the SyncManager launches a new thread or when a thread terminates.
A thread can terminate in two different ways: *Success* or *Failure*. On successful termination the peer we had a connection to is removed from the SyncList and added to the SyncHistory. In a failure case the peer's state in the SyncList is set to choked and a random penalty time between 2 and 10 seconds is calculated. As long as the penalty time is running no further `TransferClient` threads are launched. Incoming connections are still accepted.
Unsuccessful termination of a thread can have several reasons. The two most common ones are:

- Too many connections on the remote peer.
- A ratio was hit. As described in Sect. 3.4 the connection may be terminated prematurely to prevent imbalances in the data exchange for reasons of fairness.

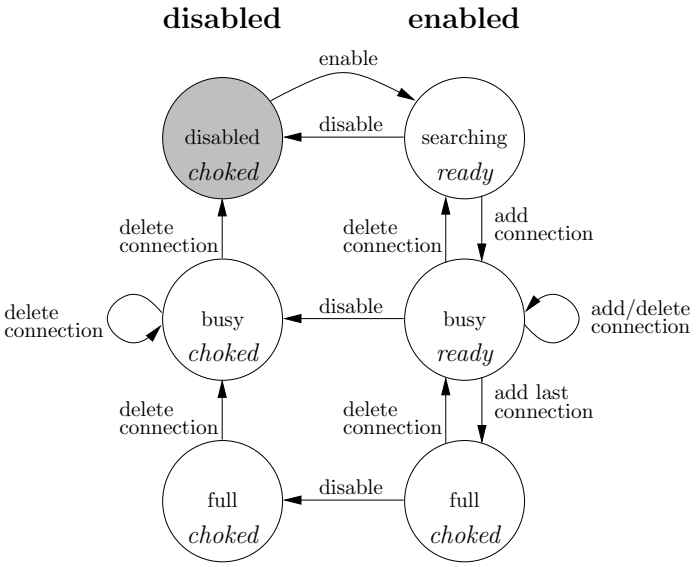


Figure 4.2: SyncManager State Diagram

The SyncManagers state diagram is depicted in Fig. 4.2. It is possible to disable the SyncManager in a clean way. When shutting down, incoming connections are no more accepted. Still running threads are left alone until they terminate themselves.

4.5 Transfer Client And Transfer Server

These two classes are by far the most complex ones. They share a lot of common code in the class `TransferCommon`. They implement a state machine whose state diagram can be found in Fig. 3.2. Its implemented exactly as shown. The data synchronization over TCP is handled completely from within these classes.
It does not make sense to explain every little detail as this information can be gathered from the source code just as well. We will focus on the big picture instead.
A transfer thread has its own main loop as well as a message queue that is used solely by the transfer thread and acts as a to do list. The message queue contains both messages that should be sent over the network as well as messages that we have received from the network. During the main loop the TCP socket is checked and arriving packets are transformed to messages and added to the queue. After that the thread processes the message queue. Messages that should be sent are transformed to network packets and sent over the network instantly. Incoming messages are decoded and handled according to their content and the internal state of the transfer thread. Following a list of important functions inside the `TransferCommon` class:

ThreadRun() contains the main loop

Send() transforms a message into a network packet and sends it over the communication channel

Receive() transforms an incoming network packet into a message and adds it to the message queue

Do_Messages() handles the message queue

Handle_xxx() these functions handle incoming messages of the given type

Process_xxx() these functions are responsible to advance in the state machine and deal with the different states of synchronization such as initiating the handshake or actually downloading chunks etc.

4.6 Data Storage Module

The data storage module can be found in `datastore.cpp`. All channels, episodes and enclosures are maintained from within this class. The module loads and saves XML files and also deals with the separation of files into chunks and pieces. It is used by the TransferServer and TransferClient threads which add new episodes and read or write piece data from single episodes. Its implementation is thread-safe. The only other access to this module happens from the GUI code which reads the content of this module and displays it in nice looking dialogs.

4.7 XML Processing

XML reading and writing is done using TinyXML. It is a very simple XML parser that helps us reading and writing the channel and episode information from the config files. TinyXML can also be used to read RSS and Atom feeds from the Internet to convert the information to the internal format.
TinyXML documentation can be found at `http://www.grinninglizard.com/tinyxml/`.

4.8 Network Packet Format

The network packet format used in this work is completely different from the one used in the former program. Its implementation can be found in `flexpacket.cpp`. The concepts used are mostly taken from the OBject EXchange (OBEX) specification [13]. We are not going to discuss this format in great detail because 99% is identical to the OBEX protocol. We will focus on the big picture and show the differences to a *FlexPacket*.

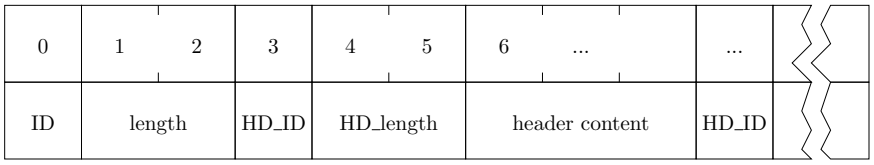


Figure 4.3: FlexPacket Byte Layout

The first byte of a *FlexPacket* contains the ID of the packet. For OBEX packets these IDs are pre-defined and only a small number such as GET(0x83), PUT(0x82) or ABORT(0xFF) may be used. There is no such limitation for *FlexPackets*. Byte 2 and 3 both encode the total length of the packet (including the ID and the length field) as 16-bit unsigned integer. Numbers in *FlexPackets* as well as in OBEX packets are always encoded in Network Byte Order (Big Endian). After the first 3 Bytes an arbitrary number of data objects called *headers* may be added to the packet. This might be a little bit confusing as the first 3 Bytes of the packet are usually referred to as header already. But OBEX refers to them as *headers* so we do as well to minimize confusion.
The first byte of a *header* defines its type. The two most significant bits of this byte define the header-type. The size of a *header* differs between header-types. There are four types of *headers*:

Unicode (0 – 63) Contains a unicode encoded text string. The length is pre-fixed with a 16-bit number. The minimal length is 3 and the length of a header with the string ”hello” is 3 + 5 = 8 Bytes.

Byte Sequence (64 – 127) Contains arbitrary bytes or text. The length is pre-fixed with a 16-bit number.

1Byte (128 – 191) Contains a single 8-bit number. The total length of this header is 2 Bytes.

4Byte (192 – 255) Contains a single 32-bit number. The total length of this header is 5 Bytes.

In OBEX the possible header-types are pre-defined. There is no such limitation for *FlexPackets*. For more information on the *FlexPacket* format have a look at the source code or the OBEX specification[13].

4.9 Analyzer

The analyser class takes care of our log files. Every packet received is converted to a semicolon separated log line and appended to the log file. 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. 4.4 for a flowchart that describes the most important part of the analyser.

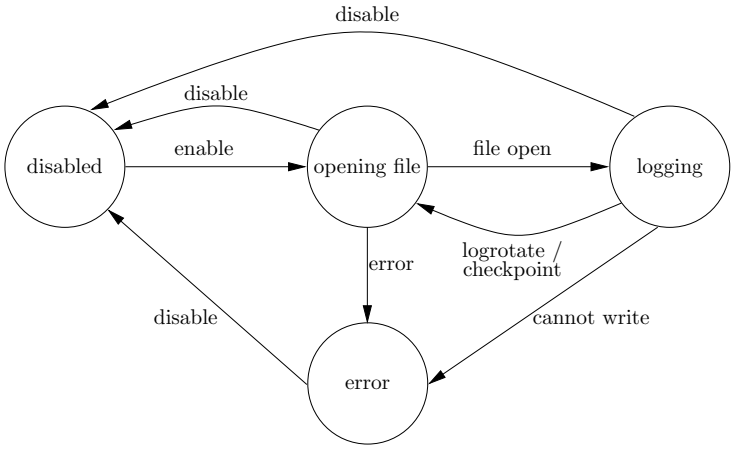


Figure 4.4: Analyser module flowchart

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.
Many options of the analyser are adjustable in `config.h`.

4.10 Command Line And Remote Control

The Command Line Interface (CLI) is automatically active if the program is compiled under Windows, Linux or Mac OSX. Its implementation can be found in `rc_commands.cpp`. The CLI runs in a separate thread. We can completely remote control the program through the CLI.

Every action is bound to a command. The most important command to get started is `help`. It outputs a list of all available commands. More help on a specific command can be obtained by typing `help` *command*.
Table 4.1 lists all currently available commands:
Adding new commands is very simple. Just have a look at the source code.
When the application is launched, it looks for one of three configuration files. Only the first one found is executed.

Command	Description
ana_filesize_flush	Set analyser file size that triggers a checkpoint
ana_logdir	Set analyser log directory
ana_max_filesize	Set analyser max file size before log rotate
analyser	Controls the analyser (enable/disable)
automator	Controls the automator (enable/disable)
automator_add	Adds a command to the automator queue
automator_addfile	Adds commands in file to the automator queue
automator_clear	Clears the automator queue
automator_q	Display the automator queue
base_folder	Sets the base folder
channel_add	Adds a channel
channel_list	Lists all loaded channels
channel_remove	Removes a channel
channel_removeall	Removes all channels
checkpoint	Triggers a checkpoint action in the analyser
devicename	Gets/Sets the devicename
episode_add	Adds an episode
episode_list	Lists all episodes
episode_remove	Removes an episode from a given channel
episode_removeall	Removes all episodes of a given channel
exit	Exits the program
log	Sends a message to the message log (use ” to embed spaces)
logrotate	Triggers a logrotate action in the analyser
my_ip	Gets/Sets the local IP address (announced in discovery)
net_timeserver	Enables the internal timeserver
peer_id	Gets/Sets the Peer ID
play	Starts playing the first audio stream found (requires fmod)
quickchat	Gets/Sets a quickchat message
router	Controls the router (enable/disable)
rpc	Controls the RPC console (enable/disable)
rpc_listenport	Sets/Gets the listen port of the RPC console
rt_broadcast	Router UDP broadcast address
rt_fadein_packets	Sets number of packets for fade-in
rt_heartbeat	Router heartbeat interval for service discovery packets
rt_maxttl	Router max TTL
rt_peer_timeout	Sets peer timeout
rt_port	Router UDP port
rt_send_mply	Router send packet multiplier
rt_seq_windowsize	Router window size for sequence numbers
save	Saves the configuration
setup_mode	Switches to Setup Mode
sleep	Sleep (ms, s, m or h, defaults to ms)
stream	Starts streaming an audio file
stream_srate	Sets the streamer sampling rate
sync_connections	Sets/Gets max number of allowed connections
sync_downloadminsize	Set transfer download limit
sync_downloadratio	Set transfer download ratio
sync_history	Displays the current sync history
sync_history_ncc	Enables/Disables new content date check
sync_history_timeout	Sets sync history timeout
sync_list	Displays the sync list
sync_listenport	Sets the listening port of the sync manager
sync_manager	Controls the sync manager (enable/disable)
sync_uploadminsize	Set transfer upload limit
sync_uploadratio	Set transfer upload ratio
xml_config	Set the XML configuration filename

Table 4.1: Command Line Interface: Available command overview

1. config.txt
2. config-client.txt
3. config-server.txt

After the configuration is loaded the program initializes all modules and starts them. Just before entering the main while loop the program tests for the file `autoexec.txt` and executes it if it exists.

Another very handy feature is an RPC-like server that also runs as thread. Its implementation can be found in `simple_rpc.cpp`. The RPC-like server listens on port 5000 for incoming connections. Any telnet client can be used to make a connection to the program. Once connected the same rules apply as for the command line. The only drawback is that the console's output will not be sent back over the network due to limitations in our debugging code (which is responsible for console output as well). Instead a message will be returned if the command was executed successfully or if it failed.

4.11 Debugging

Debugging is done using debug messages. There are two functions that 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 time stamp with the 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. It can be changed in `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 `#include "debug.h"` can be used.

Chapter 5

Testbed

5.1 Ad Hoc WLAN Setup

The ad hoc network consisted of 19 HP iPAQ devices running Windows Mobile 2003 Second Edition as well as 5 ThinkPad T60 running Windows XP. All devices communicated over their integrated WLAN interface. The podcasting application is written in C++ natively for Windows CE. Since it is not possible to access RAW sockets in Windows CE we were forced to use User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) packets. The devices were configured to use ad hoc mode with the SSID *da_podcast* and an IP address in the private range 192.168.0.X/24. 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] 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! [2]

[1] To make the devices use another channel laptops have to be used to create the ad hoc network on a different channel and then let the devices join one after the other.

[2] See Sect. 4 for more information on setting the time.

Chapter 6

Measurements

We decided to focus on real life behavior of our application. Therefore all measurements have been conducted on ETZ G floor where we have lots of interference with other access points.
After analyzing the results of a measurement we often had to change the program or add functionality which usually led to a change in its performance.
These conditions mean that:

- Successive runs of the same measurement often produced similar results. This consistency among multiple measurements is important to validate our results.
- Some of the runs also showed huge differences in performance which are attributed to the instability of the WLAN drivers and the operating system itself. We discuss these irregularities in detail if they occur.
- We have to be careful when comparing runs that are based on different revisions of the podcasting application.

6.1 Experimental Parameters

In a first attempt we wanted to know how good the application performs in a static situation. We arranged the devices on a table with a maximum distance of one meter. Power cords were always connected. We varied the following parameters:

- number of devices
- sync history strategies
- disconnect strategies (with or without ratios)
- discovery packet intervals
- number of files to exchange
- number of channels to test

Every measurement was conducted three times in total. We discuss every single measurement including graphs of all three runs in Appendix A.
In the following section we will discuss several aspects of the synchronization on a small number of selected measurements.

6.2 Overhead

In this test, we measured the overhead that is created during a synchronization. The devices were configured to contain one channel with one episode which in turn contained one enclosure of 5MB. We also varied the distribution and disconnect strategies but we could not see any difference. The results have also shown that it does not matter how many devices are in range.
The overhead is a fraction:

$$\text{overhead} = \frac{\text{number of management bytes transferred}}{\text{total number of bytes transferred}}$$

whereas

$$\begin{aligned}\text{number of management bytes transferred} &= \text{total number of bytes transferred} \\ &\quad -\text{bytes of payload content transferred}\end{aligned}$$

In the case where content is transferred in only one direction, the downloading device achieves an overhead of only 1.5%. Unsurprisingly, as the serving device does not receive any content its overhead remains 100%.
The other case to consider is when both devices have content to exchange so that payload data is flowing in both directions. Here, both devices achieve an overhead of approximately 3%.

6.3 Single File Distribution

In this test, we wanted to know how fast a single file can be spread across all other devices. How long does it take until the last device has downloaded the file completely? All devices contained a common channel. One device (the server) contained one episode with an enclosure of one file (size: 5MB). We started the application on all devices at the same time.

6.3.1 Performance and Fairness using Ratios

This section compares the effect of download ratios.

1. No ratio. If a devices finds a file to download during a synchronization the file is downloaded completely.
2. Ratio enabled. A device may download a total of 1MB of payload data until the serving device disconnects the connection. Both devices react with a random penalty timeout. This makes it possible for other devices to create a connection to the serving device.

The following discussion contains two types of figures:

Download Time (left hand side)

- The x-axis denotes the number of devices used in this run
- The y-axis shows the time in seconds since the start of the application

Legend	Description
connected	The time it took for a single device to successfully connect to another device for the first time (mean over all devices)
completed	Time it took to completely download the file (mean over all devices)
completed min	Time when the first device was able to download the file completely
completed max	Time when the last device was able to download the file completely

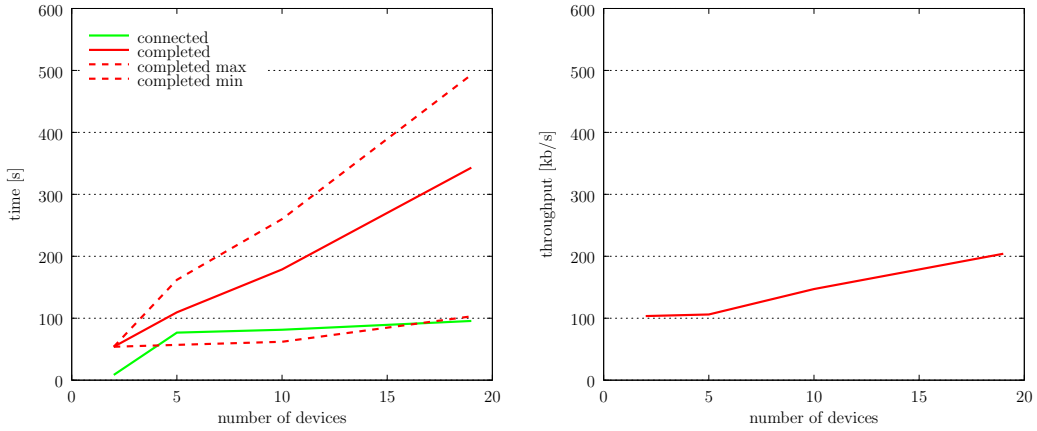
Transfer Speed (right hand side)

- The x-axis denotes the number of devices used in this run
- The y-axis shows the cumulative transfer speed of all devices during a synchronization session in KB/s. The speed of 0 KB/s when a device is idle (i.e. has no connection to another device) is not taken into account as we are only interested in the actual troughput. To calculate the throughput we measure the time of a connection. We start the stopwatch just before opening the socket and stop it just after the socket was closed. The stopwatch returns the number of milliseconds between start and stop.

Additionally we count the total number of bytes transferred (i.e. everything that we send over the socket including overhead such as meta information). The get the throughput we divide the total number of bytes transferred by the time measured.

Whenever possible we tried to use the same scale for all figures.

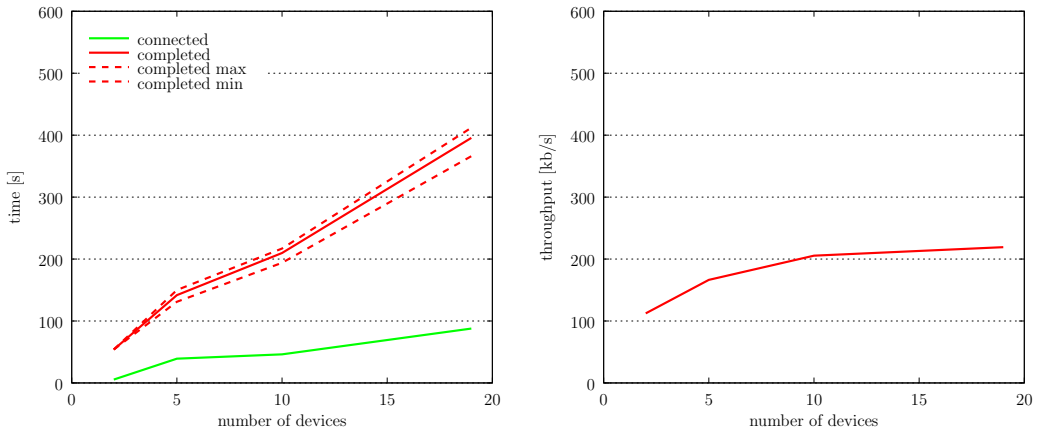
The discovery interval was two seconds for all runs.



(a) Download Time (b) Transfer Speed

Figure 6.1: Performance and fairness: synchronization every 30 seconds, discovery interval 2 sec, no ratio

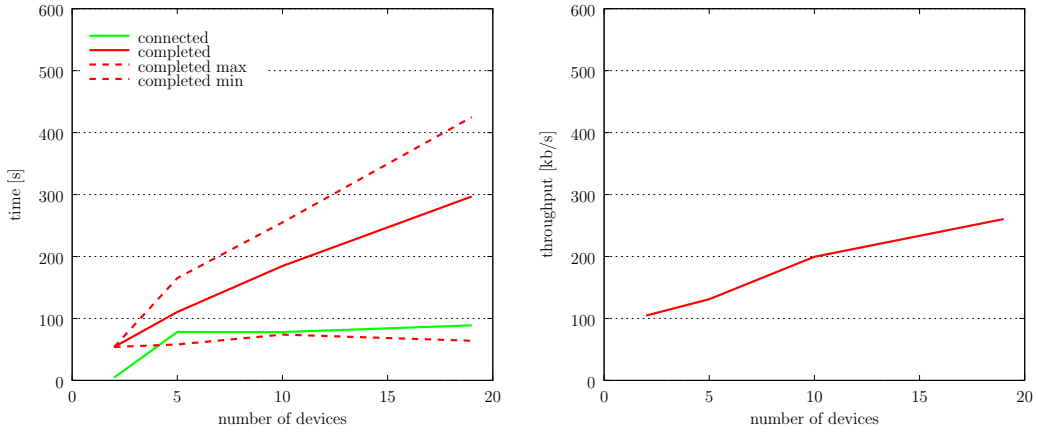
If we allow a device to download a file completely it is obvious that the first device able to connect to the server will have a very good download time. Fig. 6.1(a) shows a download time of roughly 100 seconds for the first device in the 19-device measurement. Of course the last device to finish the download has to wait a very long time until it can finally connect to a device that offers the file. Not enforcing ratios results in a very unfair environment.



(a) Download Time (b) Transfer Speed

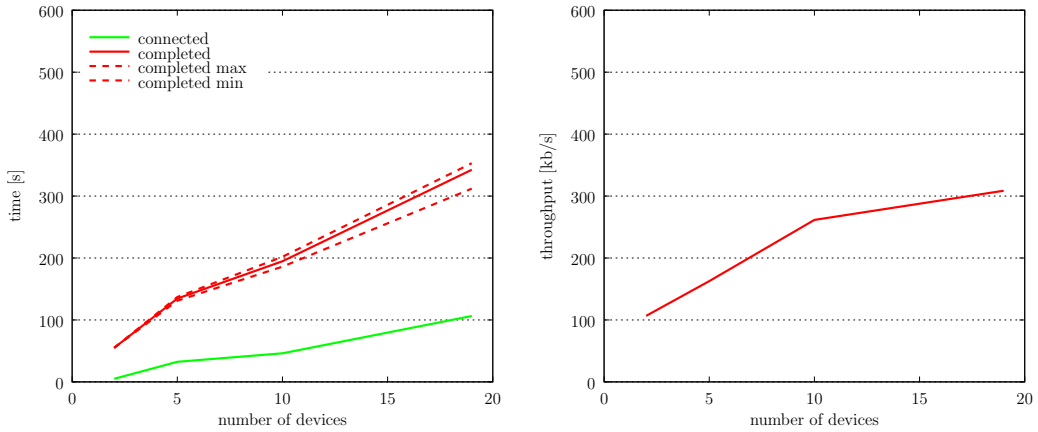
Figure 6.2: Performance and fairness: synchronization every 30 seconds, discovery interval 2 sec, ratio 1MB/synchronization

A download ratio of 1MB splits our 5MB file into 5 synchronization sessions which is enough to create a fair environment. In Fig. 6.2(a) we can also see that the last device finished the download faster than in the case with no ratios. Fig 6.2(b) reflects that as well with a small increase in throughput.
As with Fig. 6.1, Fig 6.3 shows the same unfairness. Interesting here is the effect of intelligent discovery packets. The overhead created by unnecessary re-synchronisations in Fig. 6.1 results in a synchronisation time that is almost 100 seconds faster with intelligent discovery packets.



(a) Download Time (b) Transfer Speed

Figure 6.3: Performance and fairness: intelligent discovery, discovery interval 2 sec, no ratio



(a) Download Time (b) Transfer Speed

Figure 6.4: Performance and fairness: intelligent discovery, discovery interval 2 sec, ratio 1MB/synchronization

The effect of intelligent discovery packet is visible again in Fig. 6.4. Performance is higher and download time is smaller than in Fig 6.2 due to more efficient handling of the WLAN channel.

6.3.2 Distribution Strategies

We compared three different distribution strategies.

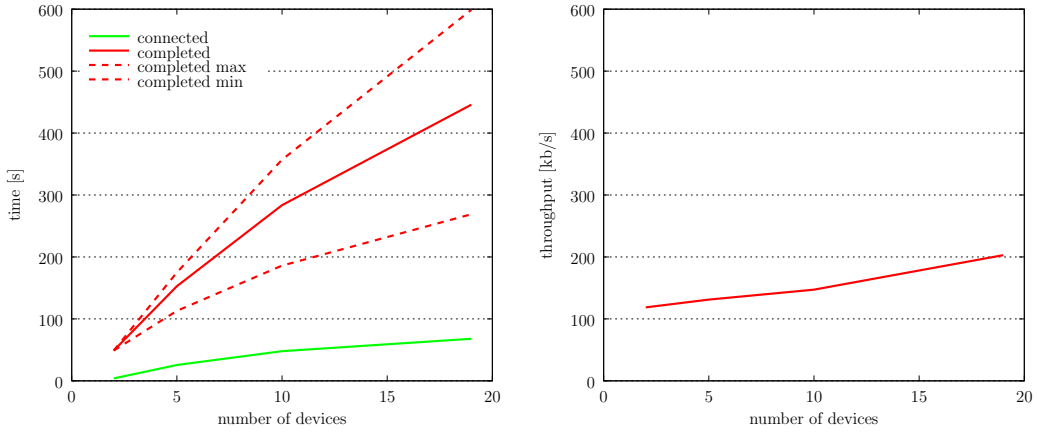
1. One-time synchronization. Every device synchronizes with every other device exactly once. This is enough to successfully spread the file among all devices. Obviously this is not very efficient.

2. Synchronization every 30 seconds. No synchronization is done with a specific peer as long as this peer is part of the *sync history*. In this mode, peers are deleted from the *sync history* 30 seconds after they have been added to it. This automatically leads to a re-synchronization as soon as the peer device is ready to accept new connections.

3. Intelligent discovery. In order to minimize the number of unnecessary synchronization attempts from the previous strategy we used the *new-content-date* field from the discovery messages to find out when we have to remove a peer from the *sync history*.

The following measurements have download ratios enabled which means that a synchronization session is automatically terminated after one device has downloaded 1MB of payload data.

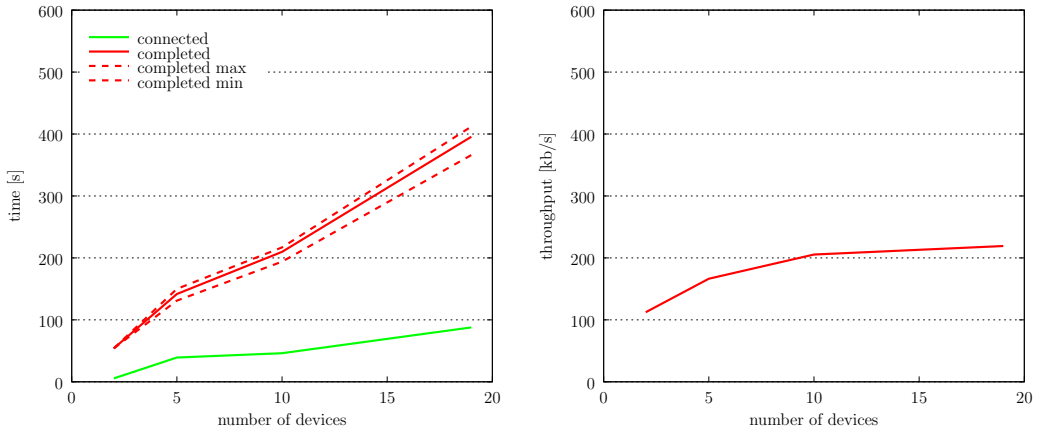
For the graphs in this section the same rules regarding scale and legend apply as in the previous section.



(a) Download Time (b) Transfer Speed

Figure 6.5: Single file distribution: one-time synchronization, discovery interval 2 sec

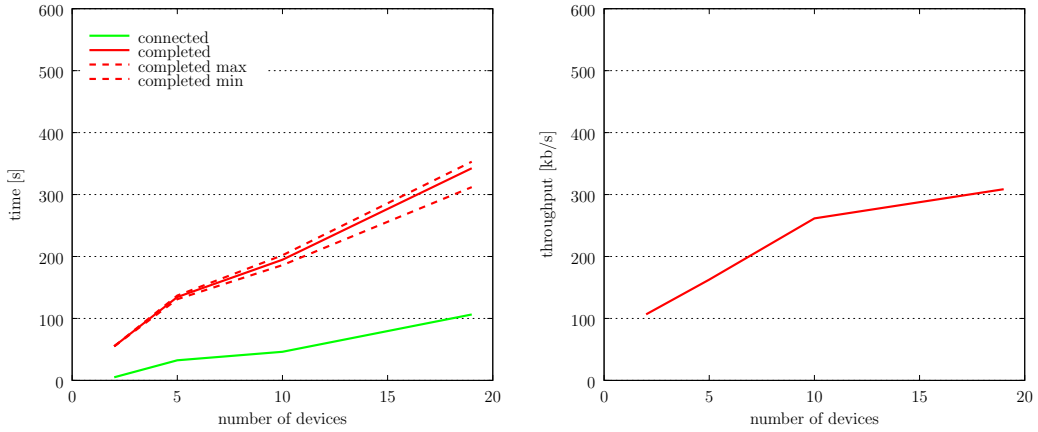
As we can see in Fig. 6.5, one-time synchronization leads to a bad distribution. It takes slightly more than 600 seconds for the last device to finish the download whereas the first device is done after only 270 seconds. With 19 devices we have 18 leechers and one server. The problem here is that one lucky device out of those 18 is able to create a connection to the server and starts synchronizing. In the mean time the 17 other devices synchronize with each other. But as they have nothing to share they are done within 30 seconds. After the lucky device has finished the download of the first megabyte all other peers only have two peers left in their *sync list*: the server and the lucky device. Ratios don't help here because a synchronization between two devices that have nothing to exchange is much faster than a download of 1MB. The rest of the synchronization phase consists of those two peers distributing the file to all other devices. This in turn leads to the bad results.



(a) Download Time (b) Transfer Speed

Figure 6.6: Single file distribution: synchronization every 30 seconds, discovery interval 2 sec

As expected a re-synchronization every 30 seconds removes the problem of bad distribution. We can see from Fig. 6.6 that the last device finished the download after barely 400 seconds. We can also see that this strategy increases fairness as the first and the last device to finish the download are only around 50 seconds apart. Fig. 6.6(b) shows only a slight increase in performance. This is due to the fact that we have a lot of unnecessary synchronization attempts which "pollute" the WLAN channel and thus, decrease the maximum achievable performance.



(a) Download Time (b) Transfer Speed

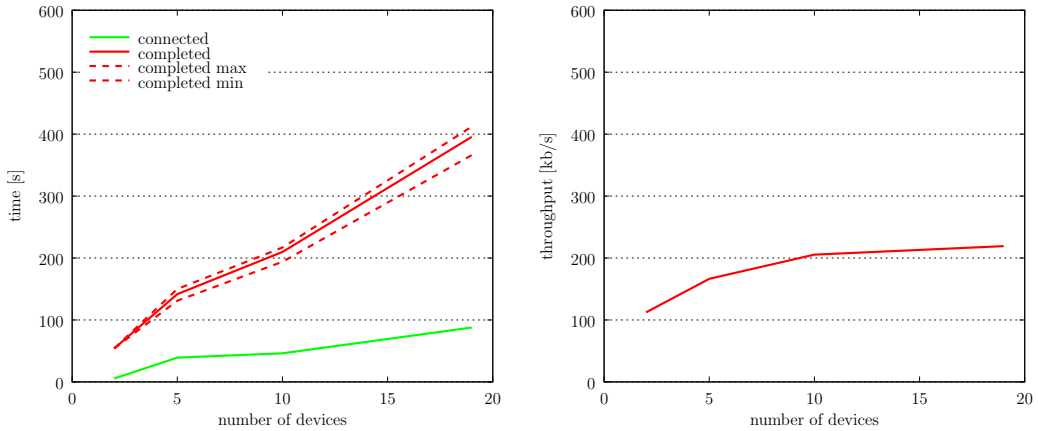
Figure 6.7: Single file distribution: intelligent discovery, discovery interval 2 sec

Introducing intelligent discovery packets in Fig. 6.7 shows an even greater improvement compared to Fig. 6.6. Especially Fig. 6.7(b) shows that optimizing the number of re-synchronization attempts significantly increases performance.

6.3.3 Discovery Packet Intervals

This test should indicate if the discovery packet interval has an influence on the data throughput. We started all our measurements with a discovery packet interval of two seconds. This means that we send a discovery packet every two seconds to the UDP broadcast address. If we send too many of those packets we risk jamming the communication channel with useless data. If we increase the interval too much it takes longer until two devices "see" each other because of our stable/unstable peer mechanism.

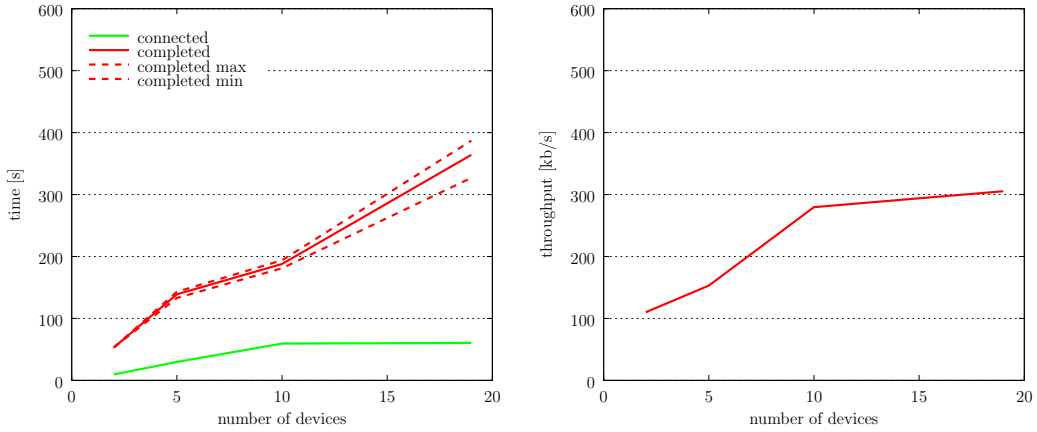
To find out if the interval has a major influence on the synchronization throughput we doubled the discovery packet interval to four seconds. The following two measurements only differ in this parameter. The synchronization strategy used was: re-synchronization every 30 seconds. As convenience we copy Fig. 6.6 as Fig. 6.8.



(a) Download Time (b) Transfer Speed

Figure 6.8: Single file distribution: synchronization every 30 seconds, discovery interval 2 sec

As we can see by comparing Fig. 6.8 with Fig. 6.9 an increase in the discovery interval also increases the transfer speed. The download times also get slightly better.

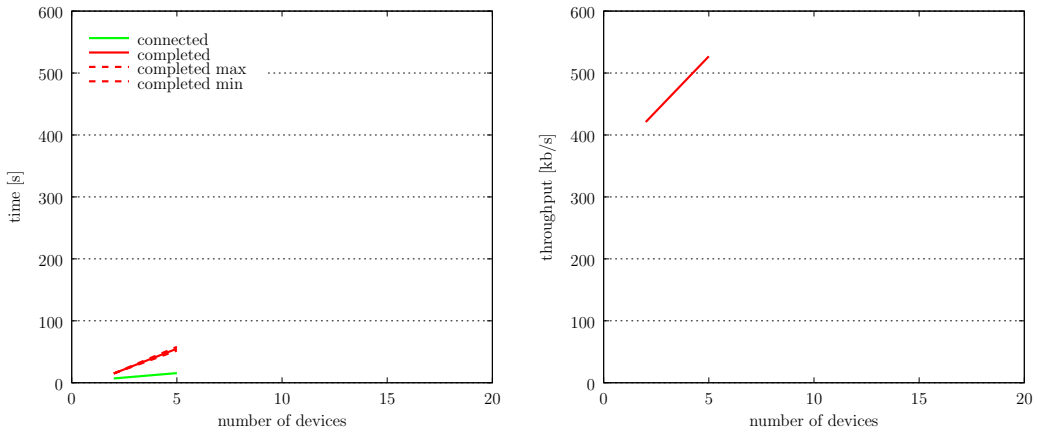


(a) Download Time (b) Transfer Speed

Figure 6.9: Single file distribution: synchronization every 30 seconds, discovery interval 4 sec

6.3.4 Laptop Run

The laptop measurement was done to verify the maximal achievable throughput. We wanted to know why the iPAQs only achieved around 100KB/s between two devices. Despite that, the laptop measurement is incomplete as we only had 5 ThinkPads.



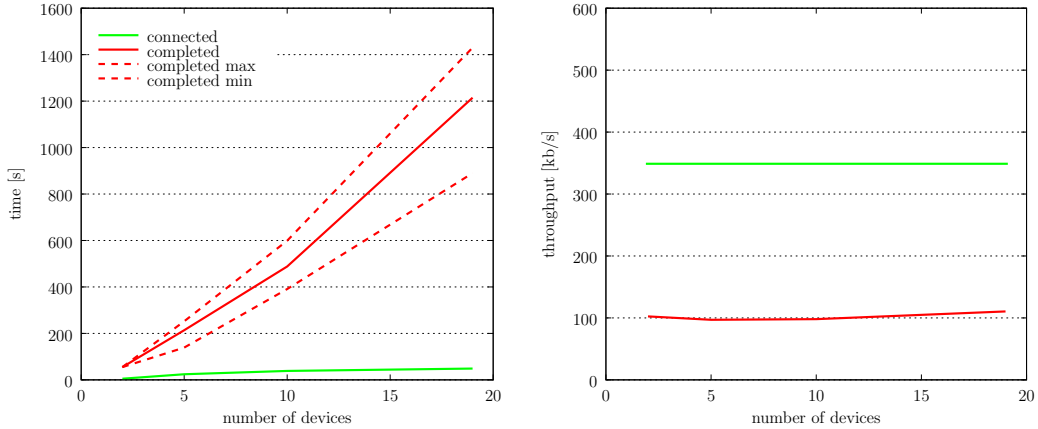
(a) Download Time (b) Transfer Speed

Figure 6.10: Single file with laptops: synchronization every 30 seconds, discovery interval 2 sec

We have to be careful when interpreting the figures. Fig. 6.10(b) shows an increase in speed in the 5-device measurement. This might be an invalid measurement or an imprecision in our analysis. Remark that the laptops achieve a throughput of 500KB/s. The file to be transmitted has a size of 5MB only and the download limit was set to 1MB per synchronization. This means that the duration of one synchronization lies around two seconds. The effect of establishing a TCP connection and all the management messages that have to be sent across the communication channel may have a huge influence on the transfer speed. To get a useful laptop measurement we would have to repeat this setup with bigger files and other disconnect strategies.

6.3.5 Server Approaches Synchronized Group

This test was done to verify the linearity of a sequential download. The measurement starts with all devices except the server. After all devices are in sync with each other the server is switched on. As there are no re-synchronizations and thus no parallel downloads this leads to a minimal distribution speed. The server has to connect to one device after another and distribute the file directly.



(a) Download Time (b) Transfer Speed

Figure 6.11: Server approach: one-time synchronization, discovery interval 2 sec

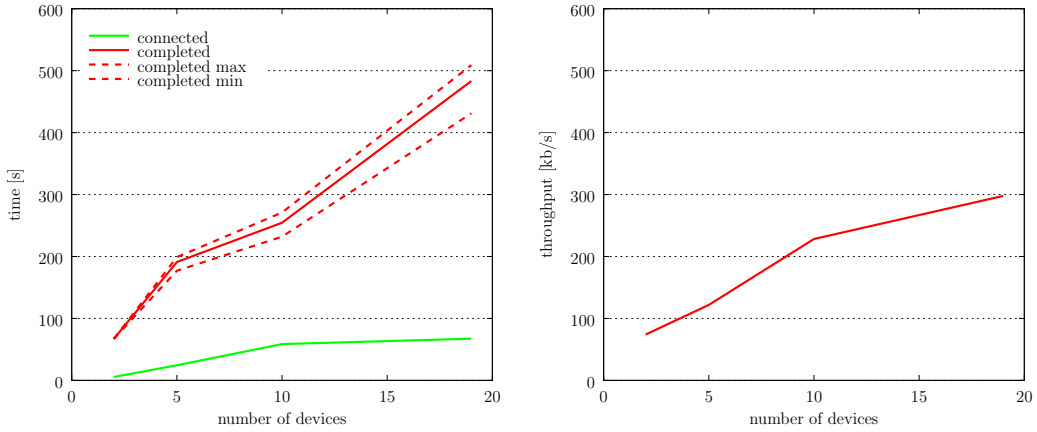
We expect a linear curve in Fig. 6.11. As a matter of fact the download times get slightly worse with more devices. We can explain this with the number of concurrent discovery packets that are sent by all devices every two seconds. Although we tried to keep them small and fill them only with important data they still have a size of a few 100 bytes per packet. With 19 devices sending their discovery packets every two seconds this effectively jams the WLAN channel. The simple discovery mechanism used in the application does not scale well for a large number of devices.
As we see from Fig. 6.11(b) the devices download with a speed of around 100KB/s. This is the actual maximum that our application on HP iPAQs can achieve. To find out the theoretical maximum that can be reached on HP iPAQ handhelds we have created a separate application that reads arbitrary data from a file on the storage card, sends the data using TCP over WLAN to another device which writes this data back into a file on the storage card. The throughput achieved is 350KB/s and is denoted in Fig. 6.11(b) as the upper curve. A short benchmark with the storage card showed a throughput of 2.5MB/s for reading and writing. The storage cards are obviously not a bottleneck. An analysis of the podcasting application tells us that we need a total number of 2 *memcpy* calls to read the data from the socket and writing it into a file. The throughput test application only needs one *memcpy* call. An additional memcpy call effectively cuts the throughput in half (i.e. 175KB/s). This and the additional computing in the podcasting application explains why we can only achieve a throughput of 100 KB/s (with peaks at 120KB/s).

6.4 100 Small Files

In this test, we wanted to know how fast 100 small files can be spread across all other devices. How long does it take until the last device has downloaded all files completely? All devices contained a common channel. One device (the server) contained 100 episodes all containing an enclosure of one file (size: 50KB). We started the application on all devices at the same time. We have chosen the file size so that all 100 episodes together have the same size as the 5MB download from the former analysis. The only difference in this measurement is the administrative overhead created by the 100 episodes as we have to send the episode information back and forth during every synchronization regardless if we can download any data or not.
For convenience we have included Fig. 6.2 here again (Fig. 6.13) as we have to compare it against Fig. 6.12. At a first glimpse both look very similar. When looking a little bit more precisely at Fig. 6.12 we can see that it takes longer to distribute 5MB spread across 100 episodes than just one episode with a 5MB enclosure. A larger overhead caused by the meta information of the 100 episodes increases the total number of bytes we have to transfer.

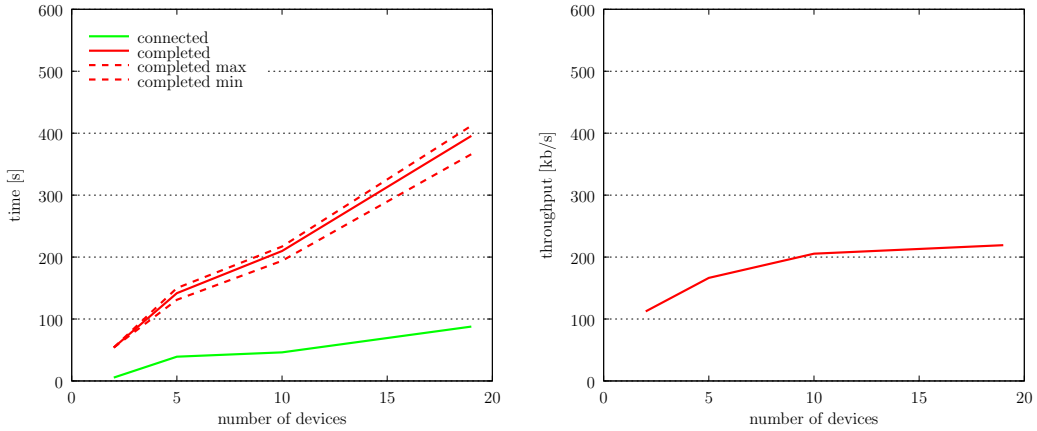
6.5 Performance of Bloom Filter

The bloom filter measurement requires two devices only. Both devices have 1, 10, 100, 1000, 5000 and 10000 channels. Only one channel is common on both devices. This channel contains one



(a) Download Time (b) Transfer Speed

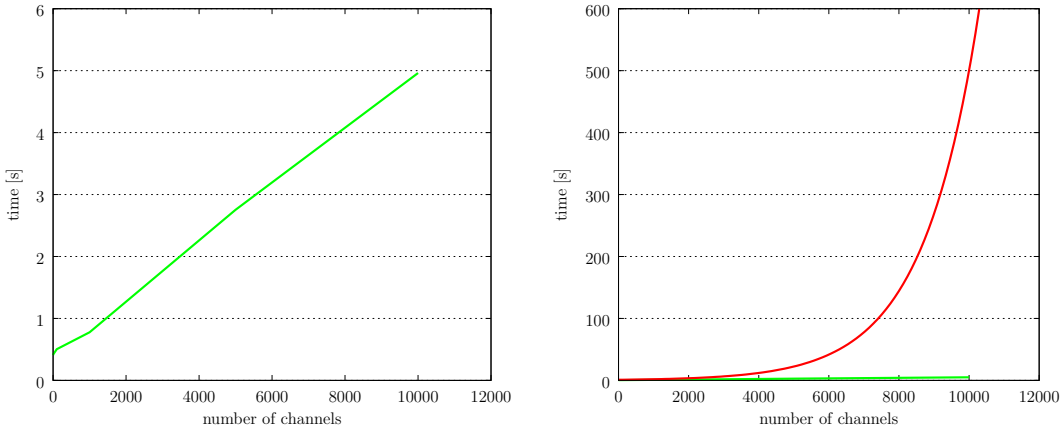
Figure 6.12: 100 small files: synchronization every 30 seconds, discovery interval 2 sec



(a) Download Time (b) Transfer Speed

Figure 6.13: Single file: synchronization every 30 seconds, discovery interval 2 sec, ratio 1MB/synchronization

episode without enclosure on one device so it will synchronize to the other device. Intelligent discovery packets were enabled to ensure that there will only be one synchronization.
The question here: How long does the synchronization take? What is the influence of the bloom filter on the synchronization?
The x-axis shows the time in seconds from the start of the synchronization session until disconnect. The y-axis shows the number of channels that are present on both devices.
The curve in Fig. 6.14(a) denotes the total time that was used for the synchronization. We started the measurement just before we opened the *socket* and stopped the time just after closing the *socket*.
As expected (or promised by the bloom filter) the increase is linear. This is because we adjust our bloom filter automatically in size so that the false positive probability remains at 1% at all times. A synchronization of 10000 channels only takes five seconds which is a very good result. With a false positive probability of 1% the program usually has to query 10 channels if it maintains 1000 channels. The meta information for one channel has an approximate size of 500 bytes. This means that we have to send 5KB of data over the communication channel. It is obvious that this does not take 5 seconds (also refer to 6.3.5). The computation time needed to query the bloom filter for 1000 (or more) channels dominates.
We know from the other measurements that the application's throughput lies around 100KB/s. As the meta information for one channel has about 500 bytes we estimated the time it would take to send this data over the network. Fig. 6.14(b) shows the estimated synchronization time (best case) without bloom filter (upper curve) and the measured results with the bloom filter



(a) Download Time

(b) Download Time

Figure 6.14: Bloom filter

active from Fig. 6.14(a) (lower curve).
The size of the bloom filter for 1000 channels lies around 1200 Bytes. What we cannot see in this graph is the actual load time of the program. We have used TinyXML which does not use stream processing for XML data. This means that the whole XML file is loaded into memory first and then parsed for channels and episodes. The channels and episodes are also kept in memory at all time. This is very inefficient but justified with the fact that the program was never intended to maintain 10000 channels. The program takes almost three minutes to load and uses 35MB of system RAM. A typical configuration with only 10 channels makes the program start instantly and taking up only 3-5 MB of system RAM.

Chapter 7

Conclusions

7.1 Conclusion & Prospect

This master thesis consists of three parts:

- Design of a wireless ad hoc podcasting application that is able to exchange and synchronize content with neighboring peers
- Implementation and testing of this application on HP iPAQ handhelds
- Measurements and analysis of various experimental parameters and distribution scenarios

We have proposed a solution that extends the concepts of podcasting to mobile users. We use peer-to-peer synchronization to exchange content directly between neighboring devices. In order to verify our proposal we have designed and implemented an innovative podcasting application in C++. The program uses UDP broadcast messages to discover devices. Among those in range, one peer is randomly selected and a TCP connection is created for data exchange. To speed up the synchronization process we use a bloom filter. Our program runs on HP iPAQ handhelds as well as on other platforms. We have tested and used the application under Windows 2000/XP, Mac OS X and Linux. The program is based on a clean design which makes it concise, easy to maintain and extensible.
Furthermore, we have conducted measurements to determine the actual performance of the proposed wireless podcasting solution. In order to examine the distribution speed of content we measured the time necessary to distribute the content from one device to all other devices in range. The results point out that the solution most fair to all devices is also the most efficient one. When distributing a large number of small files the overhead increases only slightly and remains at a low level. We can also show that the use of a bloom filter significantly speeds up the synchronization of a large number of channels. Finally, we have observed that the iPAQs used were not fast enough to fully utilize the capacity of WLAN.

7.2 Problems

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

7.2.1 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 but we were not able to integrate it into our program due to numerous compilation errors.
- Debugging gets very difficult if the program crashes at some point. In eMbedded Visual C++ development environment there is no way to display a useful 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 functionality 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. This function does not imply or enforce an actual scan for BSSIDs which often results in empty or outdated lists returned.
- There is still a race condition inside the TransferServer / TransferClient threads which sometimes leads to a lock up in the running synchronization on both devices. As there is only one concurrent connection allowed this usually leads to a complete lock-up of those devices until the end of the measurement. We have added a watchdog timer that checks the threads periodically if they are still exchanging data and resets them if they lock up. Upon resetting a thread a log entry is also created. This workaround avoids the condition even though it does not solve the original problem. This problem occurred in measurements with ten or more devices only.

7.2.2 Hardware

- The position of the devices can have a huge influence on packet loss. If a device is moved around a few centimeters the overall packet loss can increase up to 30% but stabilizes as soon as the device stops moving. 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 when the devices are close to each other.
- 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 to compare log files of one device with those of another device. We have added a simple timeserver functionality that helps to keep this problem as small as possible. As long as the timeserver itself is one of the iPAQs all devices will inherit the clock drift from this device.
- During excessive TCP data transfers it happened more than once that 2 devices totally locked up their TCP stack. They weren't able to send or receive anymore data although the rest of the operating system still worked without further problems. This forced us to reset the affected devices and repeat the measurement.

7.3 Outlook

Considering our results various suggestions for improvement emerge:

- Some of the measured parameters were limited by the hardware in the iPAQs and not by the capacity of the WLAN channel. Laptops are not affected by these limitations in speed. Measurements with a network consisting of laptops might show even more interesting effects.
- The use of a bloom filer for channel synchronization has been proven to be very effective. To speed up the synchronization process even more another bloom filter could be used for episode exchange.
- Although the use of TCP in data exchange simplifies the implementation a lot we could use UDP packets for synchronization. Additionally UDP packets could be used to broadcast the content in order to update more than one device at the same time.
- We have not conducted any dynamic measurements with moving nodes. It would be interesting to see how good our solution works in a mobile scenario.
- The implemented solution synchronizes content that was added by the user only. A problem here is that two devices can only exchange data if they both share a common channel. The application could be extended to cache popular channels that were not subscribed by the user. An intelligent caching strategy could improve the distribution of content.

Appendix A

Measurement Data

This chapter contains the graphs for all three runs of a measurement. Two different types of figures can be found in this chapter:

Download Time (upper row)

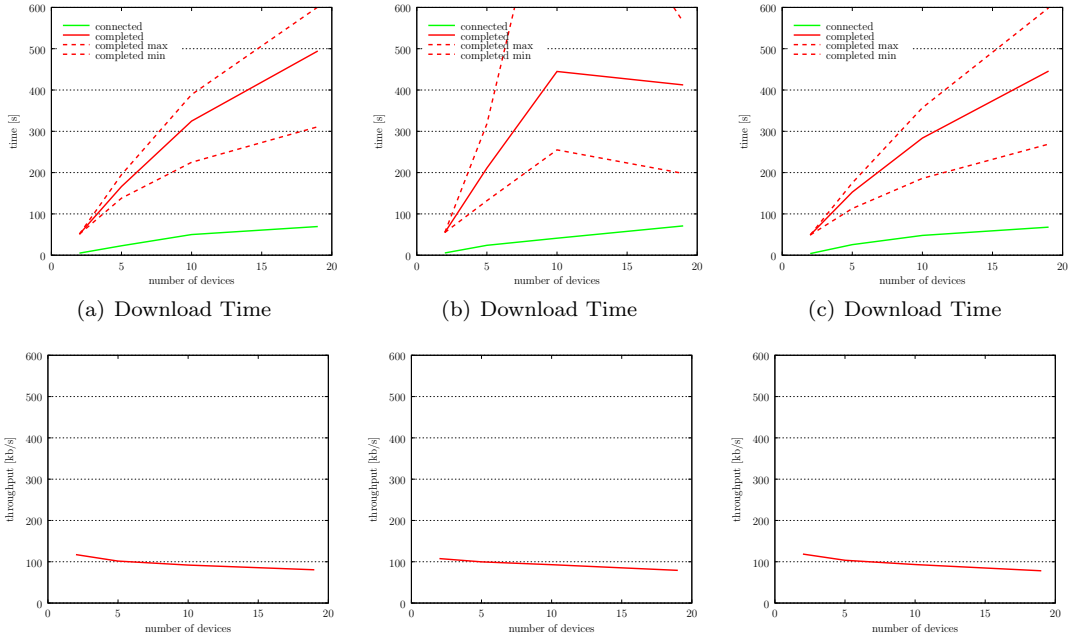
- The x-axis denotes the number of devices used in this run
- The y-axis shows the time in seconds since the start of the application

Legend	Description
connected	The time it took for a single device to successfully connect to another device for the first time (mean over all devices)
completed	Time it took to completely download the file (mean over all devices)
completed min	Time when the first device was able to download the file completely
completed max	Time when the last device was able to download the file completely

Transfer Speed (lower row)

- The x-axis denotes the number of devices used in this run
- The y-axis shows the average transfer speed of one device during a synchronization session in KB/s. The speed of 0 KB/s when a device is idle (i.e. has no connection to another device) is not taken into account as we are only interested in the actual throughput. To calculate the throughput we measure the time of a connection. We start the stopwatch just before opening the socket and stop it just after the socket was closed. The stopwatch returns the number of milliseconds between start and stop. Additionally we count the total number of bytes transferred (i.e. everything that we send over the socket including overhead such as meta information). To get the throughput we divide the total number of bytes transferred by the time measured.

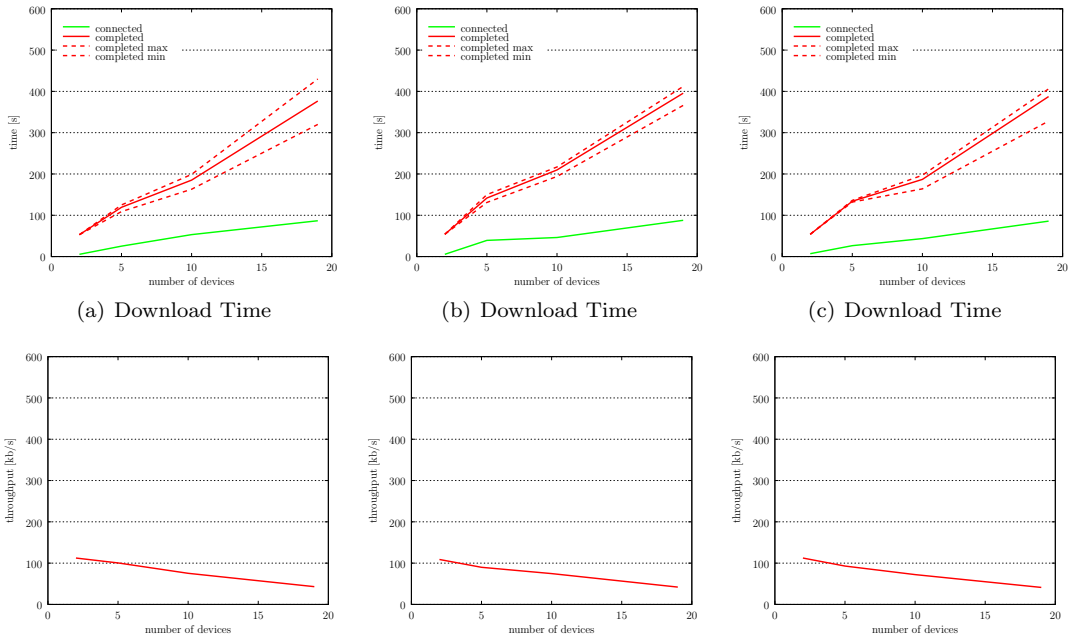
As visible in Fig. A.1(b) there was a problem with the 10-device measurement. Early versions of the program contained a bug that sometimes could lead to a stall in the synchronization. Depending on when this happened the download times could increase drastically.
Fig. A.3(c) shows a drop in the miminal download time. This was also due to a bug in the application where the download ratio was not always calculated correctly. In this case one of the devices obviously made a very good deal. As the serving device did not recognize that the download limit was hit the downloading device was able to finish its content synchronization very quickly.



(a) Download Time (b) Download Time (c) Download Time

(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

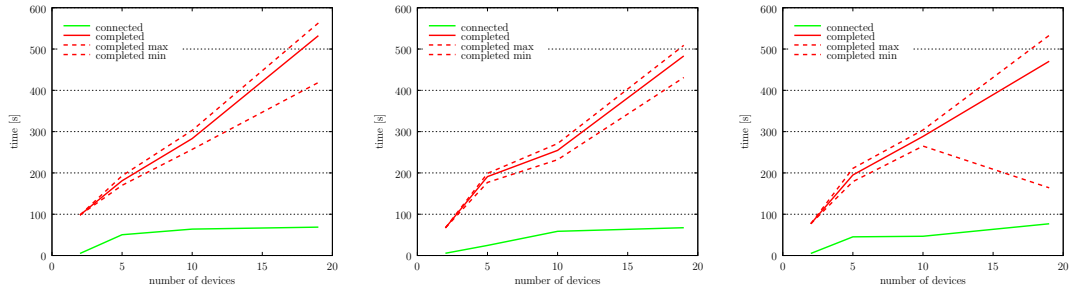
Figure A.1: Measurement 1: Single file, one-time synchronization, discovery interval 2 sec



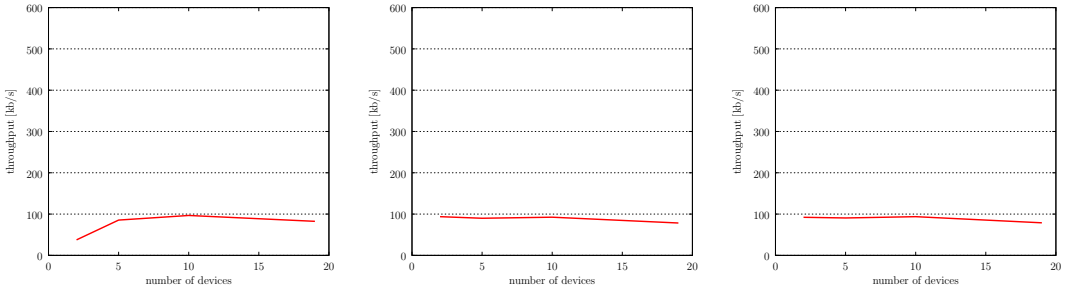
(a) Download Time (b) Download Time (c) Download Time

(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.2: Measurement 2: Single file, synchronization every 30 seconds, discovery interval 2 sec

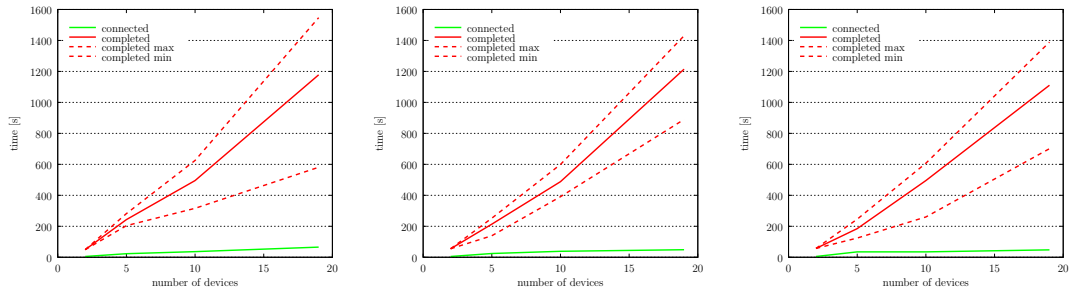


(a) Download Time (b) Download Time (c) Download Time

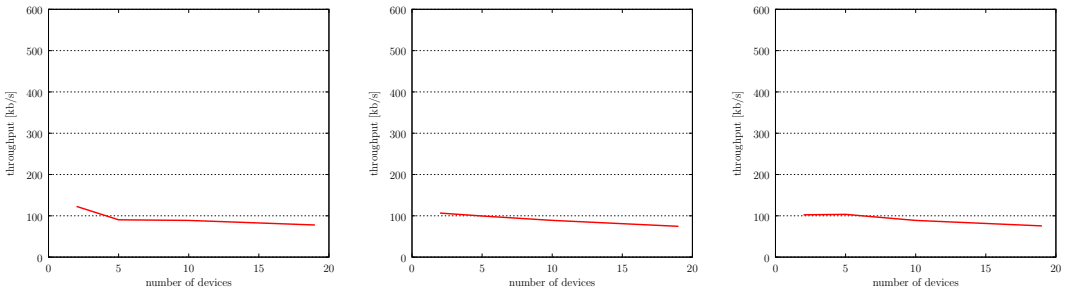


(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.3: Measurement 3: 100 small files, synchronization every 30 seconds, discovery interval 2 sec

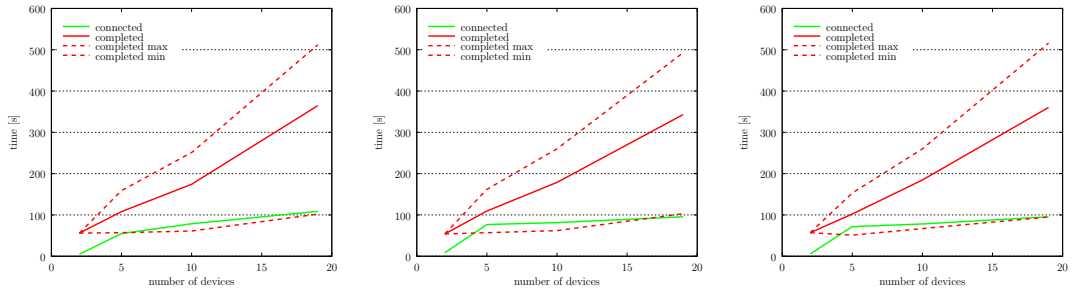


(a) Download Time (b) Download Time (c) Download Time

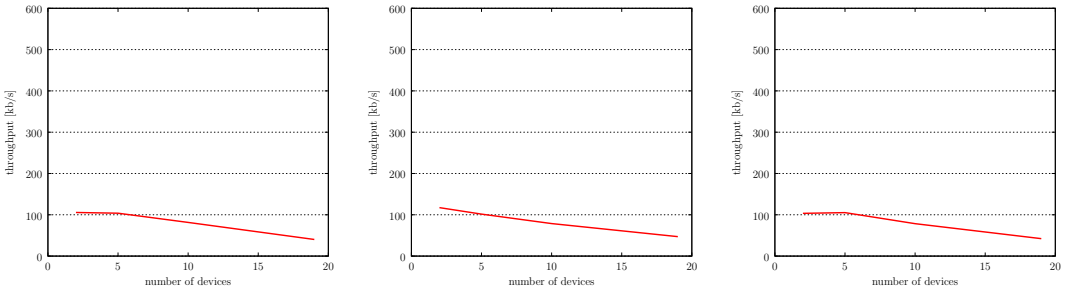


(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.4: Measurement 4: Single file approach, one-time synchronization, discovery interval 2 sec

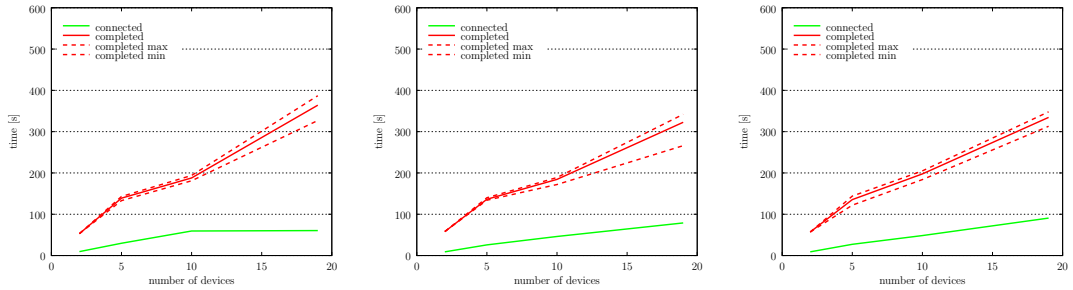


(a) Download Time (b) Download Time (c) Download Time

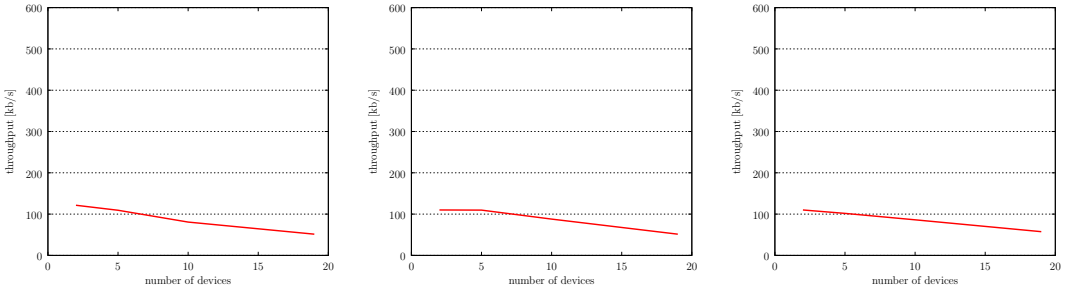


(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.5: Measurement 5: Single file, synchronization every 30 seconds, discovery interval 2 sec, no ratio

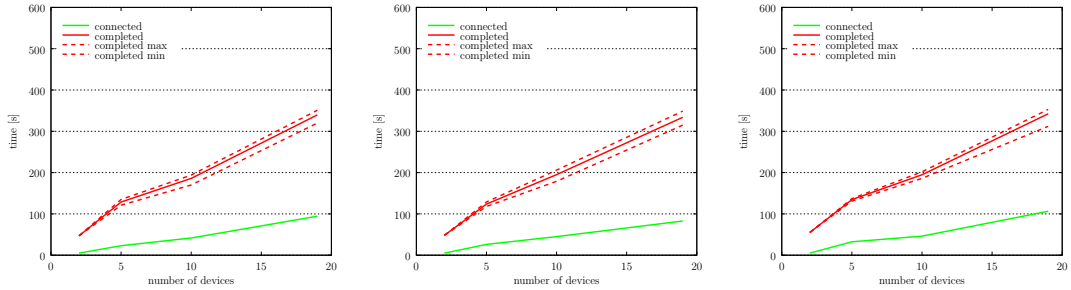


(a) Download Time (b) Download Time (c) Download Time

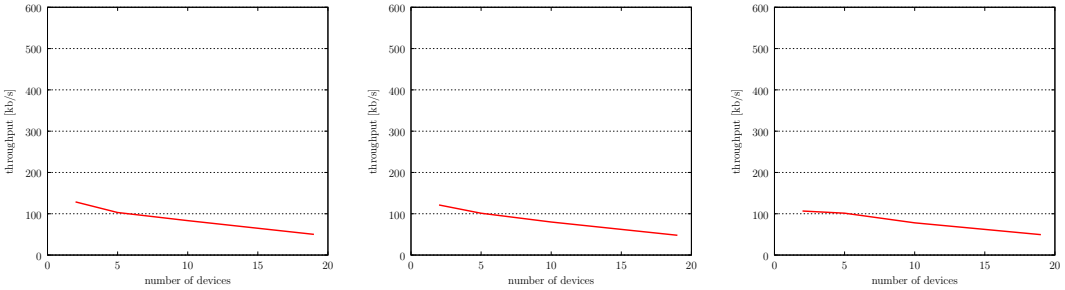


(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.6: Measurement 6: Single file, synchronization every 30 seconds, discovery interval 4 sec

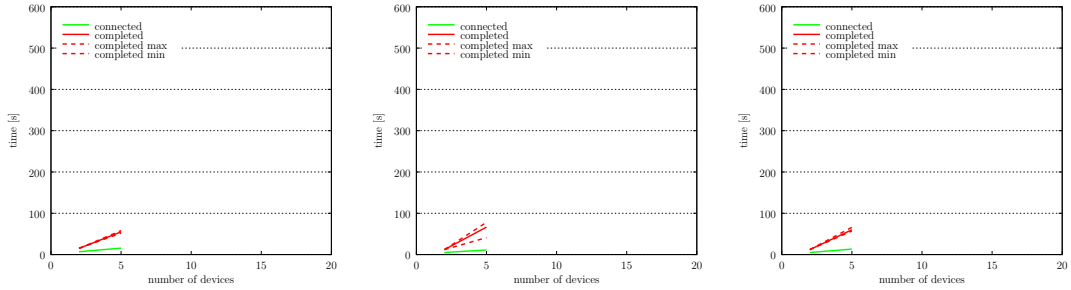


(a) Download Time (b) Download Time (c) Download Time

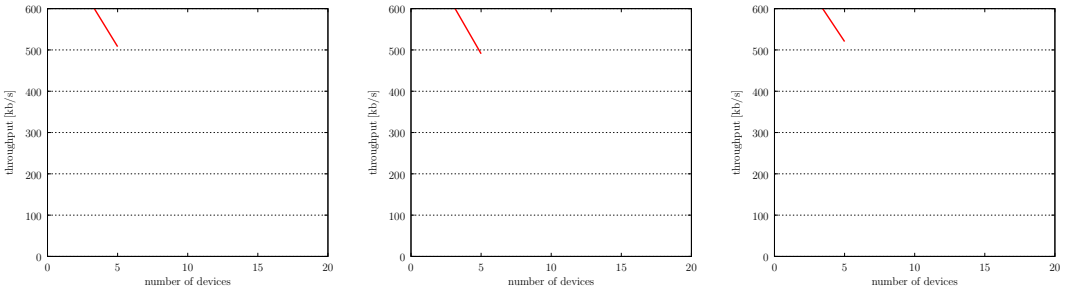


(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.7: Measurement 7: Single file, intelligent synchronization, discovery interval 2 sec

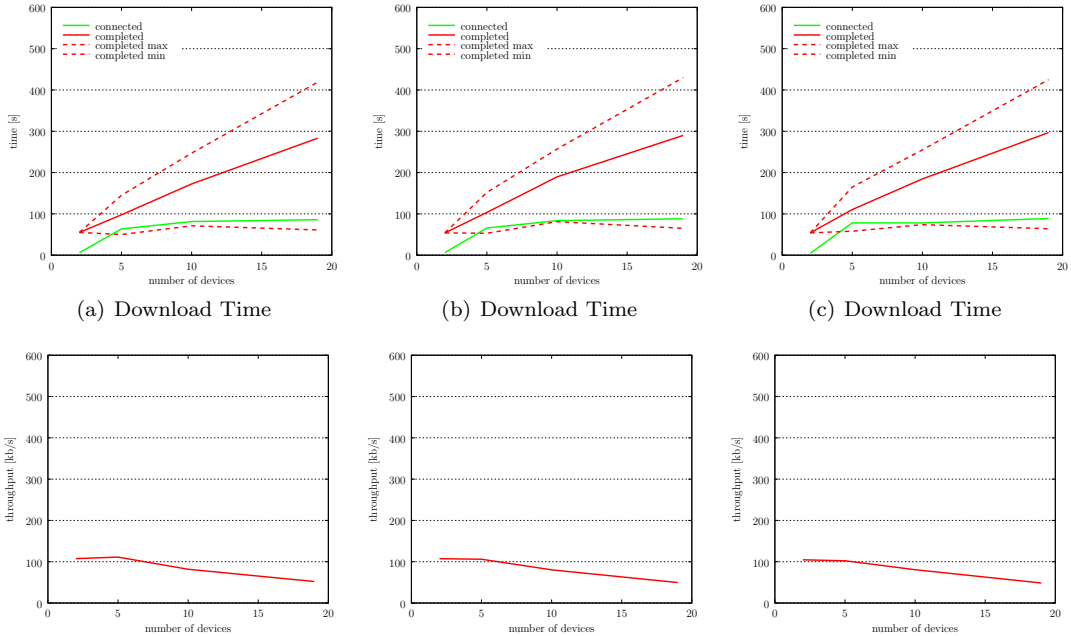


(a) Download Time (b) Download Time (c) Download Time



(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

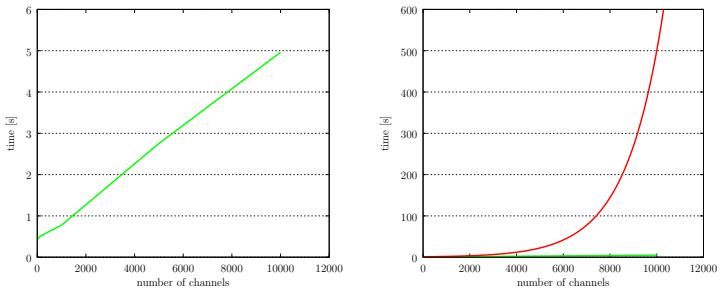
Figure A.8: Measurement 8: Single file with laptops, synchronization every 30 seconds, discovery interval 2 sec



(a) Download Time (b) Download Time (c) Download Time

(d) Transfer Speed (e) Transfer Speed (f) Transfer Speed

Figure A.9: Measurement 9: Single file, intelligent synchronization, discovery interval 2 sec, no ratio



(a) Download Time (b) Download Time

Figure A.10: Measurement 10: Bloom filter

Appendix B

Logfile Format and Analysis

The log files are saved in Comma Separated Value (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
```

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

Column	Content
ISO Time	Time when packet was logged.
Timestamp	Time when the packet was sent from the originating device. See PI_Timestamp() in App. D.2.2 for more information.
Source IP	The IP address of the device that created the packet.
Peer 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. 4.1.
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`
Event Nr	The event number that specifies the event logged. This number is unique for every event. See `sync.h` or `transfer.h` for more information.
Description	A human readable description of the event.
Remote IP	The IP address of the peer that is connected to a TransferServer or TransferClient.
Remote Port	The port of the connected peer.
Remote Status	The current device status of the remote peer (ready, choked, etc).
Remote PeerID	The peer_id of the connected device (currently disabled in `analyser.h`).
Total RX	The total number of bytes that was received during the last synchronization.
Total TX	The total number of bytes that was sent during the last synchronization.
Data RX	The total number of payload data received during the last synchronization.
Data TX	The total number of payload data sent during the last synchronization.
Uptime	The duration in seconds of the last synchronization.
Episode ID	The episode id of the added/deleted/... episode.
Channel ID	The channel id of the added/deleted/... channel.
New Content	The current new content date of this device.
Uptime (ms)	The duration in milliseconds of the last synchronization.

To change the log file format refer to the file `analyser.cpp`. The layout and the content can be changed in function `Analyser::BuildLogLine()`.
The files can be analysed using Microsoft Excel or if they are too big by the use of a perl script and/or grep et al.

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 DVD at `/ma_podcast/software/installation/`

1. Install ActiveSync (ActiveSync42_Setup.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*.

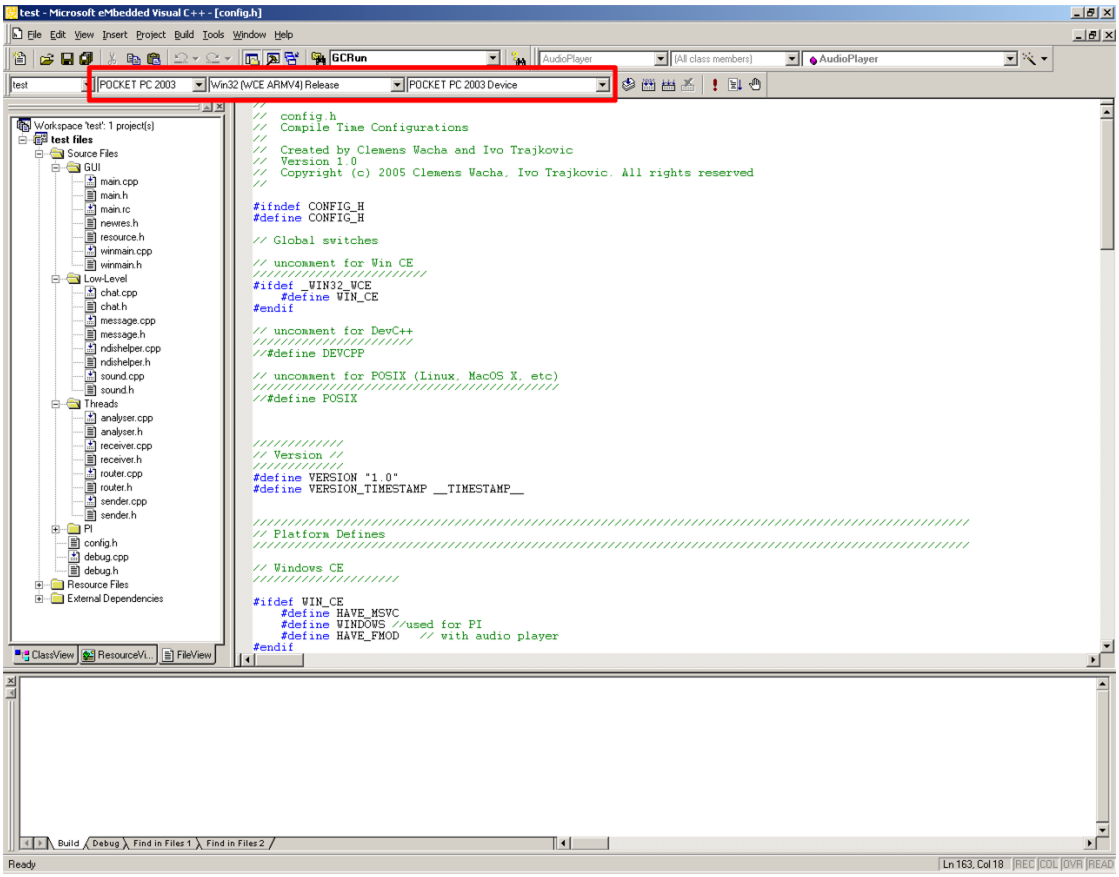


Figure C.1: eMbedded Visual C++ main view

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 /ma_podcast/software/wireless_podcast/api/wce4/armv4 copy the file fmodce.dll to the same folder on the device where you installed the wireless_podcast.exe otherwise you will encounter an error message:

```
Cannot find 'wireless_podcast' (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 Standard Template Library (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 DVD at /ma_podcast/software/tools/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 /ma_podcast/software/wireless_podcast (if not already done by the batch script).
2. Start the *File Explorer* application inside the emulator.
3. copy the wireless_podcast.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 wireless_podcast.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 wireless podcasting application also compiles on Windows using DevC++ [1]. Open the DevC++ project file located at `/ma_podcast/software/wireless_podcast/wireless_podcast.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. 4.10 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. 4.10 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.

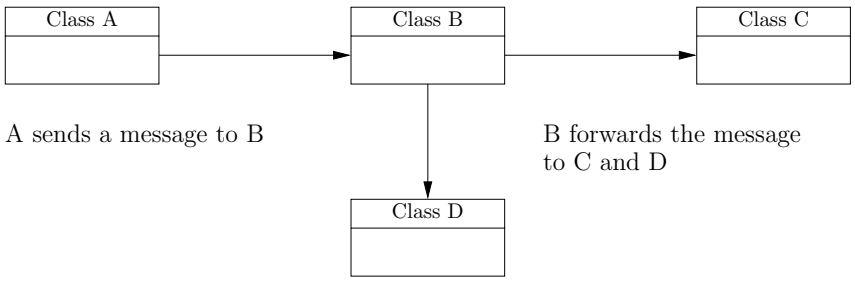


Figure D.1: Message flow

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. Unplug any cables attached to the device, so that it only runs on battery.
2. Hard-Reset the device 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
3. After the iPAQ has booted for the first time, complete the introduction and wait until the device has rebooted.
4. Use the iPAQ Backup application to restore ma_podcast-1_10_09.pbf (the file can be found in `/ma_podcast/software/iPAQ_backup`).
5. Optionally the internal device name can be set under *Settings->System->About->Device ID*. Set it to ma_podcast-XX, where XX represents the last two digits of the TIK-Inv. number. If this step is omitted ActiveSync will not be able to distinguish the devices.
6. 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 a device connects to the 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 want to use *Partnership Mode* 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. Remark that there is no time synchronization done in this mode. You can use the timeserver feature of the *Wireless Podcast* application to synchronize the time with all devices.

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: ma_podcast-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 = ma_podcast
- Owner->Name = podcast-00
- 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: GMT London, Dublin
- Settings->Power: Advanced: Deactivate auto poweroff

- Settings->Power: USB Charging activated (fast charge)
- 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>, Hold Button 1 to <Rotate>, Hold Button 3 to iPaq Wireless
- Settings->Buttons->Lock: Disable all buttons in standby.
- 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 assign in Settings->Buttons: Button 1 to <your application>

After re-installing the backup you have to change the name of the owner from podcast-00 to anything else. Also change the IP address to something other than 192.168.0.0.

Appendix F

Schedule

F.1 Milestones

Nr	Due	Description
1	Week 45	Application Design completed
2	Week 1	Software ready for deployment
3	Week 9	Measurements completed
4	Week 13	Analysis completed
5	Week 14	Documentation completed

F.2 Week Tasks

Week 42, 2006	(16. - 22. October)	Orientation
Week 43, 2006	Start of semester	Orientation
Week 44, 2006		Papers read Schedule created All administrative work done
Week 45, 2006		Application design, API enhancements
Week 46, 2006		Implementation Basic CE GUI tests Low level functions
Week 47, 2006		Implementation
Week 48, 2006		Implementation
Week 49, 2006		Implementation
Week 50, 2006		Implementation
Week 51, 2006	Mid-Term Presentation	Implementation
Week 52, 2006		**** Christmas Holidays ****
Week 1, 2007		Basic application testing
Week 2, 2007		Static Measurements / Analysis
Week 3, 2007		Static Measurements / Analysis
Week 4, 2007		Static Measurements / Analysis
Week 5, 2007	End of semester	Dynamic Measurements / Analysis
Week 6, 2007		Dynamic Measurements / Analysis
Week 7, 2007		Dynamic Measurements / Analysis
Week 8, 2007		Dynamic Measurements / Analysis
Week 9, 2007		Dynamic Measurements / Analysis
Week 10, 2007		Analysis / Documentation
Week 11, 2007		Analysis / Documentation
Week 12, 2007		Analysis / Documentation
Week 13, 2007		Analysis / Documentation
Week 14, 2007		Documentation
Week 15, 2007		Delivery of report
Week 16, 2007		Report deadline

Appendix G

Acronyms

AP Access Point

API Application Programming Interface

AODV Ad hoc On Demand Distance Vector

bps bits per second

BSSID Basic Service Set IDentifier

CDF Cumulative Density Function

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

HTTP Hyper Text Transfer Protocol

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

IRI Internationalized Resource Identifiers *(RFC 3987)*

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

OBEX OBject EXchange

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

URI Uniform Resource Identifier *(RFC 3986)*

URL Uniform Resource Locator

WLAN Wireless Local Area Network

Bibliography

[1] Nottingham M. and R. S. (Eds.), *The Atom Syndication Format*,
RFC 4287 Version 1.0, December 2005

[2] RSS Advisory Board, *Really Simple Syndication Specification*,
Version 2.0, August 2006,
http://www.rssboard.org/rss-specification

[3] BitTorrent, *Protocol Specification*,
December 2006,
http://www.bittorrent.org/protocol.html

[4] Duerst M., Suignard M., *Internationalized Resource Identifiers (IRIs)*,
IETF RFC 3987, January 2005

[5] Bloom B., *Space/time tradeoffs in hash coding with allowable errors*,
Communications of the ACM, 13(7):422–426, 1970

[6] Wikipedia, Bloom filter,
http://en.wikipedia.org/wiki/Bloom_filter.

[7] Wikipedia, Podcasting,
http://en.wikipedia.org/wiki/Podcast.

[8] Karlsson G., Lenders V., May M., *Delay-Tolerant Broadcasting*,
In Proceedings of the ACM SIGCOMM Workshops, Pisa, Italy, September 2006

[9] Trajkovic I., Wacha C., *Ad Hoc Communication with Handhelds*,
Semester Thesis TIK-2006-02,
TIK, ETH Zurich, February 2006

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

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

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

[13] Infrared Data Association, *(IrDA) Object Exchange Protocol*,
Version 1.3, January 2003

[14] IETF Zeroconf Working Group, *Zero Configuration Networking (Zeroconf)*,
http://www.zeroconf.org/

[15] Apple Bonjour, *Instant Networking and Dynamic Service Discovery*,
Technology Brief, April 2005
http://pdf.euro.apple.com/pdf/pn=BonjourTiger/MacOSX_Bonjour_TB.pdf
http://www.apple.com/de/macosx/features/bonjour/

[16] Cheshire S., Krochmal M., *Internet-Draft DNS-Based Service Discovery*,
August 2006
http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt
http://www.dns-sd.org/