PREDICTING END-TO-END NETWORK CHARACTERISTICS ON MOBILE DEVICES

Master Thesis by

Stephan Hegge

Committee:

dr.ir. I.A. Widya
ir. A. Peddemors (Telematica Instituut)
dr.ir. B.J.F. van Beijnum

ABSTRACT

In this report, we present a study in which we look into the possibilities of predicting end-to-end network connection characteristics on mobile handheld devices based on historical data. These characteristics include throughput, packet-loss and round trip time. We base these predictions on a relevant timeframe in the past and the interface they were recorded on. We also present a software framework that performs the measurements, the storage and the prediction of end-to-end network characteristics. This framework helped us researching the possibilities of predicting end-to-end network characteristics. We designed this framework to solve several issues encountered in the mobile environment.
We ran two experiments to research the possibilities of predicting network characteristics. The first experiment showed us that there are patterns of network characteristics in time. Using this collected data, we show in the second experiment that there are possibilities of predicting network characteristics.
We conclude that there is a possibility to predict end-to-end network characteristics based on historical data and that our framework is able to do this.

CONTENTS

ACKNOWLEDGEMENTS

With the completion of this Master assignment, I complete the final part of my education and will obtain my Master of Science degree. Without the exceptional support of my parents, this would not have been possible. Therefore, I would like to dedicate this report to them, to thank them for their support during my education.
Of course, this assignment would not have been possible without the supervision of Arjan Peddemors and Ing Widya, whom I would like to thank very much for their guidance during this Assignment.
For the duration of my assignment, I was an intern at Telematica Instituut. Without the possibilities Telematica Instituut offered me, I would not have been able to finish this assignment within eight months.
Finally, I would like to thank all my friends at Witbreuksweg 395-R for their support and friendship during my college days.

Enschede July 2007
Stephan Hegge

1. INTRODUCTION

In this day and age, digital battery powered mobile devices have seen a tremendous growth in usage. More and more applications are being designed and used for these devices. When these mobile devices first appeared, they only offered simple stand-alone services like a personal calendar or a personal contact list. Nowadays, these devices are also used in a networked environment (i.e. internet) for example in a mobile digital media platform (music, video, etc).
Gaining the processing power to provide complex services, new applications emerged like VOIP telephony next to telephony over the standard GSM network [1]. Other services included streaming video from internet servers; watching TV over the UMTS network, offer video conferencing, etc.

Figure 1: Mobile Handheld Device

With the coming of these complex internet services, the need for bandwidth increased. This need for more and different bandwidth demands over the air resulted in more wireless connection technologies becoming available. Some of these new technologies are Bluetooth, GPRS, EDGE, WCDMA and 802.11b/g. For versatility, manufacturers combined all these new technologies in mobile handheld devices. This allowed the devices to connect to the internet in several different ways.

Due to user mobility, these mobile devices often find themselves in different environments. Combining user mobility with the numerous internet connection facilities causes a frequent change in network connection quality.
Some internet applications designed for mobile devices rely heavily on the internet connection quality. The uncertainty of the internet connection quality used by mobile devices causes these applications to perform worse. Examples of applications that suffer performance hits due to this uncertainty are:

- Voice over IP software applications
- Data streaming software applications, like mp3 music players
- Applications that schedule use of network resources

These applications are able to adapt to networking conditions but uncertainty of the network quality makes the adaption process difficult. For example, a music streaming application could set its streaming bit-rate based on the given maximum interface speed (e.g. 11megabit/second for 802.11). Configuring applications on these values often causes erroneous behavior in practice. For example, it is possible that 11megabit/second is higher than the actual network supports. This causes buffer under runs in the music streaming application and thus gaps in the playback of music. These applications would perform better if they had a more accurate indication on the network conditions. In the example, the music streaming application could beforehand lower the streaming rate and provide gapless playback of music.

Thanks to the processing power of current mobile devices, it is possible to extend them with extra functionality. We show in this research that it is possible to get an up-front indication of the end-to-end network characteristics. These characteristics include Round trip time, throughput and packet-loss.
Because of frequent change in the network characteristics as experienced by mobile devices, we will investigate if the network characteristics, when observed in time, show repeating / periodic patterns. If this is the case then we can use these patterns to predict network characteristics for

roaming devices that use network connections in their daily activity pattern if the device networking usage shows a similar periodicity.

In this work, we investigate to which extent it is possible to provide up-front information about end-to-end network characteristics based on measurements done in the past. With this concept, it is possible to increase the performance of applications on mobile handheld devices that use network resources to provide their services. For example, applications could adapt to the network conditions before they make a connection to an end-host.

In [21] the authors show that network conditions are predictable. This research focuses on fixed networks, where our research focuses on mobile networks. In [3] the authors discuss software that automatically and dynamically selects a server using historical QoS statistics. The authors designed this software for fixed networks and limited it to remote server selection. We designed our solution to supply information only; it does not make any decisions on the network policies. Next to server selection, our solution can also be deployed for other purposes such as interface selection. In [4] the authors describe a software framework that automatically selects the path with the highest throughput for multi-homed mobile hosts. Again, our solution does not select a network interface (and thus network path) for an application, but gives information about end-to-end connection characteristics per interface available at a time. Applications can use this information themselves to select the most appropriate interface based on their service requirements. Furthermore, our solution provides network characteristics estimations based on historical data, it does not do active measurements on demand to determine the end-to-end path characteristics.

We present three contributions in this work. The first is the predictability of network characteristics as experienced by the mobile devices using observations from the past. Our second contribution is the framework to measure network characteristics on mobile devices. Finally, with some modifications, this framework provides software modules for applications to acquire estimates of up-front network characteristics using observed data from the past.

By observing network characteristics for two weeks, we first research and obtain the time-cyclic patterns in the network characteristics of a mobile device. Using these time-cyclic patterns of the network characteristics, we will research whether it is possible to predict network characteristics using these time-cyclic patterns.
To do our research on the prediction of network characteristics, we design and implement a software framework that is able to collect network characteristics and is able to provide predictions based on the collected characteristics.

Chapter two of this report describes issues that applications encounter in the mobile environment. We use these issues to motivate our research. This chapter also describes in more detail how other authors solved these mobile environment issues and how our solution solves these. The third chapter describes the concepts we had to research in order to be able to measure and predict end-to-end network characteristics. Chapter four shows the implementation of the software framework. The fifth chapter describes the experiments we did to research the predictability of network characteristics. Chapter 6 contains the results of the experiments. In chapter seven we present our conclusions.

The Dutch Freeband Communication Research Programme (Awareness project) supported this research under contract BSIK 03025.

2. MOTIVATION

Most software applications using network resources on mobile devices do not have an accurate network quality indication. Since they make use of best effort internet services, they provide their services in a best effort manner. Because this is by no means any indication of the quality of the network, providing best effort services often produces problems like buffer under runs for music streaming applications or poor voice quality in VoIP applications.
To motivate our research, we will present three important issues encountered in the mobile environment that we solve with our solution. After these examples, we will describe how other authors solve some of the issues that are at hand in these examples. We will wrap up by explaining how our solution solves these issues.

2.1. MOBILE ENVIRONMENT ISSUES

Interface selection

Devices often have multiple possibilities to connect to the internet. By being able to connect to the internet through different interfaces, devices have the ability to manipulate the network path to an end-host. If the device has multiple wireless interfaces enabled, applications often select the interface with the best first-hop network characteristics. Because the applications do not have an overview of the remaining part of the end-to-end connection, this selection does not necessarily have to be the most fitting option for the requirements of the application. Another interface that performs less on the first hop (higher delay or lower throughput), may actually be more suited in certain situations. We will explain one of these situations in the following example.

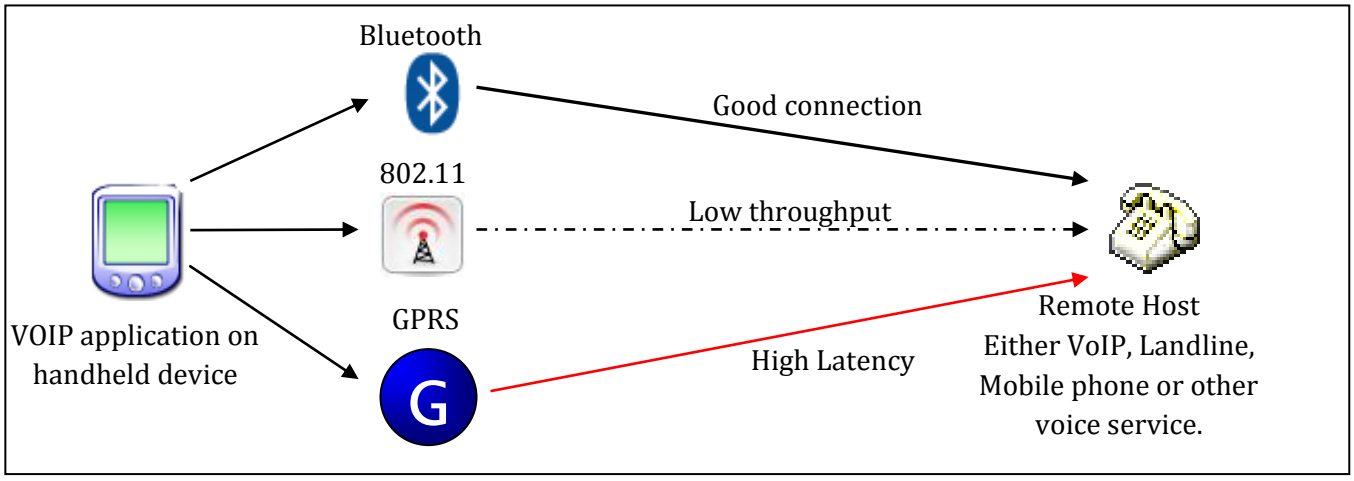


Figure 2: Voip and Interface Selection

In figure 2, we have depicted a situation where a VoIP application with video calling capabilities can choose between three different interfaces (Bluetooth, 802.11 or GPRS) to connect to the remote host. The VoIP application does not have any indication of the quality of the network connection other then the first hop characteristics, which the device provides in the form of signal strength and maximum interface speed. Consider the interface selection process of high quality video streaming application; it might go as follows:
Based on the first-hop characteristics, the VoIP application will first use the 802.11 interface to connect to the end-host. Suppose that the 802.11 access point connects to the internet through a cheap internet subscription like home ADSL or ISDN. As a result, this internet connection is of

very poor quality. Realizing after a while that this internet connection does not suffice for a high quality video stream, the application will try to connect using another network interface. GPRS, in general, has an unacceptable delay (>500 ms). After trying, the application will determine that this interface is also not suitable for a high quality video stream. Using the Bluetooth internet connection, the application determines that it provides suitable network conditions (low delay, acceptable throughput, low jitter).
This interface selection takes a very long time; especially when an application has to turn on all these different interfaces, setup a connection and wait on time out etcetera. Even worse is the moment the characteristics of an interface change drastically, this could result in very low speech quality or even worse: connection loss.
If more interfaces are available at a certain time and an application knows information about the end-to-end path for each of different the interfaces, then an application could immediately select the correct interface. This would save much time in connection setup.

Adaption to network conditions

Most network applications are already able to adapt to different network conditions. However, the network applications adapt during use of their services. For example, VoIP applications could use a lower voice quality when the measured connection throughput is low. If a network application had information about the network conditions before it connects to a remote host, it could adapt before using the end-to-end connection.

Predicted Throughput on different times using UMTS	MP3 Streaming bitrate
1 Mbit/s	192 kbit/s – 320 kbit/s
200 kbit/s	128 kbit/s – 160 kbit/s
120 kbit/s	32kbit/s – 96kbit/s

Table 1: Predictions and bitrate

Suppose a music streaming application that streams music over an UMTS internet connection. Since the application does not have any indication of the underlying network conditions, it does not know the optimal streaming bit rate. If the streaming rate were set too high, then it would experience buffer under-runs. If it were set too low, then it would not use the connection to its full potential. If the streaming application would have an indication of the throughput rate as presented in Table 1, then the streaming application could select the corresponding throughput rate as seen in Table 1.

Server Selection

Popular files on the internet can often be found at more than one download location. This concept is called mirroring. The download locations are often spread geographically around the globe. The purpose of offering popular files at multiple locations is to distribute server load and allow better (i.e. faster) access for users from all parts of the world. When a user wants to download a popular file, the user often intuitively selects the physically closest server; this server has the highest chance to provide the fastest download rate. However, since the current internet topology is built as a star network and not as a mesh network, it is possible that due to routing a server at a remote physical location performs better than one at a physical close location.
If a downloading application has information of network characteristics for each different server, then it could carefully select that host which provides the best characteristics.

2.2. RELATED WORK

In [4] the authors propose "a new wireless access paradigm for multi-homed hosts based on a session layer bandwidth aggregation mechanism". They realize this by implementing a thin module called "Session layer Bandwidth Aggregation (SEBAG)". This module is able to combine the bandwidth of different interfaces, achieving a higher throughput. Using a packet-scheduling scheme called "Expected Earliest Delivery Path First" (EEDPF), the module distributes the data load among the different wireless network interfaces based on interface speed.
The implemented module operates at both client side and server side. It operates between application and transport layer, making it transparent to applications. When the client model detects multiple internet connections, it uses all of the available connections to connect to the remote host.
With SEBAG, the authors solve the "interface selection" issue we stated. This achieved by sending data over all interfaces and using the EEDPF packet-scheduling scheme. However, since SEBAG only improves throughput by using all available interfaces to connect to the end-host, there is no feedback on other network conditions to applications. Therefore, SEBAG cannot be used to address the "server selection" and "adaption to network conditions" without changing the end-hosts. Another issue is that SEBAG requires a server module in order to operate. That means that SEBAG only works for those end-hosts that have the server module installed.

The authors of [3] describe a mechanism that selects servers based on shortest download time. This mechanism uses past QoS data according to the day of the week and the time period to select the fastest mirror. The system that the authors have implemented acts as a proxy server and is transparent to the clients behind it. After the system obtains a mirrors list for a specific file, it measures QoS data by downloading a small amount of bytes from each of the mirror servers and calculating the throughput rate of the link. The system then stores this throughput information in a database, and uses it for server selection.
The mechanism that the authors provide is able to solve the issue of "server selection". Because there is no feedback to applications, this work does not offer a solution to the "Adaption to network conditions" and "interface selection" issues. The prediction of network characteristics in our work was inspired on this research, since we used a similar method to predict network characteristics.

The research in [5] describes a system that actively selects an interface during usage of a specific end-to-end connection. Using the Stream Control Transmission Protocol (SCTP), this system sets up a multiplexed connection to the remote host through all local interfaces. SCTP works much like TCP, but it supports multi-homing and path selection. For more information about SCTP and how the authors modified SCTP, we refer to [5]. By performing active measurements on all the different active paths from the local host, the system actively selects that network path which provides the lowest round trip time in combination with the highest bottleneck bandwidth. The network path is selected by using the different network interfaces available in multi-homed hosts. When network characteristics change, the system can easily switch to another connection that provides the best network conditions.
Just like in [4], this system only solves the "interface selection" issue, but it does not let application choose which interface to use. Although it is a rather lightweight solution, it is difficult to use it on large scale. Because SCTP is not compatible with TCP, this system can only be used with end-hosts that support SCTP.

In [21] the authors do research on TCP Throughput prediction. Their research in many ways resembles ours; the authors created a tool that is able to provide TCP throughput predictions based on past measurements. Using Support Vector Regression (a machine learning technique), the authors created a system that is able to accurately predict network throughput. Using a tool called *PathPerf* the authors verified that in the best case, 90% of the predictions made using this framework were within 10% of the actual value. In the worst case, 35% of the predictions were within 10% of the actual value.
There are some vital points where our research differs from this. First, our research focuses on the mobile domain. The tool the authors created of [21] could perhaps also be deployed in the mobile environment. At the time at time of writing this report however, the tool was not yet available. Therefore, we were not able to verify whether this was possible.
Our framework is also able to provide characteristics estimations to applications. The authors of [21] do not clearly point out if there is a feedback mechanism in their tool.
Finally, our framework supports predictions of multiple different network characteristics like Packet loss, round trip time and throughput.

2.3. OUR SOLUTION

We solve the different issues described in Section 2.1 by providing predictions of the network characteristics. We base these predictions on network characteristics as the device experienced them in the past. We say experienced, since the device only uses characteristics that it can collect itself (figure 3). These characteristics include throughput, round trip time and packet loss.
Our solution solves "Interface selection" by having separate predictions available per network interface. This makes it possible for applications to select the interface that best matches their connection requirements. We solve "Server selection" since our solution is able to provide up-front estimations on a per host base. Lastly, our solution solves "Adaption to network conditions" by providing up-front estimations of network characteristics, prior before applications set up an end-to-end connection.

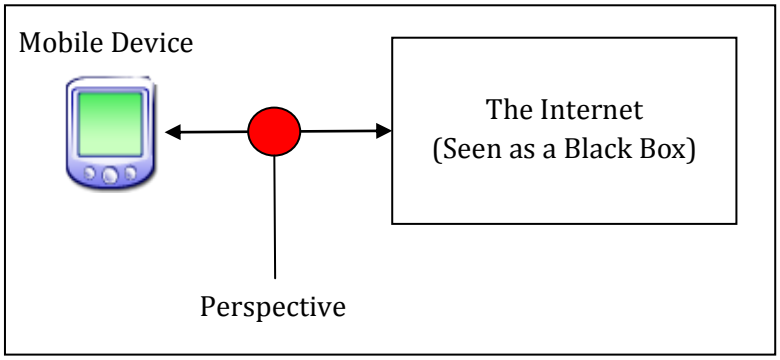


Figure 3: Device Perspective

For our solution to work however, we need historic network characteristics. When there are no past measurements available, our framework is unable to provide accurate estimations of the network conditions. Before this is possible, our framework needs information about network characteristics from the past, generated by an application that generates internet traffic. As soon as internet traffic comes available, our framework uses this information to estimate network conditions for a new end-to-end connection with the same conditions (time, end-host etc.). Since our framework works with averages over the past, it gets more accurate as more data is collected.
Another issue of our solution is when the network environment as seen by the mobile device deviates from the environment observed in the past. In our current solution, this would produce inconsistencies in the database, and as a result, inaccuracies in the network characteristics predictions.

Our framework does measuring passively. This means that is uses internet traffic generated by other applications to measure network characteristics. It does not generate internet traffic on its own. Therefore, either a user on a regular base or an automated process needs to generate internet traffic to obtain historical network characteristics.

3. MEASURING END-TO-END CHARACTERISTICS

In order to validate the possibility to predict network characteristics based on past measurements, we created a framework that helped us proving these concepts. Our software is able to measure network characteristics of end-to-end connections as seen by the mobile device during the use of an end-to-end connection. It can record these characteristics and provide predictions based on these recorded network characteristics.
Before we were able to create this framework, we had to research some basic concepts on measuring end-to-end network characteristics that we discuss in this chapter.
First, to collect network characteristics, it is important to define which characteristics are important and which are not. For example, to some applications the number of hops to an end-host is important, where for other applications only throughput is important.
After we have defined important characteristics, we will describe how these characteristics are collected. We will discuss measuring methods and the methods we use to derive network characteristics from raw network traffic.
Next, we will discuss how our software stores network characteristics and how it uses the stored characteristics for prediction of network characteristics.
Finally, we will look into possible solutions for the initial state problem (i.e. when no prediction can be made because there is no historical data available).

3.1. RELEVANT CHARACTERISTICS

To determine relevant characteristics to record, we looked at the parameters used in Quality of Service monitoring. Characteristics like throughput, round trip delay, packet loss and jitter are among the most basic QoS parameters that are meaningful for most IP-based services [2]. To provide applications with basic information about end-to-end connection conditions, we should include these characteristics if possible. There is however, a vast list available of other relevant characteristics to measure as well, which could account for good or bad end-to-end network connection conditions in a particular situation. For example, the maximum transmission unit size for an interface or the number of hops passed to reach an end-host. Although both these examples help determine the end-to-end connection condition, change in these two affect the four basic characteristics described above.
We will not focus on more than these four basic characteristics. We made it possible to extend the system to measure more characteristics and let the framework provide these characteristics to applications that want this information. To provide this functionality, we added a plug-in system, which allows programmers to extend the software with their own measuring methods.

3.2. COLLECTING DATA AND DERIVING CHARACTERISTICS

There are many methods to measure end-to-end network characteristics. Most methods are either active or passive. For active measurements, an application generates traffic to measure network characteristics. For example, it is possible to run an application that actively measures throughput by downloading data from an end-host or one could use the Ping tool to measure round trip time to an end-host. Passive measurement methods use network traffic of other applications to measure network characteristics.
Our preference lies with passive measurements. First, passive measurements do not influence the network load, since no extra traffic is generated to determine network characteristics. Secondly, using passive measurements, we measure the traffic that the user generates. This

makes sure that our software also considers user behavior patterns and can use them for prediction of network characteristics.

To be able to measure characteristics of all the end-to-end connections in use, the system has to be completely transparent to applications. Combining this with passive measurement techniques, we can choose between two different methods to achieve this.

Socket Masking
It is possible to design a measuring mechanism on top of the Transport layer, which manages end-to-end host connections. This technique is called 'Socket Masking' and is described in [13]. In this technique, a programmer replaces the Socket library as provided by the Operating System. By implementing the same functions as the default Socket library, applications do not see the difference between the original socket library and the modified one. The modified one contains code to monitor network parameters. During the usage of a TCP connection, the measuring mechanism can then perform measurements of several QoS parameters as described in the previous section.
Using socket masking also has some disadvantages. For example, in some cases, it is not easy to replace the current socket implementation on systems. On the Windows CE platform for example, the Socket library is fixed in the ROM area of the device. It is therefore difficult on this platform to change the Socket implementation.

Packet Logging
Another method to measure network characteristics is to run a service, which gathers network data at the bottom of the stack (as low as possible). This service records all the raw data packets that pass through this low layer. Another application can then use these raw data packets and derive network characteristics from this data. For example, a good way of measuring round trip time is to measure the delay between a TCP SYN and a TCP SYN-ACK packet.
One advantage of logging data and making statements based on this low-level data is that software can easily (without using many resources) take out the relevant data from a log file and store it. A disadvantage is the fact that if the logging is constantly enabled the log files might grow very large (mainly due to unimportant traffic), so a throttling mechanism should be introduced in order to keep the use of disk space at minimum.
In our software, we chose for this method to gather network characteristics. Using this method to gather characteristics brings forth the need for methods to derive network characteristics from raw packet traces. We will discuss these methods in the following three sections.

3.2.1. Deriving Round Trip Time (RTT)

TCP SYN and SYN-ACK packets

In this research [14], the authors describe a method they use to derive round trip time from a packet trace. They calculate the difference of time between sending a TCP SYN packet and receiving a TCP SYN ACK packet. We explain the process in figure 3: the sender sends out an SYN packet, when the receiver receives this packet, it sends back the SYN ACK packet. When ignoring the processing delay of receiving and sending TCP packets, we can calculate the round trip time by calculating the difference in time between sending the SYN packet and receiving the SYN ACK packet. An issue with this method appears

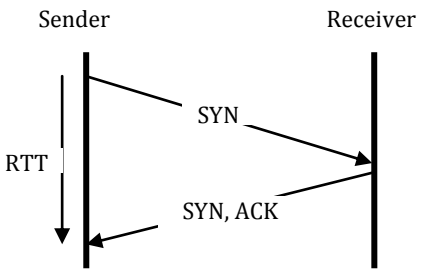
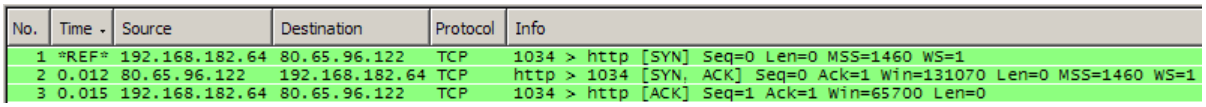


Figure 4: RTT estimation

when an intermediary host in the network path intercepts these packets (e.g., catching the SYN packet and sending a SYN-ACK packet back before the end host has responded with SYN-ACK). The RTT would then show an incorrect view since it only calculated using a part of the network path instead of the whole network path.



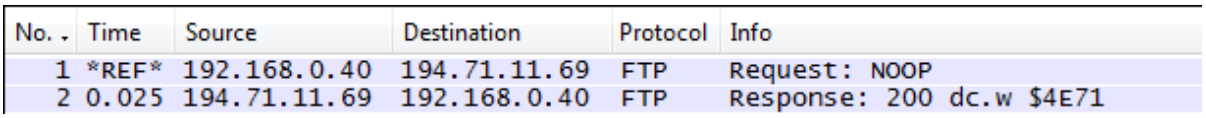
No.	Time	Source	Destination	Protocol	Info
1	*REF*	192.168.182.64	80.65.96.122	TCP	1034 > http [SYN] Seq=0 Len=0 MSS=1460 WS=1
2	0.012	80.65.96.122	192.168.182.64	TCP	http > 1034 [SYN, ACK] Seq=0 Ack=1 Win=131070 Len=0 MSS=1460 WS=1
3	0.015	192.168.182.64	80.65.96.122	TCP	1034 > http [ACK] Seq=1 Ack=1 Win=65700 Len=0

Figure 5: RTT Estimation using SYN-SYN_ACK packets

In figure 5, we see an example of a RTT estimation using SYN-SYN_ACK packets. The sending host (in this case 192.168.182.64) sends out a SYN packet to destination 80.65.96.122. In the case of Figure 5, the measured delay is 12 milliseconds.

Application Layer Protocol

Another way to derive round trip time using TCP packet traces is to make use of one of the application layer protocols like HTTP or FTP. Consider the official FTP protocol specification described in [20]. The NOOP command stands for "no operation" and does the following: *"NOOP (NOOP): This command does not affect any parameters or previously entered commands. It specifies no action other than that the server send an OK reply."*
This gives us the opportunity to use this command and others to get an indication on Round Trip Time. It works according to the same principles used in deriving RTT using TCP SYN and SYN-ACK packets. To calculate the round trip time, we take the difference between the time the command is sent and the time the response is received. Although we get an actually less accurate indication due to extra overhead, this method is less dependent on the lower protocol layers. Figure 6 gives an example of RTT calculated using FTP.



No.	Time	Source	Destination	Protocol	Info
1	*REF*	192.168.0.40	194.71.11.69	FTP	Request: NOOP
2	0.025	194.71.11.69	192.168.0.40	FTP	Response: 200 dc.w $4E71

Figure 6: RTT Estimation using FTP

3.2.2. Deriving Packet Loss

In [15] the authors describe two methods to derive packet loss percentage from a TCP packet trace. They describe a "Naïve" and an advanced method. The simple method uses TCP sequence and TCP Acknowledgement numbers. Whenever a TCP Acknowledgement number does not match the correct TCP sequence number, packet loss has occurred. If we correct for retransmissions, then we get a good indication of the packet-loss percentage. The downside of determining packet loss based on TCP packets is that actual loss might be much higher. This is because in wireless links much of the correction facilities are already done on a lower level. However, this loss will be shown in round trip time or delay jitter figures.

3.2.3. Deriving Throughput

In order to be able to derive throughput, we first have to define throughput. We define throughput as the number of bytes transferred between two endpoints within a certain period. This includes TCP retransmissions, to make up for lost TCP packets that the software does not see in the packet trace.

To explain our method of deriving throughput out of a packet trace, we define two concepts to help us determine the throughput. The first one is the flow. This is a period of time during which a host transfers a number of bytes to the other host. For example, the time an FTP server transfers data bytes of a file to another FTP client. The flow consists of TCP packets. We define the packets as our second concept.
To derive throughput, we start at a packet which payload is larger than a certain threshold (to cope with ACK streams and underutilized connections). We call this threshold the packet-size threshold. This packet marks the start of the flow. To determine all the packets that are part of the flow, each packet after the starting packet has to satisfy two conditions:

- Each consecutive packet has to originate from the same end-host as the starting packet.
- TCP packet payload size has to be larger than the packet-size threshold.

To determine the end of the flow, we check for disconnections (TCP packets with either Reset or Finish flag set) or we check whether the inter-packet time is longer than a certain threshold (we call this threshold the time threshold).

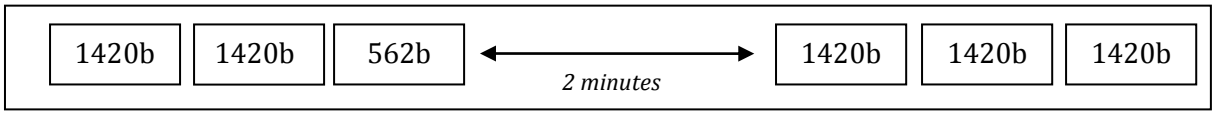


Figure 7: Inter Packet Gap

Figure 7 explains the situation. If we calculate throughput on this whole stream of packets, then we would get a wrong indication of the throughput speed, it would be much lower. To be able to control the accuracy of the throughput, we check whether the number of TCP packets within a flow exceeds a predefined value (packet threshold).
To calculate the throughput, we sum up the number of payload bytes of each separate TCP packet, calculate the duration of the flow and then calculate the number of bytes transferred per second.
Our method has some drawbacks. First, it is possible to have multiple TCP streams sending at full possible data-rate; this will negatively influence the measured throughput of one stream. In addition, a relatively large number of bytes have to be transmitted in order to get an accurate measurement of throughput.
When measuring TCP throughput, one has to deal with TCP slow start. If all packets in a flow are used, TCP slowstart will give an inaccurate view of the average throughput. With fast and steady links, TCP slow start often does not take very long. However, on links where the delay is high (and throughput is high too, like UMTS) it takes rather long before the maximum throughput rate is attained. In Figure 8 on the next page we see a throughput plot created by Wireshark [16]. We recorded the packet trace of a UMTS connection to a web server in germany. This picture shows the slow start on the connection. As can be seen on the picture, it takes rather long (1,5 seconds) before the average throughput rate is achieved.
In order to get a more accurate throughput indication, we have decided to eliminate slow start from our measurements. To reduce the effects of slowstart, we ignore a predefined number of packets at the start of a flow (slowstart threshold).

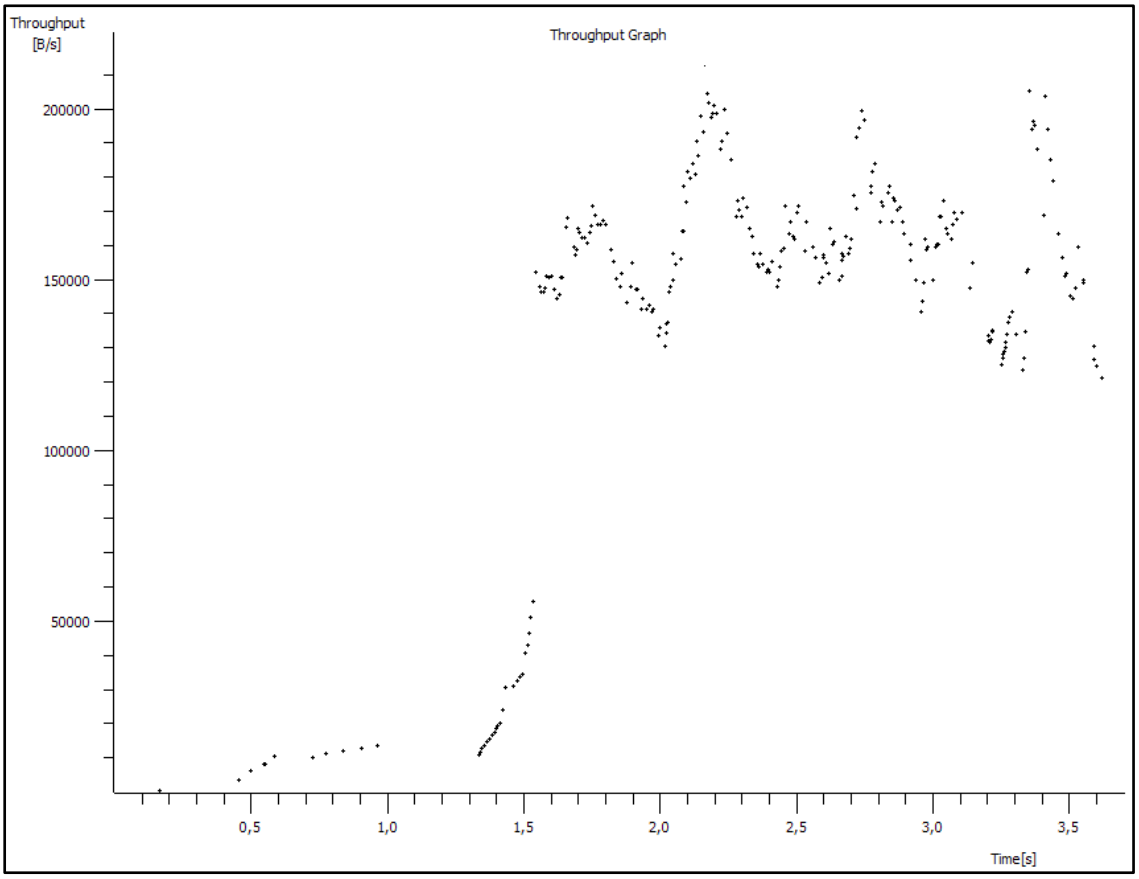


Figure 8: TCP Slowstart

Using extensive testing on several packet traces, we determined the values for these thresholds. We will now discuss the threshold values we chose for deriving the throughput.

Packet -size Threshold
Initially we chose for a value of 1420 bytes. This is the MTU (maximum transmission unit) of the 802.11 interface. However, during test of this value, we found that sometimes the MTU size of UMTS and GPRS traffic is below this threshold. Therefore, we chose a value of 100 bytes; this covered all the packet traces we tested.

Time Threshold
As a maximum inter packet gap value, we chose 5 seconds. This might seem very short, especially since TCP timeout is on 2 minutes. However, to get a valid indication of the maximum throughput, we have decided to set this on a very short period. Using several packet traces consisting of GPRS, UMTS and 802.11 traffic, we thoroughly tested this value in order to verify whether this was not to short.

Slowstart Threshold
The best method to determine this value is to actually look at the packet trace itself, and assign a value dynamically. However, since we did not have the time to program a method that dynamically assigned this value; we chose an easy solution by testing several values: 0 (for reference, 50 and 150. We checked how this value influenced the derived characteristics.
The reported throughput for the 0 packet ignore was 113961,8 kbytes/s. For 50 and 100 118355,4 kbytes/s and 116930,3 kbytes/s were reported. Although in this particular graph (figure 8) we see that the slowstart is over within the first 50 packets, we decided to ignore the

first 100 packets of a stream. We chose to do this since it slowstart heavily depends on delay variation.

Packet Threshold
We chose this value based on the number of packets we discard due to slowstart. We determined that we at least needed 100 packets for an accurate view on the throughput. Therefore, we set this value to 200 as we discard 100 packets due to slowstart.

3.3. Storing and providing Characteristics

We want to provide a prediction of network characteristics based on relevant past measurements. To do this, we need a way to combine relevant past measurements together. In [3], the authors describe a system that achieves this. As soon as a measurement sample comes available, the system calculates the average over the values corresponding to the same day of the week and the time period.
Our system works in a similar way, but because mobile handheld devices have limited storage space and processing power, we needed a method that uses less storage space. We are using the following method: We take a reoccurring period, for example one week. We split up the week in time slots. If the time period of a slot is set to one hour, we have a total number of 24 * 7 = 168 time slots within one week. If we take a reoccurring period of one month, there would be a total number of 24 * 28 = 672 time slots. In each time slot, information of the network characteristics at that time are stored, along with the end host and the interface used. Each time a new sample of the same class, interface and end host comes available, the average in the corresponding time slot is updated with this new value. Using this method, we do not have to store all the measured characteristics, thus saving up storage space.
Because at some points in time, measurement samples are more variable then other times, the fixed time period method can reduce accuracy when needed. To solve this, we could use of variable time periods. The time period is then increased where accuracy is not needed (e.g. during night times) and can be decreased for better accuracy. This concept is explained in figure 9.

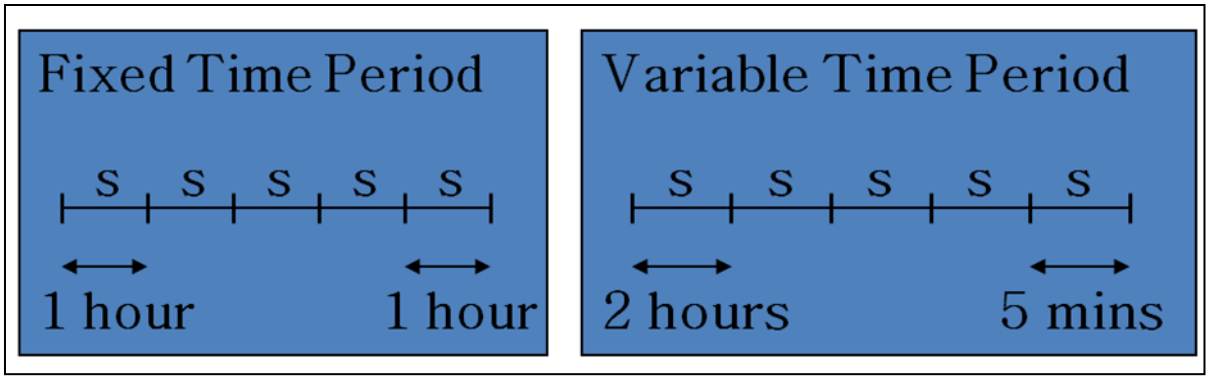


Figure 9: Fixed Time period and Variable Time Period

When we want to make a prediction of the network characteristics, we take the corresponding time slot and provide the network characteristics stored therein. To give an indication of the precision of the prediction, we also store a correction value along with the average of a measurement. This correction value is comparable with the standard deviation, but it is not the same. The difference lies in the calculation method. Where normally the standard deviation is calculated over the whole data set, this formula 'updates' the recorded deviation value with the new value. Using this formula, we do not need the complete history of average values in order to

get an indication of the standard deviation. The algorithm we use to calculate this correction value is described in detail in [18] and is based on the following formulae:

$$m_{\mathrm{new}} = \frac{n\, m_{\mathrm{old}} + x_{\mathrm{new}}}{n+1} = m_{\mathrm{old}} + \frac{x_{\mathrm{new}} - m_{\mathrm{old}}}{n+1}$$

$$s^2_{\mathrm{n-1,new}} = \frac{(n-1)\, s^2_{\mathrm{n-1,old}} + (x_{\mathrm{new}} - m_{\mathrm{new}})\,(x_{\mathrm{new}} - m_{\mathrm{old}})}{n} \qquad \mathrm{n} > 0$$

$$s^2_{\mathrm{n,new}} = \frac{(n)\, s^2_{\mathrm{n,old}} + (x_{\mathrm{new}} - m_{\mathrm{new}})\,(x_{\mathrm{new}} - m_{\mathrm{old}})}{n+1}$$

In these formulae, *m* denotes the average, *x* represents the new measured characteristic, *n* is the number of measurements, and *s* represents the correction value.

3.4. Initial State

One problem with our solution to store and provide characteristics is encountered when there are no relevant measurements available to provide a prediction. In this case, we can use of some other techniques described in this section.

Prediction based on basic network parameters

In combination with the Network Abstraction Layer described in [9], it is possible to provide information about maximum link speeds, reception level etc. With this information, there is a possibility to give a quality indication of the first hop parameters. Since this is only first-hop information and not information of the complete end-to-end path, it is not always very accurate, especially when the first hop is not the bottleneck link.

Comparison with other experience

When the device makes a connection to an end-host that does not have an entry in the database, our software can compare information with other end-hosts that have a similar network path. When the network paths are relevant, then the system can use the stored characteristics from this host for prediction of the network characteristics.

In order to use this technique, our software also needs to store the network path to an end host. To obtain the network path to an end host, we can use the trace route tool. This is not always reliable however, since a path trace done with trace route has some flaws [6].

We can also determine path similarity by using locality information of the end hosts. This information can be collected using WHOIS queries, or using the NetGeo [11] database. NetGeo provides locality information based on IP address. While these locality information providers are not perfect [12], they still are accurate enough to provide locality information of the end-hosts. Based on this information we can assume that if we connect to these end hosts from the same physical location, that the first hops of the end-to-end paths (given the usage of the same network interface) closely resemble each other.

4. FRAMEWORK DESIGN AND IMPLEMENTATION

This chapter will describe the design and implementation of *UFNET (Up Front Network Estimation Toolkit)*. UFNET is a collection of software modules bundled together to form basis software to collect, store and predict end-to-end network characteristics. We discuss how we built up the UFNET framework and which design choices we made. The concepts discussed in chapter 3 help us define the specifications and build up the framework.
We use this framework to research the existence of network characteristics patterns in time as seen from the mobile device. In addition, the framework is used to research the accuracy of predicting network characteristics using the patterns. We also designed this framework to solve the mobile environment issues discussed in chapter 2.
In the first section, we start with formulating the design considerations of the framework. After that, we provide the design overview of the framework and we end by supplying detailed information on the implementation on the framework. In this same section, we also describe issues we encountered during the implementation of the framework.

4.1. DESIGN CONSIDERATIONS

Before we created our software, we had to consider the following concepts:

1. **Lightweight**
   Because we targeted our framework at Mobile Handheld Devices, we needed to keep the framework as lightweight as possible. This means that we had to keep the use of resources as limited as possible. First, we had to limit the use of storage space. For example, it is possible not to store all collected packets when a packet trace is collected, but just a subset of the collected packets. The use of processor power also had to be limited in order not to impede the end user in normal day device use.
2. **Extendibility**
   As we described in Chapter 3, one should be able to extend the framework, making it able to measure additional characteristics. For example, it is possible to extend our framework to measure the number of hops in an end-to-end path. Since other programmers want to do this, we had to make the process of adding additional characteristics measurements easy to understand.
3. **Platform Independent**
   The framework should not be platform dependent. We designed it in such way that it can be converted to other devices. We achieve this by using as little as possible functions of the operating system. Of course, it is difficult to achieve this completely. Now, our framework only works on Windows CE 5.0 (the operating systems that use this core are Windows Mobile 5.0 and Windows Mobile 6.0), but with effort it is possible to make it work on a number of different platforms.
4. **Control & Personal Data**
   We designed our framework in such way that users are able to stop the collection of statistics whenever they please. Since the framework stores much data and from this data a lot of personal information can be derived from this data, it forms a privacy risk. In the current implementation, we did not take any steps to encrypt this data.
5. **Presenting data**

In order to make it easy for programmers to get predictions of network characteristics, we had to design the framework to present its predictions in an intuitive way. We have done this by implementing a simple application interface that provides predictions needing only a small amount of function arguments.

4.2. IMPLEMENTATION OVERVIEW

To provide insight in the layout of the framework we have created a class diagram (see diagram 1 on page 22 and see appendix 1) and a general layout diagram (Figure 10) that provides insight in the total view of the framework. We will start with the general layout to give a basic view on the framework.

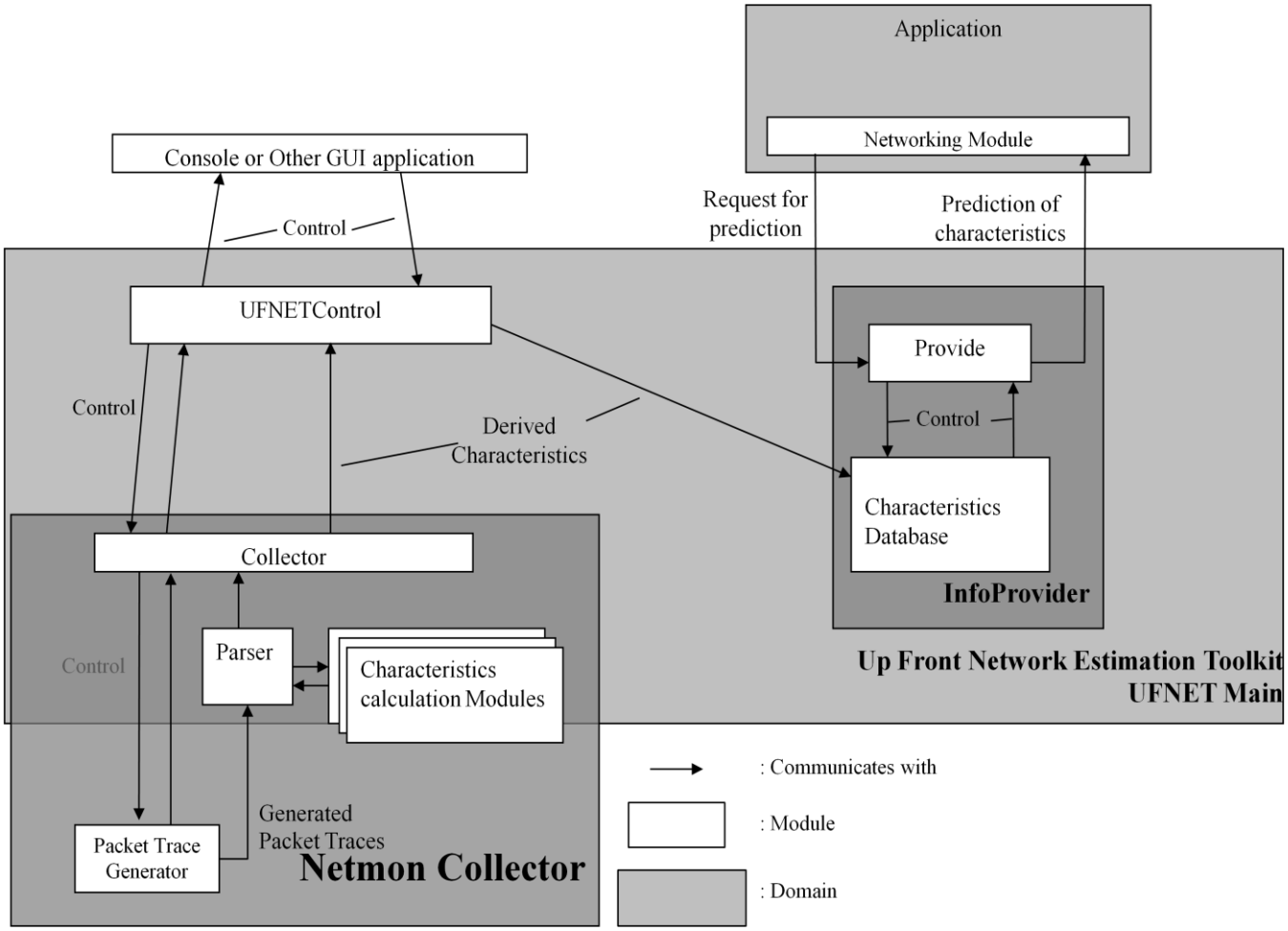


Figure 10: Basic General Layout

We have split the framework in three domains (collection of software modules): the Netmon collector domain, Info Provider domain and the UFNET main domain.
The UFNET main domain manages communication between the different components. The UFNETControl module located in the main domain controls the different software components within the UFNET framework.
The modules within the Netmon collector domain perform the actual network characteristics gathering. The packet trace generator monitors all traffic on the mobile device, it captures all the packets that the mobile device received and sent. When the collector signals the packet trace generator to calculate characteristics, the parser reads out all the packet traces and starts to

derive network characteristics using the characteristics calculation modules (which do the actual calculation process). When this process completes, it sends the derived characteristics to the UFNETControl module.
Within the InfoProvider domain, we have defined modules that read out the database and provide the characteristics to applications. It is up to applications what to do with the predicted characteristics. Our framework only collects and provides characteristics; it does not make any decisions regarding policies.

'Statistics'
Network characteristics are stored within c# objects called 'Statistic' (not depicted in figure 6). This object is used throughout the UFNET framework to transfer information about the derived network characteristics to the separate modules. It contains basic information on the derived network characteristic like the time the framework recorded it, the value and descriptive information about the type of the network characteristic.

4.3. UFNET IMPLEMENTATION

We will describe the UFNET implementation in detail in this section. We start by defining the platform we work on, after that we explain our implementation in detail.

Platform
We have implemented our software for the Windows Mobile 5.0 platform. We chose for this platform because it is very well documented and we had all software available to be able to program the devices. The device we used for testing was the HTC TyTN, a Pocket PC device running Windows Mobile 5.0.

Development Language
We chose for Microsoft.NET as our development environment. The .NET-developing environment provides us with many functions that we have to use for development of our project. In order to provide us with Pocket PC specific platform functions, we also made use of the OpenNETCF Smart Device Framework 2.0 platform [17]. The language used is C#. We chose C# since it allows for easy extension and it is independent of the device used as long as it is running Windows Mobile 5.0. For Operating System functions that were not available within the .NET or the OpenNETCF framework, we created and included some native DLL files.

4.3.1. IMPLEMENTATION DETAILS

Consider the class diagram on the next page, it shows the class diagram of our framework. We did not include all classes in order to keep the diagram conveniently arranged. We derived the real implementation from the basic general layout as we presented it in Figure 6. The collector interface is part of the UFNET main domain, where the *NetmonCollector* class is an implementation of this collector interface (and in our case part of the *NetMonCollector* domain as shown in figure 10).
To start the characteristics collection process, an application has to supply an implementation of the Collector interface, and use the *CollectStatistics* function. We have provided the *NetmonCollector* class as the default. The *NetmonCollector* class is responsible for capturing and storing packet traces. We have described this class in detail in section 4.2.3. When the *CalculateSatistics* function is called, *NetmonCollector* starts to derive network characteristics from the stored packet traces. These characteristics are stored in *Statistics* objects and are

afterwards send to *DataBaseUpdater*, which updates the database. The *ConnectionEstimate* class (which is part of the *InfoProvider* domain) uses this database to supply applications with upfront network characteristics estimations.

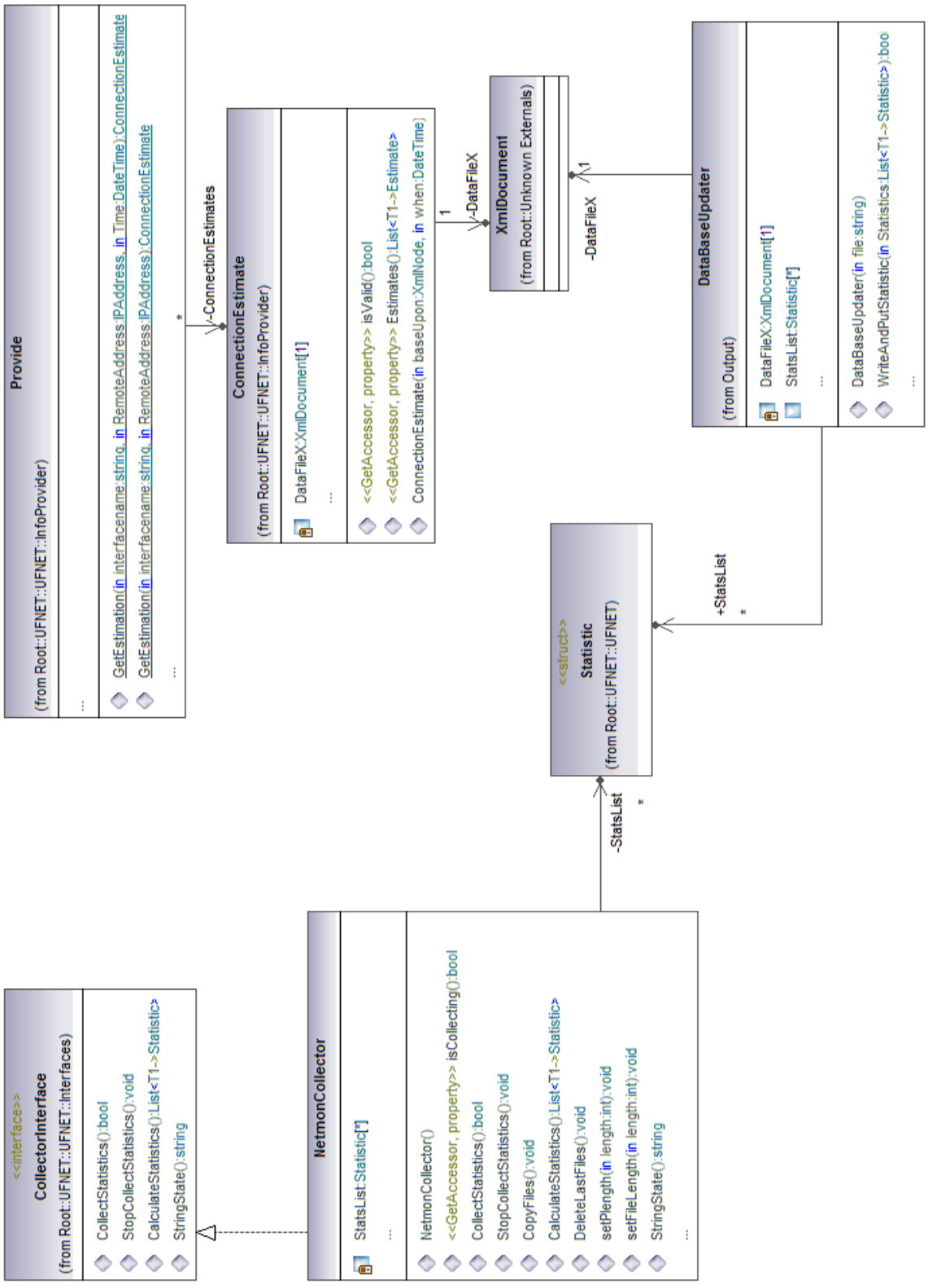


Diagram 1: Limited class diagram of UFNET framework

XML Database

```
<StatisticsXML>
        <dX>
                <tYY+1>
                        <iZ_W>
                                <Statistic 1>
                                </Statistic 1>
                                <Statistic N>
                                </Statistic N>
                        </iZ_W>
                </tYY+1>
        </dX>
        <dX+k>
        …
        </dX+k>
</Statistics XML>
```

Figure 11: XML layout. X = day number, Y = hour, Z = local interface, W= End host IP address

To implement the database we chose for a XML solution. XML allows for easy data exchange to other processes. In addition, .NET offers a vast range of possibilities to work with XML files easily. We examined the possibility of using a SQL solution with Microsoft SQL server Mobile Edition. However, after some consideration it seemed that it is not possible to create a SQL database on the storage card, limiting the storage capacity severely. See figure 11 for the layout of the XML file. We chose for a fixed time period as described in section 3.3. Characteristics are stored per day, per hour, per interface and end host. To use a dynamic solution also described in section 3.3 we have to update the *DatabaseUpdater* class. We did not choose for this due to time constraints of the project. According to the specification, the XML should fall within the *InfoProvider* domain. However, since we use a separate XML file for a database, the xml file does not fall into a specific domain as described in figure 10. The storage procedure is more extensively described in section 4.2.3.

Predictions

To provide predictions of network characteristics, we use the *ConnectionEstimate* and the *Provide* classes. Both classes fall into the *Info Provider* domain. The *provide* class is the main entry point for applications that want to use the up-front estimations that the UFNET framework offers. The class accepts a point in time, end host and interface to use. Based on this data, it will look up if predictions are available. When predictions are available, it will return a *ConnectionEstimate* instance that contains the different characteristics that are stored in the database. We left out the functionality of advising a network interface. It is still possible however, to get network characteristics of a network interface at a given time. The programmer then has to supply the different network interfaces available to get characteristics for all the devices.

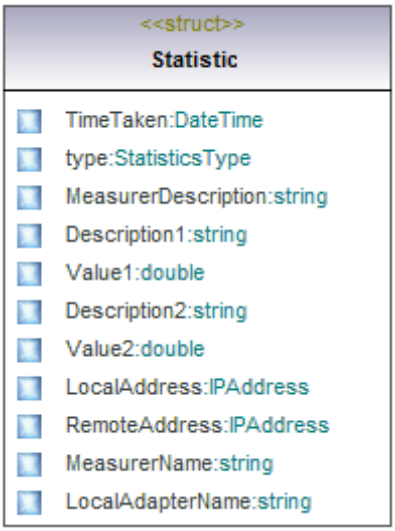


Figure 13: Statistic Structure

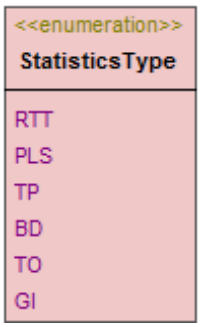


Figure 12: Statistic Type Enumeration

Statistics

Statistic is a data structure that contains information about a Network Characteristic. Figure 12 describes the layout of the C# structure. All the variables are customizable except for the type field. An enumeration determines the values of this variable. To be able to describe characteristics that are not represented in the enumeration, we included a General Information field. We have included a basic type for each of the characteristics that we

can provide with our framework. Figure 13 shows the different types available. These are Round Trip Time, Packet Loss, Throughput, Battery Drain, Time Out, and General Information. Note that we added Battery Drain to include characteristics regarding the battery usage of the device during measurements. The *collector* implementation generates instances of the *Statistic* structure. After they are generated, the *collector* implementation sends them to the *database updater* implementation.

Database Updater
The Database updater updates the XML file as described in the Basic Implementation section. This is initiated by the Collector implementation. Since Updating database might be a lengthy operation due to large file sizes, we chose to let the controlling application decide when the database should be updated. In the current implementation, we have embedded the different database updater implementations in to the NetmonCollector package. See following section for more information regarding the database updater.

4.3.2. COLLECTOR IMPLEMENTATION DETAILS

In this section, we will describe the layout and the implementation of the *NetmonCollector* domain. *NetmonCollector* uses the Netlog[8] tool to obtain packet traces of the network traffic that is sent and received by the mobile device. We will first explain how Netlog works. After that, we will go into detail on each of the classes that the class diagram appendix 1 shows.

Netlog
During the early stages of the framework development, we did not know how to generate network statistics easily. The obvious choice was to use socket masking .The idea was to build a socket implementation on top of the one already existent in Windows Mobile 5.0. This socket masking technique would then perform measurements during the usage of the socket. Some measurement methods were to calculate the delay between a connection request and a connection acknowledgement or to measure the amount of data stream coming through the socket per second. However, this would be a crude measurement since we would not have been able to deal with extra delay the operating system causes.

Figure 14 depicts the measuring point of the netlog module. This figure depicts the measuring point from an application point of view. As can be seen, it is in between the NDIS driver and the Socket Stack.

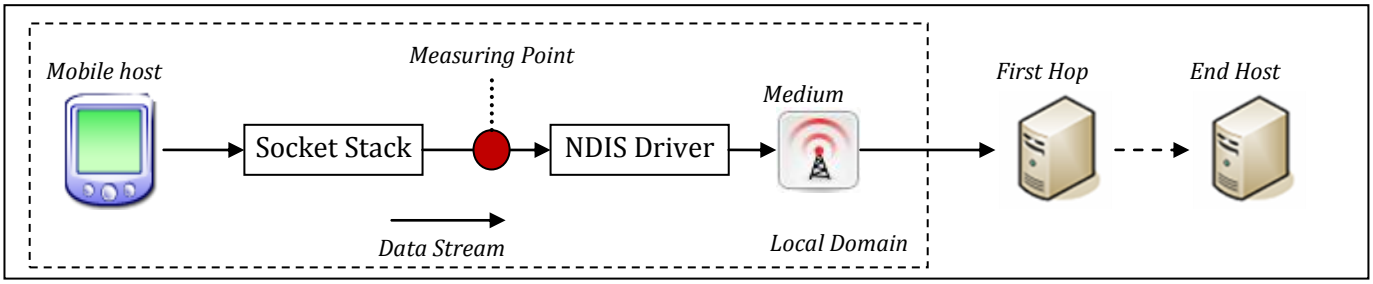


Figure 14: Measuring Point

Windows CE operating system, a tool called 'Netlog' exists. This tool is used to store raw data packets as received by NDIS (Network Driver Interface Specification)[7]. Using the raw stored data packets, we were able to derive statistics from these packets directly. For more information regarding 'Netlog', see [8].

The netlog.dll driver is loaded when the NDIS driver finds it. This driver normally is controlled by NetlogCTL.exe. However, it was not possible for us to use a console to get output from the netlogctl.exe module. Therefore, we made some changes to the netlogctl.exe module and added the static functions to netlogctl.dll. This dynamic link library allows for control of the netlog.dll module during runtime within the same process. Figure 15 describes the overall flows of the data and the different modules within Windows Mobile 5.0.

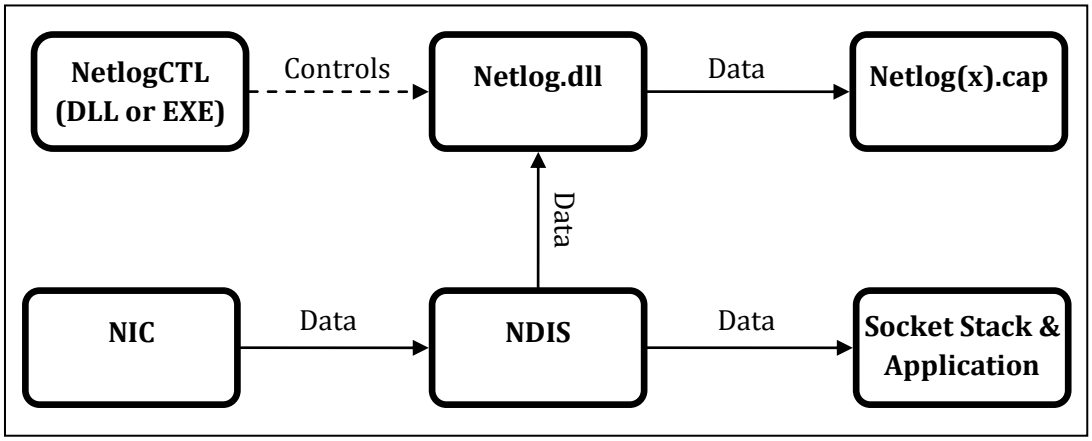


Figure 15: Netlog

Figure 16 depicts a representation of the data that netlog collects. The data is stored in the .CAP file format as defined by Microsoft. This file format is used in e.g. Microsoft Network Monitor. We wrote our own parser for this file format since the timestamp accuracy of the Microsoft Log Parser tool[22] is not sufficient to calculate statistics.

MAC frame header	IP Packet	TCP/UDP	MAC frame footer

Figure 16: Captured Data

Datacopyer

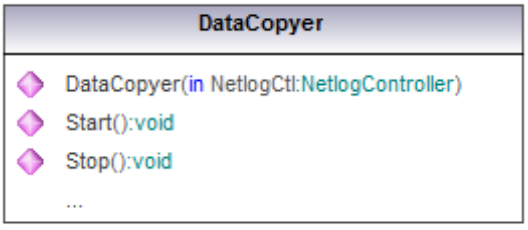


Figure 17: Datacopyer Class

Figure 17 depicts the main functions of this class. This class provides packet traces copying functionality, which we need to free up system resources. Once the Netlog subsystem is activated, it generates packet trace (.cap) files in the main memory of the handheld device. These files are stored here because in general, this memory is faster than an external memory device, like a SD card. These cap files are initially not larger then 1,5 megabytes, although this size can be changed using the *netlogcontroller* class. Netlog creates files with an index fashion. When a netlog file is created, it has the file name of netlogK.cap, with K being the current index number starting at K=0. Initially, the netlog.dll driver would switch between K=0 and K=1, thus overwriting the old file. For safety purposes, we decided to keep the files in case something would go wrong with the experiment. Therefore, we changed it to the method just explained.

The *DataCopyer* class detects any change in the default netlog directory. As soon as a new netlogK.cap file appears in the default netlog directory, the file the *DataCopyer* is copies the file to the Storage Card. On the storage card, the files are kept in a temporary directory until the NetlogParser has processed them.

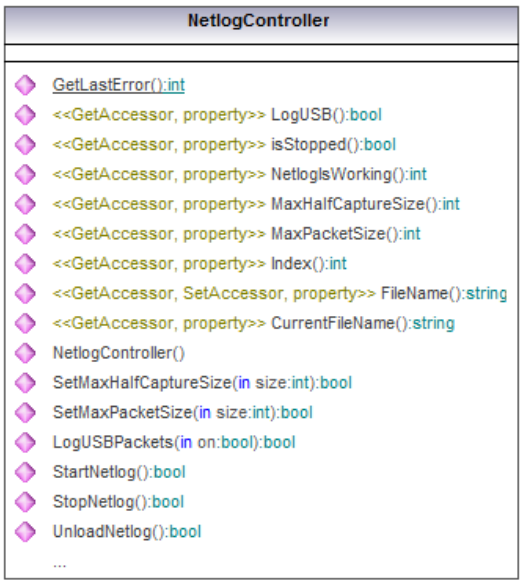


Figure 18: Netlog Controller class

The *DataCopyer* class needs the *NetlogController* class as an object in its constructor. This is needed in order to stop the netlog subsystem when files need to be copied.

Netlog Controller

The netlog controller class (see figure 18) functions as a wrapper class for native c code. Originally, the netlog subsystem comes with a NetlogCtl.exe in order to send control functions to the netlog.dll, which acts as a driver. This executable file takes console input and outputs to a console as well. It was not possible to read console output in an easy way on the Pocket PC platform. To be able to control the netlog driver dynamically, we wrote a DLL file that takes of the functions of netlogctl, but offers the static functions to programs that want to make use of the functionality. To provide the wrapper functions from native C code to managed C# code, we created the *netlogcontroller* class. This class keeps track of the state of the netlog driver and allows changing some of its properties. Figure 20 gives an overview of the variables that can be changed using this class. Note that some of the functions described in this class were initially not available, although they were described in the netlog API. We implemented these features by changing some code in the netlog.dll file and compiling it. This was possible for us to do since the Windows CE 5.0 (the core of windows mobile 2005) Platform Builder came with the source code of the netlog subsystem.

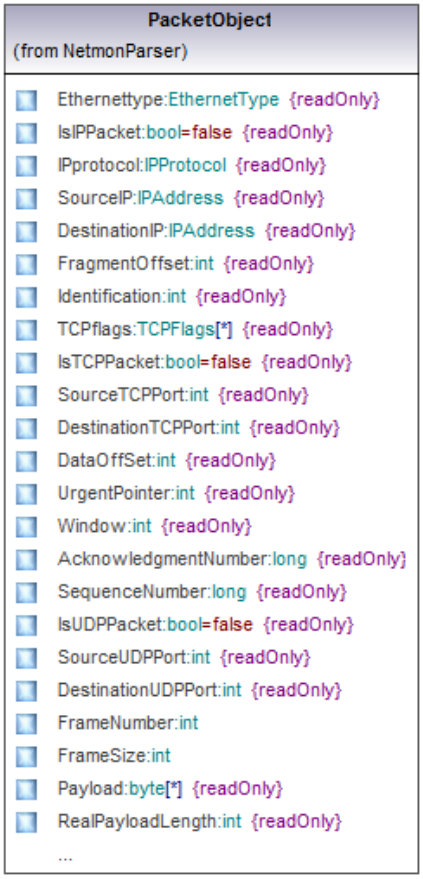


Figure 20: Packet Object Class

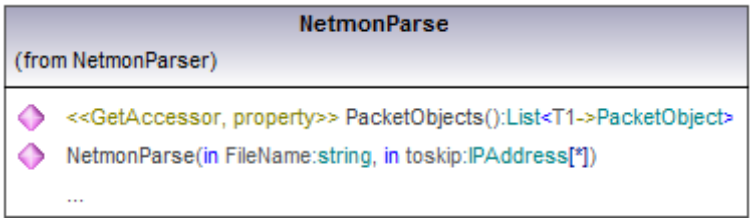


Figure 19: NetmonParse Class

NetmonParse and Packet Object

NetmonParse (figure 19) offers the functionality to parse cap files and convert them to packet objects. The general layout of *NetmonParse* can be found in Figure 19 while the general layout of the *PacketObject* class can be found in Figure 14. *NetmonParse* takes in the constructor a string to the filename it has to parse. When requested, it returns an Array of packet objects that were parsed from the .CAP file.

Additional functionality that should be added to this class is the possibility to add multiple files to be parsed. This would allow the class to concatenate all the files, and return all the relevant

information from the files into one list of Packet objects. In the current implementation, all files have to be separately sent to the *NetmonParse* class.
Packet objects contain a large collection of all the data that is stored in an Ethernet Frame. Not all information from the packet trace is stored in a Packet Object. For example, checksum data is not stored. This is because we did not use it in the characteristics calculation process. If needed, one can easily change the *Packetobject* and *NetmonParse* classes to add the storage of these variables. The packet objects are used for calculation of network characteristics. Each separate packet object represents a frame as depicted in Figure 16.

IStatistics
The *IStatistics* interface (Figure 21) provides an interface for deriving statistics in the *NetmonCollector* class. As soon as the framework starts collecting packet traces, the framework initializes the various implementations of the *IStatistics* interface that are available in the *IStatistics* collection in *NetmonCollector*. These need initialization since they might need more data then the packet traces provide (e.g. Battery drain). We chose for the *IStatistics* solution in order to provide extendibility for the methods of characteristics generation. We have made a couple of implementations, which we will discuss in the next section. The current *NetmonCollector* implementation only uses a predefined set of *IStatistic* implementations. It is possible to extend the characteristics derived by adding an implementation of the *IStatistics* interface.

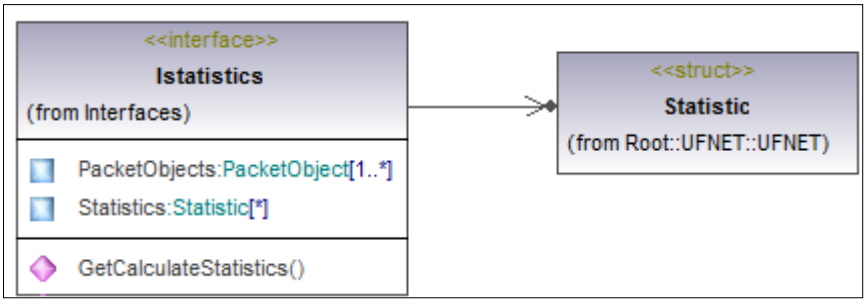


Figure 21: *IStatistic* Interface and *Statistic* Object

IStatistic Implementations
The implementations currently available in the UFNET framework are:

- **FTP_RTT**: Provides Round Trip Time estimation based on FTP commands and their reply
- **RTT**: Provides Round Trip Time estimation based on SYN-SYN_ACK packets
- **SimpleThroughput**: Provides Throughput estimation based on number of bytes counted within a certain time period
- **SimplePacketLoss**: Provides Packetloss estimation based on SEQ-ACK analysis

We only use TCP in our measurement techniques. As described in section 3.2, TCP allows for (relatively easy) measurement of some basic QoS variables. To derive the characteristics, we implemented the principles described in section 3.2.

IOutput and Implementations
For our experiments, we needed more methods to output network characteristics. Therefore, we created an *IOutput* interface with several implementations of the interface. Figure 22 describes the layout and the relations of this interface.

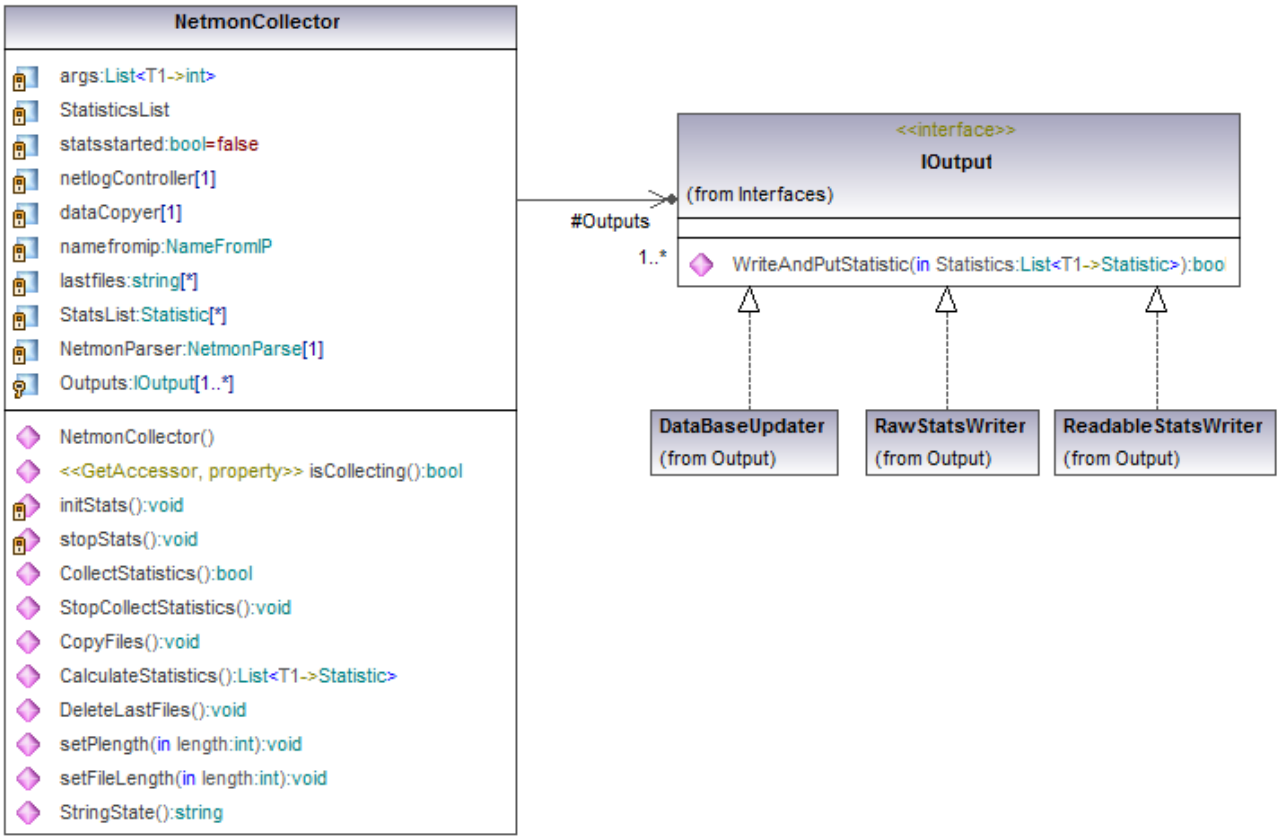


Figure 22: IOutput and Implementations

According to Figure 10, our initial design showed that the *DatabaseUpdater* class should be part of *InfoProvider* domain. However, since we chose for an offline database file in XML form, we have included the procedures to update the database in the *NetmonCollector* domain. Along with it, we have included an *IOuput* interface that the various output modules implement. Like the different *IStatistics* implementations, the framework does not automatically add all the different *IOuput* implementations to the output list. They have to be manually added and executed in the monitoring applications.

RawStatsWriter and ReadableStatsWriter

We added these two implementations of the *IOutput* class for use in our experiment. They output the characteristics as the *IStatistic* implementations supply them. The difference between *RawStatsWriter* and *ReadableStatsWriter* classes is that the *RawStatsWriter* class writes the characteristics in a binary format (compressed) where the *ReadableStatsWriter* class makes a textual impression of all the different statistics.

5. EXPERIMENT EXECUTION

In this chapter, we will describe how we executed our experiments and the realization of the testing application. We used the UFNET Framework to execute two experiments.
In the introduction of this report, we described that we want to use cyclic patterns in time of network characteristics to predict network conditions. In the first experiment, we collected data to use for our predictions. This experiment also helped us determine whether there are cyclic patterns. In the second experiment, we test how well we can use the data collected in experiment to predict network conditions.
In 5.1, we will first describe the setup, difficulties and execution of the first experiment. In section 5.2, we describe the second experiment. We end this chapter with section 5.3, where we provide some information on the difficulties we had during the creation of the test application.

5.1. FIRST EXPERIMENT: DATA COLLECTION

For the first experiment, we ran an application for two weeks that executed download sequences at random times during the day. The download sequences were start in a regularly, in order to acquire enough measurements to be able to make predictions for each of the 168 timeslots in one week (see section 3.3). Each download sequence consisted of downloading a small fragment of a big file from ten different hosts. To prevent overlapping of the download sequences, the application scheduled a new download sequence as soon as a sequence finished. The maximum waiting time was selected randomly between 10 and 20 minutes. We did this in order to get enough measurements to get an average of that hour, and not just a single sample. To get average values taken over multiple weeks, we measured two weeks long.
One person carried the device with him for the duration of the experiment. We made sure that during the day, the device had multiple sources of internet available. Therefore, we placed an 802.11 access point where this user resided most often.
When the test application started a download sequence, it randomly selected a network interface. When the framework recorded a characteristic, it stored information of the interface used to obtain it and the time it was taken. There was a 20% chance that the application selected the Cellular line (UMTS or GPRS) interface and an 80% chance that it selected the 802.11 interface to connect to the internet. We chose these values to reduce the usage of UMTS and GPRS traffic, which costs much money. If an internet connection was already available (for example in the form of a USB or Bluetooth connection) then the test application used this internet connection for download. When the download sequence completed, the test application disabled the network interface it used to connect to the internet.

Collected data
As a download source, we needed a file that is mirrored at different locations. At first, we used Tucows HTTP mirrors as download source. However, during the test, we found out that our UMTS connection provider buffered all the HTTP data we collected, rendering our test results useless (see transparent proxy on UMTS section below).
In order to overcome this problem we selected FTP mirrors instead of HTTP mirrors. Since Tucows servers do not provide any files by FTP, we had to change our source. We chose for several UBUNTU mirrors spread across the globe. Our test file was the latest UBUNTU cd image. This file was 732293120 bytes long. We had 10 mirrors in our test list:

ftp://ftp.snt.utwente.nl/pub/linux/ubuntu/edgy/ubuntu-6.10-desktop-i386.iso	Utwente NL

ftp://ftp.tiscali.nl/pub/mirror/ubuntu-releases/edgy/ubuntu-6.10-desktop-i386.iso	Tiscali Adam NL
ftp://ubuntu.mirrors.skynet.be/pub/ubuntu.com/releases/edgy/ubuntu-6.10-desktop-i386.iso	Brussel BE
ftp://ftp.uni-kl.de/pub/linux/ubuntu.iso/edgy/ubuntu-6.10-desktop-i386.iso	Uni Kaisers. DE
ftp://mirror.ox.ac.uk/sites/releases.ubuntu.com/releases/edgy/ubuntu-6.10-desktop-i386.iso	Uni Oxford UK
ftp://se.releases.ubuntu.com/mirror/ubuntu-releases/edgy/ubuntu-6.10-desktop-i386.iso	Uni Sweden
ftp://ubuntu.task.gda.pl/ubuntu-releases/edgy/ubuntu-6.10-desktop-i386.iso	Poland
ftp://ftp.ale.org/mirrors/ubuntu-releases/edgy/ubuntu-6.10-desktop-i386.iso	Atlanta US
ftp://ftp.citylink.co.nz/ubuntu-releases/edgy/ubuntu-6.10-desktop-i386.iso	New Zealand
ftp://ftp.ecc.u-tokyo.ac.jp/UBUNTU-CDS/edgy/ubuntu-6.10-desktop-i386.iso	Tokyo JP

We chose for this list to keep file conditions and server conditions the same. We derived the location information for each of the links from the information that is listed on the UBUNTU mirrors website [19]. For the duration of the experiment, the user never left the city of Enschede, located in the Netherlands. During office hours from Monday to Friday, the user was at his work in the city of Enschede while during other hours the user was at his home, located on the Campus of the University of Twente. There were two days the user did not go to the working location, which was on the Monday and the Wednesday of the second week.

Generated traffic per file

Because UMTS traffic has to be paid for (about 50 eurocents per megabyte for the used subscription), a requirement was to generate not too much data, but enough to derive statistics. Therefore, we had to find an optimum number of bytes to download of each file. Since UMTS has a high variance in packet delay[24], an optimum value for every case was difficult to find. Using a download time constraint would not suffice, since it often happened that after a connection has been set up, the connection idled for a couple of seconds and did not generate traffic. In Figure 23, we see a throughput trace from a file download. It shows that the data connection idled for at least three seconds before traffic was sent to the mobile host. This could be due to the web server took long to send the reply, but it shows us that we cannot use the time constraint to determine the length of a single download.

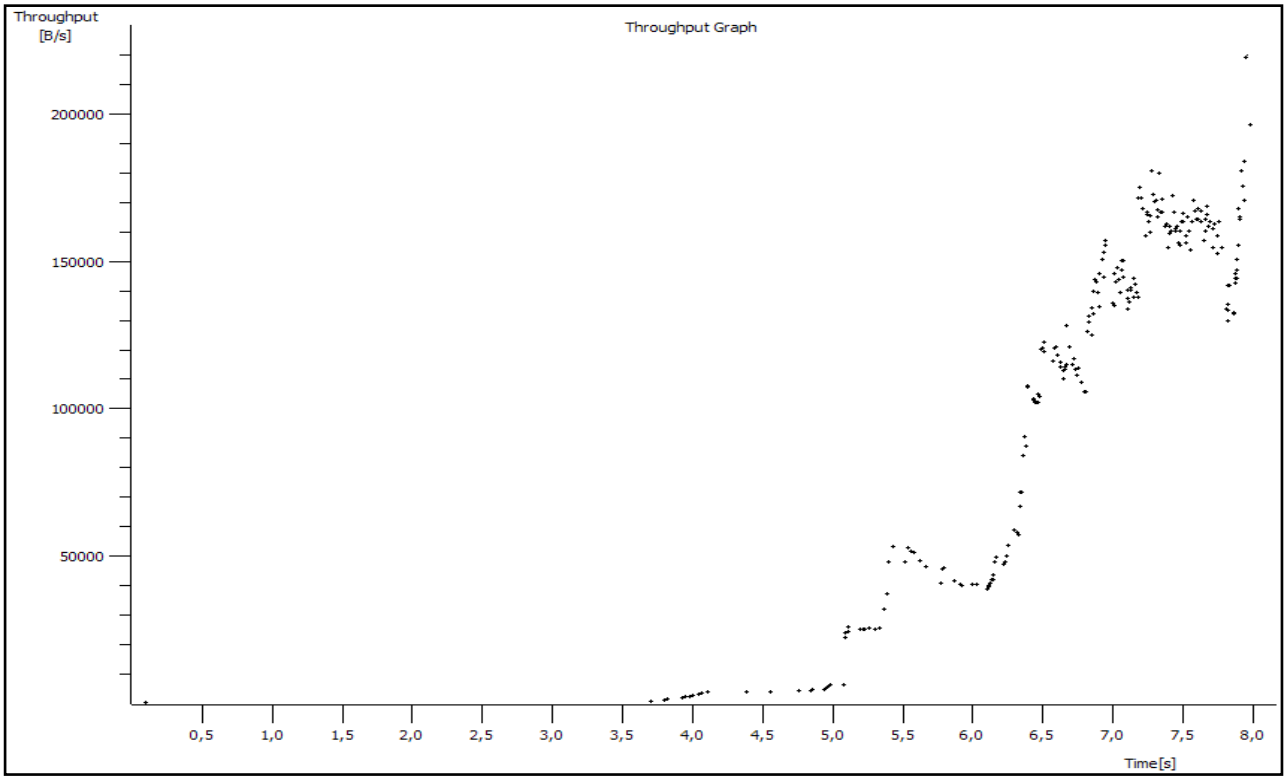


Figure 23: Throughput Trace of UMTS traffic

Because we cannot use a time constraint, we used a data constraint. To select an optimum value of data to download we had to see how much packets were generated with a particular number of bytes. We had to consider the design issues of our throughput derivement as described in section 3.2.3 (e.g. packets discarded due to slowstart, packet length to derive throughput etc.). To generate enough packets we decided to offer the framework a minimum amount of packets by downloading a preselected number of bytes.
In figures 24 and 25 we see two traces that represent a download from a server in Switzerland. Both connections were made over a UMTS connection and we defined no download time constraint. The download would stop after either 200kb (figure 24) had been downloaded or after 800kb (figure 25) had been downloaded. As we can see in the figure of the 200kb download, more than 50% of the packets of the total download are 'wasted' on TCP Slowstart. In the 800kb version however, only a small percentage is 'lost' due to slowstart. Since 800kb would generate too much data (cost requirement) and 200kb would generate inaccurate results, we chose to stop each download after 600kb.

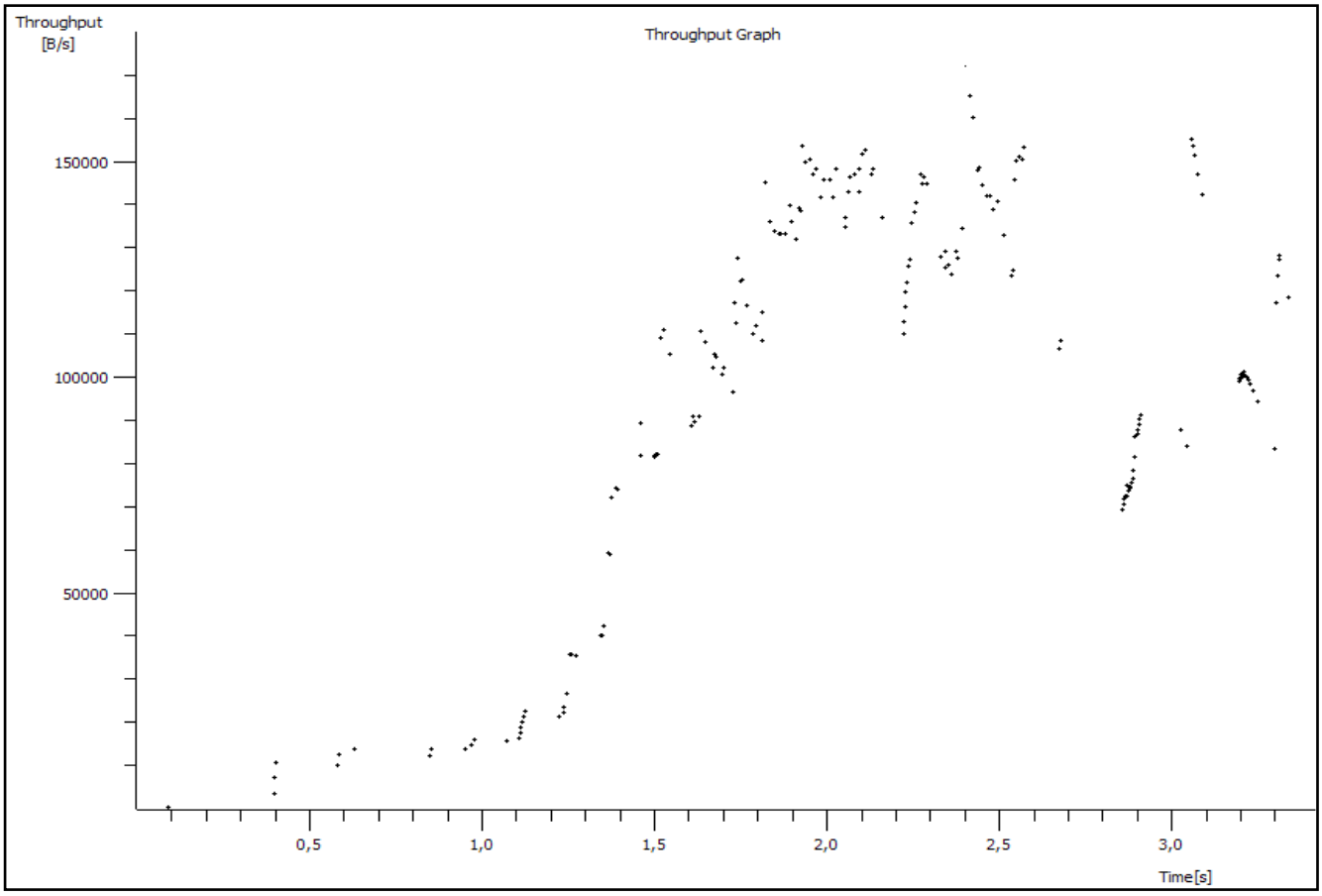


Figure 24: Throughput trace of 200 kilobyte download using UMTS

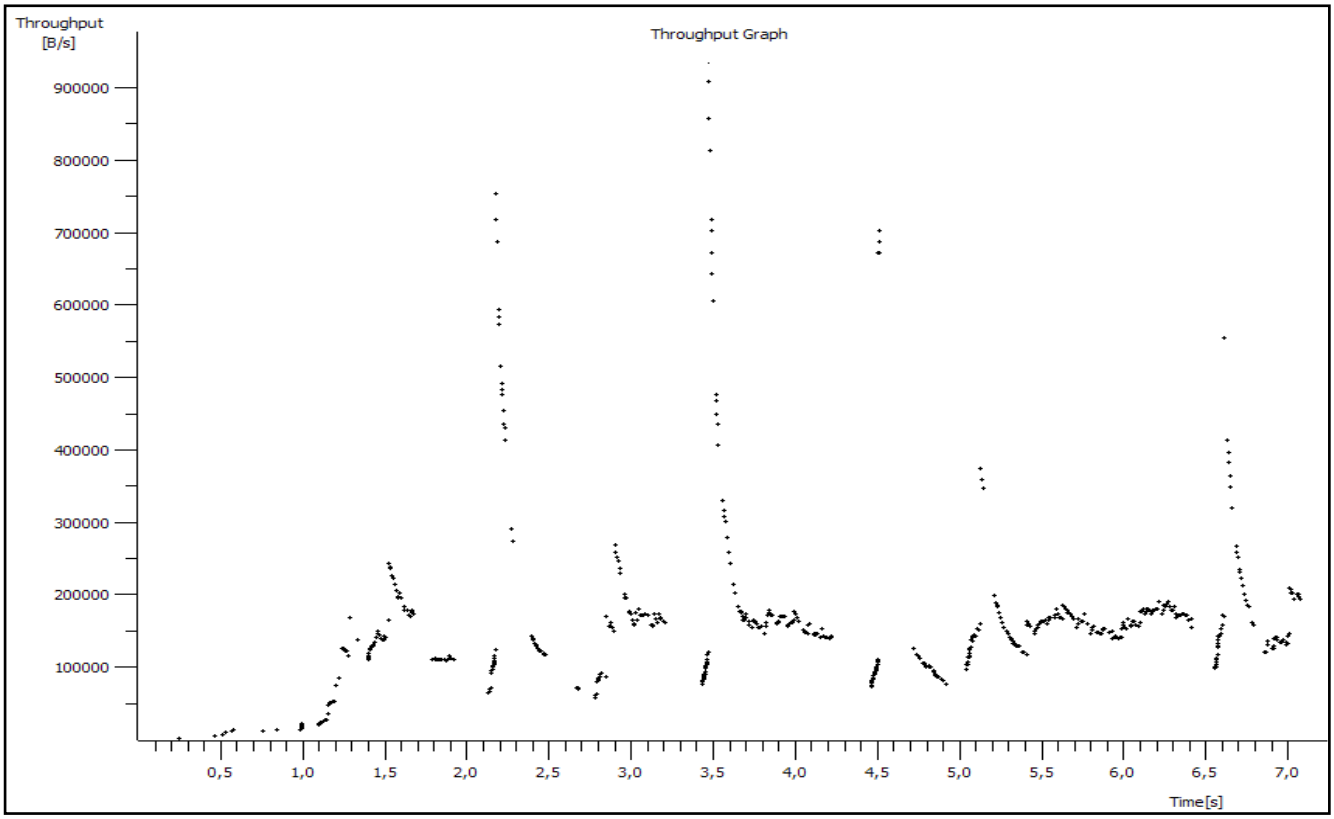


Figure 25: Throughput trace of 800 kilobyte download using UMTS

Timeout

The current socket implementation on windows mobile 5.0 did not support connection time. As a result, we had to program our own timeout implementation. For the connection timeout, we took a value of 30 seconds. If the application received no data for over 30 seconds then it assumed that connection has been lost and continued with the next download location, if an internet connection was still available. We had to use Socket timeouts; else, our test application would stop functioning if a connection had been lost.

Transparent proxy on UMTS

During the execution of the first experiment, we found out that the UMTS RTT measurements had inconsistent values. We discovered this since our UMTS measurements from *ns-tucows.tucows.com* (a server in Australia) gave a round trip time of 90ms where the same measurement on 802.11 had a response of around 350ms. Take note that our mobile end host at the time of measuring was located in Enschede, The Netherlands. We also discovered that the throughput of this particular server was much higher when using UMTS than the throughput using 802.11. It appeared that Vodafone (our mobile service provider) had a proxy server installed in order to speed up UMTS transfers. Consider figures 26 and 27.

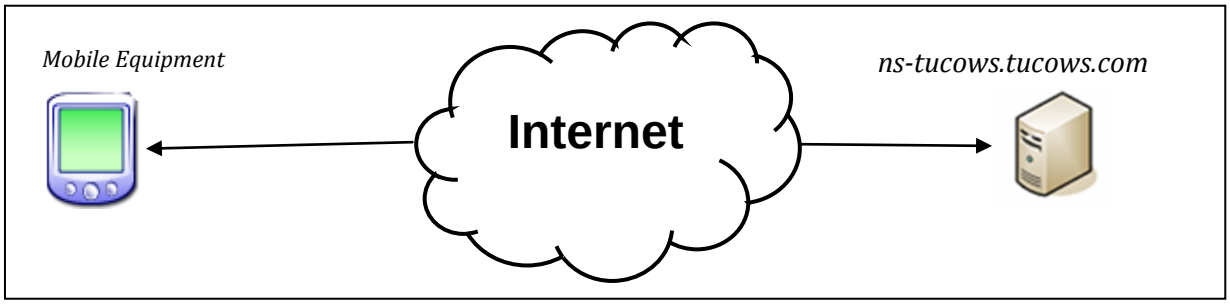


Figure 26:Assumed UMTS connection situation

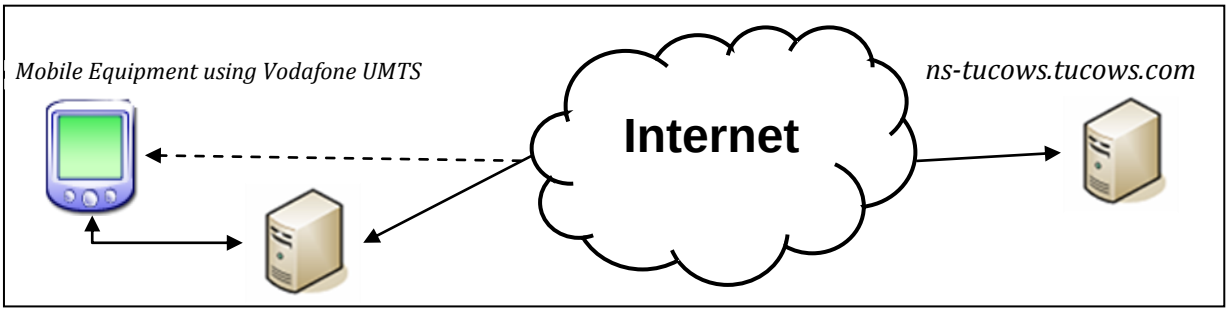


Figure 27: Real UMTS connection situation

In figure 26, we have depicted the situation as we assumed it. The mobile equipment had a direct connection to the end host, therefore making our method of deriving network characteristics using TCP packet traces valid. It is often the case however that a computer running Network Address Translation (NAT, a technique to multiplex an internet connection) is between the mobile equipment and the internet. This normally is not of concern since most NAT computers only map the port number different and directly route the traffic from remote endpoint to requesting local endpoint, without changing the packet contents.
In figure 27, we have depicted the real situation. In reality, an intercepting host resided between the mobile equipment and the internet. This host acted as a proxy; it cached all HTTP requests coming from the mobile host. It works by intercepting TCP traffic that is sent from the mobile host. This host is called a transparent proxy, since the mobile host did not have any indication that this proxy server was changing the network traffic. For example, when the mobile host sent a TCP SYN packet (to establish TCP connections) to a remote server on the internet, this host intercepted it and immediately sent a reply back to the mobile host with SYN-ACK, even before the connection had been correctly set up with the remote host. This prevented measurements based on TCP SYN-ACK packets because we only measured the round trip time of the first part of the path. In addition, when the proxy server cached the file of a HTTP request, the test application downloaded from the proxy server and not from the remote host. This would give a wrong indication of the network throughput. In order to cope with the transparent proxy we chose for FTP downloads instead of HTTP downloads.
We verified the existence of a transparent proxy by taking a HTTP web server on the internet that we maintained. Performing a packet trace on this server showed us that the proxy server on the Vodafone side did a “conditional HTTP GET” each time it wanted to download something from our server. After the first request, the proxy server cached the “HTTP GET” command and closed the connection to our server. It then sent the data directly to the mobile host, without retrieving it from our server. The proxy server still did pretend that it was our server sending the mobile device the data.

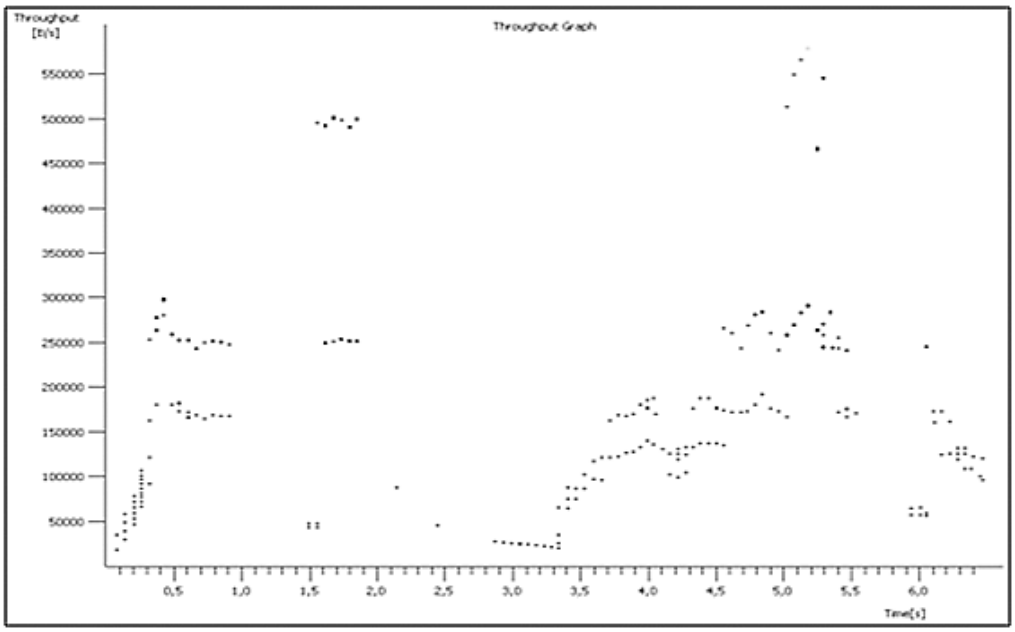


Figure 28: Throughput trace as seen by our server

The proxy server also acted as a data buffer. We verified the buffering from the fact that we see different throughput values. Figures 28 and 29 show this. We took this measurement from a packet trace of an FTP connection. In figure 28, we see the throughput graph as seen from our internet server. On figure 29, we see the throughput graph of the mobile host. What can be observed from first figure is a buffer mechanism. When the proxy server filled its buffer, the throughput speeds lower considerably until the buffer is empty. When this happens, the throughput speed increases again until the buffer is full. In addition, we see much higher throughput rates on the first figure. In [23] the authors give a thorough description of the buffering process and the transparent HTTP proxy.

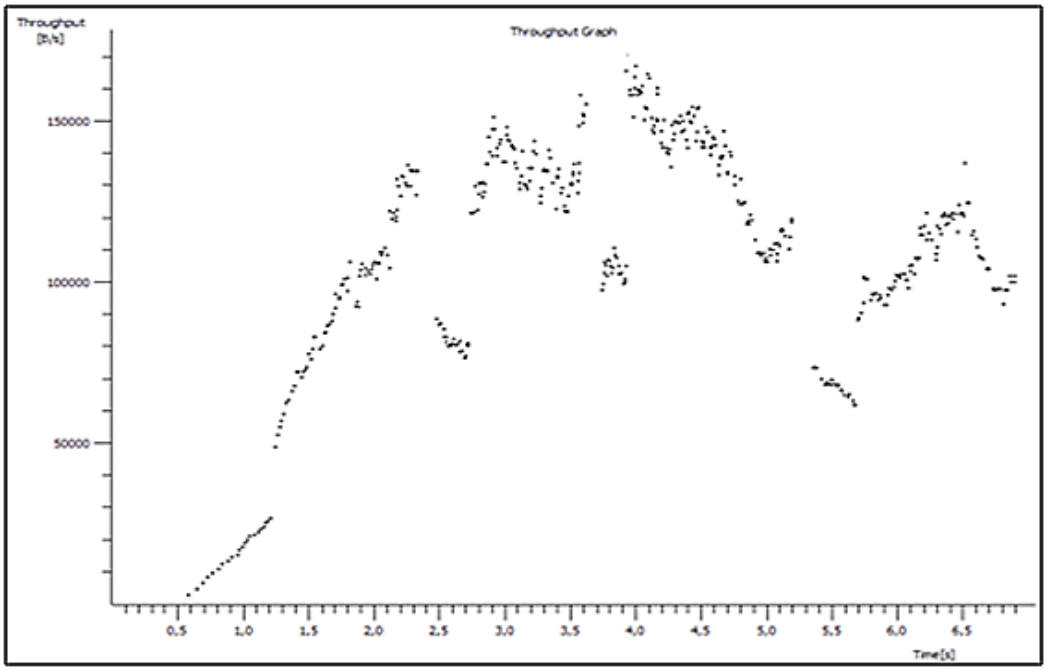


Figure 29: Throughput trace as seen by the mobile host

5.2. Second Experiment: Predicting Characteristics

This experiment resembled the first experiment; we also collected characteristics on both interfaces using the same constraints as we defined them in the first test. Since we only wanted to see how accurate our predictions were, instead of downloading from ten different hosts, each time a download sequence started the application randomly selected one host from the list we presented in section 5.1. Before the application started the download, it requested a prediction of the network characteristics from the UFNET framework. The application then stored this prediction. The measured network characteristics are stored the same way as in the first experiment, although this time in a different database in order to keep the original database as we built it with the first experiment free of change. Using these settings, we were able to compare the actual measured statistics with the predicted value. Since we did not need to generate accurate values of the network conditions, we ran the test for one week and we did only one measurement per hour.

5.3. Test Application Difficulties

Most of the difficulties we encountered during the implementation of the test application had to do with discovering native functions in the Windows Mobile 5.0 development environment. We describe two issues that took much of our time to solve.

Device suspend
In order to perform the test, we needed to make sure that the device kept executing our test code during the download of a file. In Windows Mobile 5.0 devices, the operating system might end up in a suspend state when the user has not interacted with the device for a while. When the device is in suspend, it halts code executing until the user wakes up the device, and with it stopping our test. This issue has also been present during the research of [9]. We used some parts of the NAL implementation of this research in order to prevent the device in going into suspend mode during the download of a file. In short, this code keeps the device alive by resetting the idle timer of the device. In order to make the code compatible with our test code and to save battery life, we added a mechanism that makes it possible to enable/disable the keep-alive code.

Enabling/Disabling Wireless Devices on Windows Mobile 5.0
We could not find any documentation on how to enable/disable power to 802.11 interfaces on Windows Mobile 5.0 using software calls. There are some undocumented features that allow enabling/disabling a wireless interface (Bluetooth, Phone or 802.11). The documentation we used in order to use this feature can be found here [10].

6. EXPERIMENT RESULTS

In this chapter, we will discuss the results of our experiments. We will go into detail on the results we got from both experiments. In section 6.1, we will discuss the results of the first experiment and in section 6.2, we will discuss the results of the second experiment.
Although the main purpose of the first experiment was to collect data for the second experiment, we also did this experiment to look into the patterns of the different network characteristics.
We only did these experiments with one user; this makes it difficult for us to get solid conclusions. It is possible however to provide a view on the predictability of network characteristics.

6.1. RESULTS FIRST EXPERIMENT

As described in chapter 5, in this experiment we intensively gathered network characteristics for two weeks. Every 10 to 20 minutes our test application downloaded 600 kilobytes from 10 different hosts across the globe. We gathered statistics for round trip time, throughput and packet loss. In this experiment, we only gathered data to verify if there is a recognizable pattern. We will describe the reoccurring patterns we observed from the collected characteristics using a small portion of the statistics we collected. Most host / characteristics combination show either dependency on user location, dependency on network / server load or, to some extent, no dependency at all.
We took the characteristics from the database we developed for the UFNET framework. That means that each point in the plots is the average on a specific weekday and hour over two weeks. As a result, there is a maximum of 24 * 7 = 168 points for the combination of each host and network interface. The total number of measurements for each combination of interface and characteristics therefore makes 168 * 10 = 1680 measurements. Unfortunately, some of the measurements failed, resulting into gaps in the plots. Some reasons for these gaps are:

- Losing internet connectivity during start of measurement
- Host did not respond on connect request after 30 seconds
- Host did not respond on FTP command within 30 seconds
- Host did not respond correctly on measured FTP command
- Measurement fell out of the boundaries specified in the test section.
- Interface lost connection to gateway (e.g., connection to 802.11g base station is lost)
- Destination network was unreachable for a specific host.
- No measurement done for the interface at this timeslot

The average number of gaps for WLAN was 212.86 (12.27%). For Cellular Line this is 808,1 (48,1%). Not all of these gaps are the result of measurement failures; the high percentage of gaps for the Cellular Interface is mostly the result of the small percentage of measurements (as explained in chapter 5). For both interfaces, the most gaps occurred in Packet Loss plots. The least gaps for cellular line are in the Round Trip Time plot and for WLAN in the throughput plot.

The person who carried the device running the test software (from here referenced as the “user”) was at the work location during normal weekdays (Mo-Fri, 8:45-17:00) and was at the home location during times not within this timeframe. We call this behavior the standard user behavior. The device that ran the test software never left the city of Enschede during the two weeks of running. During these two weeks, there were two days that the user deviated from the

standard behavior pattern. This was on Monday and Wednesday of the second week. During these days, the device was at the home location instead of the work location.

Characteristics dependency

We looked at the combination of the host and its characteristics. We observed that each of these showed dependency on user location, dependency on network / server load or no dependency at all. We did not observe any host that showed evident dependency on both user behavior and network behavior. In figure 30, we see an example of both concepts; dependency on user location and dependency on network / server load. Poland and Utwente NL show dependency on user location while Kaiserslautern DE shows dependency on network / server load. Again, these are just observations; the main purpose of this experiment was to collect data and not to research the nature of the observed patterns.

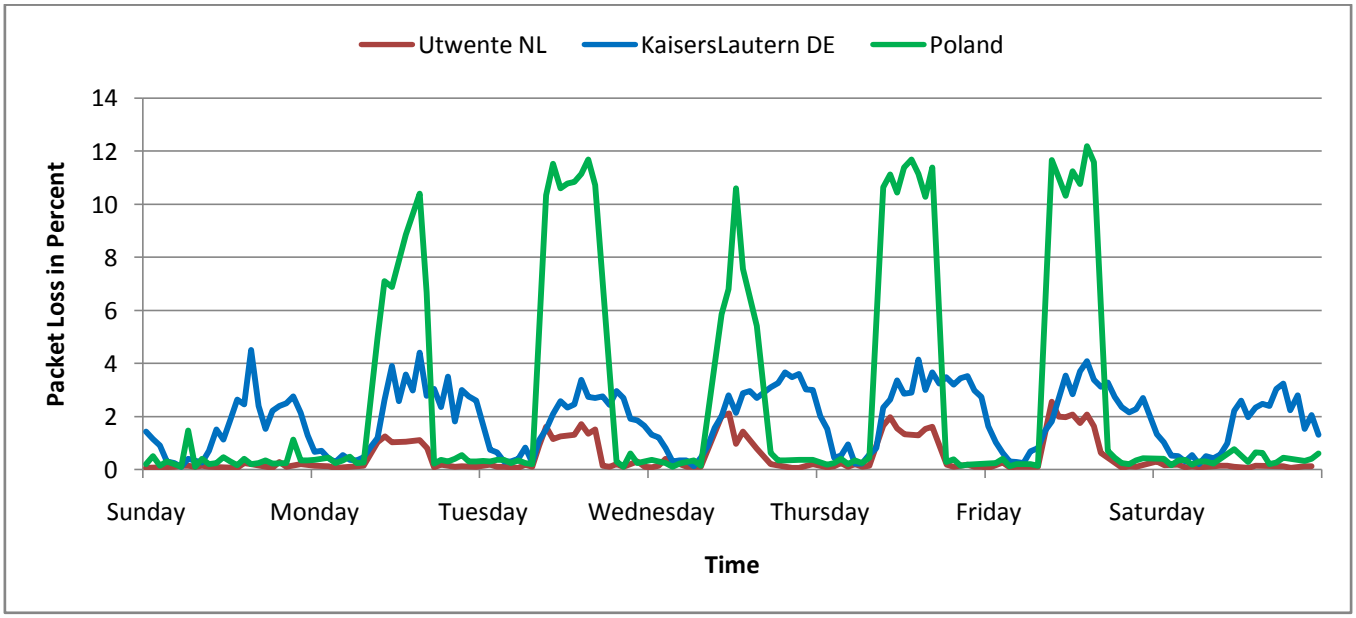


Figure 30: Average Packet Loss on 802.11g over two weeks

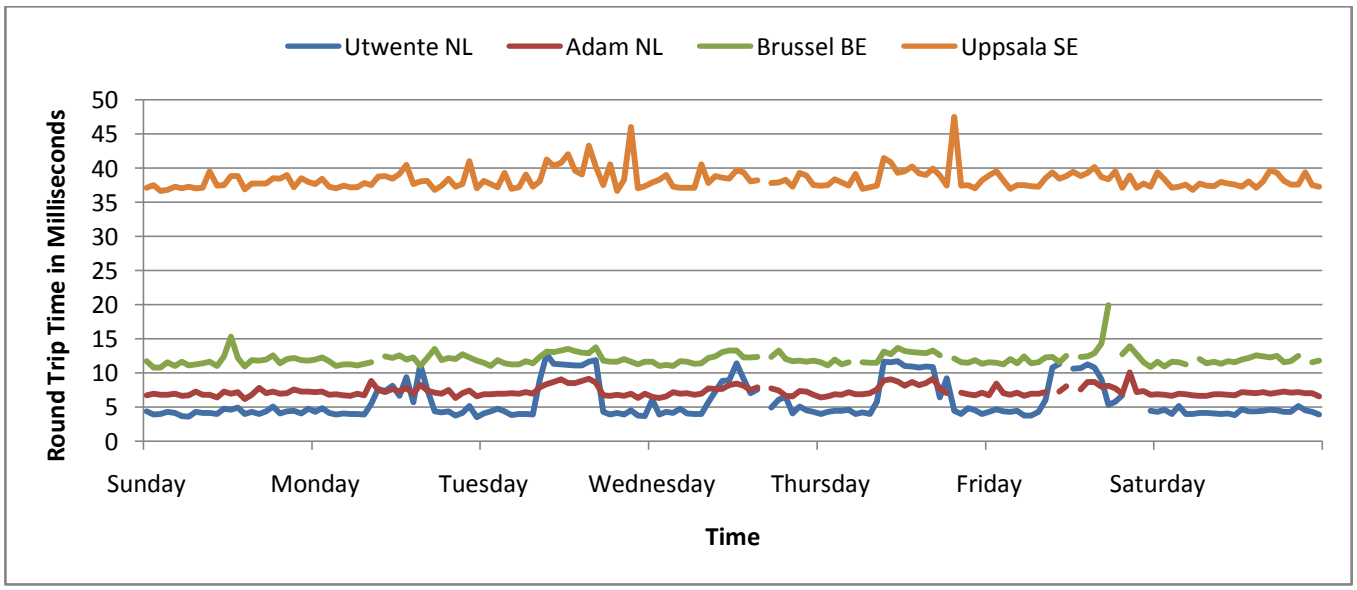


Figure 31: Average round trip time on 802.11g over two weeks

Dependency on user behavior

Consider figure 31. In this figure, we see that the round trip time of the server Utwente NL shows different values at the time the user was at the work location. There is a small increase in round trip times as well for Amsterdam NL and Brussel BE although this increase is marginal.

The throughput and RTT figures of the cellular line interface (UMTS or GPRS, figure 32) show even more dependency on user behavior. Although it can be seen that the geographically far hosts show a higher RTT (figure 32), the location of the user has much more effect on characteristics measured on the cellular line interface. The big decrease in RTT / increase in throughput are because at the work location, the device had an UMTS connection while at the home location, only a GPRS connection was available. In figure 33, one can observe the deviation of the user from his standard behavior. We notice dips during the days that the device was at home location instead of work location (figure 33, on Monday and Wednesday).

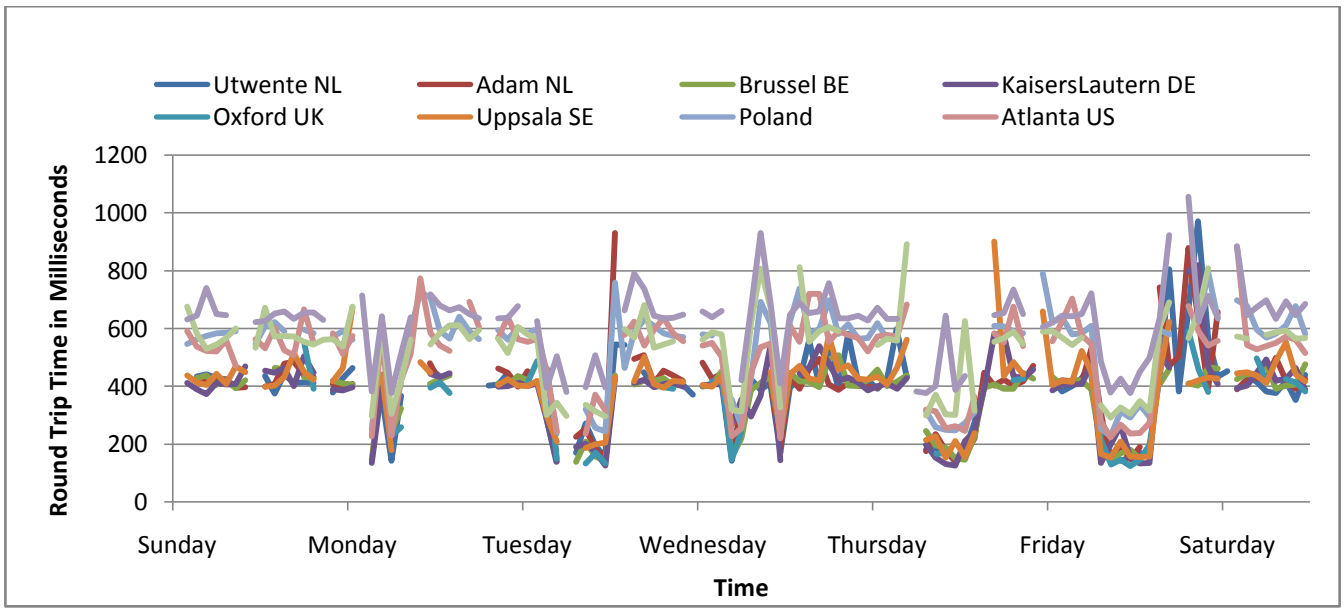


Figure 32: Average round trip time on Cellular Line over two weeks

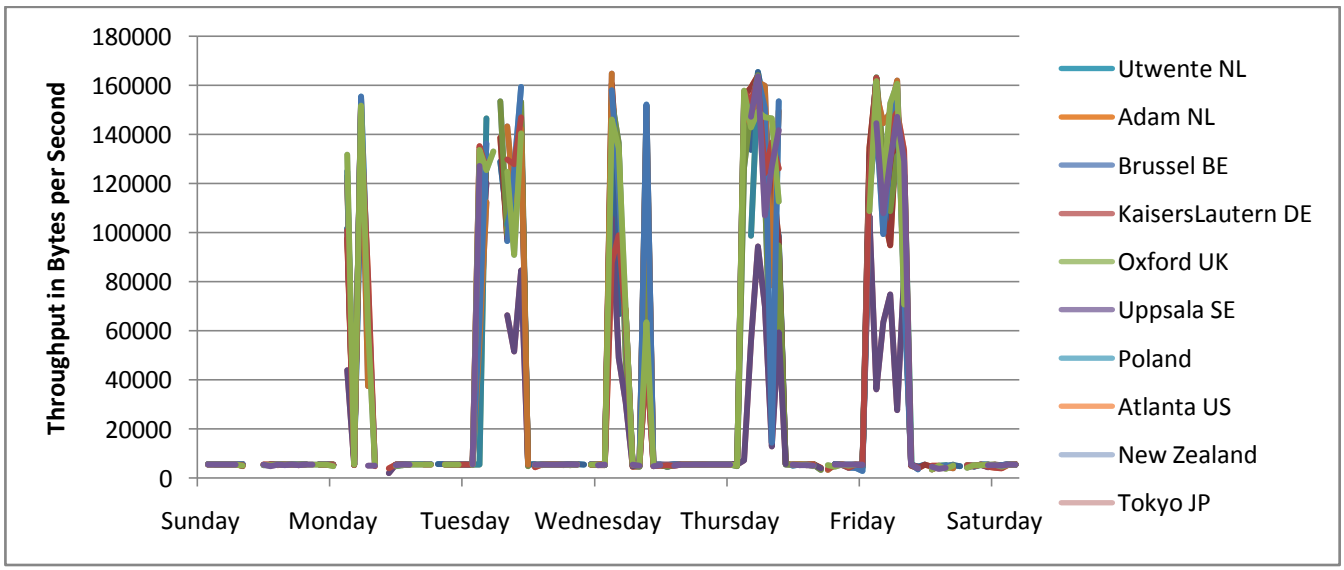


Figure 33: Average Throughput on Cellular Line over two weeks

Dependency on server / network load

In figure 34, we see an example of dependency on network / server load. Kaiserslautern DE showed daily decrease and increase in throughput (the throughput plot resembles a sine wave). This means that for some characteristics in combination with a host, the characteristics were more dependent on server load or network load. It is interesting to note that due to this phenomenon the geographically far server in New Zealand sometimes was faster than the geographically close server in Germany was (figure 34).

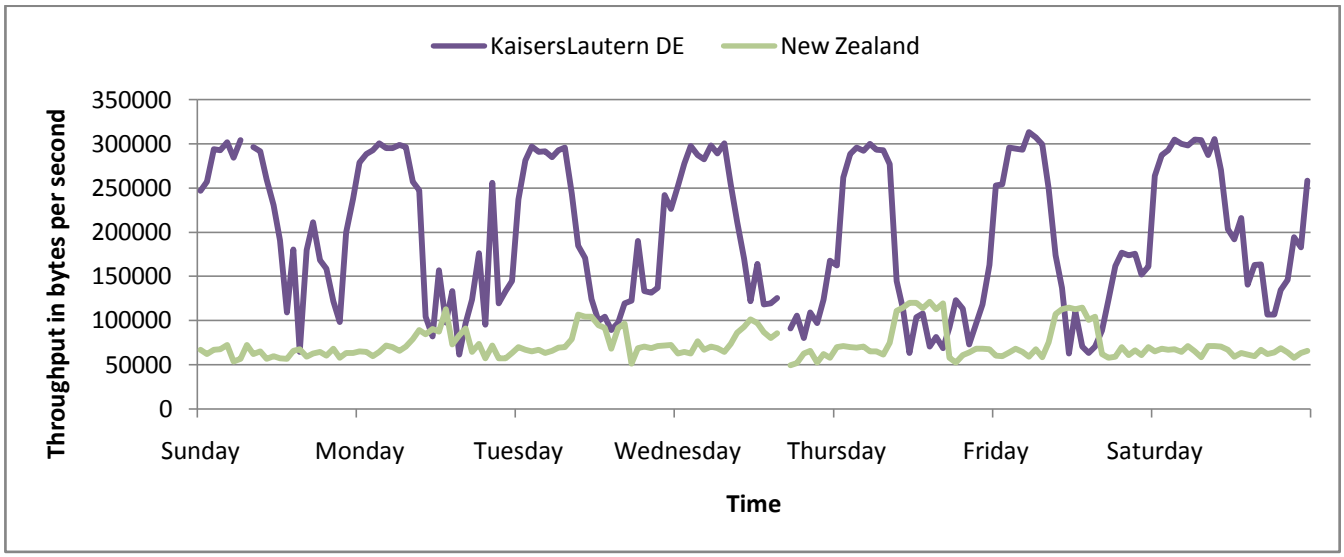


Figure 34: Average Throughput on 802.11g over two weeks

No dependency

On some hosts, characteristics did not show much change over the overall course of two weeks. Take for example figure 31. From the plot of Uppsala SE, one cannot determine whether the user was at the home location or not. A pattern of overall server load is not evident either (at least, not as evident as it is in figure 34). This behavior is even clearer for cellular line technology. Consider figure 35. The plots show inconsistent behavior (no clear pattern).

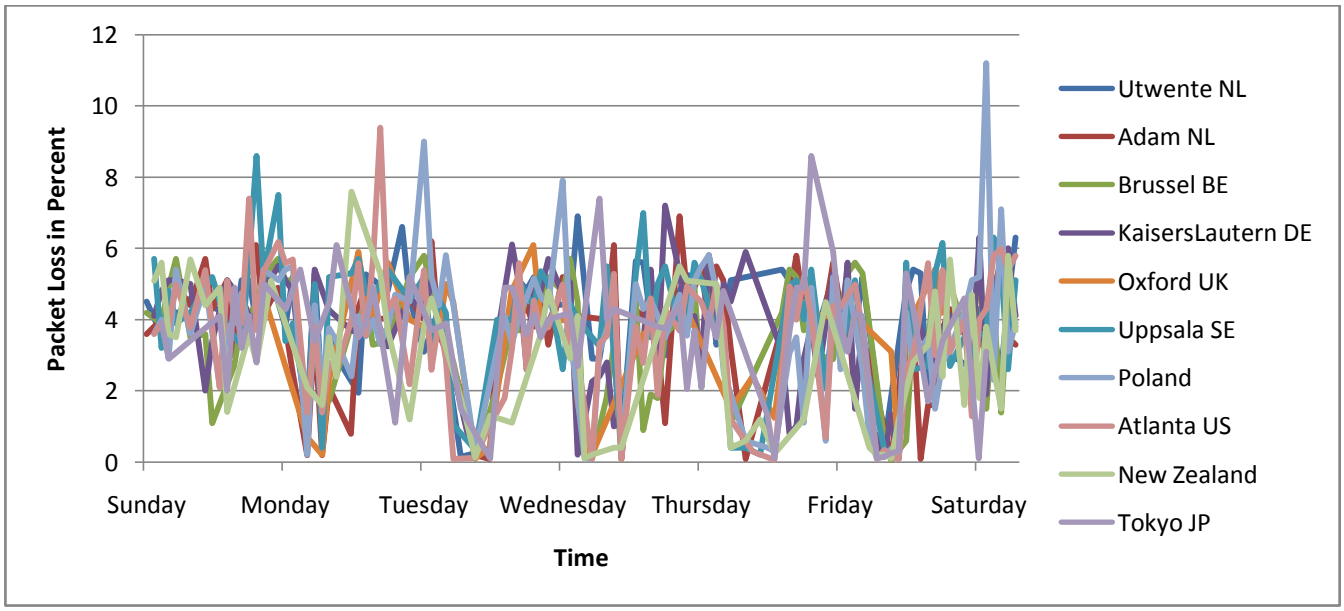


Figure 35: Average Packet Loss on Cellular Line over two weeks

6.2. Results Second Experiment

Our second experiment was to use the measurements we have accumulated in phase 1 as predictions to our measurements in this experiment. We did this to verify the accuracy of our predictions. In this experiment, the user did not deviate from the standard behavior pattern. The user went to work for 5 days and was at the home location during the evenings and the weekends. Because in the second week on Monday and Wednesday of the first experiment the user was not at work, we see inaccuracies in the predictions of Monday and Wednesday. One setback was the unavailability of an 802.11 internet connection on Sunday, therefore, Sunday is missing on the 802.11 plots. Due to unavailability of the device we used in the first experiment, we changed the device for the second experiment. The type of the device was exactly the same. We made sure that the conditions on the device were the same; we exactly copied all the settings from the other device to this different device. We will discuss six plots: one for each separate combination of interface and measured characteristic. Each plot depicts the Predicted value from the UFNET framework and the Real value (measured at the time) (the measured value after the download sequence completed) and the absolute difference in prediction and measured value. We will discuss each plot and will provide accuracy information for each of them. The average errors we present are the averages of the green line (average absolute prediction errors in value and percentage).

Round Trip Time on 802.11

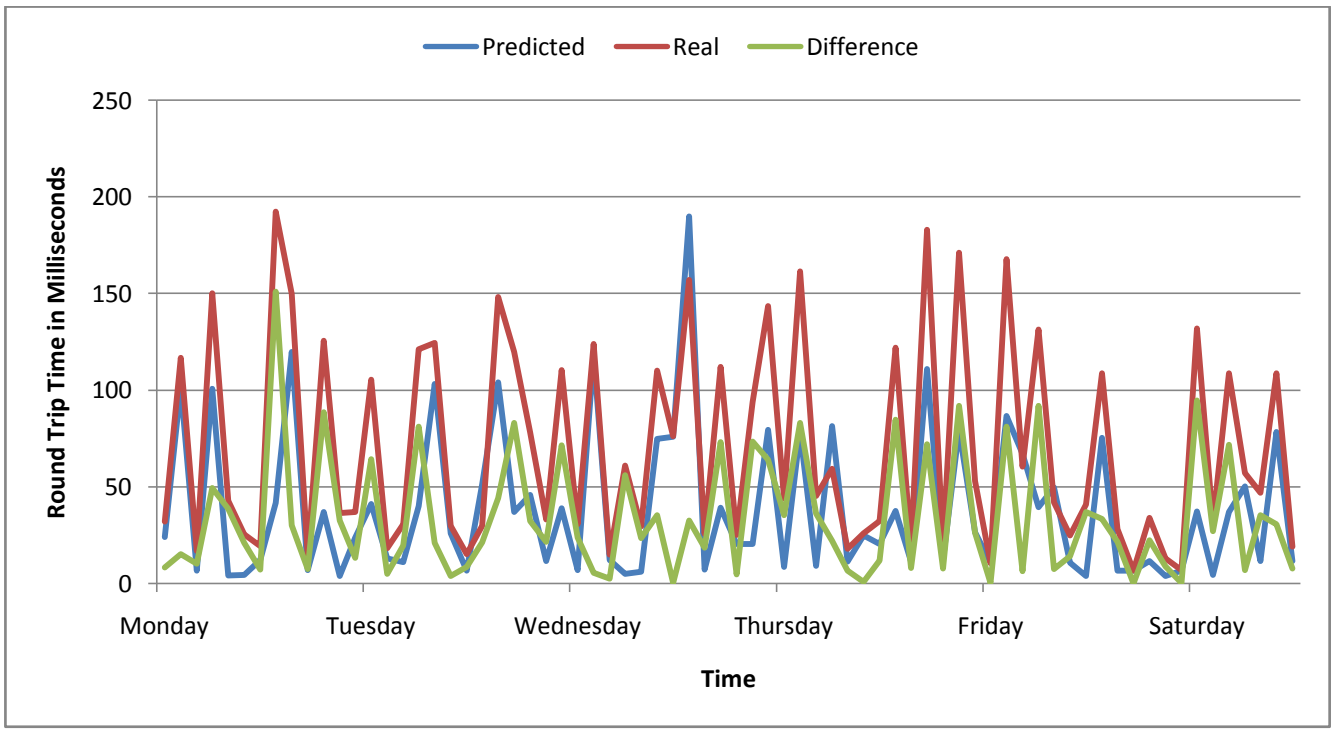


Figure 36: Predicted vs. Real values RTT 802.11

Consider figure 36. We see that the prediction follows about the same trend as the real value, but the real value it is substantially higher.
The average prediction error was 34 milliseconds. The real values show a dramatic increase over the averages. Therefore the average prediction error percentage is rather high namely 183%. The average of the prediction samples over all hosts is E=38,8 milliseconds and the

standard deviation is σ= 38,2. The average of the measured samples (real) is E=70,4 milliseconds and the standard deviation is σ=53,0 milliseconds.
We do not have an explanation of this increase in RTT. The Utwente server, for example, showed a dramatic increase of Round Trip Time. The average round trip time in the database of the Utwente server was 6.3 ms, however the newly measured values showed an average of 36,1 ms. The use of a different device for this experiment could have caused this, or increased network traffic could have caused this. It is striking however, that this change is only observed for RTT on 802.11. If we reduce the real measured values to 20% of their value, we get an average error of 7ms and an average error percentage of E= 58%. This shows there was a condition (on the device or on the network) that increased the RTT by 80%.

Round Trip Time on Cellular Line

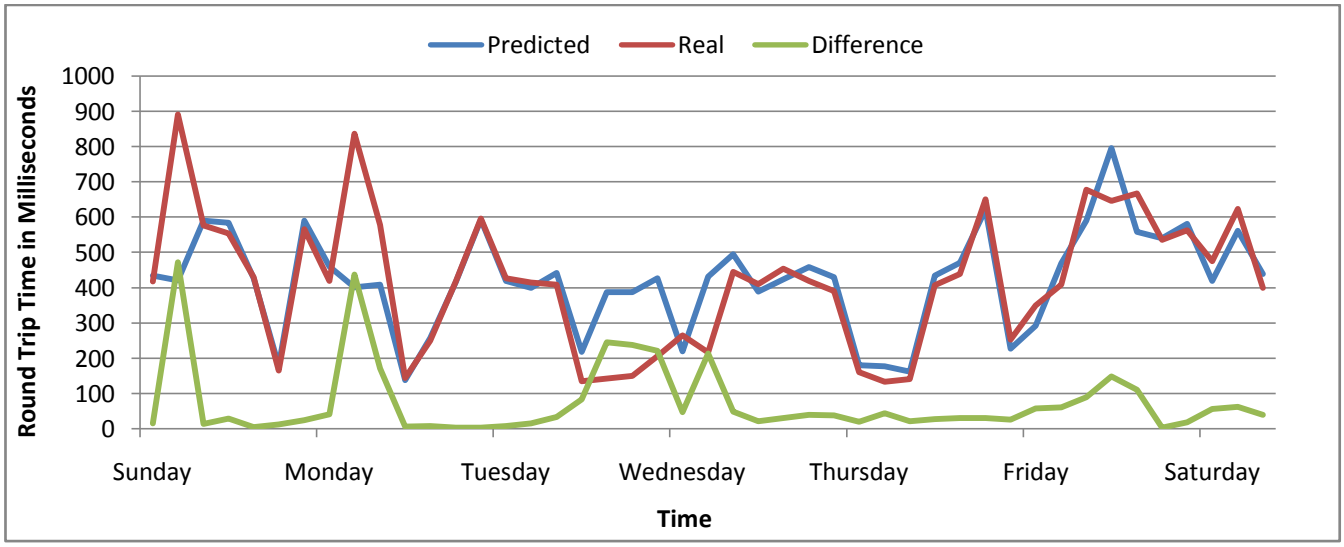


Figure 37: Prediction vs. Real values RTT Cellular Line

This figure depicts the much better prediction over all different hosts on RAS than on 802.11. The average prediction error was 76ms. The average prediction error percentage was 18,8%. The average value for the predicted samples was E= 425,3 milliseconds and σ= 144,7 and for the real values E= 427,5 milliseconds and σ= 196,1. Note that the values are smaller because the measured values of RTT are higher, therefore an increase of 30 milliseconds does not have as much impact as it has on 802.11. This is explained by the high standard deviation. However, the figure shows consistent values, and although the standard deviations are high, we still have quite a good prediction, given the high variation in the RTT values of the cellular line interface. Note the prediction error on Monday and Wednesday, which is due to the user deviating from the standard pattern in the second week.

Throughput on 802.11

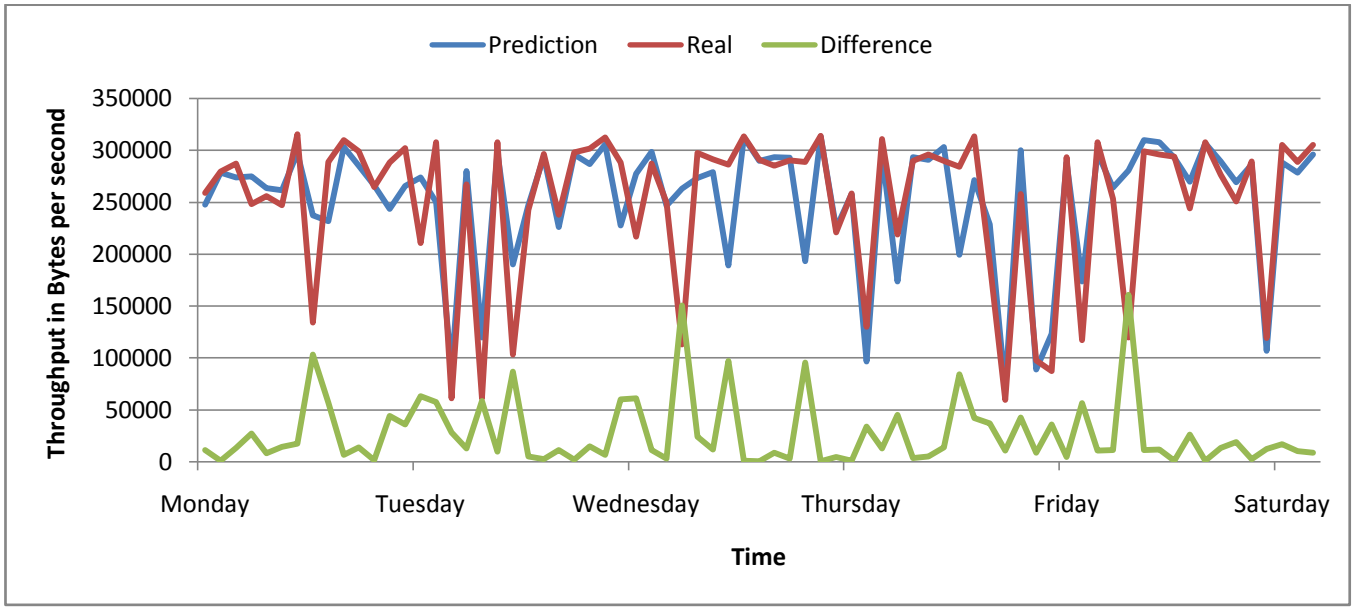


Figure 38: Prediction vs. Real values Throughput 802.11

Contrary to what the plot of the RTT of 802.11 shows, we see a very consistent plot in figure 38. The overall predictions were much better as in the case of RTT .The average prediction error was 27408 bytes/second and the average prediction error percentage was 12,8% (the lowest of all measured characteristics). As for the averages, the average of the prediction samples was E= 253192 bytes per second with σ= 52989 and the average for the real samples was E= 250423 bytes per second with σ= 67615).

Throughput on Cellular Line

Consider figure 39. The average error is 16107 bytes / second and the average error percentage is 91%. Prediction samples average is E= 35748,5 with σ= 55906 and the average of the real samples is E= 38186,5 with σ= 56554 The Standard deviation is bigger than the average because there is a very big difference in bytes per second between GPRS and UMTS (150kb/s vs 5kb/s). The peak drop of predictions on Wednesday is due to inconsistent measurements as is seen in the RTT graph of Cellular Line. Although on Friday, the user did have a UMTS connection, there still is a large error on the real value and the predicted value. This could be caused by temporary network congestion since the UMTS measurements done on that day where all done to the same end-host. If we correct our readings for these errors (there were two errors of more than 1100% and one of more than 500%) then we get an average prediction error of 13130 and an prediction error percentage of 21,7%.

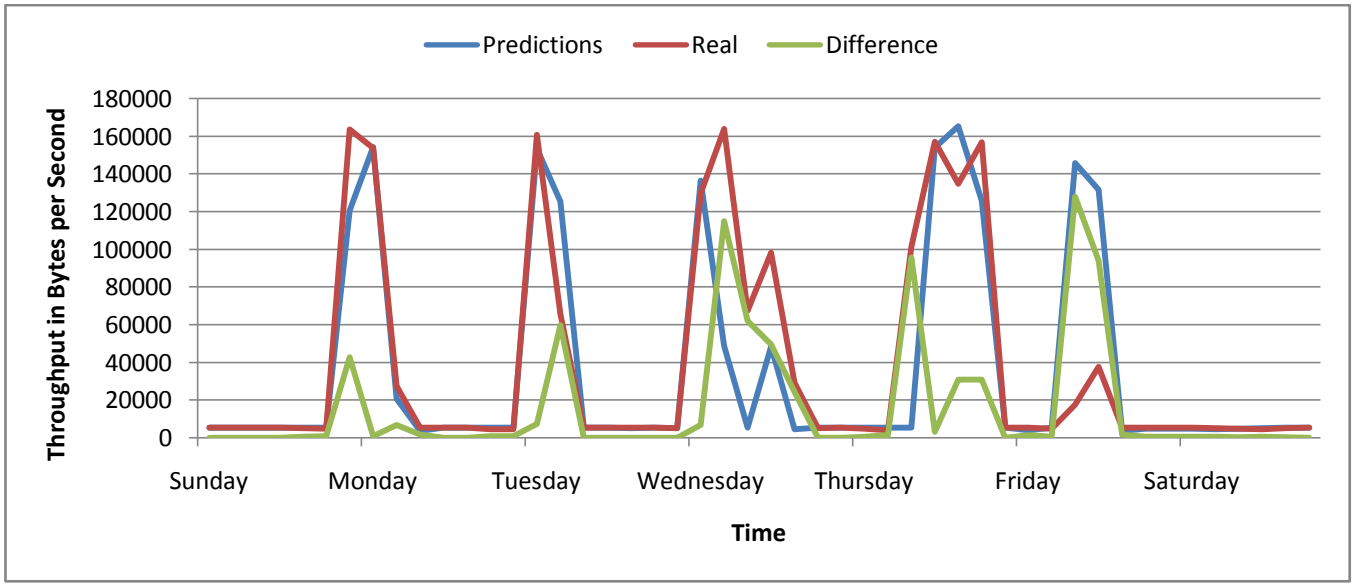


Figure 39: Prediction vs. Real values Throughput Cellular Line

The figure below (figure 40) shows a zoom in on the bottom area of the plot. We see that there is still some difference, but the predictions are still quite accurate.

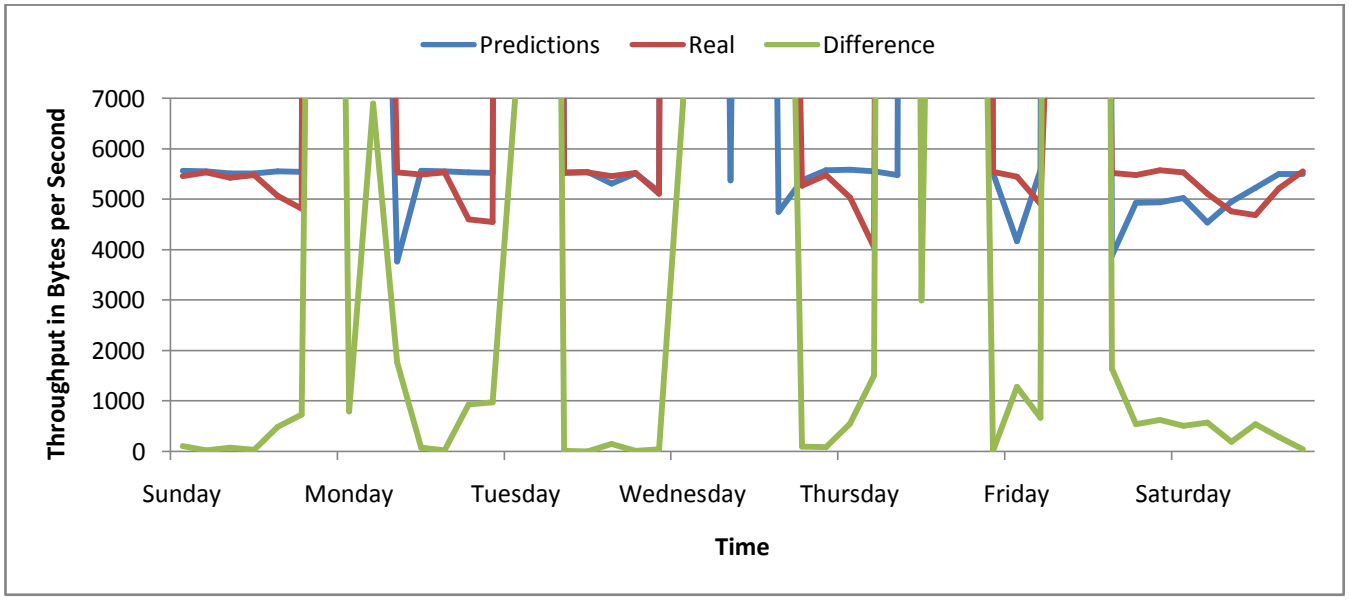


Figure 40: Prediction vs. Real values Throughput Cellular Line 2

Packet loss on 802.11

The average prediction error was 0,64% packet loss (figure 41 on the next page). The average error percentage was 66% (note the percentage of lost packets vs. error percentage). The average over all prediction samples was E= 1,9% with a standard deviation of σ= 2,7. The average over all Real samples was E= 2,1% with a standard deviation of σ= 3,1.
Although the prediction error shows 66%, compare this with the error of 0,64%. This explains why the error rate is large, because packet loss values are very small.

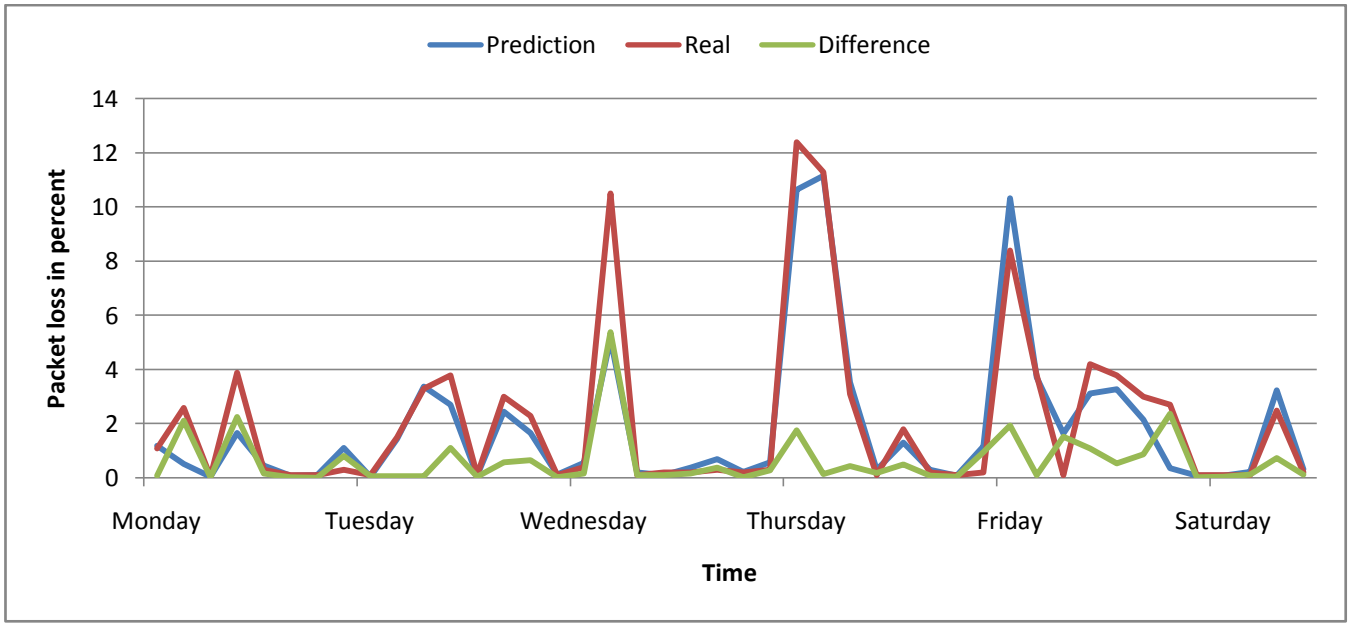


Figure 41: Prediction vs. Real values Packet Loss 802.11

Packet loss on Cellular Line

This plot of packet loss on the cellular line interface (figure 42) shows the least consistent values. The average prediction error was 1,86% and the average error percentage of each value was 100,3% Prediction samples E= 3,7% σ= 1,3 Real samples E= 2,9% σ= 1,4. Also considering the other plots of this section, we can state that packet loss value on the cellular line interface is the least predictable of all the characteristics. We expected this since the packet loss characteristic on the cellular line interface did not show any pattern (see figure 35 in the previous section).

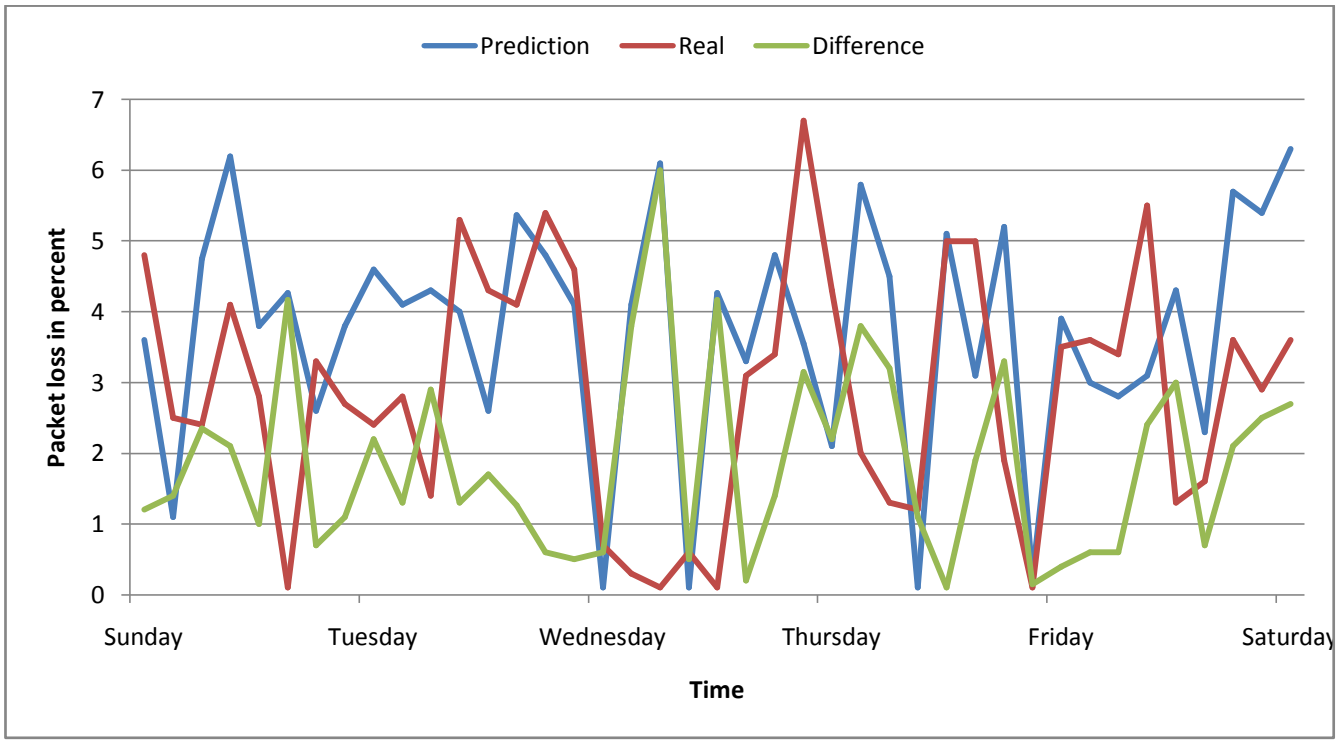


Figure 42: Prediction vs. Real values Packet Loss Cellular Line

Here we present a table with our prediction results.

Interface	Characteristic	Error	Percentage Error	Predicted average and SD	Real average and SD
802.11	RTT	34ms	183%	E=38,8ms σ=38,2	E=70,4ms σ=53,0
802.11	TP	27408b/s	12,8%	E=253192b/s σ=52989	E=250423b/s σ=67615
802.11	PLS	0,64%	66%	E=1,9% σ=2,7	E=2,1% σ=3,1
Cellular Line	RTT	76ms	18,8%	E=425,3ms σ=144,7	E=427,5ms σ=196,1
Cellular Line	TP	16107b/s	91%	E=38186,5b/s σ=55906	E=38186,5b/s σ=56554
Cellular Line	PLS	1,86%	100,3%	E=3,7% σ=1.3	E=2,9% σ=1,4

7. CONCLUSIONS AND FUTURE WORK

Conclusions

In our work, we have examined the possibilities to the predictability of network characteristics on mobile devices using measurements done in the past. As we have shown in chapter 6, most of these characteristics are predictive due to their cyclic patterns; we observed that these show dependency on network / server load or dependency user location. Predictability got as good as an average error percentage of 12,8% on throughput predictions of 802.11 and as bad as an average error percentage 183% on RTT predictions of 802.11. Although we only did this experiment with one person, it means that there is a possibility to predict network characteristics using historical data, on the condition that the environment of the mobile device does not change drastically. This immediately shows the biggest flaw in using past measurements to predict network characteristics: as soon as there is a change in the characteristics patterns, the predictability of the network characteristics drop.

The framework we designed solves the issues we described in the second chapter of this report. Network applications that would make use of the features of this framework would benefit from the up-front estimations, since they would be able to adapt to network conditions. These applications include VoiP software or media streaming software. It would allow the software to select the best quality settings, giving the user the best quality available. The current framework can be used by applications to measure the network characteristics during usage. Since the operating system runs all the time on the mobile devices, for the best results, the framework should be incorporated within the operating system, making it possible to provide up-front estimations to all applications that want information about the network conditions.

However, for this solution to be optimal, network conditions should be measured all the time and not just only when the user uses the network possibilities of the mobile device. With the current devices, this is not possible, since this would put a huge strain on the battery use. With the first experiment, even with the steps we took to limit battery usage we had to charge the battery every night, since it would not last for 24 hours.

Future work / recommendations

Since we did not have enough time to implement sophisticated methods of deriving network characteristics using the packet traces we collect, it is possible to improve the accuracy of the network predictions by improving the methods that derive network characteristics. We only stored network characteristics by the interface they were collected and which time period. To improve predictions, it is possible to store information about the conditions they were captured (e.g. connected network, signal strength). To give a better view on the quality of the predictions, a method should be implemented to assign a value judgment to the predictions. We only had time to use a correction value, which is updated every time new characteristics are stored in the database. However, we did not have enough time to verify whether this is good indication of the quality of the predictions.

To get a better view on the network quality, more methods should be implemented that derive network characteristics from the packet traces. For example, jitter or number of hops to a host.

Finally, a solution should be found to the issue that appears as soon as the network characteristic patterns change drastically. One could implement the solutions as we described them in the initial state section of chapter three. However, more research is necessary to cope with this issue.

8. REFERENCES

[1] Go Mobile with Skype, http://www.skype.com/download/skype/mobile/ , *retrieved from skype.com in april 2007*

[2] J. Gozdecki, A. Jajszczyk and R. Stankiewicz, "Quality of service terminology in IP networks", *Communications Magazine, IEEE*, vol. 41, no. 3, pp. 153-159, 2003.

[3] K. Mase, T. Kuribayashi and A. Tsuno, "A dynamic server selection method using QoS statistics" *Electronics and Communications in Japan (Part I: Communications)*, vol. 87, no. 7, pp. 43-54, 2004.

[4] B. S. Manoj, M. Rajesh and R. R. Ramesh, "SEBAG: A New Dynamic End-to-End Connection Management Scheme for Multihomed Mobile Hosts" *7th International Workshop of Distributed Computing* (IWDC 2005) pp. 524-535, 2005.

[5] S. Kashihara, T. Nishiyama, K. Iida, H. Koga, Y. Kadobayashi and S. Yamaguchi, "Path selection using active measurement in multi-homed wireless networks", *Proceedings of International Symposium on Applications and the Internet 2004* (SAINT'04) 2004, pp. 273-276.

[6] L. Amini, A. Shaikh and H. Schulzrinne, "Issues with inferring Internet topological attributes", *Computer Communications*, vol. 27, no. 6, pp. 557-567, 2004.

[7] Network Driver Interface Specification, http://www.microsoft.com/whdc/device/network/ndis/default.mspx , *retrieved from Microsoft.com in April 2007*

[8] Netlog, http://msdn2.microsoft.com/en-us/library/ms886701.aspx , *retrieved from Microsoft.com in April 2007*

[9] A. Peddemors, I. Niemegeers and H. Eertink, "An extensible Network Resource Abstraction for Applications on Mobile Devices", *Proceedings at the Second International Conference on Communication System software and Middleware. COMSWARE 2007*, Jan. 2007

[10] Controlling the Radio Devices, http://www.teksoftco.com/articles/article%20007/radiodevices.htm , r*etrieved from teksoftco.com in April 2007*

[11] "NetGeo" Verifia Inc., http://www.netgeo.com/index.htm, *retrieved from netgeo.com in March 2007*

[12] N. P. Venkata and S. Lakshminarayanan, "An investigation of geographic mapping techniques for internet hosts", *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 173-185, 2001.

[13] J. Lucio Maia and S. Donizetti Zorzo, "Socket-masking and SNMP: a hybrid approach for QoS monitoring in mobile computing environments" in *Proceedings. 22nd International Conference of the Chilean Computer Science Society, 2002.* pp. 106-114, 2002

[14] J. Hao and D. Constantinos, "Passive estimation of TCP round-trip times", *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, pp. 75-88, 2002

[15] P. Benko and A. Veres, "A passive method for estimating end-to-end TCP packet loss", *Global Telecommunications Conference, 2002.* (GLOBECOM '02), vol. 3, pp. 2609-2613, 2002

[16] Wireshark, http://www.wireshark.org/ , *retrieved from Wireshark.org in April 2007*

[17] OpenNETCF Smart Device Framework 2.0, http://www.opennetcf.org/PermaLink.aspx?guid=e014642d-b028-451a-9cbd-fea5ab140462, *retrieved from OpenNETCF.com in April 2007*

[18] Donald E. Knuth, "The Art of Computer Programming, volume 2: Seminumerical Algorithms", 3rd edn., p. 232. Boston: Addison-Wesley, 1998.

[19] Ubuntu|Download ubuntu, http://www.ubuntu.com/getubuntu/download, *UBUNTU May 2007*

[20] J. Postel, J. Reynolds, "File Transfer Protocol (FTP)", *IETF Request for Comments 959,* October 1985

[21] M. Mirza, J. Sommers, P. Barford, X. Zhu," A Machine Learning Approach to TCP Throughput Prediction", *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (SIGMETRICS '07), pp. 97-108, 2007

[22] ScriptCenter Tools: Log Parser 2.2, http://www.microsoft.com/technet/scriptcenter/tools/logparser/default.mspx, as viewed on June 2007

[23] R. Chakravorty and I. Pratt, "Performance Issues with General Packet Radio Service", *Journal of Communication and Networks* (JCN), vol. 4, no. 2, pp 266-281, 2002

[24] M.C. Chan and R. Ramjee, "TCP/IP Performance over 3G Wireless Links with Rate and Delay Variation", *Journal of Wireless networks*, vol 11, no1-2, pp 81-97, 2005

APPENDIX 1

Netmon Collector Class Diagram. A detailed class diagram of the Netmon Collector module within the UFNET framework.