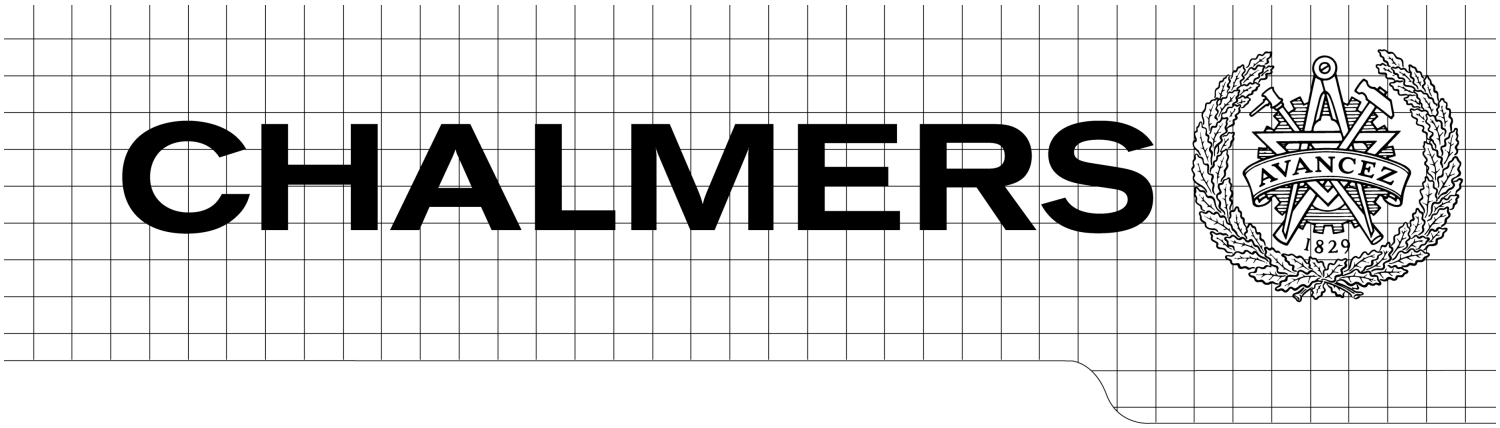

BETULA HUC WITH TERMINAL MODE

A Terminal Mode implementation and evaluation

Master of Science Thesis Networks and Distributed Systems

ERIC ERLANDSSON
DAVID JOHANSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

Betula HUC with Terminal Mode
A Terminal Mode implementation and evaluation

Eric Erlandsson,
David Johansson

Examiner: Peter Lundin

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2011

Preface

The proposal for this thesis project was made by **Mecel**. They wanted to have a thesis project about Terminal Mode to learn more about the concept and to prepare themself if they were to construct their own Terminal Mode client. We are very grateful to Mecel for having us as their thesis workers. Mecel has supported us through our work and have provided us with the necessary equipment to do a good thesis project.

Especially we want to thank **Magnus Alinder** and **Jakob Hjelmåker** at Mecel. Magnus was our supervisor, he gave us guidance and helped us with any questions that we might have. Jakob Hjelmåker helped us in getting a Terminal Mode prototype device from Nokia.

We also want to thank our examiner at Chalmers, **Peter Lundin**. Throughout the thesis project he has seen to that we have been on the right track and he have given valuable feedback on the report.

Abstract

This report deals with the concept Terminal Mode, which is a recently developed standard by Nokia and Consumer Electronics for Automotive for connecting mobile phones to displays and input devices. The concept aims to be an industry standard for integration of mobile applications with the car environment. The master thesis project will implement a Terminal Mode client and the report will deal with the different parts needed to create a Terminal Mode client. Terminal mode is capable of using different physical transport methods but in this report only USB is considered, the reason for this was that the not fully developed Terminal Mode server prototype phone provided by Nokia only supported USB. FPS measurements were made but the results were not satisfying because there was a limitation in the prototype device. The report ends with a deep analysis of the standard. The analysis main focus is on security mechanisms in the protocol but also other systems with the same purpose as Terminal Mode is analyzed and compared. The outcome of the analysis was that Terminal Mode is secure if the security mechanisms are used correctly and that Terminal Mode can become a leading standard for enabling apps in the car. The big question is if car manufacturers will trust the classification of applications made by the smartphone manufacturers and if the customers want to pay for these features.

List of Abbreviations

A2DP Advanced Audio Distribution Profile

DHCP Dynamic Host Configuration Protocol

FPS Frames Per Second

HFP Hands-Free Profile

MTM Mobile Trusted Module

N8 Nokia N8 Terminal Mode Prototype Device

PAN Personal Area Network

PCR Platform Configuration Register

RLE Run-Length Encoding

RTP Real-time Transport Protocol

TCP Transmission Control Protocol

TPM Trusted Platform Module

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

VNC Virtual Network Computing

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

This master thesis is about the concept Terminal Mode [1]. Terminal Mode is a proposed industry standard for the integration of mobile applications into the car environment [2].

A Terminal Mode client used for demonstration together with a Nokia N8 Terminal Mode prototype device has been implemented. The implementation process and the results will be described in the report. Furthermore an analysis of the Terminal Mode concept has been performed.

The following section will give a brief introduction to why Terminal Mode is needed, what the purpose of this thesis is and the limitations of the master thesis project.

1.1 Background

Mobile phones have improved greatly the last years. The smartphones produced today usually has a music player, internet radio, GPS, web browser and the possibility to buy other applications from app stores. At the same time there is a growing demand for infotainment systems in cars which would allow users to use some of these features.

Such an infotainment system can be built in several ways. One way is to construct a system with functionality similar to a smartphone directly into the car. Another way is to integrate the smartphone in the cars infotainment system so that all the functionality that is available in the smartphone will be available in the car. It is this second approach that Nokia chooses in their newly proposed industry standard Terminal Mode.

With Terminal Mode it is possible to view the smartphones display on the cars screen. Furthermore the smartphone is controlled by the cars key input, either by a touch sensitive screen or by buttons. Terminal Mode also enables the smartphone to send audio to the car. So in many ways it is just as if you would build in the smartphone into the cars dashboard. However there are several advantages with Terminal Mode compared to a smartphone built into the dashboard. One is that the smartphones display can be scaled to a larger resolution which fits the cars screen. There are also mechanisms which make it possible for the car manufacturer to only allow some content from the smartphone in the cars infotainment system.

Mecel is a systems and software development company which develops solutions for the automotive industry [3]. They have a product called Mecel Betula SDK which is a Bluetooth platform that simplifies the implementation of Bluetooth products in automotive systems. Mecel Betula HUC is an application built on this SDK which shows how the integration of Mecel Betula SDK could look in an infotainment system. Mecel wants to integrate Terminal Mode into Mecel Betula HUC.

1.2 Purpose

The purpose of this thesis project is to develop a Terminal Mode client integrated into Mecel Betula HUC that can be used for demonstrations, as well as to test the Terminal Mode concept. The Terminal Mode client should be developed according to the standard so that it can work with any smartphone supporting Terminal Mode. However it will only be tested together with a Nokia N8 Terminal Mode prototype device. An evaluation of the Terminal Mode concept will also be performed. The evaluation will go through different parts of the Terminal Mode standard, it will be compared against other similar concepts and the authors will give their opinions about the concept in general.

1.3 Scope

The finished program should work on a test system which consists of a PC with Windows XP installed and a smartphone supporting Terminal Mode. It will not be implemented in a real environment such as in a car or on specialized hardware for that use.

The Terminal Mode client will not support all extension messages. It will be developed to work together with the Nokia N8 Terminal Mode prototype phone. Therefore services not supported by the Nokia N8 Terminal Mode prototype phone will not be implemented. Device attestation and security mechanisms will not be implemented in the Terminal Mode client, however a deeper analyze of these parts will be performed. Support for audio will not be implemented, however it

Figure 1: An example how a integration into Mecel Betula HUC can look like. The image has been modified.

is still possible to pair Mecel Betula HUC with a Bluetooth device and transfer audio outside of Terminal Mode. Mecel Betula HUC supports the Bluetooth profiles HFP and A2DP which makes it possible to use audio from the Nokia N8 prototype phone in it.

2 Theory

The Terminal Mode standard is built upon several open standards such as VNC[1] (Virtual Network Computing), RTP (Real-time Transport Protocol) and UPnP[2] (Universal Plug and Play). RTP is used for audio and since audio will not be implemented further description of it will be unnecessary, however an analysis of when RTP should be used instead of Bluetooth for audio is done in section 6.7. To fully understand the implementation developed in this master thesis, and also to understand the reasoning in the results and analysis section, the reader is encouraged to read this section. The VNC, UPnP and the Terminal Mode standard will be explained (but not fully described in detail since the standards is available to download). Also image resizing and colour models used by the implementation will be described.

2.1 Terminal Mode Specification

This section will briefly describe some parts of the Terminal Mode Specification version 1.0 [1] which is used by the implementation.

2.1.1 Network setup

Terminal Mode requires that underlying systems provides IP connectivity between the Terminal Mode server and the Terminal Mode client. But despite that the communications at lower layers are not performed by Terminal Mode, the Terminal Mode specification still specifies some rules which must be followed.

The client and server must support USB version 2.0 or higher. Furthermore they must support the USB driver CDC/NCM, but also support for other drivers is optional. A good thing with CDC/NCM is that it offers high transfer rates. Other transport methods such as WLAN and Bluetooth can be used as well. Theoretically it is possible to use any protocol as long as they carry IP packets and deliver enough bandwidth to support Terminal Mode.

No matter which transport method that is used the Terminal Mode server must have a DHCP server. The DHCP server must assign an IP-address to the DHCP client on the Terminal Mode client. The transport method must also support UDP and TCP.

2.1.2 UPnP

Terminal Mode uses UPnP [4] to discover and control devices. UPnP is an open network architecture which allows devices joining a network to communicate and set up network services with each other without manual configuration. UPnP builds on established internet standards and is therefore independent of any operating system or programming language. Some of the underlying standards that UPnP uses are XML, SSDP (which is a protocol used for discovery of network services) and SOAP (which is a XML based protocol that allows applications to exchange information over HTTP [5]).

UPnP consists of three basic building blocks which are devices, services and control points. [6]

A device is a unit on the network, e.g. a network printer. The device has some information about itself such as manufacturer name, model name, serial number and which services it has. The information about a device is specified in XML format and SSDP broadcast messages is used to discover new devices on the network.

A device can have a number of services. The services have state variables which models the services state, a state variable can e.g. be a printer's print queue. These state variables can be controlled with actions. If the state variable is a printer's print queue then there can e.g. be actions for getting information about the jobs in the queue or adding a job to the queue. SOAP messages are used to invoke an action. The services also publish information when a state variable is changed, this is called *eventing*. A device may subscribe to these events.

The third building block is the control point. The control point is capable of discovering and controlling devices. After a control point has discovered a device it first gets a list of its services and fetches its associated service descriptions, then it invoke actions to control it.

[1] VNC is a registered trademark of RealVNC Ltd. in the U.S. and in other countries.
[2] UPnP is a registered trademark of UPnP Implementers Corporation.

There exist a number of usage models in UPnP. These models describe how the devices interact with the control point. Terminal Mode uses a model called the 2-Box Pull Model. In this model the control point and client are integrated into one unit. The Terminal Mode server is a standalone device which can be controlled by the Terminal Mode clients control point. This means that it is the client who discovers the server and invokes actions on it.

UPnP uses standardized device and service descriptions which are documents that defines the specific devices and services. These descriptions specify among other things which services a device provides and which actions the service has. In the Terminal Mode specification the device and service descriptions are given. A Terminal Mode server device is called TmServerDevice. In this device description it is specified that it has two services, a TmClientProfile and a TmApplication-Server. These services must be supported by all Terminal Mode servers, however the clients control point only need to support the TmApplicationServer, support for the TmClientProfile server is optional. If the control point supports the TmClientProfile service then several client profiles can be registered at the server. These client profiles notify the server about client preferences, settings and capabilities.

The TmApplicationServer has a number of actions which are described in table 1. These actions are used to control the Terminal Mode server. The direction IN shows arguments which are used in the call, the direction OUT shows the return values. If a control point does not support multiple client profiles the ProfileID argument must always be set to 0 when invoking actions.

Table 1: TmApplicationServer service with its actions

Action	Direction	Arguments
GetApplicationList	IN	AppListingFilter
	IN	ProfileID
	OUT	AppListing
GetApplicationStatus	IN	AppID
	OUT	AppStatus
LaunchApplication	IN	AppID
	IN	ProfileID
	OUT	AppURI
TerminateApplication	IN	AppID
	IN	ProfileID
	OUT	TerminationResult

2.1.3 Display Output and Control Input

In Terminal Mode VNC are used for two things. To view the Terminal Mode servers display on the client and to control the server from the client by using key input. VNC will be described in the next section, however Terminal Mode specific information about VNC are given below.

Before initiating a connection the Terminal Mode client must know that the VNC server is started on the Terminal Mode server, this is handled by UPnP mechanisms. As soon as the VNC server has been started the VNC client residing at the Terminal Mode client can start a connection which is established by a TCP/IP socket.

In addition to the standard messages in the RFB protocol, Terminal Mode specifies 19 different extension messages. In table 2 information are given about the extension messages which are used in the thesis.

A description of the extension messages in table 2 are given below.

1. *Server display configuration* - Sent by the server during initialization, in response to the Set Encodings message. Contains information about which display configurations that are supported by the server.

2. *Client display configuration* - Sent in response to the Server display configuration message. Contains information about which display configurations that are supported by the client.

Table 2: Used extension messages

Message #	Sender	Receiver	Message
1	Server	Client	Server display configuration
2	Client	Server	Client display configuration
3	Server	Client	Server event configuration
4	Client	Server	Client event configuration

3. *Server event configuration* - Sent by the server to provide information about event mapping. Contains among other things which keys the server support such as device keys, knob keys, multimedia keys and if the server support pointer and/or touch events.

4. *Client event configuration* - Similar message as the server event configuration, but sent by the client to provide information about event mapping.

Run-Length Encoding
Terminal Mode can also use RLE which is a way of encoding a set of pixels and it is of good use when several pixels next to each other on a row are of the same colour. The RLE-encoding would then send information about how many pixels that have the same colour before the pixels colour value is sent. This is good for all colour encodings that don't fill up all bytes with bits for the colour values but instead have some padding bits. See table 6 for how many padding bits each colour encoding have. The padding bits for RGB343 allows a run length of 64, RGB444 allows run length 16, RGB565 needs an extra byte and then allows 256 in run length and ARGB888 allows a run length of 256. The run length sent should be incremented by one, for example a zero sent means one.

2.2 VNC Specification

VNC is a system for remote control of a graphical user interface. The Terminal Mode standard uses VNC to transport the display content from the Terminal Mode server to the Terminal Mode client. VNC uses the RFB (Remote Framebuffer) protocol [7] which is designed to run on a simple client with light weight hardware. Therefor the resource demanding operations of the protocol have been focused to the server side so that the client can be as simple as possible. Published versions of the protocol are 3.3, 3.7 and 3.8. The Terminal Mode standard requires that support for version 3.7 and 3.8 is available.

The different versions have a set of messages that can be exchanged between the server and the client. For flexibility and adoptability the standard also support pseudo encodings that can be additional encodings for the image data or used to define additional messages that can be used by the server and the client. This is used by the Terminal Mode standard to send image data with a new RLE and to send extra information to the server such as device keys pressed on the client, client capabilities and other TM specific messages. In the Terminal Mode standard these messages are called extension messages and it will be discussed more in the section about the Terminal Mode standard.

The following text in this section will describe the connection setup, different messages exchanged between the server and the client, and how the display updates are handled.

2.2.1 Connection Setup and Initialization

The RFB protocol uses TCP over IP for data transport between the server and the client. It uses one socket for each connection and the standard port is 5900, which the server listens on. The client initializes the connection by connecting to this port. After the socket has been set up the initialization and handshaking messages in table 3 are exchanged.

1. *ProtocolVersion* - First message sent from the server to the client containing the highest RFB version the server supports. The message is sent immediately after the client has connected

Table 3: RFB initialization messages

Message #	Sender	Receiver	Message
1	Server	Client	ProtocolVersion
2	Client	Server	ProtocolVersion
3	Server	Client	SecurityTypes
4	Client	Server	ChosenSecurityType
5	Server	Client	SecurityResult
6	Client	Server	ClientInit
7	Server	Client	ServerInit

and the content is 12 bytes of ascii encoded characters which tells the highest protocol version available on the server. The format is "RFB xxx.yyy\n" where xxx describes the version number before the decimal and yyy the version number after the decimal.

2. *ProtocolVersion* - As a response to the ProtocolVersion received from the server, the client sends a message of the same format containing the highest RFB version supported by the client. The response cannot be higher than the version received from the server in the first message and the version sent is the version to follow during the whole session.

3. *SecurityTypes* - The server continues by sending a list of the security types it supports. A number represents each security type.

   First it sends an unsigned byte which tells the client how many security types that will be sent. This is followed by an array of unsigned bytes which each represents a security type. The length of the array is the number indicated in the first byte. If the number of security types is zero this indicates that the connection failed. An array of characters which representing a reason message will in that case be sent to the client.

4. *ChosenSecurityType* - The client decides which of the proposed security type to use and indicates this by sending a single byte to the server containing that security type number. After this message specific data for the chosen security type is exchanged. This however doesn't apply when Terminal Mode uses the protocol since the standard doesn't support any authentication on this level. The only valid security type for Terminal Mode is *1*, which means no authentication.

5. *SecurityResult* - The server sends a single byte with the value *0* if the security handshake was successful or *1* if the handshake was not successful.

6. *ClientInit* - The client sends a single byte that tells the server if it should allow several clients to be connected at the same time or if the server should give the connecting client exclusive access and disconnect the others. A non-zero value means that several clients could be connected and a zero value means that other clients should be disconnected.

7. *ServerInit* - The server responds to the ClientInit with a ServerInit message containing information about the framebuffers width and height, the servers name and the pixel format used by the server. This pixel format will be used when the server sends display updates if the client doesn't specify another format with the SetPixelFormat message.

2.2.2 Display Updates

Each display update is called a framebuffer update. A framebuffer update contains one or several rectangles, which represents a snapshot of a rectangular area in the framebuffer. Each update needs to be requested/pulled by the client by using a framebuffer update request message. This means that the server will not continuously send updates of the framebuffer to the client, it will only send one framebuffer update message in response to one framebuffer update request.

One flag in the framebuffer update request message tells if the corresponding framebuffer update should be incremental or not. If a full update is requested then a full snapshot of the framebuffer will

be sent to the client and if incremental updates is requested the update will only contain rectangles where data in the framebuffer has changed. The update will be sent first when something has been updated in the framebuffer. The framebuffer update response to a framebuffer update request could this way come direct or in an arbitrary time interval.

This design solution of the RFB protocol eliminates the risk of having buffers filled with data received and it will not have old inaccurate data existing in a buffer or on the network. By adjusting the rate of the framebuffer update requests, i.e. by introducing a delay, the client could decrease the needed bandwidth. This could also be done at the server by introducing a delay before sending the framebuffer updates.

Each framebuffer update request message is small and contains the coordinates of a rectangle in the framebuffer that the client want to get updates from. It also contains the flag that marks if it wants a full or incremental update as mentioned above.

Table 4: FramebufferUpdate header message

# bytes	Type	Description
1	U8	Message Type (Value always 0)
1		Padding
2	U16	Number of Rectangles

Table 5: FramebufferUpdate rectangle message

# bytes	Type	Description
2	U16	X Position
2	U16	Y Position
2	U16	Width
2	U16	Height
4	S32	Encoding Type
Depending on encoding type	X	Pixel Data

The content of the framebuffer update message is shown in table 4 and table 5. The first table shows the framebuffer update message header and the second describes the rectangles sent. Number of data with bytes to receive for each rectangle depends on the encoding used and the size of the rectangle.

The RFB protocol defines several different encodings that can be used in the framebuffer update message. The only mandatory encoding however is the raw encoding which all clients and servers must support. The raw encoding sends the pixel data pixel by pixel which requires much bandwidth. The server is not allowed to send in any other format unless the client announces support for it with the set encodings message. The Terminal Mode standard introduces a pseudo encoding which is called RLE. This encoding is described more in the theory section about Terminal Mode.

2.2.3 Messages

Except for the FramebufferUpdateRequest and the corresponding FramebufferUpdate message already described, the RFB standard defines a few other messages that is used by Terminal Mode. The interesting messages are the following.

KeyEvent is sent from the client to the server and each message represents a key press or key release. One byte in the message tells if the key was pressed or released and an unsigned long (4 bytes) represents the keys identifier.

PointerEvent is sent from the client to the server when the mouse is moved and/or clicked in the framebuffer. An array of flags in the message represents the state of the buttons, if they were pressed or not. A pointer event needs to be sent twice to represent a mouse click, one message when the button is pressed and one message when the button is released.

SetEncodings is sent from the client to the server and the purpose of the message is two things.

First it tells the server which encoding formats the client supports. This is the encodings that the server can choose between when generating the framebuffer update messages.

The second purpose of the message is to inform the server about which pseudo encodings the client supports. Two already defined pseudo encodings is the desktop-size pseudo encoding used to change the framebuffer size and the cursor pseudo encoding which tells the server that the client can draw the cursor locally. The standard support different addons in form of pseudo encodings. The client tells the server which possible addons it have support for by using pseudo encodings. However after a client has sent a pseudo encoding it supports it can't assume that the server supports it until it get a confirmation that it does. The server can in this way ignore pseudo encodings that it doesn't support.

SetPixelFormat is sent from the client to the server after the initialization but before the first framebuffer update request. It tells the server how the pixels colour values sent in the framebuffer update messages should be encoded. It does not define how several pixels should be encoded to save bandwidth.

2.3 Image

The following sections will explain how image resizing is done, how different colour models work and how different models can be converted between each other. The reader will need this knowledge to understand how the image received from the Terminal Mode server is processed before displayed.

2.3.1 Resizing

The reason to take up the subject image resizing, or scaling, in this thesis is that image scaling on the Terminal Mode server is not mandatory. If it was mandatory then all image scaling could have been done at the server.

Since Terminal Mode servers can have different screen resolutions the image needs to be adopted to fit the screen size of the client. Without scaling and if the resolution is not the same this could be done either by clipping or framing. Framing is when the resolution on the Terminal Mode server is lower than on the display so a frame is added around the image. Clipping is when the resolution on the Terminal Mode server is higher than on the display so only a part of the image can be displayed.

An unscientific method to scale an image is to go through the image and on calculated intervals duplicate the rows and columns. To understand how scaling is done in a more scientific way we need to think of each pixel as a sample point in the image representing a specific colour. To increase the size of an image the number of samples needs to be increased and the problem to solve is which colour the new samples should have. The same method applies if the image size is decreased but then the number of samples needs to be reduced. Figure 2 shows an example of how new samples will be placed when scaling an image from 4 x 4 pixels to 5 x 5 pixels. The process of estimating sample values when scaling is called interpolation. [8]

It exist several methods for interpolation. Three interpolation methods and another method for image resizing will be explained here. The first two of the following methods have been implemented in the Terminal Mode client implementation in this master thesis and the others are described to give the reader a wider understanding of the subject.

Nearest Neighbor Interpolation is the most simple interpolation technique. Each new sample gets the value from the nearest existing sample in the original image. Because of its simplicity straight edges doesn't look good on the resized image [8]. Images that have been scaled using nearest neighbor interpolation will have pixels duplicated to fill up the missing ones.

Bilinear Interpolation is a better-looking but more computer-demanding algorithm than nearest neighbor for image scaling. If the new samples that needs to be calculated is not exactly where an old sample was (or exactly on the same x-axis or y-axis as the existing samples) it would take a weighted average of the 4 closest samples of the new sample. The new sample will get the colour

value calculated by a weighted average of the old sample values. The closer to an old sample the new sample is the greeted impact of the calculated value the old sample will have and samples far away will have less impact. [8]

Bicubic Interpolation is the interpolation standard used when zooming in photo editors such as Adobe Photoshop and Corel Photopaint. It uses the 16 nearest old samples to calculate the new sample value. This result in higher complexity for the algorithm which requires more computing power but also it generally does a better job at keeping details in the image. [8]

Seam carving is another method of image resizing compared to the interpolation algorithms previously described. The seam carving technique is content aware and uses a seam of connected pixels from top to bottom or from left to right. Inserting or carving out seams of an image can change the image aspect ratio. The seams are identified by an energy function that makes the seams not go through more detailed parts of the picture that should be unchanged. [9]

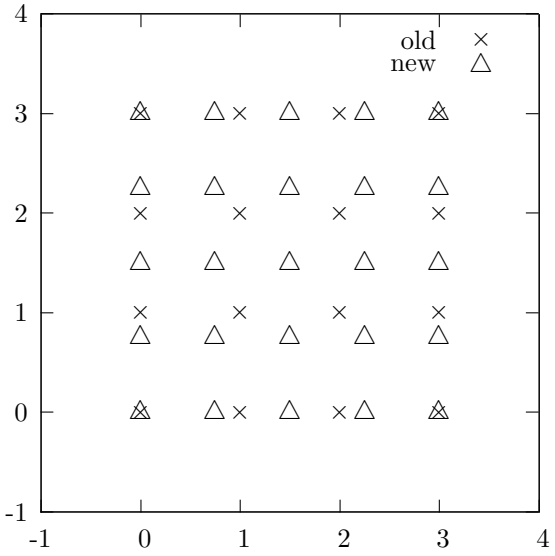

Figure 2: Example of 4x4 image converted to 5x5 image.

2.3.2 Color Models

Representing colours can be done in several different ways. Often they are represented with one value for red, one for green and one for blue. Depending on the capabilities of the monitor and how much space that can be used the different values can be of different sizes. Another aspect of which colour model to use is available bandwidth since a colour model which use fewer bits per colour demand less bandwidth when they are transmitted.

The most straightforward model is ARGB888 which uses one byte for each colour and one byte as padding. See table 6 for different colour encodings and figure 3-6 to see how the pixels are represented in memory.

Table 6: Pixel colour encodings

	Bits for					
Name	red	green	blue	padding	Bits for Colours	Total Bits
ARGB888	8	8	8	8	24	32
RGB565	5	6	5	0	16	16
RGB444	4	4	4	4	12	16
RGB343	3	4	3	6	10	16

Figure 3: 4 bytes representing a pixel of colour model ARGB888

Figure 4: 2 bytes representing a pixel of colour model RGB565

Figure 5: 2 bytes representing a pixel of colour model RGB444

Figure 6: 2 bytes representing a pixel of colour model RGB343

3 Material/Hardware and Methods

This section will describe the hardware and the methods used when the Terminal Mode client was developed in this mater thesis project.

3.1 Hardware

The hardware used in this master thesis project consisted of the following.

- 2x Standard PC laptops
- Nokia N8 Terminal Mode Prototype device
- Cables

The laptops was standard Dell model D630 running both Windows and Ubuntu. The Nokia N8 smartphone was a prototype device which has software installed that has Terminal Mode server capabilities. The device can be ordered from Forum Nokias Device Distribution program for 499 €[10]. It was however not retrieved through the Device Distribution program but through Mecel's connections with Nokia. The N8 was received nine weeks into the thesis project.

3.2 Programming Environment

When deciding which programming language and programming environment to use there were a few things to consider. A Terminal Mode client can be created in several programming languages but the client should also be integrated into Mecel Betula HUC which is developed in C. To simplify integration with Mecel Betula HUC, C was chosen as the programming language for the Terminal Mode client. In addition to the standard C libraries the windows.h library was used. The reason for this was that the client should be developed under Windows XP and because support for sockets, threads and semaphores was needed.

Mecel uses Visual Studio 2005 with the revision handling system Rational Clearcase integrated into it. Since C was chosen as the programming language and the client should be developed under Windows XP it was decided to use Visual Studio 2005 with Rational Clearcase as the programming environment.

An application for testing needed to be built as well, the only requirement on this was that it should work with the Terminal Mode client under Windows XP. Since Visual Studio 2005 was already used C# was chosen.

3.3 Software Development Method

The development method chosen when implementing the Terminal Mode client was a U-model which Mecel uses. All steps were however not used and the model was changed to fit this thesis project. The steps that was used were software require analysis, software design, software construction, software integration and software testing. These steps are shown in Figure 7.

The step software integration was also done a bit earlier then in the original U-model. After the different parts were integrated incremental updates and software testing was done until the product was finished.

The steps that were removed from the original U-model was the first two steps System Requirement Analysis and System Architectural Design, as well as the last two steps System Integration Test and System Testing. The reason for not using these steps was that most of these parts were not applicable when developing a Terminal Mode client.

For the demo application no developing method was chosen, but it was decided to use a more agile developing style since it was a smaller application that changed depending on how the Terminal Mode client was developed. Both for the Terminal Mode client and the demo application tasks were mainly solved by programming one and one. However when problems arose pair programming was often used until a solution to the problem was found.

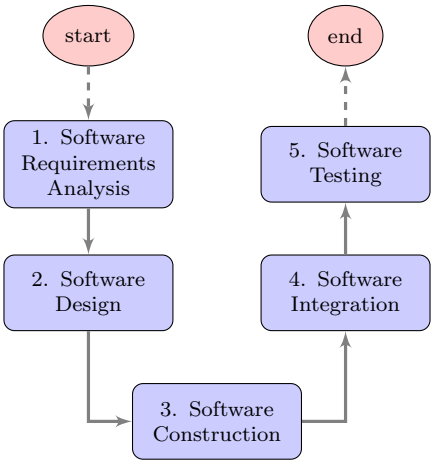

Figure 7: Software development flow chart

4 Implementation

In this section it is described how the Terminal Mode client was implemented. What problems arised and how they were solved. It is also described how the initial demo and testing program was developed. Furthermore the integration into Betula HUC is described.

4.1 System Setup

Nokia have chosen to use the USB device class Network Control Model to connect the N8 smartphone to the computer. An advantage of using NCM is that several ethernet frames can be aggregated into one single transfer. The reason to do this is to enable higher data transfer rates. The specification for NCM is built on the ECM specification with some modifications [11].

Problems arise when drivers were not available for windows at the time. Sources on the Internet only provided one driver which was from a German company named The Sycon. This driver was a demo but it was anyway not compatible with the N8. Other drivers existed but they had to be paid for. Nokia didn't provide any driver in the prototype kit either but they referred to the latest Linux kernels.

Linux kernels have support for NCM from version 2.6.38 which at the time is the latest stable Linux Kernel. The solution to get the N8 to communicate with the computer was to install Ubuntu 10.04 then compile a customized kernel and install it. After communication with the phone was established in Ubuntu Virtualbox was install with a Windows guest operating system. Virtualbox was configured to create a network interface in the Windows guest operating system, which was a bridge to the Linux interface for the phone. This allowed the virtualized Windows to communicate with the phone.

Two issues with this solution for connecting the phone to the computer were discovered. The phone sometimes did require several disconnects and reconnects before it was operable. The reason for this is unknown but it is mentioned in the prototype kit manual from Nokia [12]. The second issue was that when the device was disconnected from the computer and reconnected again the Windows guest operating system needed to be paused and then started again.

A later improvement to this virtualization solution was to use a standalone Linux computer bridging the network interface (usb0) created by the phone with the ethernet interface (eth0). DHCP didn't work over this solution but multicasts from the UPnP server in the phone worked and was received by the computers connected to the ethernet port of the bridge. Figure 8 shows this setup.

Configuring a bridge in Linux between the interface eth0 and usb0 is done with the code in Listing 1. Support for bridging in Linux is included in a default kernel.

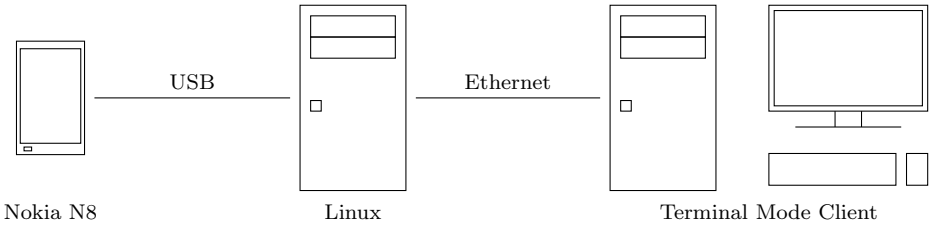

Figure 8: System setup

```
1 ifconfig eth0 0.0.0.0
2 ifconfig usb0 0.0.0.0
3 brctl addbr bridge
4 brctl addif bridge eth0
5 brctl addif brdige usb0
6 ifconfig bridge up
```

Listing 1: Commands to configure a Linux bridge

4.2 Structure

This section will describe the general structure of the Terminal Mode client, how it's built up and how internal and external communication is done.

4.2.1 Interface

An interface was created to be used against Betula HUC. The interface has a number of functions which makes it possible for Betula HUC and the Terminal Mode client to communicate. At startup Betula HUC calls a function in the interface that initializes the Terminal Mode client. Another function is used for shutting down the Terminal Mode client. These functions are further described in section 4.4.

It also has a function for receiving pointer events from Betula HUC. The pointer event is handled differently depending on which status the program has. This is described further in section 4.3.3.

The input of the functions is described in table 7. All the interface functions are of type integer and returns 0 to Betula HUC. These functions could as well return another value then 0, e.g. if some additional error handling are added but this is not necessary at the time.

4.2.2 Modules & Threads

Except for the interface the Terminal Mode client is divided in three different modules, UPnP, VNC and Manager. These modules are further described in their corresponding chapters. When the Terminal Mode client is initialized three threads are started. The functions in the modules are run either by these three threads or by Betula HUC, the threads are described below.

The VNC Thread is responsible for setting up a connection with the VNC server, it sends framebuffer update requests to the VNC server and receives framebuffer updates. It takes care of these updates, processes the data and writes them to the framebuffer.

The UPnP Thread handles the discovery of the Terminal Mode server. It keeps track if it periodically receives messages from the UPnP messages from the Terminal Mode server. If it does, the device is available, otherwise it is unavailable. If a discovered device becomes unavailable the UPnP thread notifies the manager thread of this.

The Manager Thread keeps track of the program state. It performs tasks such as printing the menu and notifies the VNC thread if it should connect to or disconnect from the VNC server. It also monitors the Terminal Mode server with help from the UPnP thread.

Table 7: Interface functions

Function		Arguments	
Name	Description	Name	Description
InitializeTerminal-ModeClient	Initialize the Terminal Mode client	char *fb_from_ui	Pointer to the framebuffer
		unsigned short int height	Height of framebuffer
		unsigned short int width	Width of framebuffer
		unsigned short int x_min	Start of x-position
		unsigned short int x_max	End of x-position
		unsigned short int y_min	Start of y-position
		unsigned short int y_max	End of y-position
		int preferred_pf	Preferred pixel format
		int* u	To keep track if framebuffer is updated.
Stop	Stops the Terminal Mode client		
PointerEvent	Passes a pointer event to the phone	int xpos	x-position of pointer
		int ypos	y-position of pointer
		char buttonpressed	If mouse is pressed down or released

4.2.3 Internal communication and global variables

Communication between the threads is performed with help of semaphores and global variables. Each thread has a variable that can be changed by the other threads. To inform or control a thread this variable is changed and a semaphore is released. The thread is notified of the event change by acquiring the semaphore.

There are several other global variables and structs used in the Terminal Mode client. For instance all the client and server settings are stored in global structs. The global variables are not protected by any semaphores or locks but no races exist so this is not a problem.

4.3 Framebuffer

Before the display data is showed to the user it is saved in a framebuffer. The framebuffer will then be used by Betula HUC, which will display the data. A framebuffer can be compared to the raw data of a bitmap-image, the pixels are saved in memory pixel by pixel from the first to the last one, row by row. Depending on the colour configuration used in the operating system the framebuffer can encode the pixels in different ways. Standard today is 32 bit colours which means that the pixel is encoded with ARGB888 encoding.

The Terminal Mode client has two framebuffers. One framebuffer is called the *shared framebuffer* and one is called the *local framebuffer*. The shared framebuffer is provided by Betula HUC and is the one that will be displayed to the user. The data in the shared framebuffer can be scaled compared to the framebuffer on the VNC server while the local framebuffer is a non-scaled copy of the VNC servers framebuffer. The reason to use two framebuffers is that when doing image scaling the original image data needs to be used and therefore it needs to be stored at the client. This is explained more in detail in the section about scaling in VNC.

The application menu buttons is printed before the VNC has connected. The image data received from the VNC server is displayed somewhere in the same framebuffer as the buttons and the application menu. After the VNC has connected and when the client knows which device keys the server supports they are printed in a reserved area in the framebuffer. This will be explained more in detail later in this section.

Reservation of the area in the framebuffer for the device keys are made after the VNC client has connected and after the server has communicated the display size. The device keys can be

placed either to the right of or below the image data from the server. To decide the position it calculates the height/width ratio of the servers display and compares it to the height/width ratio of the client's framebuffer. If the client's height/width ratio is higher than the servers it will place the device keys below the image in the framebuffer because there it would be some space left.

The image from the N8 is always displayed in landscape mode. Thus for most resolutions the device keys are placed below. When a place has been chosen it reserves the space by setting a value that tells in which area of the framebuffer the VNC client is allowed to put the image it receives from the server.

4.3.1 Application Menu

The manager is the module responsible for printing the application menu. After a device has been found it calls the *printMenu()* function which prints the menu. An argument to this function is an array of application ids for menu icons that should be printed. The printing function keeps an array with information about which application icons it has printed to the framebuffer. This is an array of structs which contains information about the height and width of the icon, the x and y position in the framebuffer and which application id that belongs to the icon. This information is saved so that it later is possible to decide which icon that was clicked. When printing the icons the program decides the next free position by looking at which position in the array that has an application id set to null. If an icon for an application is missing a default icon is printed.

4.3.2 Device Keys

After the VNC client has connected and received the server event configuration with information about which device key the server supports the VNC client calls the function *printButtons()* which prints the device keys to the framebuffer. *printButtons()* goes through the data about device keys support and identifies which device keys the server has support for. There is a set of pre-defined icons stored in the program and if the server has an icon for a device key it will be printed to the framebuffer. If an icon for a button doesn't exist that button is ignored. The code is easy to change to allow for more buttons. The *printButtons()* function will like the *printMenu()* function keep an array of which icons that have been printed. The reason is like the array for the menu to later be able to decide which button that was clicked.

4.3.3 Mouse Click Events

The function *FramebufferMouseClick()* is called when a mouse click has occurred in the framebuffer. Depending on in which state the program is and where in the framebuffer the mouse is clicked any of the following can happen:

- Display click
- Device key click
- Application icon click
- Click outside of buttons or display

The code first checks if the menu is visible or not with *programControl.menustatus*. If the menu was active it checks if any of the buttons where pressed by looping through each of them and compare the coordinates of the mouse click with the coordinate of the buttons. If the menu wasn't active and the display was visible the program checks if the display was clicked or it will loop through the printed keys to see if any of them were printed. These steps are showed in the flow chart in Figure 9. If the program detects that the coordinates of the mouse click was outside of the display, the device keys and the placation icons the function will return.

4.4 Initialization and Shutdown

Initialization and shutdown is handled by two functions in the Terminal Mode client.

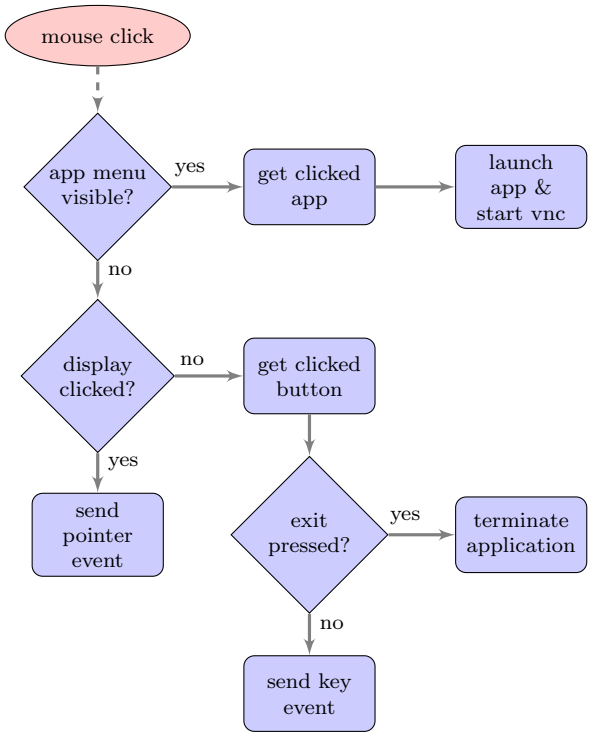

Figure 9: Flow chart of a mouse click event

The initialization function is the first function called when the Terminal Mode client is starting. As arguments the function takes information about the framebuffers memory address, its height and width. These values are saved to the global configuration values. The function and its arguments are showed in table 7.

It continues by creating three threads for VNC, UPnP and the Manager. These three threads continue the execution of the Terminal Mode client and the initialization function returns so that the Betula HUC can continue the execution. The thread handlers for each thread are saved in global configuration variables so that they later can be used by the shutdown function.

The shutdown function sets the controls of the different modules to shutdown and then it waits for the threads to close using the thread handlers. After the threads have closed it cleans up the local framebuffer if it was used and returns so that Betula HUC can continue the execution.

4.5 Manager

The Manager module controls the VNC module in the program. It is designed to be able to control a RTP module too but now it is only controlling the VNC module. Its main functions is to:

- Print the application menu when a device is discovered
- Monitor the application running on the Terminal Mode server
- Reconnect the VNC client if the connection is interrupted
- Shutdown the VNC client if the Terminal Mode server is disconnected

The manager runs in a separate thread and is built around two nested while loops. This design is chosen so that the inner loop is executed while a Terminal Mode server is connected and the outer loop is waiting for a Terminal Mode server to be found by the UPnP module. When a Terminal Mode server is found the code in the inner loops starts it execution.

The inner loop is executing its control sequence at least every second or when a ManagerStatusSemaphore has been acquired. See Code Listing 2 for a pseudo code showing the layout of

the loops. This is done by giving the program one second to acquire the semaphore, otherwise it will timeout and the control sequence can be executed. This design makes the manager react immediately if an event notifies it while it still will execute at a fixed interval.

Table 8: Module statuses

Module	Status	Description
VNC	DISCONNECTED	Status when VNC is disconnected from server
	WAITING	Status when VNC is waiting to connect
	CONNECTING	Status when VNC is connecting
	HANDSHAKE	Status when VNC is in handshake phase
	INITIALIZATION	Status when VNC is in initialization phase
	CONNECTED	Status when VNC is connected to server
	CONNECT_FAILED	Status when VNC failed to connect to server
Menu	SHOWMENU	Set when the menu is visible
	NOMENU	Set when the menu should not be visible

Table 9: Module control actions

Module	Action	Description
VNC	CONNECT	Tells VNC to connect
	DISCONNECT	Tells VNC to disconnect
	SHUTDOWN	Tells VNC to exit thread
Manager	NEWDEVICE	Tells manager that a device has been found
	NODEVICE	Tells manager that no device is available
	SHUTDOWN	Tells manager to exit thread
UPnP	RUN	Tells UPnP should operate as normal
	SHUTDOWN	Tells UPnP to exit thread

4.6 UPnP

For the UPnP part a control point which could handle a TmServerDevice with a TmServerApplication was created. At first it was decided to use an UPnP SDK to simplify the implementation. But due to problems in connecting the control point to the N8 the first control point was abandoned. Another control point was created but this time it was created from scratch.

The first control point is described in section 4.6.1. The rest of this section is dedicated to the second control point.

4.6.1 Generated UPnP Stack

For the development of the UPnP functionality an UPnP SDK was chosen. The SDK that was chosen was the Developer Tools for UPnP Technologies [13]. Developer Tools for UPnP Technologies is a set of tools which e.g. can be used to create devices, control points and services. It can build devices for both Windows and Linux, and it can generate code for both C and C#.

The reason for choosing Developer Tools for UPnP Technologies was that it was one of few SDKs that supported Windows. Furthermore it offered good tools and documentation which simplified the implementation.

Building the Control Point

One of the tools in developer tools which are called Device Builder was used to create the control point residing at the Terminal Mode client. To create the control point the necessary devices and services must be imported. Devices and services used in Terminal Mode are specified in the Terminal Mode specification expressed in XML format. [1]

The control point must handle a TmServerDevice with a TmApplicationServer to be able to communicate with the Terminal Mode server. So to create the control point the XML files for the device and the service were imported into Device Builder and the UPnP stack was generated.

```
1  while true:
2    ManagerStatusChangeSemaphore.acquire()
3    if ManagerControl == SHUTDOWN:
4      return
5    else if ManagerControl == NEWDEVICE:
6      Show Menu
7
8      while true:
9        ManagerStatusChangeSemaphore.acquire()
10       Check Running Program
11       if ManagerControl == SHUTDOWN:
12         return
13       else if MangerControl == NODEVICE
14         RFBControl = DISCONNECT
15       else if RFBStatus == DISCONNECTED or RFBStatus == CONNECT_FAILED
16         Print Menu
17         RFBControl = WAITING
18       if ManagerControl == NODEVICE
19         RFBControl = DISCONNECT
20         RFBStatusChangeSemaphore.release()
21         break
```

Listing 2: Manager pseudo code

The control point did not work at first. The problem was a bug in the code created by device builder which did that it only worked with IPv6 and not with just IPv4. Since the control point should work on Windows XP this problem had to be solved. There were some difficulties in solving the problem and contact were taken with the developer of the tools. The developer pointed out a function which returned an error because the lack of IPv6 support. By commenting out this error and making the function return 0 instead the problem was resolved.

The control point which was created by the developer tools had eight functions for handling the Terminal Mode server. It had functions for discovering and removing devices which are standard functions for all UPnP devices. The other six functions are the ones described in the TmApplicationServer service. It consists of two eventing functions and four actions. The two eventing functions was AppStatusUpdate and AppListUpdate which the Terminal Mode server uses to notify the control point of updates. The last four functions were the actions GetApplicationList, LaunchApplication, GetApplicationStatus and TerminateApplication which are used to control or get status from the Terminal Mode server.

Interface

To keep changes in the generated UPnP code to a minimum an interface was created. In the generated code only calls to this interface was inserted. This allows the control point created by the SDK to be easily replaceable without any larger changes in the Terminal Mode client. When a device is discovered or removed the control point runs a function in the interface which takes the necessary steps so that the manager thread are notified about the device. Also when the control point receives an AppStatusUpdate/AppListUpdate it runs a function in the interface. When the Terminal Mode client want to use one of the actions the corresponding function is called in the UPnP interface which in turn invoke the action in the control point and waits for a reply.

Problems

The UPnP stack worked together with a TmServerDevice created to simulate the Terminal Mode server. This device was created by just changing a value from control point to device when generating the UPnP stack in the Developer Tools. This way a real Terminal Mode server could be simulated several weeks before the N8 arrived.

Once the N8 arrived the integration with it and the control point started. It didn't work right away by just connecting the N8 instead of the TmServerDevice created by the Developer

Tools. To figure out why this was the case an analysis was made to see what differed between the TmServerDevice created by Developer Tools and the TmServerDevice residing in the N8.

Wireshark were used to analyze the messages sent by the devices and one major difference could be seen. The N8 only used newline character to indicate a new line meanwhile the other TmServerDevice used both carriage return and newline characters. The control point did only accept devices that used both carriage return and newline characters. To cope with this some changes were made to the UPnP stack so that it would accept devices which just used the newline character. However this did not solve the problem.

The analysis continued and a lot of effort was put in understanding the code generated by developer tools, where the control point should accept the Terminal Mode server and why it didn't. Several things were tried such as replacing values and hard coding some parts but none of the taken measures helped solving the problem.

About two weeks after the N8 arrived a decision was made. Since no significant progress towards a working solution were done and since there were no good ideas left how to get it to work within reasonable time the control point was abandoned. Instead a new control point was created and this time it was created from scratch.

4.6.2 Differences between the control points

The new control point should work in the same way as the old one but one part that was left out was eventing. By using eventing The TmApplicationServer can notify a control point when information about its applications has changed. This is done through a AppStatusUpdate or a AppListUpdate message. These functions could be used to make the program a bit faster when closing an application but it would not be a large performance gain. Considering that and the limited time it was decided to leave this for future work.

4.6.3 Device Discovery

The first step when starting the Terminal Mode client is that it must find a server to connect to. UPnP devices sends periodic notifications on the network to allow other devices to discover them, these messages are sent as SSDP messages on the multicast address *239.255.255.250* and port *1900*.[4] So to find the N8, the Terminal Mode client first needs to listen after multicast messages on the network. When it receives a message it looks if this is the device that it is searching for. This is done by looking for the string *"urn:schemas-upnp-org:device:TmServerDevice:1"*. If a message which contains this schema is found it knows that the correct device is on the network hence it has discovered a Terminal Mode server. It will then save the URL to the device which will be used to communicate with it.

There are two ways in which the program stops without user intervention. The first way is when a task fails to complete such as a socket closes prematurely, which should only occur if it has lost its contact with the Terminal Mode server. The other way is when the control point loses contact with the Terminal Mode server device.

Since the N8 periodically sends notify messages the control point can listen for these messages and if it stops receiving these messages it knows that either the network is broken or the device is unavailable. The messages from the N8 are normally received every fifth second but if the control point has not received a message for eleven seconds it assume that the Terminal Mode server is unavailable. Eleven seconds has been chosen to allow one round of notify messages to get lost and to allow some delay on the network. This timeout value can easily be changed. When a timeout occurs in the UPnP thread it will notify the manager that the Terminal Mode server is unavailable. The manager will then in turn tell the VNC client to disconnect.

If the user has not told the program to stop, the control point will still listen for notify messages. If it then finds a TmServerDevice it has once again discovered a device and the Terminal Mode client can connect to the Terminal Mode server.

4.6.4 Actions

To control and get information from the TmApplicationServer on the Terminal Mode server four actions are used, LaunchApplication, TerminateApplication, GetApplicationStatus and GetAppli-

cationList. To invoke an action a SOAP message encapsulated in a HTTP request message should be sent to the Terminal Mode server. [4] Since the messages needed to be created from scratch, the easiest way to find out what kind of messages that the N8 accepted was by looking at actual messages that it did accept. These messages can be found by listening to the communication between the N8 and the prototype client with Wireshark. The messages were first tested to see that the N8 could be controlled by them and that it correctly sent a reply to the Terminal Mode client.

However this was not enough since it should work with any smart phone. To resolve this problem the messages needed to be generated in a way which followed the UPnP Device Architecture [4] and the Terminal Mode specification [1]. Furthermore the messages received, which also are SOAP messages but encapsulated in HTTP response messages, needed to be parsed to get the actual response.

So after finding out what messages that the N8 accepted the next step was to generate the messages from scratch according to the UPnP Device Architecture and the Terminal Mode specification so that the messages would work with any phone. At this point the design that would be used for invoking actions was clear. It consisted of the following five steps for each action.

1. *Generating message* - Generate a SOAP message with its HTTP header.
2. *Connect to Terminal Mode server* - Create a socket to the TmServerDevice residing at the Terminal Mode server.
3. *Posting message* - Sending the generated message on the socket.
4. *Receiving message* - Receiving the response on the socket.
5. *Parsing response* - Parsing the received message to get the requested value.

Generating and posting messages When generating the messages to send, the first step was to analyze which parts of the message that was the same for all messages and which parts that differed. The HTTP header does not differ so much between different actions. Some fields are not changed between the messages and have therefore been hardcoded.

Other fields do vary, the *POST* and *HOST* fields vary depending on which Terminal Mode server it should connect to. In these fields the correct URL respectively IP and port must be set. The input to these fields have been stored from the periodic notify messages that the Terminal Mode server sends out.

Another field that varies is *CONTENT-LENGTH*, which is the length of the SOAP message. The value of this field hence depends on the length of the SOAP message. The last field that vary and the only field that depends on which action that is used is the *SOAPACTION* field, this field will consist of the string "urn:schemas-upnp-org:service:TmApplicationServer:1#*Action*" were *Action* is the specific action e.g. LaunchApplication.

The data of the HTTP message is a SOAP message. The SOAP message used for the action LaunchApplication can be seen in listing 3. The envelope part of the messages is always the same. The body on the other hand differs between each action. For other SOAP messages then the one for LaunchApplication, the fields with the text LaunchApplication are replaced with the corresponding action. The other part that changes for each action are the arguments, which were described in table 1.

The messages are generated by using the memcpy() function on different strings and variables, the result is stored in a buffer. After the message has been constructed a socket is opened to the TmServerDevice. The buffer consisting of the message is then sent on the socket.

Receiving and parsing messages After the SOAP message are sent the Terminal Mode client waits for a response. When the response has been received the message must be parsed. The first step in the parsing is to check that the body of the SOAP message is properly formatted. The N8 sends in some cases the character sequence "<" and ">" instead of the proper signs '<' and '>'. If the SOAP body contains these character sequences they are replaced with the correct characters.

```
<?xml version="1.0"?>
<s:Envelope
  xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
  s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <s:Body>
      <u:LaunchApplication
            xmlns:u="urn:schemas-upnp-org:service:TmApplicationServer:1">
        <AppID>0x123456</AppID>
        <ProfileID>0</ProfileID>
      </u:LaunchApplication>
    </s:Body>
</s:Envelope>
```

Listing 3: SOAP message for the action LaunchApplication

The second step in the parsing is done in different ways depending on what action it is responding to. For the three actions GetApplicationStatus, LaunchApplication and TerminateApplication the response consists of a single XML value. To get this value a function searches for the XML tag that contains the response. In LaunchApplication this means that the response is a string which lies between the XML tags *<URI>* and *</URI>*. When this string is found it is saved to a buffer which the calling function has passed a pointer to. The last action, GetApplicationList does not receive a single value from the TmServerDevice, instead it receives a list of applications. This list is also saved to a buffer defined by the calling function.

4.7 VNC

This section will describe the implementation of the VNC module, which is responsible of receiving the image from the Terminal Mode server, and how it updates the client's framebuffer. VNC and extension messages used in VNC by Terminal Mode are described in the theory section of this thesis.

The structure of the VNC client is a thread running two nested loops. The outer loop is used so that the thread will never finish executing and the inner loop is executed when the VNC client has connected to the server. Before the client has connected to the server the program is waiting in the first loop but it has not yet entered the second loop. The functions of the loops are shown in the pseudo code in code listing 4.

```
while true:
  RFBStatusChangeSemaphore.acquire()
  if RFBControl == SHUTDOWN:
    return
  else if RFBControl == CONNECT:
    Connect

    while true:
      send RemoteFramebufferUpdateRequest
      receive Message
      if RFBControl == DISCONNECT:
        break
```

Listing 4: VNC pseudo code

When receiving a message the first byte is read to identify the message type. Then the function responsible for each type of message is called to handle the message. Message types can be framebuffer update or an extension message. Instead of using a recv() call directly the program first do a select() call on the socket to see if there is available data to read. This is done so the recv() call is not blocking in the case the manager tells the VNC client to disconnect. The select() call is done with a timeout of 1 second and between every select the value of RFBControl is checked.

4.7.1 Connection Setup

As described in the initialization section all threads, the VNC thread included, are created at the beginning of the execution. Before a device has been discovered the VNC client is not supposed to do anything but wait for information about which server to connect to. This is done by trying to acquire the RFBStatusChangeSemaphore which will be released by the manager when a device is found. When the semaphore has been acquired the RFB control variable is checked for which action to take.

If the action is set to connect then the VNC client will use the port and server address defined in the global serverConfig data structure and start the connection.

If the connecting process in some step would fail (recv() call or send() call returning 0 or a negative value) the VNC client would inform the manager by setting its status to connect failed and then release the ManagerStatusThreadSemaphore to start the execution in the manager. After notifying the manager the VNC client would go back to the start state where it tries to acquire the RFBStatusChangeSemaphore so it then can make a new connection.

4.7.2 Display Updates

When the program receives a message and it has been identified as a Framebuffer Update the function *ReceiveFramebufferUpdate()* receives the rest of the message and processes the update. The program will execute the following steps for this process. Each of the steps are implemented in a separate function so the code is easy to follow and more understandable.

1. Receive number of rectangles
2. For the number of rectangles do:
   (a) Receive rectangle data
   (b) Convert data to 32bpp if needed
   (c) Update local framebuffer
   (d) Update shared framebuffer, scale if needed

1 The first thing the framebuffer update message contains is information about how many update rectangles to expect. For each of these rectangles the following parts will executed.

2a First for each rectangle information about the size, position in the framebuffer and encoding is received. The information about the framebuffer together with the framebuffer data will be saved into a struct. If the encoding is RAW the size of the data that should be received is calculated by the height multiplied by the width multiplied by the pixel size, 2 bytes if RGB343, RGB444 or RGB565 is used and 4 bytes if ARGB888 is used. The program is constructed by a while loop which do recv until all data has been received. The recv length is set to the size of the update minus the size of the data already received.

If the encoding is RLE then each pixel would be received one at a time because the program doesn't know how many bytes to receive. The data would then be masked out so information about the run-length and the pixel value will be separated. The number of bits for data is showed in table 6, which the number of bits for RLE could easily be calculated by subtracting bits for colour from total bits. The code for receiving the pixel data and extracting the colour value and the run-length is shown below.

```
recv(serverConfig.socket,(char *)&buf,(nbrColourBits + nrRunLengthBits)
    /8,0);
runLength = (buf >> nbrColourBits) + 1;
colorValue = buf & colorMask;
```

One obvious performance issue here is that a lot of recv will be needed to do. Since the recv is a system call they will use a lot of resources. Another design alternative would be to receive into

a buffer and read from it instead. Then the number of recv-calls could be minimized. This will be discussed more in the discussion section.

When the run length and the colour value is extracted from the received bytes the framebuffer will be filled up with the number of pixels specified by the run-length all having the same colour value. After this step have completed a struct will contain information about a the rectangle that has been received.

2b If the data is not sent in ARGB888 format but RGB343, RGB444 or RGB565, the data will need to be converted to ARGB888 before next step can be executed. This needs to be done because the framebuffer shared with Betula HUC have the pixels encoded in ARGB888 format.

Converting a colour value from RGB343, RGB444 or RGB565 to ARGB888 is done by masking out the bits for each colour. When the colour values are identified they are left shifted so they get in the correct position of a 24-bit colour and the most significant bit gets in the correct position. Then they all are anded together into one single value.

These two shifting steps are done at the same time. Below is the code that shows how this is done.

```
pixelARGB888 = (((long) pixelRGB565 & 0xF800)<<8) | (((long) pixelRGB565 &
    0x7E0)<<5)
             | (((long) pixelRGB565 & 0x1F)<<3)

pixelARGB888 = (((long) pixelRGB444 & 0xF00)<<12) | (((long) pixelRGB444 &
    0x0F0)<<8)
             | (((long) pixelRGB444 & 0x00F)<<4);

pixelARGB888 = (((long) pixelRGB343 & 0x380)<<14) | (((long) pixelRGB343 &
    0x78)<<9)
             | (((long) pixelRGB343 & 0x7)<<5);
```

An obvious flaw exists with this method. The problem is that zeroes are shifted in when the extra bits are added. This means that if a colour was 1111 1 it will be 1111 0000 after the bits are shifted which will be slightly darker than the optimal conversion. This will be discussed more in the discussion section.

2c Copying the data to the local framebuffer is done row by row. The source position to copy from and the destination position to copy to are calculated for each row. The code used is showed below. r is the struct which contains the data about the rectangle.

```
for(row = 0;row < r->height;row++){
  unsigned long int copyFrom = (row * r->width) * 4;
  unsigned long int copyTo = ((row + r->y_position)
                              * serverConfig.framebuffer_width
                              + r->x_position) * 4;
  unsigned char *fbPos = clientConfig.localframebuffer + copyTo;
  unsigned char *dataPos = r->data + copyFrom;

  memcpy(fbPos, dataPos, r->width * 4);
}
```

2d Updating the shared framebuffer is the last step. If the size of the shared framebuffer is bigger than the local framebuffer, scaling of the image will be needed. If the shared framebuffer is smaller than the local framebuffer the image is clipped.

Scaling is a computing demanding operation so only pixels that have been updated will be scaled. The pixels being updated are defined in the update rectangle. This is done by the scaling function that sets an area of which coordinates to scale.

The first thing the function does is to calculate how much the framebuffer should be scaled. This value is later used when converting coordinates of a pixel in the shared framebuffer to a coordinate in the local framebuffer. To ensure that all pixels is updated an extra frame of pixels around the updated ones will be scaled. The scaling function is put in two loops, one going through the pixels horizontal and one vertical. One performance improvement is to put some of the calculations in the outer loop so they only will be done once for each row.

Each pixel in the scaled shared framebuffer will get the value from pixels in the local framebuffer. This can be done either by nearest neighbor interpolation or bilinear interpolation. A settings flag defines which is used. For an explanation of how these algorithm works see the theory section.

First the start and stop values for the inner and outer loops are calculated. These values are then checked so they don't start outside of the framebuffer, don't end outside the framebuffer, that the stop value is bigger than the start value. For each of these pixels the following steps are executed:

1. Calculate x,y coordinates of the neighbor[3] pixels. This is rows 2-8 in code listing 5.

2. Insert the coordinates into the pixel structures. This is rows 17-25.

3. Get the colour values and put them into the pixel structures. This is row 27.

4. Calculate the ratio of how much colour that should be used from the neighbor pixels. This is row 29-32.

5. Calculate the red, green and blue values from the surrounding neighbor pixels. This is rows 34-45.

6. Calculate the position in the framebuffer and insert the colours. This is row 47.

Note that code listing 5 is not the fully code for bilinear interpolation. There exist some special cases when a pixel has the exact same x or y coordinate as one or more of it neighbor pixels.

4.7.3 Extension Messages

When an incoming message has been identified as an Terminal Mode extension message by the message type it is sent to *ReceiveTerminalModeMessage()* which checks the type of the extension message. Depending on the message type it will execute *ReceiveServerDisplayConfiguration()* or *ReceiveServerEventConfiguration()*.

ReceiveServerDisplayConfiguration() will receive the whole Server Display Configuration message into a struct using one *recv()* call. Then it saves the received data to the global configuration variables. After this it will send the corresponding Client Display Configuration message using the *SendClientDispalyConfiguration()* function. *SendClientDispalyConfiguration()* uses a pre defined message which it sends to the server. It uses the pixel height and width from the display but the other values are set to unknown.

ReceiveServerEventConfiguration() will like *ReceiveServerDisplayConfiguration()* put the whole message into a struct. It saves the received data to the global configuration variables and then calls the *printButtons()* which prints the device keys received to the framebuffer. After this it continues by calling the *SendClientEventConfiguration()* function which sends the corresponding Client Event Configuration Message. The Client Event Configuration message sent will use the same language and country codes as received from the server sent, it will set support to knob keys and multimedia keys to none, device keys will be the same as the value received from the server. It also set support for pointer events but no touch events.

[3]The pixels northwest, southwest, northeast and southeast.

```
1  // This code can be put outside of the second loop
2  float y = py/scale;
3  float y1 = (float)ceil(y);
4  float y2 = (float)floor(y);
5
6  float x = px/scale;
7  float x1 = (float)floor(x);
8  float x2 = (float)ceil(x);
9
10
11 /* q11 = top left
12    q12 = top right
13    q21 = bottom left
14    q22 = bottom right */
15
16 // This code can be put outside of the second loop
17 q11.y_pos = (unsigned int) y1;
18 q21.y_pos = (unsigned int) y1;
19 q12.y_pos = (unsigned int) y2;
20 q22.y_pos = (unsigned int) y2;
21
22 q11.x_pos = (unsigned int) x1;
23 q12.x_pos = (unsigned int) x1;
24 q21.x_pos = (unsigned int) x2;
25 q22.x_pos = (unsigned int) x2;
26
27 GetColour(&q11); GetColour(&q12); GetColour(&q21); GetColour(&q22);
28
29 q11_ratio = ((x2-x)*(y2-y))/((x2-x1)*(y2-y1));
30 q12_ratio = ((x2-x)*(y-y1))/((x2-x1)*(y2-y1));
31 q21_ratio = ((x-x1)*(y2-y))/((x2-x1)*(y2-y1));
32 q22_ratio = ((x-x1)*(y-y1))/((x2-x1)*(y2-y1));
33
34 r = (unsigned short int)((q11.red * q11_ratio)
35   + (q12.red * q12_ratio)
36   + (q21.red * q21_ratio)
37   + (q22.red * q22_ratio));
38 g = (unsigned short int)((q11.green * q11_ratio)
39   + (q12.green * q12_ratio)
40   + (q21.green * q21_ratio)
41   + (q22.green * q22_ratio));
42 b = (unsigned short int)((q11.blue * q11_ratio)
43   + (q12.blue * q12_ratio)
44   + (q21.blue * q21_ratio)
45   + (q22.blue * q22_ratio));
46
47 *(unsigned long int *)(clientConfig.framebuffer.address + (py *
       clientConfig.framebuffer.width + px) * 4 + 0) = (unsigned long int)b |
       ((unsigned long int)g)<<8 | ((unsigned long int)r)<<16;
```

Listing 5: Bilinear interpolation, calculation of one pixel value

4.7.4 Control Input

The input that can be sent to the VNC server comes from the interface. The VNC client provides two functions for sending pointer events and device keys, *SendPointerEvent()* and *SendKeyEvent()*.

SendPointerEvent() takes arguments of x-position, y-position and which mouse button that was pressed. It creates the message and sends it on the VNC socket. SendKeyEvent() takes arguments of device key and if it was pressed down or released. It creates the message and sends it on the

VNC socket.

4.8 Demo Program

Implementation to Betula HUC was planned to be done late in the project. So to be able to do debugging a demo program in C# was implemented. Later it turned out that the demo program would be the one used in the final product since the Terminal Mode client was never integrated into Betula HUC. The reason for this was lack of time from Mecel which had to work a bit with Mecel Betula HUC to make the integration possible.

The user interface of the demo program consists of a picture box that is used to view the framebuffer and control the Terminal Mode server, some boxes for settings and start/stop buttons.

When starting the program the preferred pixel format and resolution must be chosen. The demo program then initializes a framebuffer and sends a pointer to it to the Terminal Mode client. Along with the pointer it also sends preferred pixel format, width and height on the framebuffer as well as a variable that the Terminal Mode client should increment when there is new data in the framebuffer.

After the Terminal Mode client has received data from the Terminal Mode server it converts the data into ARGB888 format and writes the data to the framebuffer specified by the demo program. When the data has been written to the framebuffer it also updates a variable. The demo program controls if this variable has been updated and when it has the demo program creates an BMP image from the data in the framebuffer. This BMP image is then showed in the picture box.

To handle events the demo program keeps track on where the mouse are at all times. When the left mouse button is clicked inside the framebuffer then the Terminal Mode client is notified about which position of the framebuffer that was clicked.

If the shutdown button is pressed the demo program notifies the Terminal Mode client so that it shuts down. It also clears the picture box and returns the user to the settings screen. Pressing the test button starts a set of tests. These tests can be used to verify that the Terminal Mode client works as it is supposed to. The tests that can be performed are to check UPnP functionality, test framebuffer updates or to control that the parsing is done correctly.

5 Results

The outcome of the implementation was a Terminal Mode client capable of discovering a Terminal Mode server, start and stop applications, connect to the VNC server to retrieve the display content and control the user interface. It was also capable of monitoring the application running on the smartphone and shutdown the VNC client if an unwanted application is launched. It was not implemented into Mecel Betula HUC which was one of the goals from the beginning. But it was integrated into the C# Demo program which functioned in the same way as Mecel Betula HUC. So from a technical point of view this was of minor importance.

Figure 10 shows a screenshot of the implementation when it has connected to a Terminal Mode server and the application menu is showed. Figure 11 shows a screenshot when it has launched an application and the VNC is connected.

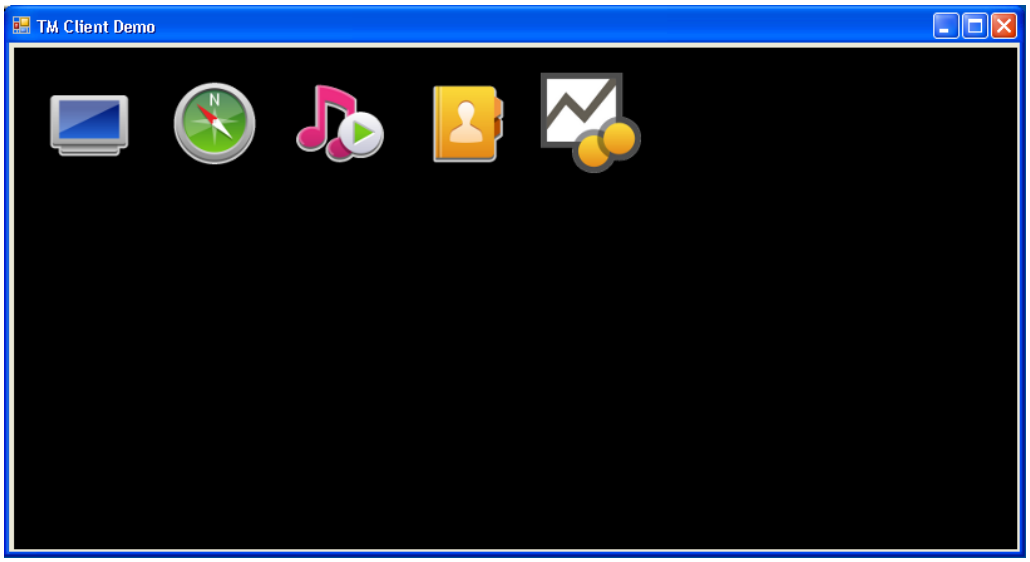

Figure 10: Screenshot of the program running showing the application menu

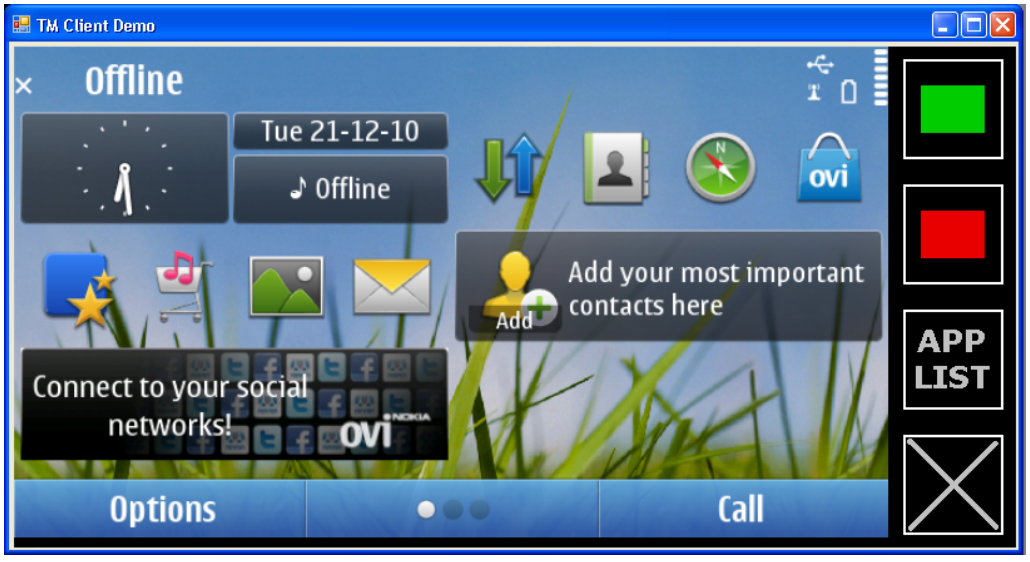

Figure 11: Screenshot of the program running showing connection to the VNC server

The following sections in this chapter will present the measurable results of the implementation such as image quality when the image have been scaled, measured FPS rate for different image and pixel encodings and how well the implementation is expected to work with other Terminal Mode servers than the one in the N8 prototype device.

5.1 Scaling

The first measurable result of the implementation is the image quality when the display has been scaled. Figure 12 and Figure 13 shows the same display when different scaling methods have been used. The screenshot in Figure 12 have been scaled using nearest neighbor and the screenshot in Figure 13 have been scaled using bilinear interpolation. The screenshot is taken from a part of the N8 home screen where both text and a picture/icon is showed. To include the full picture here wouldn't show the effects of scaling, therefor a partial screenshot have been chosen to illustrate this.

We can clearly see that the image scaled by the bilinear method has a smoother look than the image scaled by the nearest neighbor method. The image scaled with nearest neighbor has sharper and more distinct edges. It also looks like it is built on larger pixels. The results confirm the expected results from the theory section.

Figure 12: Result after nearest neighbor scaling

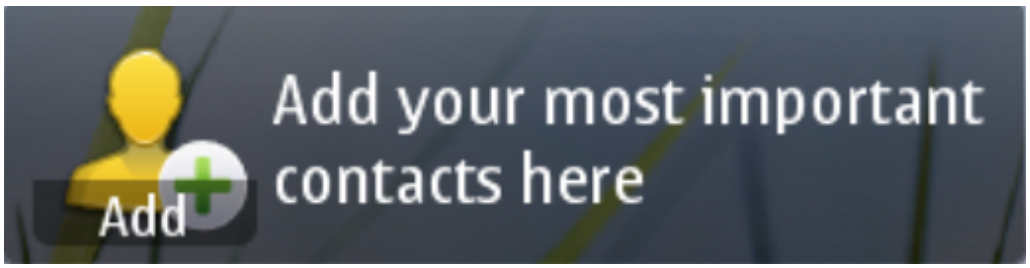

Figure 13: Result after bilinear interpolation scaling

5.2 Experiments

To help in the evaluation of the terminal mode concept and to see how well the TM client performed a number of tests were planned. One test was to compare how the different transport methods USB, Bluetooth and Wi-Fi performed which FPS rate the program could receive when using different transport methods and if there was some interference when using Bluetooth and WLAN at the same time. These tests were however not performed because of the limitations in the N8 prototype device.

Since the N8 prototype device only supports USB as its transport method tests were only performed on USB. The test which was done consisted of measuring the FPS rate when sending full size images. The same image was sent 2000 times (20 runs with 100 image updates each time) in a row. From this output the average FPS rate was calculated.

This test was performed on the TM client with nearest neighbor and bilinear scaling as well as on the prototype client. The four color models RGB343, RGB444, RGB565 and ARGB888 was tested.

The mean values of the test results for the different color models, without and with RLE, for the different clients and scaling methods, can be seen in table 10 and table 11. The entire test results can be seen in Appendix A.

Table 10: FPS Results, without RLE

	Prototype Client	Nearest Neighbour	Bilinear
RGB343	0.18	0.37	0.36
RGB444	0.17	0.39	0.34
RGB565	0.17	0.40	0.34
ARGB888	0.04	0.05	0.05

Table 11: FPS Results, with RLE

	Prototype Client	Nearest Neighbour	Bilinear
RGB343	1.22	1.16	1.13
RGB444	0.71	0.88	0.86
RGB565	0.19	0.15	0.16
ARGB888	0.14	0.08	0.07

The FPS rate with RLE is generally much higher than without RLE. One exception from this exists and that is with the color model RGB565, here the results are about the same or even slower than without RLE. The first pixel in each run-length takes up 3 bits extra, so for RGB565 that doesn't have any overhead this means that 3 bytes is sent instead of 2 bytes. RLE will still be better even for RGB565 when a large amount of the data have the exact same colour, however when the color differs a lot from pixel to pixel then the extra byte needed by RLE will instead decrease the FPS rate.

A clearly increase in performance can be seen in the TM client compared to the prototype client when not using RLE. The results of the FPS test are discussed in the chapter 7.

5.3 Interoperability

The aim when developing the Terminal Mode client has, except that it should work with the prototype device, been that it should work with any Terminal Mode server, either directly as it is or at least with very small modifications. This section will present how the Terminal Mode client has been developed and how it is working in relation to interoperability.

To get the Terminal Mode client as interoperable as possible it was important to follow the standards. The basic functionality in the VNC part of the client has been implemented according to the standards. It has also been tested against a VNC Server (RealVNC Server version 4.1.3) to verify that it works. Support for VNC version 3.7 and 3.8 has been implemented and verified by testing. The test was performed by connecting the client to the VNC Server, it was verified that the correct messages was sent for the two versions of the protocol.

The Terminal Mode extension messages which have been implemented are Server Display Configuration, Client Display Configuration, Server Event Configuration and Client Event Configuration. These extension messages are supported by the Nokia N8 prototype phone. No more extension messages should be required for a Terminal Mode client to be interoperable with any Terminal Mode server.

The UPnP control point has been designed to follow both the UPnP device architecture and the Terminal Mode standard. However all the functionality in the control point has not been implemented. A *M-SEARCH* message should be sent by the control point when it connects which tells UPnP devices on the networks that it searches for devices. This way the devices can respond directly when a control point joins a network. Since this message is not implemented the Terminal

Mode server must be connected first after the Terminal Mode client has been initialized. Other than that eventing was also left out which were described in section 4.6.2.

Except the *M-SEARCH* message and lack of support for eventing the control point follows the standards. A thing that is not mentioned in the standards is about how UPnP devices should handle new lines in the messages. The Terminal Mode client sends messages with only newline characters, however it accepts messages which use both newline and carriage return characters. The first control point which was developed by Developer Tools for UPnP uses both carriage return and newline characters. If the Terminal Mode client should need to send newline and carriage return characters to be interoperable with another Terminal Mode server then this can be easily changed in the code.

Even if all messages should be sent according to the standard different devices do not always send the exact same messages. Because of this the parsing of the received messages must be done in a way which allows some small differences but at the same time follows the standards. The parsing in the Terminal Mode client does allow some minor differences in the messages such as additional spaces. It can also handle invalid XML tags which were described in section 4.6.4. No flaws in the parsing have been found but since the client has only been tested with one Terminal Mode server it is hard to verify that the parsing is done correctly.

6 Terminal Mode Analysis

This section will leave the Terminal Mode client implementation that have been the main subject of this report so far and instead make a deep analysis of the Terminal Mode standard. The main focus in this analysis will be on security, which security mechanisms that exist and what values they protect. But security will not be the only thing analyzed. Alternative solutions to the Terminal Mode standard will be described and compared to Terminal Mode as well as the concept in general. How driver distraction is prevented and an analysis of what would be needed for the car systems to communicate sensor values and other data to the Terminal Mode server will be performed. The question about when RTP should be used for audio instead of Bluetooth will also be answered.

The authors' thoughts about the Terminal Mode concept will be left out to the discussion and the conclusion section.

6.1 Security Mechanisms

The Terminal Mode standard specifies three security mechanisms, Content Attestation, Device Attestation and Link Layer security. One section each will be dedicated to describe them here. What they protect will be described in section 6.3.

6.1.1 Content Attestation

The Terminal Mode standard provides a mechanism for attestation of framebuffer updates. The purpose of this mechanism is to ensure that the framebuffer updates comes from the intended source. The mechanism start with that the client sends a content attestation request message before it sends a framebuffer update request. After the corresponding framebuffer update the server then will send a content attestation response message. The message flow is showed in Figure 14.

The response message contains hashes of different data sent from the server to the client. The hashes can be encrypted with a key so only the client can read them and so that only the server can generate correct hashes.

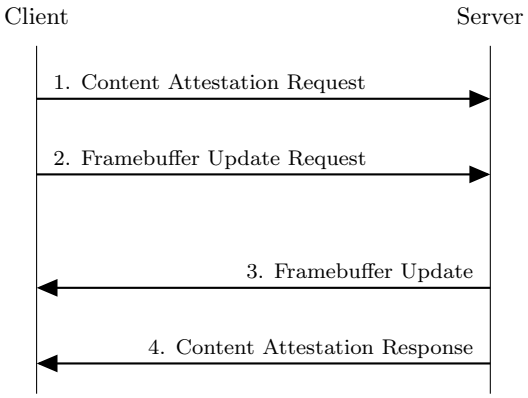

Figure 14: Content attestation mechanisms message flow

Request message

The message sent from the client is the framebuffer content attestation request message. The data in the message except for the headers are showed in Table 12. The purpose of this message is to tell the server which values to attest and to initialize the session key used by the server in the response. The session key is a symmetric key and it is encrypted with the server public key, which comes from the device attestation mechanism, so only the server can read it. The use of session keys is optional so encryption is not a must. It is however up to the client to choose which level of security that is acceptable.

Description of the parts in the content attestation request message showed in Table 12:

Table 12: Contents of content attestation request message

#	Description
1	Random nonce
2	Definition of what should be included in the response
3	Signature type
4	Session key information
5	Session key

1. 16 bytes of random nonce

2. Definition of what should be included in the response. Bit *0* tells to include all context information pseudo rectangles. Bit *1* tells to include the last framebuffer update. Bit *2* tells to include the number of bytes sent since the last content attestation response message.

3. Definition of which algorithm to use for signature generation. Terminal Mode standard only defines value *1* which means an HMAC-SHA-256 hash. Value *0* means that the signature should be left out.

4. Definition of which type the session key is. Value *0* means that no session key will be used. Terminal Mode standard only defines value *1* means to use a 128-bit symmetric session key. It will be encrypted with the server's public key retrieved during the device attestation phase.

5. If a session key is defined in the previous field a session key will included. The session key will be encrypted with the server's public key.

Response message

After the content attestation request, the framebuffer update request and the framebuffer update messages have been sent the content attestation response message is sent. It contains hashes of the different values requested by the client. The data the message contain except for the headers are showed in Table 13. If a session key was used all data except for the header is encrypted with the symmetric session key that only the server and the client can have knowledge about. The headers consist of information about the message type and then the encrypted payloads length.

Table 13: Contents of content attestation response message

#		Description
1	Signature Flag	
2	Error Code	
3	a	Nonce from the clients content attestation request message
	b	Signature flag, same as 1
	c	Hash of framebuffer context information data
	d	Hash of framebuffer content
	e	Number of bytes sent since last attestation message
4	Signature over the data in 3	

Description of the parts in the content attestation request message showed in Table 13:

1. Defines which fields that have been attested and included in the hash. Bit *0* tells if all context information pseudo rectangles have been included. Bit *1* tells if the last framebuffer update has been in clouded. Bit *2* tells if the number of bytes sent since the last content attestation response message has been included.

2. Contains information if the signature flag was the same as the client sent or if there wasn't any session key, content attestation was not implemented or if any other error occurred.

3. These fields are the source of the hash. Fields a-e are optional. They will be included if the client asked for them and the server can generate them. If the server doesn't have the capability to do so the signature flag will be changed and the error code will indicate that this has happened.
   (a) The nonce sent by the client in the content attestation request message.
   (b) The signature flag used by the server.
   (c) If context information pseudo rectangles should be included the hash will be here.
   (d) If the last framebuffer update should be included the hash will be here.
   (e) If the number of bytes sent since last content attestation response message the number of bytes will be here.
4. If a signature was used the hash over the values in 5-9 will be sent.

6.1.2 Device Attestation

Terminal Mode has a mechanism for verifying that a Terminal Mode server is compatible with the Terminal Mode client and that it is running approved software. This mechanism is called device attestation. Software components in Terminal Mode that can be attested are VNC server, UPnP server, UPnP control point, RTP server and RTP client.

To use device attestation the Terminal Mode server should have a device attestation server and the Terminal Mode client should have a device attestation client. The Terminal Mode client knows if the Terminal Mode server has a device attestation server after it has received the response from the UPnP action GetApplicationList. If it has a device attestation server then the response from the GetApplicationList action should contain this.

The device attestation server is launched by using the UPnP action LaunchApplication, which in response gives an URI (Uniform Resource Identifier) so that the device attestation client can connect to it. When the URI is received the device attestation protocol is launched. The device attestation protocol consists of two messages, attestationRequest and attestationResponse. The client sends a attestationRequest containing which component it want attested. The server attests this component and then sends an attestationResponse to the client which verifies it. When the software component has been attested the device attestation server can be terminated, this is done in the same way as it does with other servers, i.e. by using the TerminateApplication action.

The device attestation protocol is described in more detail below. Even more details about the device attestation can be seen in the Terminal Mode standard [1].

Prerequisites

To use device attestation the Terminal Mode server must have a X.509 device certificate for its device key pair from the server device manufacturer as specified in the Terminal Mode standard. X.509 is a standard for defining digital certificates. It must also have one or more X.509 manufacturer certificates from client manufacturers. The device key pair should be stored securely with the server device by using TPM (Trusted Platform Module) or similar. The Terminal Mode standard recommends a hardware-based MTM (Mobile Trusted Module).

Attestation Process

The attestation process which consists of an attestationRequest and an atttestationResponse message can further be divided into five steps.

1. The client picks a random nonce.
2. The client sends the attestationRequest message to the server.
3. The server receives the message and attests the component.
4. The server sends a attestationResponse message to the client.
5. The client verifies the content of the attestationResponse message.

These five steps will be described below.

1. A random nonce is picked by the client to be used in the attestation process. The nonce is a 20-byte random number. This nonce will be sent to the server in the attestationRequest message.

2. An attestationRequest message is sent from the client to the server. The attestationRequest message is an XML message, the items in the message can be seen in Table 14. The text below describes the contents of the table.

Table 14: Contents of device attestationRequest message

#	Description
1	Version number
2	trustRoot
3	Nonce
4	componentID

1. The Terminal Mode version used, currently 1.0
2. Trust root used for verification by the Terminal Mode client. 32-byte SHA-256 hash of the client manufacturer's public key.
3. The nonce is the random number generated by the client.
4. The software component to attest. There are six options for this, VNC server, UPnP server, UPnP control point, RTP server and RTP client as well as the option to attest all components.

3. After the Terminal Mode server receives the attestationRequest it needs to do some calculations, for this it uses the TPM. The TPM contains a number of 160-bit registers called PCRs (Platform Configuration Registers). The PCRs is used to let a relying party obtain unforgeable information about the platform state. [14]

The platform can be seen as a number of components. These components can receive and pass control to other components. The first calculation done by the client is to measure another component, i.e. compute its hash. If the hash matches an expected value then a URL to this component and optional a hash of the public part of the application key are extended into PCR 10 (PCR 10 is reserved for this use). This is done by using the TPM_Extend command.

A quote Signature is then created by using the TPM_Quote operation. As input to the TPM_Quote operation the PCR and the nonce received from the client is used. This operation signs the PCR using a certified device key.

4. After the server has completed its calculations it sends the attestationResponse to the client. The attestationResponse is a XML message which consists of the items in Table 15. The text below describes the contents of the table.

1. The Terminal Mode version used, currently 1.0
2. Result is 0 if attestation was successful, 1-5 if attestation was unsuccessful. The number indicates what went wrong. This field is optional.
3. Attestation field contains attestation of one component. There can be several fields like this if several components should be attested.
   (a) The software component that was attested.
   (b) The old value of the PCR 10 which is a 20-byte value.
   (c) A structure over which the quoteSignature is calculated. Contains the current value of PCR 10.
   (d) The calculated quoteSignature. 128-bit or 256-bit value.

Table 15: Contents of device attestationResponse message

#	Description	
1	Version number	
2	result	
3	a	componentID
	b	oldValue
	c	quoteInfo
	d	quote Signature
	e	URL
	f	application PublicKey
4	device Certificate	
5	manufacturer Certificate. Can contain several instances	

(e) URL to the attested component.

(f) Public key part to the attested application. May later be used to authenticate transferred data. The key pair should be a RSA 1024-bit or RSA 2048-bit device key. This field is optional.

4. X.509v3 certificate issued by the Terminal Mode server device manufacturers. Contains public part of the RSA 1024-bit or RSA 2048-bit device key.

5. One or several X.509v3 certificates. Issued for the server by the client manufacturer.

5. To verify the quoteSignature the client performs a number of steps. It performs a number of calculations and compares the output to the values received in the attestationResponse message. If the calculated values match the received values, then the attestation has been verified.

6.1.3 Physical, Link

The Terminal Mode standard depends on the link layer & physical layer to be secure. The proposed connection methods USB, WLAN and Bluetooth will be analyzed but other connection methods could be used.

Securing the link layer is supposed to keep an attacker out of the system but the user is still able to connect other equipment which can be of threat to the system.

For both Bluetooth and WLAN the client and the server is unable to verify that enough security is used. If the server and the client could verify it, this would still be useless since it could only verify the security on the endpoint. Somewhere else on the link lower security could be used.

USB

The host authentication relies on that a USB device is only connected to one device at the time. It would be impossible for an attacker to connect to the USB but the user could still connect another device. If this device is compromised then it could open up for a man-in-the-middle attack scenario.

WLAN

WLAN authentication mechanisms like WPA or WPA2 must be used. These mechanisms keep an attacker out of the system and they are considered to be secure as long as the password is complex enough.

Bluetooth

When a Bluetooth PAN connection is used to connect the Terminal Mode client and server Bluetooth authentication and pairing mechanisms must be used. The Terminal Mode standard relies on the security of the Bluetooth protocol but it cannot verify that security actually is enabled and used.

6.2 Security Mechanisms Analysis

6.2.1 Content Attestation

If correctly used the content attestation mechanism will prove the authenticity of the framebuffer updates and the framebuffer context messages. The data can then be verified to come from the VNC server attested with the device attestation protocol.

When using the content attestation mechanism the client first generates a session key that should be used. The client then encrypts this session key with the servers public key from the device attestation protocol so it only can be decrypted with the servers private key. The session key is then sent encrypted to the server. At this stage the only two parts that can know the session key is the server and the client. When the server has generated the attestation response message it encrypts it with the session key. When the client receives the message it will decrypt it with the session key. If the message has been modified during the transmission the values will not be readable since no one else than the server could create a message encrypted with the correct session key.

However the use of a session key is optional and if it isn't used the mechanism can not guarantee the authenticity of the messages, it will only make it harder for an attacker to accomplish an attack where the framebuffer update content have been changed.

Since the content attestation mechanism only impedes an attack when the session key is not used and not fully protect from an attack it will only give false security if it is used in an incorrect way.

Symmetric & Asymmetric keys

Both symmetric and asymmetric keys are used by the content attestation mechanism. It is known that symmetric encryption is way much faster than asymmetric encryption. But since the maximum data size to encrypt is maximum 122 bytes it seems unnecessary to use a symmetric session key. Instead the response message should be encrypted with the servers private key. This will make the message readable by every part that can capture it but the asymmetric key pair makes it impossible for no one else than the server to forge the message. The message is decrypted with the servers public key. This would be a more straightforward and easier approach.

Key Size

According to [15] keys with the size 112 bits would have a lifetime to at least 2030 and that 128 bits keys would have a lifetime far beyond 2030. This in combination that the session key can be changed between every message which makes a key old the second a new key is used makes cracking the symmetric key unfeasible.

6.2.2 Device Attestation

Device attestation uses TPM with X.509 certificates and attestation mechanisms standardized by the Trusted Computing Group [16]. It is by the use of TPM and its mechanisms that the client can check that the server is from a compliant manufacturer and that it is running approved software.

TPM controls that no change has been made to the components so that they are behaving as expected. This way by the use of TPM together with certificates from the manufacturer the Terminal Mode client can be sure that the Terminal Mode server is running approved software.

The question is if TPM is secure or if it is possible to hack it. To analyze step by step how TPM works and which security flaws it might have would be a too large task for this master thesis project. Instead focus has been put on specific parts and searching the Internet for known flaws in TPM.

A few successful attacks against TPM has been found, [17] and [18]. These attacks are however very complex and requires physical access to the hardware. So it is highly unlikely that an attacker would go to this extent for the uses it can have in Terminal Mode.

TPM uses public key encryption with at least 1024-bit RSA keys. The 1024-bit keys may be possible to break today [15] with enough resources and time but to launch this attack during a Terminal Mode session would be unfeasible. Also 2048-bit key sizes can be used which would make it nearly impossible to break the key.

Another question is if the X.509 certificates pose a security risk. There exists some exploits, [19] and [20] in the X.509 certificates but no indication that they have been compromised has been found.

6.3 Security Mechanisms Usage

The question of what should be protected needs to be asked. Computer security has in the past usually been to protect from an attacker but with Terminal Mode the system needs to be secured against the user as well. The reason is that the user should not be able to exploit the Terminal Mode standard and connect a device with software that could change the display content to something else than what is intended by the certified Terminal Mode server. If the user succeeds with this the user could show anything on the screen while driving, something that a car manufacturer want to avoid.

The security needs also, as traditional, secure the Terminal Mode devices from ordinary threats from an attacker. An attacker should not be able to eavesdrop the connection, control the Terminal Mode server or modify the content sent to the client.

This security review will not focus on client security like code exploits and what can be done with the computer running the client software if it is exploited. It is of course important to protect the client so an attacker can't gain access to parts of the car, or control the car, but this is left to the developer of the Terminal Mode client to secure.

6.3.1 Security threats

The following security threats have been identified as potential targets. *1a* and *2* was taken from the Terminal Mode standard but *1b* and *3* are also potential threats not mentioned in the standard.

1. Showing content from an uncertified Terminal Mode server
   (a) User intended
   (b) By attacker
2. Leaking content to an attacker
3. Unauthorized access to the Terminal Mode server

6.3.2 Protection against security threats

This section will describe how the available security mechanisms protect the system from the security threats mentioned above.

1a. One of the threats in Terminal Mode is that a user might want to be able to view any content from the smartphone in the car while the car manufacturers only want to allow specific applications. If neither device attestation nor content attestation is used a user could easily show any content it wish.

If only device attestation is used the Terminal Mode client can be sure that the attested component is correctly implemented on the Terminal Mode server. So if device attestation is used it is not possible to modify these components by e.g. jailbreaking the smartphone. However if the user could redirect the traffic to another component with similar functionality then the user could still do anything with the system.

If only content attestation is used without a session key then the Terminal Mode client can e.g. be sure that the framebuffer updates sent by the Terminal Mode server has not been modified.

If only content attestation is used without a session key then it will be harder to connect to another VNC server because content attestation response messages needs to be generated. This message can only be generated by a server supporting the content attestation mechanism which is a Terminal Mode specific feature and not standard in VNC.

By using both device and content attestation in a correct way with the session key in the content attestation messages request is encrypted with the Terminal Mode server's public key. The response message is encrypted with the session key. This ensures the Terminal Mode client that the right component sends the data and that the data has not been modified.

1b. The security against an attacker is similar to the security against a user with the exception of that the attacker doesn't have access to the link layer. Link-layer security does not matter for the user but an attacker must first break the link-layer security mechanisms to be able to launch any attack. If USB is used the attacker must get physical access, if WLAN is used the attacker must break WPA/WPA2 and if Bluetooth is used the attacker must break the security mechanisms used by Bluetooth. It is not very likely that the attacker is successful in accomplishing any of this.

But if the attacker in some way could bypass the link-layer security mechanisms then it would still have to deal with device and content attestation in the same way as a user does. By using a man-in-the-middle attack the attacker could bypass the security mechanisms offered when using only device attestation or content attestation, either by redirecting the traffic to another component or by changing the content of the messages. However if both device attestation and content attestation are used the attacker would have to figure out the servers private key to do any harm. As mentioned in section 6.2.1 it is unfeasible to do this.

2. Leaking content to an attacker is a confidentiality breach of security. In VNC the framebuffer updates is sent without encryption and the only mechanism protecting the display content, or other content as well, from leaking is the link layer security.

This means that when USB is used the client manufacturer needs to be sure that the device is connected directly to the Terminal Mode server. This needs to be fulfilled so that an attacker cannot insert a device between the server and the client to create a man-in-the-middle attack.

When Bluetooth is used the Terminal Mode server needs to ensure that security mechanisms is correctly used when setting up the Bluetooth PAN.

If the WLAN security is breached, either if WEP or WPA/WPA2 with a weak password is used, or an attacker gets access to the link layer it could easily eavesdrop the link and listen to all content sent. It could also set up a man-in-the-middle attack by acting as both the Terminal Mode server so it gets all packets and then relay them to the intended destination.

3. Unauthorized access to the Terminal Mode server is a integrity breach of security. The Terminal Mode standard doesn't have any protection against unauthorized access to its UPnP services or the VNC server. The only protection mechanism is the link layer security. If an attacker succeeds in breaching the link layer security then the Terminal Mode server is completely open and accessible for the attacker. It would also be possible for an attacker to connect to the VNC server since the Terminal Mode server doesn't verify the client.

6.4 Driver Distraction

Driver distraction is something that has to be carefully considered. The Terminal Mode client can't show arbitrary data on the display which could distract the driver. To tackle this problem the Terminal Mode standard have a mechanism called context information which contain information about what type of content is sent to the client.

The context information is sent as a pseudo encoding together with the data in the framebuffer updates. It has the form of several rectangles each describing the context of that specific area in the framebuffer. If several rectangles have information about the same area the last one is valid. The context information should be sent before the framebuffer data so the client can decide if the data is allowed to be shown or not. The responsibility to decide if data is approved or not is put on the client.

The specific context information pseudo rectangle massage and its content are showed in table 16.

The text below describes the content of the message.

1-4. Information about the rectangle position and size in the framebuffer

5. Encoding type -524 for context information

6. Application ID

Table 16: Content of context information message

#	Description
1	X-Position
2	Y-Position
3	Width
4	Height
5	Encoding type (-524)
6	Application ID
7	Trust Level for Application Category
8	Trust Level for Content Category
7	Application Category
8	Content Category
9	Followed content rules

7. The trust level is information about which part that provides the context information. The available trust levels are *no trust*, *user configuration*, *self-registered application*, *registered application* and *application certificate*. For *self-registered application* the application specifies the context information and for *registered application* the provided data is under control of the VNC and UPnP server. *Application certificate* is a trust level where the context information is delivered by a third party certification. The user or the application can't change the values if the trust level is *registered application* and if device attestation is used in combination with content attestation of the context we can be certain that this information is not tampered with.

The content and application can be divided into the following categories.

Application Categories:

- Unknown Application
- UI Framework
- Phone call application
- Media applications
- Messaging applications
- Navigation
- Browser
- Productivity
- Information
- Social Networking
- Personal Information manager
- UI less applications
- General System

Content Categories:

- Text
- Video
- Image
- Vector Graphics
- 3D Graphics
- User Interface
- Car Mode
- Miscellaneous content

If no context information is sent the client should assume that all values is set to zero. This means that the context should be interpreted as an unknown application and that the user is responsible for the content.

6.5 Interaction with car

Tons of data is available today in the car and this data could be visualized and used by the head unit. The reason to consider car interaction is that application running in the head unit in systems like this could take advantage of this data.

One of the things that is missing in the Terminal Mode standard is how data could be exchanged between the systems in the car and the applications in the phone. This will however be included in

the next version of Terminal Mode[21]. If data from the car should be available in the applications on the phone as of today using the Terminal Mode standard then the data needs to be retrieved in some other way than with Terminal Mode, for example through a Bluetooth connection or a WLAN connection to the car system. It would in this case be application dependent.

The Terminal Mode standard could be extended to support reading and writing values to the car. Values that are writable needs to be controlled and only a subset should be writeable. For example it should be able to read the velocity of the car but not change the velocity. The cars head unit and the Terminal Mode client should be the part responsible of the security. It should keep track of what values that are writable and ensure that this is followed. The obvious reason for this is that an application on the phone shouldn't be able to take control of the car.

Adding functionality for car interaction could be done in the following three ways:

1. Another extension message in the RFB protocol
2. Extra messages for UPnP
3. Another custom protocol with the only purpose to exchange data with the car

The third method is the most likely to be used in the next version. A server for car interaction could then be started with UPnP.

Regardless which method is used for the communication an initialization phase needs to exist. This phase should inform the client which values that are read and writeable from the server. The values could then be exchanged by three different models. These models will be described in the following text.

1. Values from the car could be sent at specified intervals. Depending on which type of data that is sent they should probably be sent at different intervals. For example the outside temperature value can be sent every minute while the cars velocity is interesting to update every second. The data that should be sent periodically needs to be requested and also the time interval needs to be specified. There also needs to be a method to stop the subscription of a message. This could be done with a re-initialization where this value is left out and not requested any more.
2. Another method for transferring data is polling. The client would then request a value and the server would respond with it. This could happen anytime and the client always initializes the procedure.
3. The third solution is to have the server send a value when the value is changed. The Terminal Mode server would then inform the Terminal Mode client of which values it wants to have updated by an initialization message. An argument to a value in this initialization message can be how often a value can be sent. If a value changes constantly the client should not have to receive all updates.

Probably all these should be used in different combinations depending on which values are transmitted, this because different values needs to be updated at different intervals.

6.6 Alternative Solutions

To know if Terminal Mode is a good concept other solutions need to be considered and compared against Terminal Mode. The requirement was that the solution should provide third party apps inside the car. The Internet was searched and the result was that many car manufactures have different infotainment systems. Some of them are Contintental AutoLinq [22], Saab IQon [23], Ford Sync [24], Kia Uvo [25], Audi MMI Touch [26], Mercedes mbrace [27], Hyundai Blue Link [28], GENIVI [29], GM OnStar [30] and Toyota Entune [31]. GENIVI is not a car manufacturer but an alliance between several car manufacturers and other tier 1 suppliers for development for in car infotainment systems. Out of the solutions mentioned only Saab IQon and Continental AutoLinQ provided a solution with an integrated app store, which enables users to download and run third party apps in the infotainment system. Saab IQon and Continental AutoLinQ which both have

application stores will therefore be described and compared to Terminal Mode in the two following sections. They are both Android-based and will in the following be referred to as Android-based systems.

6.6.1 Saab IQon

Saab IQon is a recently announced infotainment system concept by Saab Automotive. The concept is built around an Android system connected to an 8 inch display in the dashboard. The system will have functions for streaming music and entertainment but they also open up for third party developers to put downloadable apps in an app store for Sabb IQon. [23]

The idea of the app store is that users will be able to download different types of apps such as entertainment, weather, traffic info or online services. Safety is preserved by the control mechanism that Saab Automotive has to evaluate and approve every app that the third party developers want to put in the app store. Having an app store Saab claims that the apps can be upgraded and the system personalized during its life cycle. [23]

One feature is that the third party apps can use is a special API which allows access to over 500 sensor values. These values can be anything from vehicle speed, location and direction to the outside temperature and the suns position. Saab hopes that these values and sensors will get the developers creative and trigger the development of third party apps. [23]

6.6.2 Continental AutoLinQ

AutoLinQ by Contintental is very similar to Saab IQon. The biggest difference is probably that Contintental in contrast to Saab is not a car manufacturer but a subcontractor. Android is used as the operating system in AutoLinQ. It also has a SDK for easier development of in car apps. An API exists so that values can be read from and written to the cars system.

6.6.3 Comparison with Terminal Mode

To compare an Android-based system with the Terminal Mode standard the following subjects were identified as interesting to discuss and compare; Apps, Hardware, Internet Connection, Cost, Ease of use & Car Interaction.

Apps

For apps to be available in Terminal Mode they need to be adopted for content awareness. The manufacturer of a Android-based system has total control of the apps allowed so they can decide which apps that are allowed to run. It would probably be harder to get an app approved for IQon or AutoLinQ app store than at a Android app store. AutoLinQ have however a feature that allow all Android apps to run when the car is standing still. The conclusion is that apps needs to be adopted both for Terminal Mode and for Android-based systems but in different ways.

Hardware

Terminal Mode requires less hardware than an Android-based system and since Terminal Mode is a lightweight solution it can probably be built into already existing infotainment system. Also the Terminal Mode server is upgradeable by replacing the smartphone providing the service. The client doesn't need to be updated other than for software updates. An Android-based system will be built into the car and can't be changed as easily as the Terminal Mode hardware. Also since the Android-based system hardware is not replaceable it needs to be of higher capacity to meet higher demands in the future. Of course one requirement for Terminal Mode is that the user has a smartphone with Terminal Mode.

Internet Connection

Today most smartphones are connected to the Internet. Therefore a smartphone with Terminal Mode will probably already be connected to the Internet. The Saab IQon system requires an extra 3G subscription to connection the Internet which results in extra costs. Android-based system users which already have a smartphone might be able to connect the system to the Internet through the

smartphone with Internet tethering. The press release from Saab about IQon or the AutoLinQ webpage doesn't however tell if this is possible.

Cost
Hardware costs for the car manufacturer will be much higher for an Android-based system than for Terminal Mode. The Terminal Mode server is provided by the mobile phone manufacturer and the car manufacturer doesn't have to pay for that. The car manufacturers will however need to pay for the whole Android-based system.

Another cost to consider is the extra 3G subscriptions. This cost will however be put on the customer.

Also the cost for app stores and infrastructure needs to be considered. Android-based system will require an organization for approving application, maintaining servers for the app store. The Terminal Mode clients will however be software only existing in the car, no infrastructure will be needed for the car manufacturer to invest in since this is provided by the manufacturer of the smartphone.

Ease of use
Android-based systems will start directly when the user starts the car while the Terminal Mode server needs to be plugged in or connected to the infotainment system. The advantage for Terminal Mode is that the user is already familiar with the apps in the smartphone and doesn't have to learn a new system. Also for example a journey can be planned in the house with the GPS application and be brought out to the car without having to redo it.

Car Interaction
The Android-based systems from Saab and Continental have APIs for data exchange with the car. Terminal Mode doesn't have this in the current version of the standard but this will come in the next version of the Terminal Mode standard [21].

6.7 RTP or Bluetooth for audio

Terminal Mode can transfer audio by either Bluetooth or RTP. The Terminal Mode server must make a distinction between speech from a phone call or application audio but it can also distinguish between several other categories of audio. The speech audio needs to be bidirectional while the application audio can be unidirectional.

RTP use UDP for sending data which means that it is a stateless connection. On the Terminal Mode server there have to exist a RTP server for sending audio to the RTP client on the Terminal Mode client. This is used when media is played on the Terminal Mode server device and sent to the client. If speech should be enabled an RTP server needs to exist on the Terminal Mode client and a RTP client on the Terminal Mode server. It is always the RTP server that sends the audio. The RTP server and client on the Terminal Mode server are started with UPnP. Technical information about RTP is left out here since it can be read in the Terminal Mode standard. RTP supports different payload types. The RTP implementation in Terminal Mode must support payload type 0 (8 bit, 8 kHz, mono) which is a RTP requirement. It must also support payload type 99 (16 bit, stereo, 48 kHz) which is specified by the Terminal Mode standard. Payload type 98 (16 bit, 48 kHz, mono) is specified by the Terminal Mode standard but optional. Information from the UPnP action GetApplicationList specifies which payload type that will be used. RTP can be used either unidirectional (audio is only sent to the client) or bidirectional (speech is sent from the client to the server and audio from the server to the client).

Bluetooth can be used for both phone calls and for streaming audio. The Terminal Mode standard specifies two relevant Bluetooth profiles for use in Terminal Mode, HFP (Hands-Free Profile) and A2DP (Advanced Audio Distribution Profile). The HFP is used for voice calls and is often used in Bluetooth handsfrees, it only has support for monophonic audio [32]. A2DP is used for streaming high quality audio in either mono or stereo [33].

The user may want to use their Bluetooth headset for phone calls instead of speaking through the head unit. If a HFP is paired with the Terminal Mode server device then audio will be sent to the headset. An advantage of using RTP and the head unit for phone calls is that the audio can

be sent in a higher quality. HFP use 8 bit mono while RTP can use 16 bit mono/stereo for audio. If RTP is used for phone calls then echo cancellation needs to be implemented. This is a feature that already exists in Bluetooth.

An interesting scenario occurs when the user has connected the smartphone to its handsfree and then gets into the car and connects the Terminal Mode client. There exist several other scenarios like this and the question is with which method the audio should be transferred. To solve this problem the Terminal Mode standard specifies a priority list that tells which transfer method that should be preferred over others for different use cases. RTP is however recommended in the Terminal Mode standard to use for transferring media sound instead of A2DP, but in the end it is the clients decision which method that should be used.

7 Discussion

During this master thesis project and the writing of this report, many topics came up worth discussing. We will first discuss the expected result and the results from the implementation part. Some optimizations that could have been done will then be discussed and then the design choices made and the motivations for these. Furthermore the authors' thoughts about the concept in general will be discussed and possible future work presented.

The results of the FPS measurements were a big disappointment. At the beginning we thought that the prototype device would perform good but before it arrived we read in the manual that it only supported 4 FPS and shouldn't be used in any public demo. When doing the tests the maximum FPS we reached was however 1.16 with out client and 1.22 with the prototype client. Just when this master thesis project started Nokia sold out their last N97-mini Terminal Mode prototype device. This device was showed at public demos and would probably have generated much better performance than the N8 did. Sadly we couldn't get our hands on one of these after we got the N8. It would have been really interesting to compare the performance between the N8 and the N97-mini, but we can say that we believe that the N97-mini would have outperformed the N8.

The attentive reader noticed that the prototype client did 1.22 FPS but our implementation only did 1.16 FPS. The reason for this is that the higher FPS was reached with RLE encoding and therefore the data size was of unknown size and was read from the socket pixel by pixel. A great performance increase would have been to create a buffer that the socket data could have been written to. Then all parts which do RLE could read from this buffer instead. If the buffer becomes empty a new system call for reading from the buffer should be made to fill it up again. This implementation would have minimized the costly recv-calls on the socket.

Another optimization that could have been made was when colours was converted from RGB343 to ARGB888. The 3 bits for red should be converted to 8 bits and to do this it should have been multiplied with a fixed number instead of left shifted. The left shift made the picture darker but it was impossible to see with the eye so we left it that way.

7.1 Design Choices

When doing the implementation part of this master thesis project we had several design choices to consider.

- The first design choice was if we should have a single threaded or a multithreaded solution. Advantages of a single threaded solution were that communication between threads didn't have to be implemented.

  The multithreaded solution had more advantages than the single threaded solution. First computers get more and more parallel processing capabilities in forms of multicore processors. The second is that the implementation gets easier when we could have one thread for each task the client have to do. Implementing several functions such as the VNC client, the RTP client and the UPnP control point in the same thread would have been possible but hard.

- Another consideration regarding the threads was if they should be started at initialization or if they should be created when needed. We decided that they should be created once at initialization so that less thread handling had to be done. With this design we minimized the risk of having a thread hanging loose.

- We thought of using or implementing a system for message passing between the threads but since the information that should be sent and received was so small we decided not to do so. Communication between the threads was done with semaphores and control variables.

- Bilinear interpolation was chosen as the interpolation technique because the calculations for bicubic interpolation would be to processor demanding. The reason to not choose the simpler nearest neighbor interpolation was that it didn't look good enough to put in a commercial product and we wanted to explore if software bilinear interpolation was possible. When bilinear interpolation was implemented it was easy to create the nearest neighbor interpolation at the same time.

- UPnP was not familiar to us in the beginning and we had to do a lot of reading to understand the Terminal Mode standard. Therefore we decided to use a UPnP SDK so we didn't have to put so much time on UPnP. After a while when the UPnP control point was integrated our knowledge about UPnP had increased and we realized that it would have been possible to implement the control point from scratch. This alternative was then chosen when we didn't manage to get the UPnP control point developed with the UPnP SDK to work.

- We thought of having an extra thread for sending key events. The reason to this was that Betula HUC could add the key event to a message queue and then it would be sent by the Terminal Mode client. With this design Betula HUC wouldn't have been affected if something hangs in the Terminal Mode client which would be a good feature. Because of the increased complexity we decided not to use this design but a more simple when the Betula HUC thread sends the key event.

- Using two or more threads for handling received VNC data and doing the scaling would probably be a good performance boost. In the part were scaling was done could easily have been multi threaded and would probably have scaled good since it should read from one place and write to another. This was however not implemented and only one thread was used for the scaling.

7.2 Terminal Mode as Concept

In chapter 6 different parts of the Terminal Mode concept has been analyzed, in this section a discussion of the concept as a whole will be done.

We believe that the idea with integrating the smart phone with the cars infotainment system is a good idea. The users which are most keen in using applications in their cars are probably also the users which most frequently use applications in their smartphones. Furthermore the smartphones and its applications are developed at high speed which can help assure the car manufacturers that their users always get the latest technology. Also the Terminal Mode concept do not limit a car manufacturer to just using Terminal Mode. Instead it can be used as an application in the cars infotainment system which can allow users who do not own a smartphone to still be able to use some basic functionality.

The use of well-known standardized protocols will make it easier for car and smartphone manufacturers to embrace the concept. Terminal Mode is also developed in a way which makes it possible for them to adapt to changes in smartphone development. E.g. USB is the most commonly used interface for connecting it to a computer but other transport methods can also be used, most smartphones supports Bluetooth but RTP can also be used for audio. So no matter in which direction the smartphone development will go, as long as there is IP connectivity between the client and the server the concept will work. Also the fact that the Car Connectivity Consortium[4] is further developing Terminal Mode will assure that several car manufacturers can easily adapt to changes in the standard.

Two important aspects of Terminal Mode is security and driver distraction. We believe that to satisfy the car manufacturers these aspects needs to work well. The security part looks good, if Terminal Mode is properly used then the security mechanisms provided should be enough. We are a bit more uncertain about driver distraction. Terminal Mode offers good ways of dividing applications into different categories. But can a car manufacturer be certain that the applications have been labeled into the right category? It is probably possible to assure this but this will lead to another question. If the applications is correctly labeled and mechanisms which make it impossible to change category for an application is in place, will a lot of applications still be available in the smartphone or will only a small subset be available for use? To make the concept Terminal Mode to a success we believe that this is one of the most important aspects to solve. Users want to use as many applications as possible in the car while car manufacturers want to be in control. If this labeling of applications goes smoothly both for existing applications as well

[4]As of today the members of the Car Connectivity Consortium are Alpine, Daimler, General Motors, Honda, HTC, Hyundai Motor Company, LG Electronics, Nokia, Panasonic, PSA, Samsung, Toyota, Volkswagen, Delphi, Denso, Ixonos and Sony.

as for future applications then we believe that Terminal Mode has a large chance of becoming a well-used standard.

One part that we felt was missing in the Terminal Mode concept was the interaction with the car. As mentioned in section 6.6 other concepts aim to have interaction with the car. This will however also be included in Terminal Mode version 1.1 [21]. With this new version we feel that the last piece of the puzzle will be added and it is now a more complete concept.

7.3 Future Work

The Terminal Mode client can be further developed for increased performance, functionality and interoperability. For performance more work can mainly be done on the framebuffer updates and for more functionality additional extension messages can be enabled. To increase the interoperability more work can be done on the UPnP control point. Other additions such as adding audio or security can also be done.

When the new Terminal Mode version 1.1 [21] comes out it will be possible to more closely integrate Terminal Mode into the car. This leads us to a proposal for a new master thesis. The object of the master thesis could be to add a new module to the Terminal Mode client. This new module should take care of interactions with the car. It should be developed according to the Terminal Mode Data Exchange Protocol [21] and communicate with a suitable data bus for use in cars.

8 Conclusions

Infotainment systems in cars are on the rise. We believe that many new infotainment systems will be seen in cars over the next couple of years. If the members of the Car Connectivity Consortium starts to get out the first implementations of Terminal Mode in the near future then Terminal Mode has a good chance of becoming one of the more commonly used solutions for this. Possibly even the most common solution for infotainment systems.

The fact that Terminal Mode can be used as an application in a cars infotainment system in addition to other functionality will increase the likelihood of that Terminal Mode will be widely used. The reason for this is that car manufacturers can still develop their own infotainment system and have Terminal Mode as a backup or as an extra feature in the system.

The Terminal Mode client developed during this master thesis project can serve its purpose and be used for demonstrations. However another Terminal Mode device than the Nokia N8 prototype device should be used because of the N8's low frame rate. Althouth there are currently no other new Terminal Mode devices available to order as of today.

References

[1] J. B. et al., *Terminal Mode Technical Architecture, Release Version 1.0*. Nokia Corporation, Audi AG, BMW AG, Daimler AG, Porsche AG, Volkswagen AG, 2010. http://www.terminalmode.org/en/developer/Downloads/.

[2] "Terminal mode," June 2011. http://www.terminalmode.org.

[3] A. Arnholm, "At the forefront of automotive technology." Web page, May 2011. http://www.mecel.se/.

[4] A. P. et al., *UPnP^{TM} Device Architecture 1.1*. UPnP Forum, October 2008. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf.

[5] "Soap tutorial," 2011. http://www.w3schools.com/soap/default.asp.

[6] "Understanding upnp," 2000. http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc.

[7] T. Richardson, *The RFB Protocol*. RealVNC Ltd, November 2010. http://www.realvnc.com/support/documentation.html.

[8] R. C. Gongalez and R. E. Woods, *Digital Image Processing*, pp. 65–68. Pearson International Edition, third ed., 2008.

[9] S. Avidan and A. Shamir, "Seam carving for content-aware image resizing," *ACM Trans. Graph.*, vol. 26, July 2007.

[10] "Forum nokia support center." Web page, 2010. https://forumnokia.secure.force.com/apex/DDP/, Login required.

[11] A. P. et al., *Universal Serial Bus Communications Class Subclass Specifications for Network Control Model Devices, Revision 1.0 (Errata 1)*. USB Implementers Forum, Inc., November 2010. http://www.usb.org/developers/devclass_docs.

[12] Nokia Corporation, *Nokia Terminal Mode Protoype Kit User Guide: N8 Delta*, Mars 2011.

[13] O. S. Projects, "Developer tools," February 2011. http://opentools.homeip.net/dev-tools-for-upnp.

[14] M. Ryan, "Introduction to the tpm 1.2." DRAFT, June 2011. ftp://ftp.cs.bham.ac.uk/pub/authors/M.D.Ryan/08-intro-TPM.pdf.

[15] B. Kaliski, "Twirl and rsa key size." Technical Newsletter, May 2003. http://www.rsa.com/rsalabs/node.asp?id=2004.

[16] "Trusted computing group." Web page, June 2011. http://www.trustedcomputinggroup.org/.

[17] W. Jackson, "Engineer shows how to crack a 'secure' tpm chip." Web page, June 2011. http://gcn.com/Articles/2010/02/02/Black-Hat-chip-crack-020210.aspx.

[18] "Tpm reset attack." Web page, June 2011. http://www.cs.dartmouth.edu/~pkilab/sparks/.

[19] A. Lenstra and B. de Weger, "On the possibility of constructing meaningful hash collisions for public keys," June 2011. http://www.win.tue.nl/~bdeweger/CollidingCertificates/ddl-full.pdf.

[20] P. H. Cameron McDonald and J. Pieprzyk, "Sha-1 collisions now 2^{52}," June 2011. http://eurocrypt2009rump.cr.yp.to/837a0a8086fa6ca714249409ddfae43d.pdf.

[21] "Terminal mode: Techical release," June 2011. http://www.terminalmode.org/en/agenda/release-1-1.

[22] "Continental automotive - autolinq™." Web page, April 2011. http://www.autolinq.de/en/default.aspx.

[23] "World first from saab: Saab iqon - open innovation in car infotainment." Web page, April 2011. http://newsroom.saab.com/news/news/worldfirstfromsaabsaabiqonopeninnovationincarinfotainment.5.741c75ab12da7f448807ffe719.html.

[24] "Sync — change songs, make phone calls, get directions - all using your voice — forc.com." Web page, 2011. http://www.ford.com/technology/sync/.

[25] "New kia technologies showcased at 2010 ces." Web page, April 2011. http://kia-buzz.com/?p=3654.

[26] "Audi sverige > nya bilar > a8 > mmi touch." Web page, April 2011. http://www.audi.se/se/brand/sv/models/a8/a8/equipment/multi_media_interface/mmi_touch.html.

[27] "Mercedes-benz - mbrace." Web page, April 2011. http://www.mbusa.com/mercedes/mbrace/overview/src-4376.

[28] "Hyundai motor america — blue link homepage." Web page, April 2011. http://www.hyundaiusa.com/bluelink/index.aspx.

[29] "Genivi alliance." Web page, April 2011. http://genivi.org/.

[30] "Navigation system, auto security, vehicle diagnostics - onstar." Web page, April 2011. http://www.onstar.com/web/portal/home.

[31] "Toyota entune." Web page, April 2011. http://www.toyota.com/entune/.

[32] A. W. et al., *HANDS-FREE PROFILE 1.5*. Car Working Group, 2005. https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=41181.

[33] M. L. et al., *ADVANCED AUDIO DISTRIBUTION PROFILE SPECIFICATION*. Audio Video WG, 2007. https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=66605.

A FPS Test Data

Table 17: FPS with RLE

	Prototype Client				Client with Nearest Neighbor Scaling				Client with Bilinear Scaling			
	RGB343	RGB444	RGB565	ARGB888	RGB343	RGB444	RGB565	ARGB888	RGB343	RGB444	RGB565	ARGB888
1	1.33	0.72	0.20	0.14	1.21	0.84	0.15	0.07	1.12	0.78	0.16	0.07
2	1.29	0.73	0.18	0.14	1.32	0.88	0.15	0.08	1.15	0.83	0.16	0.07
3	1.32	0.75	0.19	0.14	1.24	1.03	0.15	0.08	1.10	0.85	0.16	0.07
4	1.34	0.72	0.19	0.14	1.15	1.01	0.15	0.08	1.10	0.80	0.15	0.07
5	1.20	0.69	0.19	0.14	1.18	0.92	0.15	0.08	1.07	0.81	0.15	0.07
6	1.24	0.72	0.19	0.14	1.20	0.95	0.15	0.07	1.14	0.93	0.16	0.07
7	1.16	0.73	0.19	0.14	1.20	0.92	0.14	0.08	1.11	0.85	0.15	0.07
8	1.22	0.67	0.19	0.13	1.12	0.89	0.15	0.08	1.21	0.82	0.15	0.07
9	1.08	0.73	0.19	0.14	1.16	0.96	0.15	0.08	1.20	0.89	0.16	0.07
10	1.22	0.74	0.20	0.14	1.25	0.91	0.15	0.08	1.20	0.87	0.15	0.07
11	1.24	0.71	0.22	0.14	1.18	0.80	0.15	0.08	1.08	0.95	0.15	0.07
12	1.20	0.70	0.19	0.14	1.17	0.76	0.15	0.08	1.13	0.87	0.16	0.07
13	1.25	0.72	0.20	0.14	1.20	0.83	0.15	0.08	1.15	0.88	0.16	0.07
14	1.27	0.71	0.18	0.14	1.11	0.82	0.15	0.08	1.13	0.89	0.15	0.07
15	1.20	0.65	0.18	0.14	1.12	0.89	0.16	0.08	1.13	0.80	0.16	0.07
16	1.26	0.68	0.20	0.14	1.05	0.89	0.14	0.08	1.16	0.93	0.15	0.07
17	1.17	0.72	0.19	0.14	1.15	0.79	0.15	0.08	1.18	0.97	0.16	0.07
18	1.27	0.72	0.19	0.15	1.20	0.87	0.14	0.08	1.09	0.92	0.15	0.07
19	1.17	0.66	0.19	0.14	1.12	0.88	0.15	0.08	1.14	0.81	0.16	0.07
20	1.07	0.71	0.19	0.14	1.08	0.75	0.15	0.08	1.09	0.81	0.16	0.07

Table 18: FPS without RLE

	Prototype Client				Client with Nearest Neighbor Scaling				Client with Bilinear Scaling			
	RGB343	RGB444	RGB565	ARGB888	RGB343	RGB444	RGB565	ARGB888	RGB343	RGB444	RGB565	ARGB888
1	0.18	0.19	0.17	0.04	0.37	0.37	0.41	0.05	0.36	0.34	0.33	0.05
2	0.18	0.18	0.17	0.04	0.36	0.37	0.40	0.05	0.39	0.35	0.34	0.05
3	0.17	0.17	0.17	0.04	0.34	0.38	0.38	0.05	0.37	0.35	0.33	0.05
4	0.18	0.17	0.17	0.04	0.38	0.41	0.41	0.05	0.37	0.34	0.35	0.04
5	0.18	0.17	0.17	0.04	0.35	0.39	0.43	0.05	0.37	0.35	0.31	0.04
6	0.16	0.17	0.17	0.04	0.34	0.38	0.40	0.05	0.38	0.33	0.34	0.05
7	0.18	0.18	0.17	0.04	0.36	0.37	0.39	0.05	0.36	0.35	0.32	0.05
8	0.19	0.18	0.17	0.04	0.36	0.40	0.39	0.05	0.35	0.33	0.32	0.04
9	0.18	0.16	0.17	0.04	0.37	0.39	0.42	0.05	0.36	0.35	0.32	0.05
10	0.18	0.17	0.18	0.04	0.37	0.39	0.40	0.05	0.35	0.33	0.34	0.05
11	0.17	0.17	0.17	0.04	0.36	0.38	0.38	0.05	0.35	0.33	0.36	0.05
12	0.18	0.18	0.17	0.04	0.40	0.42	0.40	0.05	0.35	0.34	0.33	0.05
13	0.17	0.16	0.17	0.04	0.38	0.40	0.42	0.05	0.35	0.33	0.34	0.05
14	0.17	0.17	0.17	0.04	0.37	0.40	0.41	0.05	0.36	0.34	0.35	0.05
15	0.17	0.17	0.17	0.04	0.39	0.42	0.42	0.05	0.37	0.31	0.35	0.05
16	0.18	0.17	0.17	0.04	0.36	0.40	0.39	0.05	0.36	0.33	0.32	0.05
17	0.18	0.16	0.17	0.04	0.39	0.38	0.40	0.05	0.36	0.35	0.35	0.05
18	0.18	0.17	0.17	0.04	0.37	0.38	0.41	0.05	0.35	0.35	0.32	0.05
19	0.17	0.17	0.17	0.04	0.35	0.40	0.41	0.05	0.33	0.35	0.37	0.05
20	0.17	0.18	0.17	0.04	0.38	0.41	0.39	0.05	0.36	0.32	0.32	0.05