LULEÅ UNIVERSITY OF TECHNOLOGY

2004:023 CIV

MASTER'S THESIS

Logging and Monitoring of TCP Traffic in SSH-tunnels

ANTON PERSSON

MASTER OF SCIENCE PROGRAMME

Department of Computer Science and Electrical Engineering
Division of Media Technology

2004:023 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 04/23 - - SE

Logging and monitoring of TCP traffic in SSH-tunnels

Masters thesis

Anton Persson

Abstract

The work documented in this thesis has been carried out at Ericsson Microwave Systems AB in Luleå. The company has developed a system for remote support. The system enables personnel working at Ericsson to access equipment at their customers' sites via the Internet or other networks. There are several reasons for providing such a standardized system. For example, if a customer calls Ericsson requesting support for a specific system, setting up a connection "Ad hoc" can be very time consuming and it is often a waste of resources. Developing a standard method for accessing the remote hardware removes these complexities. The system developed by Ericsson is called "The Remote Support Gateway", or RSG.

Historically, when Ericsson's customers needed support, a support engineer would be sent to their site to do the job. The customer could observe the work being conducted on their systems just by standing next to the engineer. With the new remote support systems they have limited possibilities to monitor the engineer. The RSG system provides some features allowing monitoring and logging of certain things, but the solution is not complete. Therefore it was decided to extend the possibilities. This thesis describes the development of those extensions.

The result is a software application which records, replays and monitors the described activities inside the RSG system. The application is divided into several parts, one for each operation.

The programs developed as part of this thesis are of practical use to Ericsson. The results are also of use for other things, which they were not originally intended, such as debugging or creating visual demonstrations of other applications.

Preface

This thesis has been the last mile on this journey we call "University education". Four and a half years at Luleå University of Technology has passed by, rather quickly I might add. I would like to take this opportunity to thank all my friends here in Luleå, especially my close friends in D99.

I would especially like to thank the following people. First, Joacim Häggmark, he was the one who offered me this job in the first place. I am also very grateful towards Andreas Lundström for proof reading my thesis over and over again. Other people, who have contributed with proofreading, are Ted Björling, Mattias Eklöf and Jonas Frisk. I would also like to thank my close friend and room mate, Linnéa Malmquist, who have made dinner for me more times than I can remember during my time here at Ericsson. Two people who have helped me out during some difficult times here in Luleå are Linda Larsson and Johan Thim, I am forever grateful.

I would also like to thank Roger Dahlén, my supervisor here at Ericsson, and all my colleagues who have assisted me. Finally I would like to thank my examiner Johan Kristiansson for giving me valuable input on my report.

Luleå 2003-12-04
Anton Persson

Contents

List of Figures

Chapter 1

Introduction

Historically, supporting large complex hardware systems required physical access. For example, if a telecommunications switch broke down somewhere, an engineer had to be sent to the site. Today, however, when information technology is readily available, complex support tasks can be handled remotely. This greatly simplifies the support work and makes it more efficient. A problem with this new approach is that the customers can no longer see the work as it is being done. In fact, much of what the support engineer did is never really known to the customer, because it can not be observed. To solve these problems, the support management systems should be extended with logging and monitoring functionality.

1.1 Background

Ericsson Microwave System AB in Luleå (formerly Ericsson Erisoft AB) has developed a system called the Remote Support Gateway, RSG. This system is used by Ericsson's support staff to access client equipment over a computer network connection. The system has become an Ericsson standard.

RSG uses the *SSH* protocol (see the glossary in appendix C) to provide encrypted connections. A support engineer may use a SSH-encrypted connection to send commands and transfer files. The encrypted connection can also be used to secure ordinary TCP traffic, by *tunneling*. Creating a tunnel in SSH means that the data in an ordinary TCP stream is routed via a dedicated channel inside the SSH stream. This provides security for unencrypted protocols such as FTP and Telnet.

Figure 1.1 illustrates a connection between a desktop computer at Ericsson's support office and a machine at the client's site. The machines, marked "Ericsson RSG machine" and "Customer RSG machine", act as special fire walls that only grants access to authorized personnel. A secure TCP connection (tunnel) has been created between the support engineer's machine and the client's hardware. There are three insecure protocols routed through the secure tunnel, FTP, Telnet and X-Windows. X-Windows (appendix A) is a windowing system, like the Microsoft Windows or Apple's Mac OS, and is used to run graphical applications on remote computers.

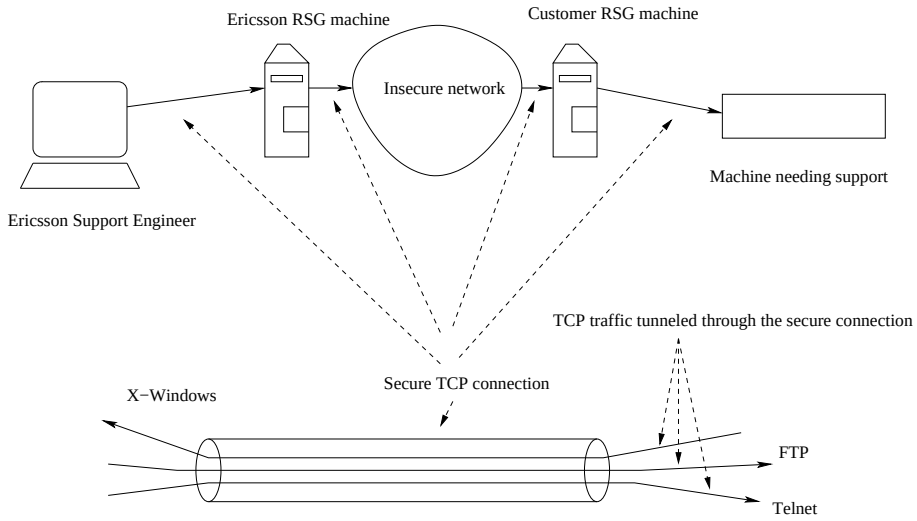

Figure 1.1: The RSG system

1.2 Purpose

The purpose of this thesis is to design a software application that solves the problems with monitoring and recording of tunneled TCP traffic. Such a solution has several benefits. The primary benefit is security, but it is also possible to record demonstrations on how to use certain applications for example. Furthermore, the recording feature may also be used for debugging, for example if a certain sequence of data caused a machine or program to crash.

1.3 Related work

There already exist several tools for monitoring network traffic at different levels of detail. There are also tools for logging information about the flow of data through networks and nodes, like *Ethereal* [C$^+$02]. The applications that are of most interest to this thesis are *XMX - An X protocol multiplexor* [Baz99] and *RFB Proxy* [Wau02].

RFB Proxy is a program that records the graphics shown on a screen. For example, consider a scenario where "Bob" is "surfing" the web. If RFB Proxy is currently running, everything that "Bob" sees on his monitor is recorded to a file. That file can later be played back by "Bob" (or someone else) to view the sites exactly as "Bob" saw when he was "surfing". RFB Proxy is design to work with *VNC* [Rea02].

XMX is an application, developed for the *X Windows* system, that enables two ore more persons to view or interact with the same application simultaneously. A special case of this is "monitoring", where one party is interacting with a program and another party is viewing their actions. XMX provides some valuable insight into what an X Windows logger must be able to do.

The difference to the work done in this thesis is that the thesis applications have more general goals. XMX can not play back any logs, and RFB proxy does not provide monitoring for example. It is also noted that RFB proxy is used with VNC while XMX is used with X Windows. The applications designed in this thesis provide logging of any TCP stream and replay of any implemented protocol. The application developed in this thesis is also specially designed for SSH, which gives a cleaner

solution.

1.4 Project description

A project can not exist without a clear definition. A software project is defined by its surrounding software, the languages used; its environment. A project also needs a set of goals and requirements that must be achieved. Some of these things will, however, evolve during the development, they are not static. They provide guidelines and scope for the project.

1.4.1 The development machine

RSG is based on the FreeBSD operating system. The system runs on any modern *x86* based PC. For development the main system has been a Pentium II based computer. Some Sun machines have been used for testing. The software used for developing the application is GNU Emacs and the GNU Compiler Collection.

1.4.2 The programming language

Since FreeBSD is a UNIX system, most of the system software is written using C, which is a "high-level" language originally designed for system programming. The language is the base for more modern languages like C++, Java and C#. The applications developed for this thesis are therefore also written in C.

1.4.3 Project goals and requirements

The goals of this project are to answer the following questions:

- How can the data streams in OpenSSH be intercepted?
- Can it be determined which protocols are used?
- Raw data logs occupy a lot of space, how can this be solved?

There is also a list of requirements:

- Implement recording and monitoring.
- Develop an extensible solution for playback.
- Minimize the impact on ordinary use.

Chapter 2

The development method

The software in this thesis is developed by one person. This means that the only communication is between the single developer and the "customer" (in this case Ericsson). This minimizes the development overhead. However, when problems occur with specific implementation details, for example, it can be a problem to not have anyone else involved with whom one can discuss the issue.

Since the X Windows part was prioritized that was investigated first. Since nothing about the inner workings of X was known in prior, more information was needed. After a literature study the development proceeded to an experimental stage, where a simple prototype application was developed. The prototype was used to draw conclusions on how to design a good solution. The useful parts of the prototype were saved and the rest was removed.

After the new version of the X Windows logger was finished, the work on OpenSSH began. The source for OpenSSH was modified to intercept X streams and route them via the logger. The method for intercepting ordinary TCP streams in OpenSSH was almost identical. Recording TCP streams was rather straight forward and could be implemented quickly. Finally a module for looking at Telnet logs was implemented.

Testing took place all through the development process, but no real evaluation was done until the end of the programming phase.

The different requirements, sub requirements and goals were discussed with the customer all through the development process. The original goals were modified over time since some of them were too complicated to be achieved during the time allocated for the project. Removed goals include "live" monitoring and automatic identification of TCP streams.

Chapter 3

Monitoring and logging TCP traffic

If a TCP stream could somehow be recorded, for example inside a switch or other network node, the data could in theory be analyzed. Encrypted TCP streams, however, may be hard to analyze. Furthermore, if several different streams are transferred via the same encrypted TCP line, it is even harder.

If it is assumed that a TCP stream can indeed be recorded, there are however additional obstacles. For example, there is no information inside a TCP stream that states which higher level protocol it transfers. Furthermore, even if the TCP stream itself is not encrypted the data inside the stream may still be. Additionally, there are so many protocols available that the results produced by any analysis may be incorrect. Even though the number of protocols that are used via RSG is somewhat limited, a major problem remains. Each protocol that is to be recognized must be understood, to some degree, by the monitoring software. This requires a deeper understanding of each protocol, and also their differences to other similar protocols. Only protocols that can be identified can be monitored in real time.

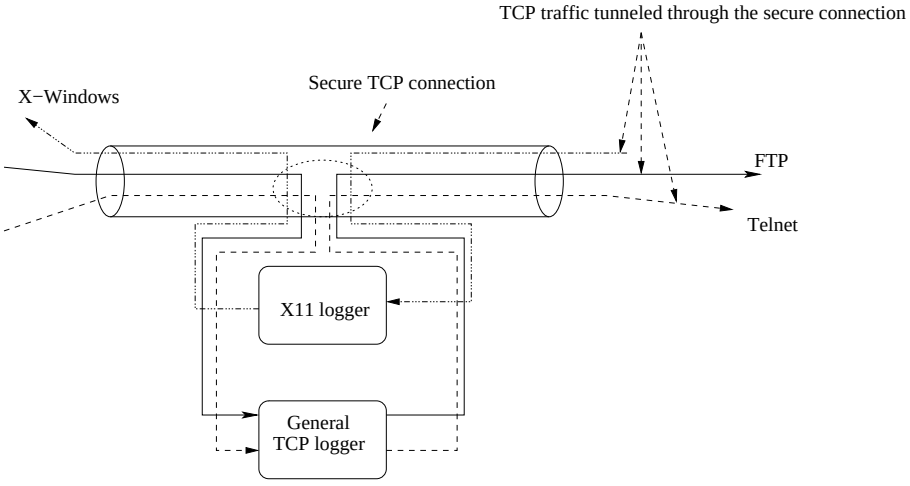

Figure 3.1: TCP traffic re-routed through the logging system.

RSG is based on the FreeBSD operating system and OpenSSH. OpenSSH supports TCP port forwarding in general and X-Windows forwarding in particular. Since tunneled data is not encrypted inside OpenSSH, a possible solution would be to add a module to OpenSSH which could record and

monitor TCP streams. An illustration of the proposed solution is shown in figure 3.1.

Since OpenSSH separates the general TCP tunneling from the X-Windows system the proposed solution will have a similar design. The general TCP logger, explained in this chapter, records the data stream with no expectations on the specific contents of the stream. The X-Windows logger, however, expects that the data which it receives contains genuine X-Windows packets and is therefore able to produce more specific logs. Since the X-Windows logger is a very large portion of this thesis, and since it is almost entirely separated from the general TCP logging, it is documented separately in chapter 4. Furthermore, since X-Windows is the only automatically recognized protocol it is the only one that can be monitored live.

The use of both a general TCP logger and an X logger has several motives:

- There must be a method of recording other protocols, since X is not the only protocol used via RSG.
- If there was no specialized X logger, the playback functions would be much more difficult to implement, since the filters implemented in the X logger modifies the packets live to force the client and server to be compatible with the player routine.
- Filtering can not be done in the general TCP logger, since it does can not recognize protocols.

An optimum solution would be if the general TCP logger could recognize the current protocol, but this has not been achieved, the reasons for this are explained in section 3.1.

Besides documenting the general TCP logger, this chapter also explains the design of another application, the Telnet processor. This program produces useful information from a log file that contains a Telnet session.

3.1 The basic design

Besides Telnet, FTP and X-Windows (also called X11) the RSG also transfers other applications such as *Citrix* and *VNC* [Rea02]. Since the ideal solution, where each protocol could be detected automatically, requires too much knowledge, an alternative solution is desirable. To record everything that is sent via the tunnels and store it on the hard drive for later analysis is an acceptable beginning. Specialized programs can then be implemented to enable the detection and analysis of different protocols in the logs. For example if a stored session is viewed using a telnet analyzer, the log would be displayed like the visual output from a telnet client. If the session was not a telnet session the module would eventually fail or produce unreadable output.

3.1.1 Intercepting a TCP stream

SSH can be seen as a method of creating a bridge between two previously separate networks. The network on which the initiating computer is located is called the local network, while the other network is called the remote network. Tunnels may then be created from either the local to the remote network, this is called “local port forwarding” and creating tunnels from the other side is called “remote port forwarding”. The term “port forwarding” comes from the fact that SSH will listen on a TCP port on one side of the tunnel. If an application connects to this port, the other end of the tunnel will create a connection to a port on another machine. The data is then transferred between

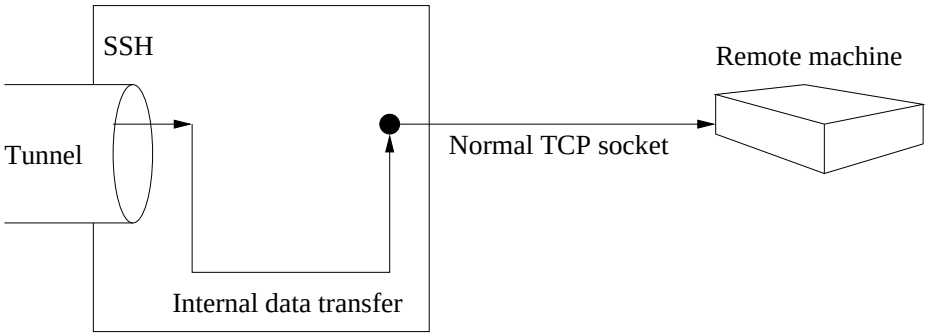

Figure 3.2: A local port forward at the remote side.

the two machines connected to the tunnel. A detailed view of a local port forward is illustrated in figure 3.2.

To intercept a forwarded local port on the remote network is quite simple. SSH will create a socket on the remote side where one end is connected to a machine and the other is connected to the tunnel. If the socket is connected to the recording program instead, and another socket is created to connect with the tunnel, the transferred data can be recorded. Compare the original procedure, illustrated in figure 3.2, with the situation during interception, illustrated in figure 3.3.

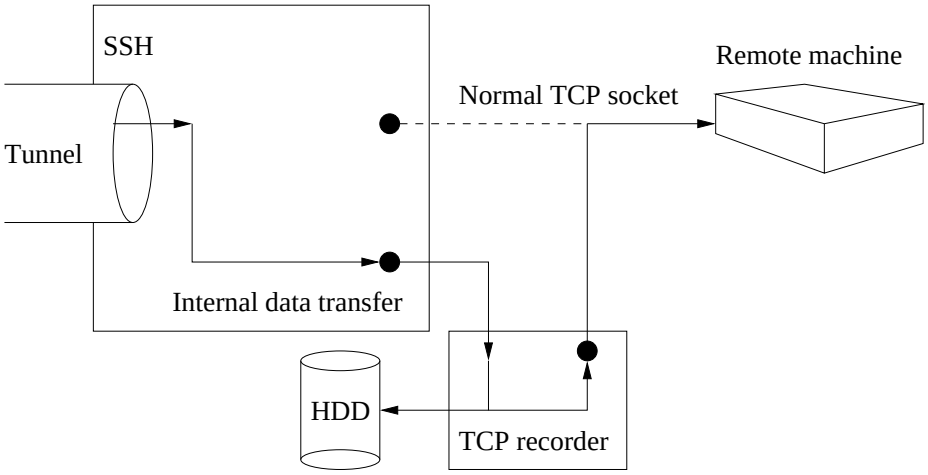

Figure 3.3: An intercepted stream.

3.1.2 The TCP recorder

When a TCP stream has been successfully intercepted inside SSH, the data stream is transferred via the TCP recorder. The data must somehow be formatted to maximize the logs usefulness. The two streams will be divided into smaller blocks. Since there are two incoming streams each block is marked with a header indicating from which direction it came. The blocks will then be written to the log file, in arrival order. The specifics of the file format are described in appendix B.1.

Technical description

OpenSSH handles two types of connections, incoming and outgoing sockets. Both types of connections results in a *file descriptor*[1].

Immediately after OpenSSH has received a file descriptor that represents a socket, the interception code is called. This code will first create an internal socket, used for interprocess communication, and then *fork*[2] the process. Both ends of the internal socket are given a *file descriptor*. The first of these two file descriptors and the original socket are given to the new child process. The second file descriptor will replace the original socket descriptor in the parent process. The child process will then execute a separate program; the program that contains the actual code for recording the data stream.

3.1.3 The telnet processor

Even from the early beginnings of computer networking, an important service has been to provide remote control of other computers. When the networks and protocols began to be standardized, the differences between computers had to be addressed. Different vendors had different ideas on how to process new line characters, how to use signals and so forth. The Telnet protocol [PR83b] was created to bridge these differences.

The telnet protocol defines the concept of the Network Virtual Terminal, NVT. The NVT is probably best described as "an imaginary device which provides a standard, network-wide, intermediate representation of a canonical terminal" [PR83b]. The minimal set of requirements specified by the NVT must be emulated at each side of the connection, at least conceptually (since most of the time there is only one real terminal). From these minimal requirements the two sides may negotiate more advanced options.

The Telnet log analyzer, or processor, must understand the basic set of operations of an NVT. It must also be able to understand the negotiation of options. Additionally the processor must keep track of the state of each NVT.

Telnet is a byte oriented protocol, each byte represents a "character". A character is either a normal, printable character, or a control or signal character. Some control characters have special meanings in Telnet, like the IAC character. IAC stands for "Interpret as command" and marks the beginning of a control sequence.

Since the telnet processor is used only on pre-recorded sessions it has no control over the negotiated options. This means that if the telnet client and server agree on using a certain option which the telnet processor does not understand, the processors can not guarantee that the output is correct. Therefore the processor can only produce "best effort" output. Even if the processor did understand all the options, it could still be prevented from producing usable output. This could happen if the client and server have agreed to use encryption, for example.

The Telnet processor will emulate one of the NVTs of the original session. This means that one of the streams in the log will be used as the incoming stream. This stream is the one that originally came from the telnet server and it is the one that is presented on the console to the user. The other stream is only used to deduce how the original client negotiated its options.

[1] A file descriptor in UNIX is used to identify everything from a normal file on disk to network connections and other I/O devices.

[2] A fork in a process means that a copy of the process is created.

Technical description

A simplified (and incomplete) version of the execution flow is illustrated in figure 3.4. The Telnet processor can be in either one of two top level states, the normal state or the command state.

When an IAC (Interpret As Command, [PR83b]) character is read from the log the processor will leave the "normal" state and proceed to the "command" state, where it will process a Telnet command.

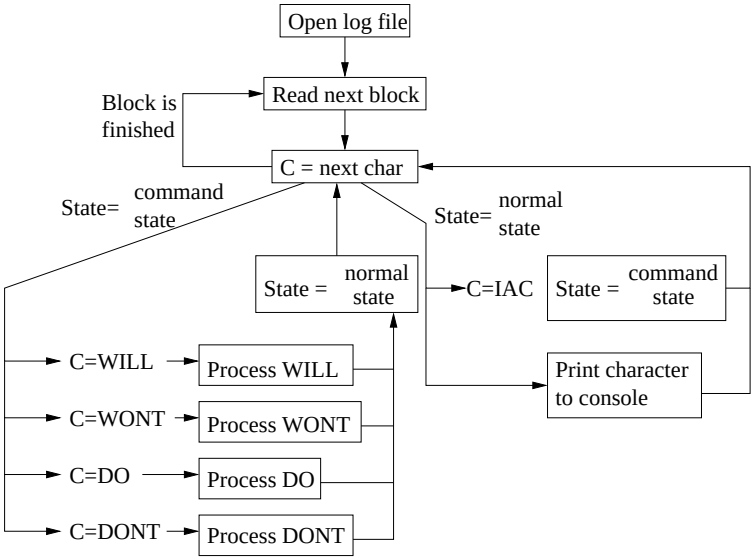

Figure 3.4: A simplified view of the execution flow.

A simple telnet command is specified as three characters. For example, "IAC DO ECHO" (interpreted as the three ASCII characters numbered 255, 253 and 1) means that the machine who sent the commands requests the other side to perform an option called Echo [PR83a]. The "DO", "DONT", "WILL" and "WONT" commands are used to negotiate *options*. Other commands (which are left out of the illustration in figure 3.4) include "IP" (interrupt process), "EC" (erase character) and others.

An option is negotiated by one of the parties, either by requesting the other party to perform an option ("DO"), or by informing the other party of the desire to perform an option ("WILL"). The other party will respond with "WILL" or "WONT", if it received a "DO", or it will respond with "DO" or "DONT", if it received a "WILL". All options defaults to "WONT", i.e. no party will perform that option if it is not negotiated first.

For example, assume that the two NVTs "Alice" and "Bob" have initiated a Telnet session. "Alice" wishes that "Bob" should echo all characters that "Alice" sends. "Alice" achieves this by sending the following three bytes: "IAC DO ECHO". If "Bob" is inclined to grant "Alice's" request, he will respond with: "IAC WILL ECHO".

The Telnet processor implemented for this thesis understands the "LINEMODE" option [Bor90]. It also recognizes when two sides have negotiated to use the Telnet *auth* [ATT00] option with or without encryption. Different options often mean that some characters will have special meanings, or that characters need special processing before they can be printed to the screen. This is the case with encryption. However, the Telnet processor does not know the encryption keys used so it cannot process the characters correctly. When a special character is detected in the stream the Telnet processor will run a dedicated subroutine that does the proper processing.

Chapter 4

Logging and monitoring X

As it was explained in chapter 3 the ideal solution where all protocols could automatically be detected is very difficult to implement. Therefore the log files produced are not always optimal, but the specialized X Windows channel in OpenSSH provides the foundation of a specialized X recorder. This advantage should be exploited to maximize the usability, and the easiest way to do that is to have two separate loggers.

4.1 Theory and facts

X Windows is a "server/client" system for displaying graphical user interfaces. X applications and servers communicate using the X protocol [N+95]. The version currently in use is X11, version 4 release 6. This section assumes prior knowledge about the details of the X system and the X protocol. Those who are not familiar with these concepts are urged to read appendix A.

The developed application that logs and monitors the X Windows protocol is referred to as the X logger, or XIIL, which is an abbreviation of "X11 logger" (note the use of "II" instead of "11").

The basic theory for logging and monitoring X is pretty simple, just record the requests sent by the client to a file together with some time stamps. The requests may be sent to a monitoring server simultaneously or they may be played back at a later time. However, it is more complicated than that in practice.

X exists for a multitude of hardware. This is one of the strengths of the system, but it also makes it complicated. One very important issue is that the order of the bytes used to construct larger words[1] may be different on different hardware. The client decides in which order the bytes should be. Thus XIIL must be able to convert between the clients byte order and the native byte order used by the machine on which XIIL runs. Additionally, when an X client connects to a server it receives two 32 bit integers called the resource ID base and the resource ID mask. From these two numbers, the X client generates resource IDs for everything that it needs to control on the display. The base and mask are almost never the same as the last time the client connected, so when XIIL transmits logs to a new X server each ID must be regenerated. Furthermore, every time the old value is detected in the stream, it has to be mapped to the new value.

[1] A "word" is a set of bits constituting the smallest unit of addressable memory. A word may be several bytes, examples are 2 or 4 bytes for one word.

Another identification method is the Atom (see appendix A for an explanation). The numerical mapping of an atom may differ from server to server which may cause problems. Therefore the playback system must discover when the playback server's mapping is not equal to the original server's mapping and replace any occurrences of the old values with the new ones.

Besides the resource IDs and Atoms the X protocol also supports extensions. Each extension is mapped to a major *opcode* (operation code) between 128 and 255, and each server may support different extensions, and may map each extension to any value in the given range. XIIL must be able to map these numbers if the original X server had different opcodes for an extension than the current playback server. Additionally, XIIL must deny the use of any extension that it does not understand since unknown requests may contain resource IDs or atoms that need to be mapped.

4.2 Design choices

The X logger is designed to work in both stand-alone mode and as a plug-in module to OpenSSH. This enables testing of functionality without involving OpenSSH and it also enables the software to be used in other fashions like recording demonstrations of how to use other software.

To enable control and monitoring of the current logging sessions to the administrator, some kind of interface is needed. A simple text interface is enough since most of the RSG system tools are console based anyway. Since there can be many connections simultaneously, some kind of administrative system has to run at all times. This resulted in a daemon where all logging sessions and the administrator can connect. The daemon is called "the control center", while the administrator application is a simple telnet-like application called "the terminal". The communication method between daemon, administrator terminal and logging sessions is the UNIX domain socket. A modular description of the logger in conjunction with OpenSSH can be seen in figure 4.1.

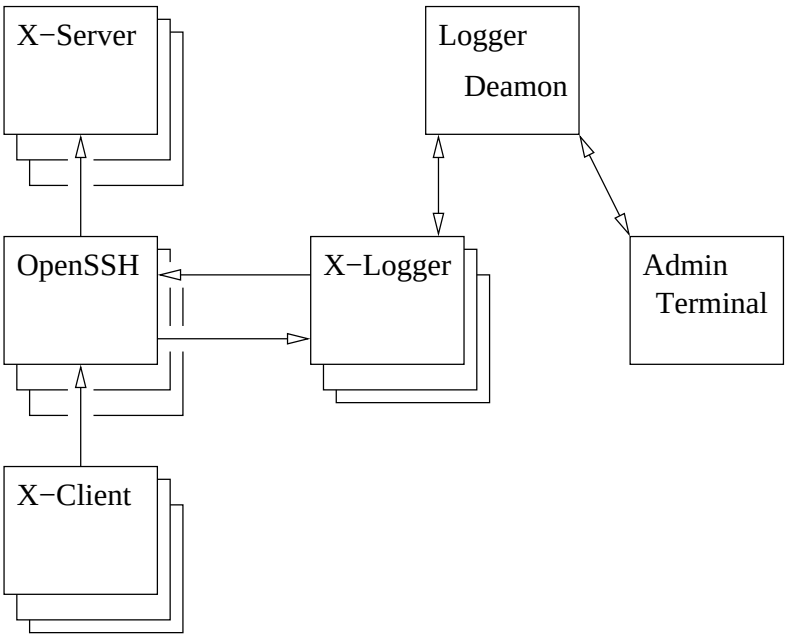

Figure 4.1: The logger and OpenSSH together.

To produce correct playback of the logs, timing is important. Therefore XIIL must generate and

save time stamps along with the data. However, if each packet (a request, reply, event or error) would have a timestamp attached to it, the timestamps could in theory occupy more than 50% of the log file. That is not preferable, and therefore XIIL allows for larger blocks of data to be stamped with a single time stamp.

Since playback and monitoring essentially work in the same way, they may use the same routines. There are only a small number of parameters that differ. Therefore a good design would be a general interface at the lower level and above that, two high level interfaces, one for playback and one for monitoring.

The streams of data that XIIL must process have to be stored in memory temporarily, i.e. buffered. There are numerous methods for buffering, each suitable for different tasks. Since each X packet must be processed in a continuous block of memory, the buffer system must not break packets, unless it can restore them. Furthermore, the following facts must be taken into consideration when deciding which algorithm to use:

- The size of X packets is highly application dependent.
- The packet size may be very irregular.
- Duplicating memory is very time demanding and should be avoided.
- X specifies a maximum request size.

The algorithm selected is a simple one. Allocate a buffer of the size specified by the X server. Add new packets to the end of the buffer, and process the packets in order. When a buffer, and all complete packets are processed, any incomplete packet at the end is moved to the beginning of the buffer and the process continues. This algorithm is pretty straightforward and it is also basically the same as the one that the *X Consortium* (see the the glossary) use in their implementation of X. For a detailed description of XIIL's buffer structure see appendix B.2.1.

There are a number of extensions to X, but due to lack of time none of them has been implemented in XIIL. However, most applications have no problems working in this "degraded" mode. Therefore the logger will simply deny the use of any extensions. This is accomplished by altering the reply sent from the server for any request for an extension. The reply is altered so that the client believes that the server does not support that particular extension.

4.3 Detailed technical description

Both the playback routines and recording (logging) routines are integrated in the same executable file. This is because both the player and monitoring routines use the same functions. It could be argued that the player and recorder should exist in different binaries and that the common routines should be placed in a library. However that solution was discarded on the grounds that more files increase complexity.

4.3.1 The logging module

During logging, XIIL controls 3 I/O streams simultaneously. The first is the client input, the second is the server input and the third is the control center channel. XIIL will also control a fourth I/O stream, the monitor input, if operating in monitor mode. When a data block (packet) arrives at an input the appropriate handler is called to process the data.

The client handler

The client handler is implemented as a state machine, and is described in figure 4.2. When the state machine begins in the initial state it expects four bytes of data, the minimum size of a request. This is indicated by setting a counter called "need" to four. The "need" and other fields that belong to the buffer structure are described in appendix B.2.1.

When the first four bytes of a request have arrived, the client handler will proceed to the incomplete state. In this state the "need" counter is recalculated to a new value, depending on the type of request. The client handler will wait in this state until the "got" counter (appendix B.2.1) matches or exceeds the "need" counter. It will then proceed to the complete state, indicating that a request is ready for processing.

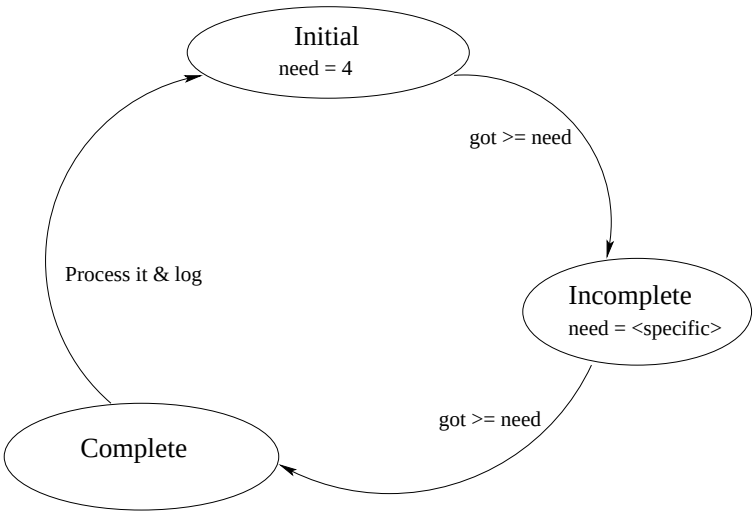

Figure 4.2: The client state machine.

There are basically two types of requests, the ones that expect replies and the ones that do not. Most of the requests do not expect replies. Each time a request that needs a reply is detected, the request's sequence number and the type is stored in a request queue. This data is later used by the server handler to identify replies.

When the client handler is finished processing the request it is saved to a log file and then forwarded to the server. The client handler will then proceed to the initial state again.

The server handler

The server handler is, like the client handler, a state machine. The different states are shown in figure 4.3. The initial state expects to receive 32 bytes before it can proceed, as this is the minimum size of a packet sent by the server. When it has received the expected amount of data it will check the first byte of the packet for the type, as declared in appendix A. If the packet contains an error XIIL will record it, send it, and then proceed to the initial state again. If the client or server is unable to proceed due to the error the connection will eventually fail. If the packet contains a reply, the machine proceeds to the incomplete state where it will wait for additional data, if needed. From there it will continue to the complete state where the reply is processed. If the packet contains an event, no processing is done. All packets are written to the log and sent to the client before proceeding to the Initial state again.

A reply is processed by a special routine that alters the content of specific replies, like a reply to a Query Extension request. The Query Extension call requests information about a special extension. The reply is altered by XIIL so that the client believes that the server does not support that extension, as explained earlier in section 4.2.

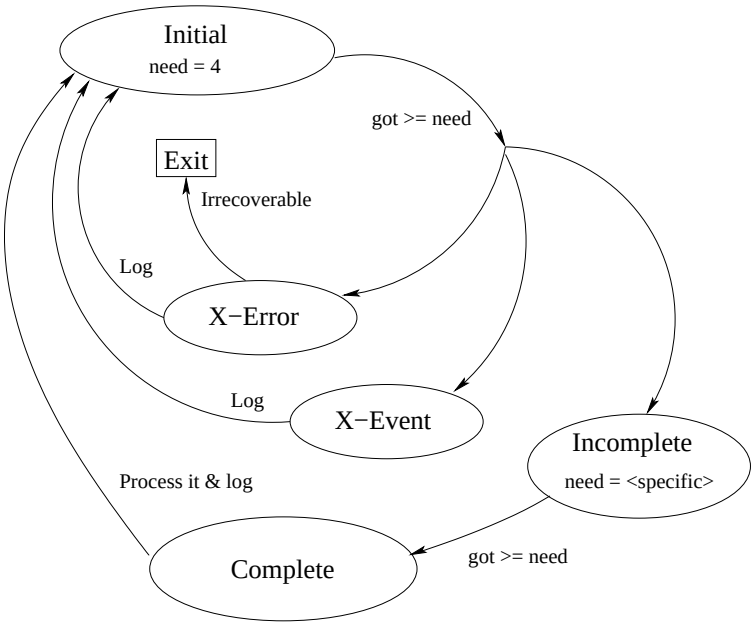

Figure 4.3: The server state machine.

The control center handler

The control center handler understands three types of commands, kill connection, send information and start the monitor. The protocol for sending these messages is explained in B.4. Basically, the following might happen when the handler receives a message:

- A kill message will abort everything and shut down all connections.
- An information request will transmit information back to the control center daemon.
- A monitor request will initiate and start the monitor module.

If the monitor module is running, the monitor handler is waiting for input from the monitoring X server, or a time out event. The time out event occurs when the timer for the next request has expired. When either one of these events have occurred the handler will call the appropriate playback functions. See section 4.3.3 for more information.

4.3.2 The X control center

The X control center is responsible for routing commands between the administrator and all the X loggers (clients). The program is run as a *daemon* and listens to two UNIX domain sockets for connections. The first socket is used by the clients and the second is used by the administrator. The control center is implemented as an event driven system with the following events:

- A new XIIL client has connected.
- An administrator has connected.
- A message from a XIIL client has been received.
- A command from the currently connected administrator has been received.
- A XIIL client has disconnected.
- The administrator has disconnected.

The commands sent by the administrator are analyzed by a parser (the different commands available are described in section B.4). If the command is intended for a specific client, the parser will generate a packet which is sent to the indicated client. When the client replies, the message will be relayed to the administrator.

The control center only allows one administrator terminal to be connected at once. If the administrator tries to connect to the control center using another terminal, he or she will be asked whether or not to override the previous connection.

4.3.3 The monitor and playback package

Since the monitor and playback are equivalent at the low level there is no point having separate descriptions.

The playback module begins with a setup stage. In this stage the player will open the log files and connect to the requested playback server. After a successful connection attempt the player will load the first packet, "client prefix", from the client log.

The playback server may expect the client to send some kind of authentication data (see appendix A). This data, if it exists, is extracted from the system using the *"xauth"* program. Any authentication data stored in the "client prefix" is replaced with the new data. Furthermore, the byte order defined in the "client prefix" is used to set up the byte order conversion functions. The modified packet is then transmitted to the playback server.

If the connection is accepted by the playback server the playback module must check the response ("setup prefix") sent by the server to make sure that it is compatible with the prefix found in the log. The playback will fail under the following circumstances:

- The playback server is using a smaller maximum request size than expected.
- The image byte order used by the server does not match.
- The image bit order used by the server does not match.
- The bitmap scan line unit does not match.
- The bitmap scan line pad does not match.

The image and bitmap properties in the log could be converted to match the ones of the playback server, but this feature was not implemented due to lack of time.

When the player has successfully connected to the server the time stamp of the first request is loaded into a timer. After that, the player starts to execute the main loop.

The main loop begins by waiting for either the server to send a packet or for the timer to expire. If the server sends a packet the server handler is called. After the server handler has been called the player will check if enough time has passed to send the next request from the client log.

Server handler

The server handler will begin by loading the received data into a buffer. The buffer is then checked to see if a complete packet has been received. If this is the case the handler will use the reply processing function on the packet. The processing function will first determine the type of the packet. If the packet is an event it will be ignored, and if the packet is an error, a message will be printed to the standard output (but the playback will continue.) If the packet is a reply the next available reply from the log file will be loaded into memory, and then the type of the reply will be extracted from a reply queue (see "request processing" below). The reply contains the sequence number of the request that generated it, this number will be matched against the number stored in the log file. If these numbers do not match the player has lost its synch and the playback will fail. If in synch, the received reply and the reply stored in the log will be processed by the reply mapping routine. This routine will take care of such matters like adding numerical representations of *Atoms* (A) to the hash table. The reply processing function will repeat until the buffer contains no more complete packets.

Request processing

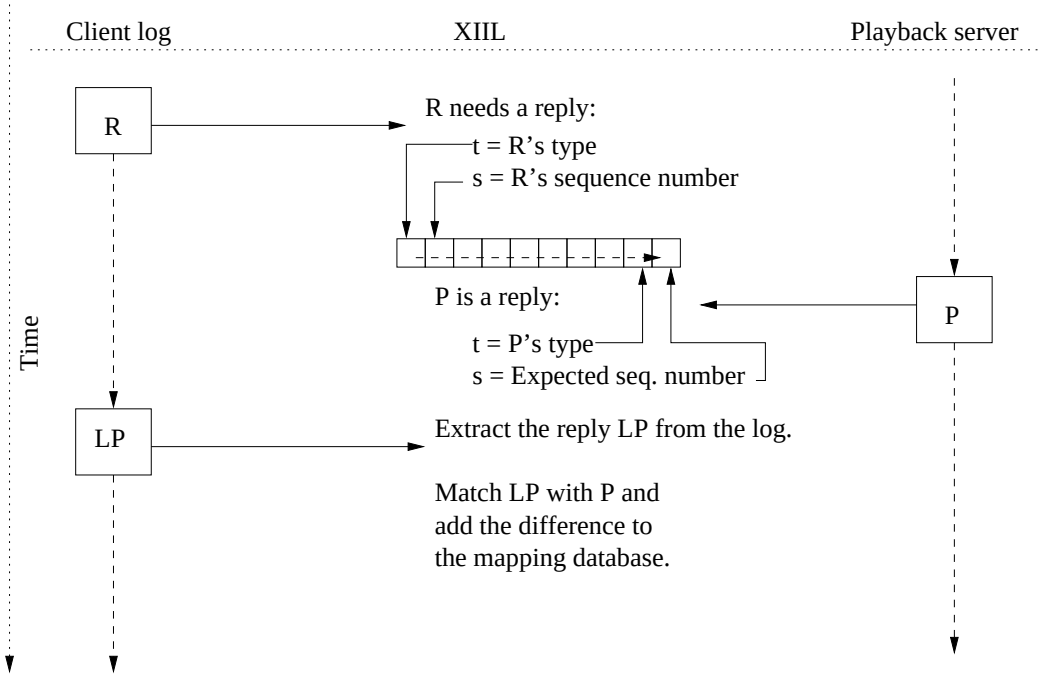

Figure 4.4: The reply queue.

If the timer for the next request has expired the player will call the request processing function. This function will load all requests in the current time block (several packets stamped with the same time stamp) into the buffer and process them in order. If a request generates a reply the request type will be added to the request queue. This is needed since a reply does not contain information

about which type of request that generated it. The use of the queue is illustrated in figure 4.4. The request is then passed to the request mapper, which will add values for "new" request IDs to the hash table, and replace old values in the request with the new ones. After the request has been modified it is transmitted to the playback server. The timer is then set to the next time block and the main loop repeats.

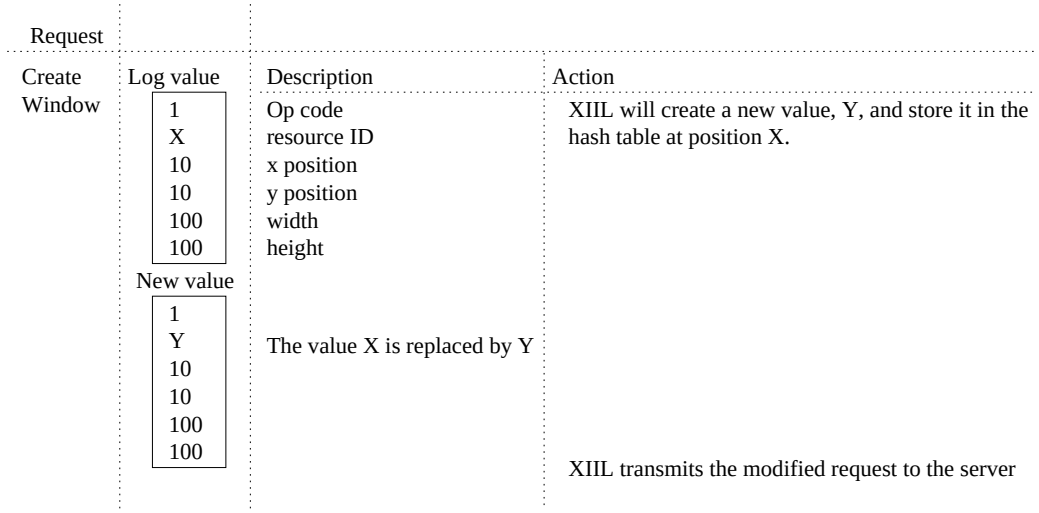

Figure 4.5: Resource mapping example, part A.

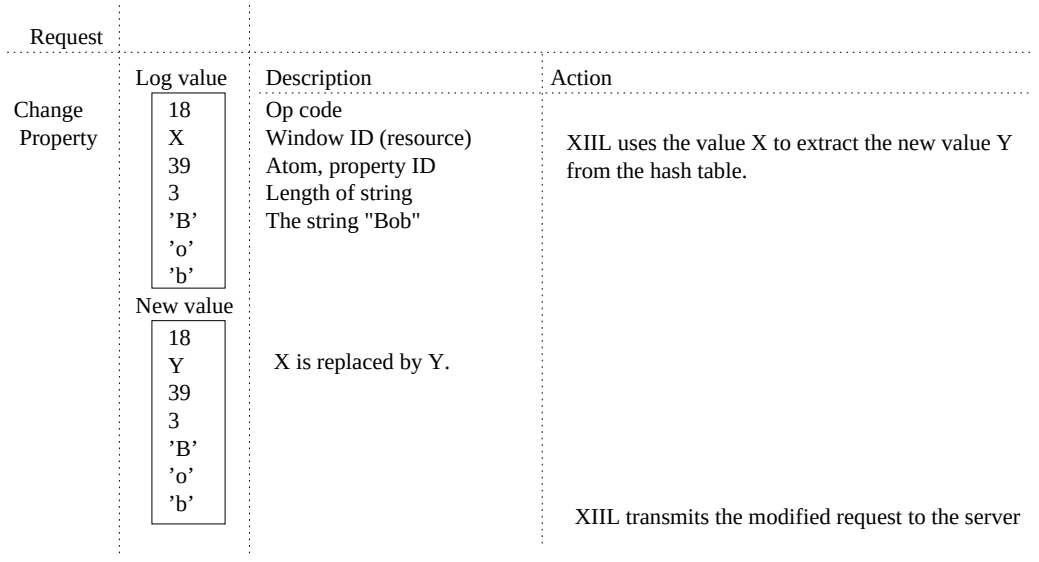

Figure 4.6: Resource mapping example, part B.

Consider a scenario where XIIL has recorded the X session presented in section A.9 and that an administrator would like to replay that. When XIIL has connected to the playback server it will receive two new values of the resource *base* and *mask* (see appendix A). These integers are used to generate new values for the resource IDs stored in the log. Figures 4.5 and 4.6 illustrates how new values are inserted in the hash table, and how the old values stored in the log file are replaced.

Chapter 5

Evaluation - Performance

Even though a system may seem to be good, at least conceptually, it can not be trusted unless some kind of testing and evaluation is conducted. This chapter will, by performance testing, show how the RSG system is affected by the additional software developed for it.

5.1 Time

The performance of the general TCP logger has been tested by transferring a large file, 10 megabytes, through the tunnel. The test creates one tunnel. The tunnel transfers 10 streams simultaneously. A total of 12 test runs were made, 6 without interception and 6 with. The results are shown in table 5.1.

With interception	No interception	Difference
1m4s	1m3s	+1s
1m3s	1m4s	-1s
1m4s	1m5s	-1s
1m3s	1m5s	-2s
1m3s	1m3s	0s
1m4s	1m3s	+1s

Table 5.1: TCP interception performance

The test shows that there is no detectable impact on the performance when using stream interception compared to when it is not used.

The X-Windows recorder was tested by using it with a graphical web browser. The performance was not measured numerically, instead the response time when using the recorder was compared to the case when it was not. While the prototype version was extremely slow the improved and delivered version almost performed as good as when not using interception. The difference is largest when comparing “startup time”. The intercepted version can be almost 1-2 seconds slower. This can be explained by the fact that using XIIL implies more processing overhead. Each X packet must be processed by two “X servers” before any visual effect can be detected.

Since logging requires disk access, it is easily understood that if large amounts of data are transferred each time unit, there will be a heavy load on the I/O system. This load can significantly decrease the performance compared to when not logging all streams.

5.2 Space

The amounts of data transferred through an RSG node may be substantial. X11, for example, will transfer all bitmap graphics raw, i.e. not compressed, which can result in large amounts of data being stored on the log-disk. When running the performance test with the web browser, 10 minutes of surfing on a graphically intense web page generated a log of 15 megabytes. Compressing the log with *Gzip* produced a file of 4 megabytes, or about 25 percent of the original size.

The bitmap data stored in the log files can theoretically be compressed with more advanced algorithms like JPEG or PNG. This routine, combined with Gzip, could yield very good results.

5.3 Additional evaluation

Other parts of the system, like the administrator terminal, should also be evaluated, however this has not been done. Further more, additional testing of the X logger and Telnet processor could also be done.

The administrator terminal could (and probably will) be tested by real users. These tests will most likely result in requests for improvements such as user friendliness and additional features. The X logger should be tested to see if there is a need for any extension to be supported. The Telnet processor can also be tested to see how it handles obscure telnet options which are not yet supported.

Chapter 6

Discussion and conclusion

This thesis has progressed as several parallel processes, with two clear main pieces, the TCP logger and the X11 logger. Although the general principles and much of the functionality is pretty much the same in both applications, they where mostly developed separately. The reason for this separation is that there where problems that could not be solved with a single solution. The general TCP logger can not produce good enough logs of X11 traffic, while the X11 logger must assume that the traffic is the X protocol. This division is also clearly visible all through out this report; everything is described from two viewpoints. Therefore the discussion of the result is also divided, and it begins with the general view of the TCP logger. However, the conclusion at the end of the chapter will merge all views into a global perspective of the project.

6.1 Discussion of the development process

6.1.1 The general TCP logger

Currently RSG is using an old version of OpenSSH, but the interception patches are developed for a newer version. However, since it was necessary to test the patches on a "live" RSG system, the patches were ported to the old version as well. The patches were not working properly right from the beginning, however, so they had to be updated. Working with two sets of patches, where almost all patches were identical, was very frustrating. The intercepting code (the patch) was moved into its own function and generalized so that both versions of OpenSSH could use the same code.

The recording program was designed and implemented in one day. Testing the code with telnet indicated that the first implementation worked. However, the code would fail on heavy data loads. This problem was caused by simple oversight. Since the sockets are put into "non blocking" mode it means that calls to read and write will return immediately, even if there is not enough data to read or if the socket is busy with writing a previous block. The code failed to check for the special case where the socket could not handle as much data as the code tried to send. This case requires that the recording code pauses for a short while, waiting for the socket to finish its current operations.

The Telnet processor was in retrospect more complicated to develop than originally anticipated. However, it did not turn out as complicated as anticipated, when the amount of documents concerning Telnet was discovered. The development work was rather straight forward coding. First a system for handling option negotiation was developed. There are many options that are recognized, but some of them are not implemented, or only partially implemented. After the option handler

was written the console output code was developed. This code is responsible for processing special characters and writing characters to the console. When testing the Telnet processor with interactive applications, like text editors, it was noted that the processor was working too fast. Therefore the processor was extended with code that emulated a key-striking pace of 10 keys per second.

6.1.2 X Windows logger

The X logger and player has been developed in two stages, the experimental prototype stage and the final development stage. Since most of the prototype was "written while learning" almost the entire prototype was discarded when the final development stage began.

The prototype

The prototype started as a simple tunnel through which the X packets were routed. The next step was to log the traffic. This was pretty crude in the early stages. All packets sent by the server were discarded while all packets from the client were simply written to file, along with a time stamp. The playback routine was equally dumb. It just read the time stamps and packets and sent them to the playback server at appropriate times, totally ignoring anything sent by the server. However, if the logs were played back to the same server immediately after recording, no difference could be seen. Although an extremely simple implementation, the results were very motivating.

Since playback did not work just by replaying the X-requests, something more had to be done. The "X Protocol Reference Manual" [N+95] explained the resource ID system and how it worked. However, it did not, in an accessible form, state which requests that generated new IDs and which requests that needed mapping. Attempts to locate the information on the web failed, and the X Consortium was hard to get in touch with. The solution was brute force studying of every request specified in the protocol. From this study, a table was generated in which a request was categorized by the following:

- It generates IDs.
- It must be mapped.
- It expects a reply.

Since the requests most commonly used need mapping, the mapping function must be quick. The natural choice here was a hash table. The hash table implemented is a very large static sized hash table. The functions that do the actual mapping and hash table insertions contain very long switch blocks.

A very important issue concerning byte order differences was discovered very early in the development and is clearly explained in [N+95]. The solution was to implement functions that convert between the real client's byte order and the byte order used by the machine running XIIL. The functions must be applied to every word used by XIIL. Failure to map everything is likely to produce very strange errors, and any such error may be very difficult to detect and locate.

The mapping routines seemed to work very good during tests. Playback was possible even when the resource ID base and mask were completely different from the original. However, there were imperfections when trying to replay logs to other X servers. For example, windows were lacking titles and windows had wrong positions etc. Further studying of [N+95] revealed that to control such things as window titles the concept of the Atom is used. There are a number of predefined

atoms in the standard, but some atoms are not always the same on all servers. These values must therefore also be mapped. This was done in a similar fashion with a hash table, and again the reference manual had to be read thoroughly for information about requests that use atoms.

The application now worked, but it was hard to use. First, it could only communicate via TCP sockets. The client handler and server handler could use already opened file descriptors to sidestep this issue, but they had to be opened by another application instead. Secondly, the X Windows logger could only understand IP numbers, but people want to use names. Thirdly, it did not support any authentication methods used by the X protocol. Furthermore, the file format used for the log files was inefficient. During replay the file had to be scanned back and forth for matching replies from the server with the ones in the log and all packets had their individual time stamp. Also, the log file had to be pre-processed before any replay could be done. The prototype was already a rather large patchwork, so it was scrapped.

The rewrite

The usable code from the prototype that could be salvaged was the mapping code, the queue functions and the hash tables. The code for handling command line options and early initialization was also still usable, albeit that code needed some modifications. Since the original buffer system had become extremely complicated the new code was designed to produce as little code as possible. However, the second iteration of the buffer system was also slow, this was due to the fact that it copied a lot of small memory chunks. Thus even that code had to be replaced. The final buffer code was a bit more complicated since it has to handle more special cases at the buffer boundaries, but it was at least fast. However, the new code prevented the monitor functionality to be implemented in real time. Thankfully, the new monitor implementation was so similar to the playback functionality that they both could be replaced with a single package. This reduction of complexity compensated more than enough for the increased buffer complexity.

6.2 Limitations

There are several limitations in both the general TCP recorder and in XIIL. As explained earlier, both the space needed to store the log files, and the time needed to access disks, can limit the performance of the system. Additionally, the general TCP logs are of no use if there is no method of analyzing the data in them. The TCP streams contain no information about time, which also prevents logs from certain protocols from being fully usable, even if a log processor was developed. This is a relatively easy operation to fix, but the time assigned to this project was limited thus, it was not prioritized.

X is a complex state machine, which means that if a log is sent to a server, all requests have to be sent, and they have to be sent in the right order. Failure to do this may put the X server in the wrong state. For example, if a request to create a window is skipped, all requests that depend on that window must also be skipped. Furthermore, the current design of XIIL relies on the fact that X uses a counter to identify replies. If the counter on the server does not match the counter in the log, XIIL will fail to identify the replies properly. However, it is probably possible to implement additional features in the program to handle this.

Another issue with X is that any application authorized to connect to the server may use any window, as long as its identity is known. This means that two programs may send requests to the same window or to any other resource, as long as both know its resource ID. The implication of this is that a resource ID may be sent through XIIL without having to be “created” first. If only one of

the X11 streams is recorded it means that the correct state of the X server can not be deduced from the log file. However, even if both X11 streams were recorded, XIIL can not use them since it does not understand that they have a relation.

6.2.1 Future work

The application described in this thesis can be extended with additional features. Some of these features require much work, while others are rather simple.

A VNC processor

There is an application called RFB Proxy [Wau02] developed by *Tim Waugh*. RFB Proxy is a tool that records VNC sessions and enables a visual replay of them, just like XIIL does for X11. If the log files generated by the TCP logger contained time stamps the log files could be "sent" through RFB Proxy. RFB Proxy would recognize the streams as VNC sessions and produce re-playable log files. The implementation of this feature requires that a log file can properly be identified as a VNC stream, and that the log file is extended with time stamps.

Possible extensions to XIIL

XIIL may be extended and enhanced in numerous ways. A graphical tool for controlling playback is one simple example. The *GUI* could enable such things as pause, fast forward and seek. These things could also be possible to implement with an interactive text interface. Another feature which have been proposed is the ability to fast forward a selected number of time units and then pause.

XIIL could be enhanced very easily to support playback of log files to an X client instead of an X server. This would create the possibility to record macros for X applications. The client playback feature can also be used to create test cases for graphical programs, for bug eliminating purposes for example.

One important part of bug elimination is the ability to recreate the circumstances under which the bug appears. With graphical interfaces it can be hard to remember, or document, the steps required to generate the fault. Client playback could be helpful in those cases. The log file could also be replayed to both the client and a real X server simultaneously. This would provide a visual representation of the process.

The problems with multiple X connections using the same resource IDs can be solved by using a common database. The common database must only be used with logs from clients that connected to the same server, and it is only necessary to use it the logs contain common resource IDs. These requirements can be verified by analyzing the log files that where running at the same time, i.e. their time frames overlap.

6.3 Conclusion

The goals and requirements described in section 1.4.3 have been achieved, and it has been proven that the interception of data streams in SSH is possible. However, as it was explained in chapter 2 the requirements have been modified during the development process to reflect that which was

actually achievable. The delivered product is a good "stepping stone" towards further development, of which some were described in section 6.2.1.

The problems that were encountered during the development illustrates that everything is not always what it seems. Complex systems have extremely many "special cases", and some times they are not discovered until they cause a (serious) problem. This shows that an important part of developing such systems is to conduct testing.

Some personal reflections by the author

There are many things which I have learned during my work on this thesis. Most of it concerns different network protocols that I have studied, and the internals of OpenSSH for example. I have also learned how important "customer feedback" is to the development of software applications. The ideas that the customers have for a piece of software are likely to evolve during the development, as they realize the capabilities of the system. The things they see as very usable features are often things they did not even think about when the original specifications were written down. This is also true for the developers, in this case me. As the work progressed I discovered new possibilities and new problems all the time. However, many of these things had to be thrown in the "to do" bin, since the time for this project was limited, but the list will probably provide "years" of fun for the people that take over now that I am finished.

In conclusion I can say that the experience I have had here at Ericsson has been most rewarding. I have been entrusted to develop my own software for a large company, and they have supported and appreciated my efforts. Had this been a couple of years ago I would probably continue my work here even after graduation, but due to the economic slowdown in the world today, I can not.

Bibliography

[ATT00] J. Altman and Editor T. Ts'o. *Telnet authentication option*, 2000. RFC: 2941.

[Baz99] John Bazik. Xmx - an x protocol multiplexor, 1999. http://www.cs.brown.edu/software/xmx/ , last verified: Nov 2003.

[Bor90] D. Borman. *Telnet linemode option*, 1990. RFC: 1184.

[C+02] Gerald Combs et al. Ethereal, 2002. http://www.ethereal.com/ , last verified: Nov 2003.

[N+95] Adrian Nye et al. *X Protocol Reference Manual*. O'Reilly & Associates, Inc., 1995. ISBN: 1-56592-083-X.

[PR83a] J. Postel and J. Reynolds. *Telnet echo option*, 1983. RFC: 857.

[PR83b] J. Postel and J. Reynolds. *Telnet protocol specification*, 1983. RFC: 854.

[Rea02] LTD RealVNC. Vnc - virtual network computing, 2002. http://www.realvnc.com/ , last verified: Nov 2003.

[Wau02] Tim Waugh. Rfb proxy, 2002. http://cyberelk.net/tim/rfbproxy/ , last verified: Nov 2003.

Appendix A

The X Windows system

The X Windows system, or just plain X, is the primary system for window based graphical applications on UNIX and UNIX-like operating systems. It is an open standard maintained by the X Consortium, which also develops an open source implementation of the system.

The main difference between X and other windowing systems like Microsoft Windows is that X takes the client/server approach. Where other systems have an elaborate API from which you directly call the different functions of the graphics system, X provides a network protocol instead.

The important thing here is to understand what is the server, and what is the client. Under normal circumstances the computer which the user is physically in front of is the client and the applications are running on the server. However, X is somewhat different. The different applications running and displaying graphics are the X clients. These applications connect to the X server which displays the graphics to the user. Thus the X-clients run on the server computer, but the X-server runs on the client computer, see figure A.1.

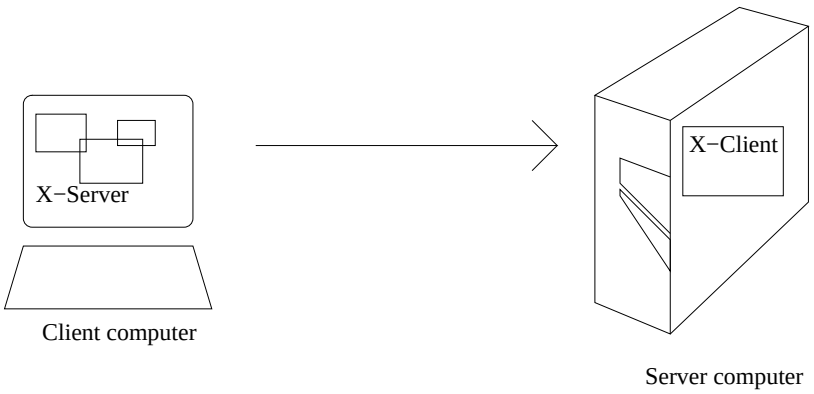

Figure A.1: The X-Windows system.

The basis of the X system is the X protocol [N+95]. The protocol describes four types of packets sent between the server and the client. The client sends requests, while the server may send replies, events and errors. So if a client wishes to draw a circle somewhere in a window it would send a request for drawing a circle. After a connection between the X-client and X-server has been established, an X session could look as the example seen in figure A.2.

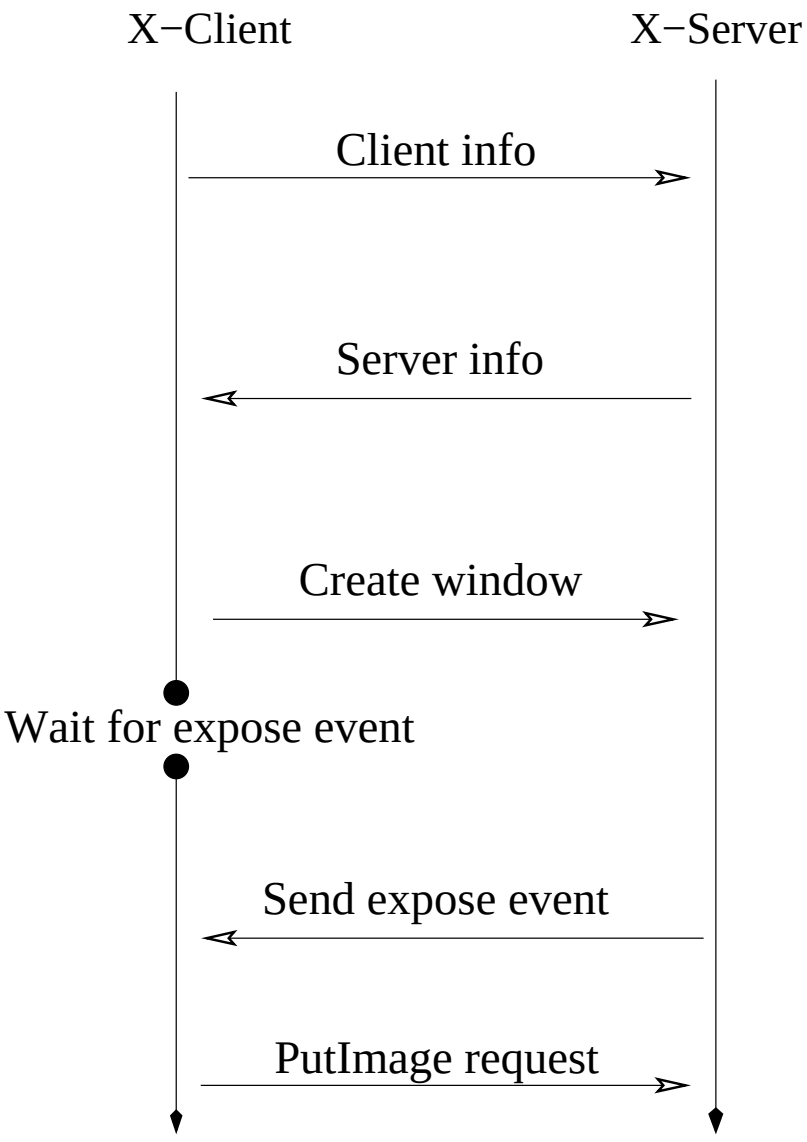

Figure A.2: An example session.

A.1 Setup

When a client has connected to an X server the protocol enters the setup phase. In this phase the client declares which version of the protocol it intends to use. Additionally the client informs the server of which byte order it uses. Furthermore, the client will also send an authentication block so that the server can verify that the client has permission to access the server.

If the server grants access to the client, the server will transmit a large information block. This block contains information about the size and depth of the display, the byte and bit order of bitmaps and images, resource IDs (A.6), different buffer sizes and so forth.

A.2 Requests

An X request is defined as a header of four bytes with zero or more additional four byte chunks. The header contains a major opcode, a data byte and a length field. The opcode 1 to 127 are reserved for standard requests while opcode 128 to 255 are used by extensions. Each extension may support many additional requests that are identified by a minor opcode, which then is stored in the data byte.

A.3 Replies

Some, not all, requests need a response from the server, these are sent as replies. A reply always contains at least 32 bytes of data and zero or more 4-byte chunks of additional data. The first byte has the value of 1. If the value is 0, it is not a reply but an error, or if it is larger than 1, it is an event.

A.4 Events

Any time during an X session the server may send the client some kind of event. An event message is 32 bytes long, with the first byte defining which type of event it is.

A.5 Errors

If the information sent in a request is somehow faulty, the X server will respond with an error message. This is indicated by the first byte being zero. Error messages are always 32 bytes long.

A.6 Resource identification

Resources like windows, pixmaps, colormaps and so forth are identified with 4 byte integers called resource IDs. When a client creates a window it sends a CreateWindow request containing the resource ID it wants to use to identify it. The ID is generated by the client using two numbers which it received from the server at the beginning of the session.

Since the resource ID base and mask may change during each connection the client may need to calculate new values each time. This must be taken care of during playback of log files.

A.7 Atoms

Window titles and sizes are examples of *properties*. A property is identified by an atom. Atoms are character strings like AM_NAME and AM_ICON_SIZE. Some of these atoms, like the two examples above, are predefined in the X standard. Since sending character strings over a network connection, compared to sending single integers, is bandwidth demanding, there are numerical mappings for these strings. The predefined atoms also have predefined mappings, but certain atoms may not have that so there is an X request for requesting that mapping. This is also an issue that need to be handled by the playback software.

A.8 Extensions

Extensions to X are packages of additional requests and replies. This enables developers to add functionality to X. An extension is used by sending the QueryExtension request. The reply will contain the major opcode to use. Since extensions may support many requests a single opcode will not suffice, therefore additional minor opcodes are used. The minor opcodes are defined by the extension. There are 128 major opcodes available, but that many are seldom needed.

A.9 An X example

To illustrate the use of atoms and resource IDs here is a small, and purely educational, example.

A client has successfully connected to the server and has received the *base* and *mask* values. The client then wants to create a small window and call it "Bob".

To create the window the client has to send a "Create Window" request, but first it has to decide how to identify it. By using the base and the mask it selects a number (resource ID) X. The client then transmits a message telling the server to create a window using the resource ID X.

The client wants to set the window title to "Bob". This is accomplished by changing the *property* "AM_NAME" for *window* X. "AM_NAME" is an atom and to change the property identified by that atom the client has to know the numerical translation of that name. "AM_NAME" is a predefined atom and has the numerical translation of 39. Therefore the client will send a "Change Property" request to the server containing the resource ID X, the atom number 39 and the string "Bob".

Appendix B

Low level detail

This chapter explains some low level details like file formats and data structures as well as giving some examples of the code.

B.1 The general TCP log file format

The file format for general TCP logs has the following structure:

```
  struct TCPLog {
    char type[4]; /* contains the string MTCP */
    uint8_t info_lenght; /* length of the information string */
    char info_string[info_length]; /* Human readable information */
    uint8_t *data_blocks; /* the logged data */
  };

  struct TCPdata_block {
    char prefix[4]; /* either ENTA or ENTB, depending on direction */
    uint32_t length; /* amount of data */
    uint8_t data[length];
  };
\end{erbatim}

\section{The X log file format}
An X log contains two separate files; one file for the requests sent
from the client and one file for the replies, errors and events sent
by the server. A X log file has the following format:

\begin{verbatim}
  struct XIILog {
    char type[4]; /* either XICL or XISL, depending on Server
                   * or Client.
   */
    uint8_t info_length; /* length of information string */
    char info_string[info_length]; /* A string of information */
```

```
    uint32_t prefix_len; /* length of the prefix */
    uint8_t prefix[prefix_len]; /* Either the client or
                                 * server prefix, see the X protocol
                                 * reference manual.
                                 */
    uint8_t *time_blocks;
  };

  struct XIILtime_block {
    uint32_t time_stamp[2]; /* seconds, microseconds */
    char type; /* either 'C' or 'S', client or server,
                * used for debugging.
                */
    uint32_t length; /* timeblocks length */
    uint8_t data[length]; /* One or more X packets */
  };
```

B.2 X log data structures

B.2.1 Fast buffer

One of the more important structures in XIIL is the buffer structure.

```
/*
 * Fast buffer
 */
typedef struct _f_buffer {
    xiiPacket type, def; /* the current and default type of the
                          * first packet in this buffer.
                          */

    int writp, readp; /* write and read ports (FDs) */
    int completed; /* 0 if first packet is incomplete !0 otherwise */

    FILE *log; /* pointer to a log file */
    struct timeval timer; /* next packet timer */

    char *name; /* the first character in the string must be
                 * 'c' or 's' (client or server)
                 */

    int block_fsize, block_fposition; /* indicates the size and position
                                       * of the current timeblock
                                       * in the log file
                                       */

    // backstore is used for swap, see f_buffer_read()
    uint8_t *data, *backstore, *pointer;
    uint32_t size; /* actually uint8_t *pointer
                    * points at a memory area twice
```

```
                    * as big as size.
                    */

    uint32_t unsent, us_len, /* index of first byte of unrelayed
                              * data, and length */
        fp, we_need, we_got; /* index of first packet, how
                              * much we need, and how much we
                              * got */

    /* playback specific fields */
    long countdown; /* used to set the timer above */
    long prefixlen;/* lenght of the first block sent by client/server */
    xConnSetupPrefix *setupprfx;
    xConnSetup *setup;
    xConnClientPrefix *clientprfx;
} f_buffer;
```

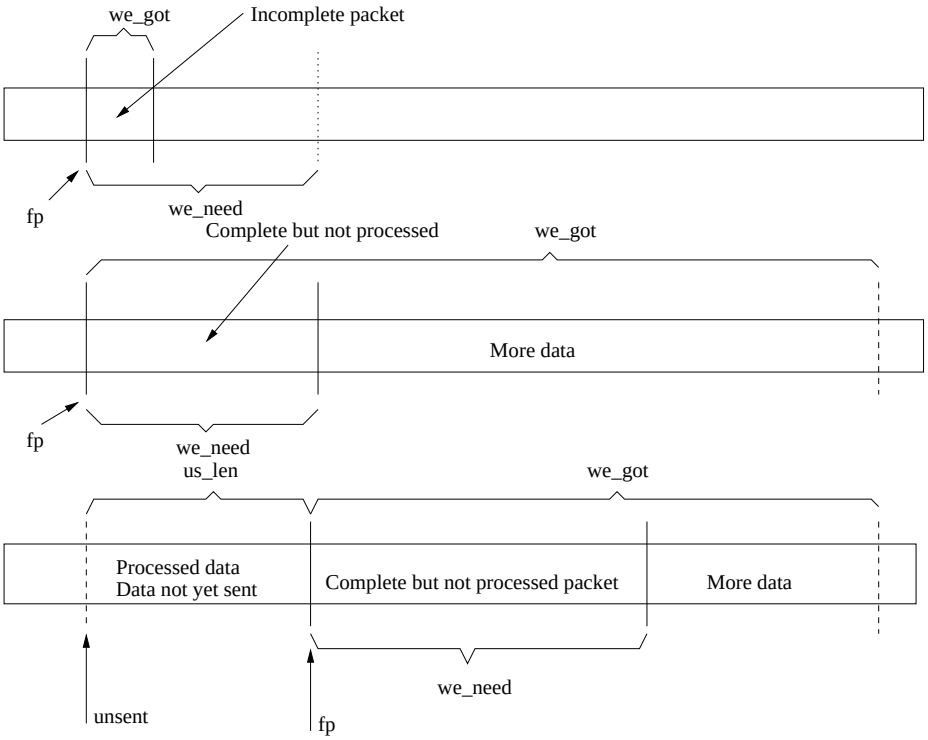

Figure B.1: A buffer at three different points in time.

The "type" field indicates the current state of the first packet, while the "de" field indicates what "type" should be set to by default. The "writ" and "read" fields state the port that the buffer should write to and the port from where it should read. These are in fact just file descriptors. XIIL must count the number of bytes it writes to the log file for each time stamp, this number is stored in "block_frize". When the whole time block is completed the number will be written to the log at the position stored in "block_fposition". An example view of a buffer in action is shown in figure B.1. The example begins when the buffer has received the initial amount of data in a packet and determined that the need is larger. At the next point in time the buffer has received another chunk of bytes and the total amount now exceeds the need. The now complete packet is processed and the

"fp" pointer is increased set to point to the next packet. To indicate that the buffer contains data that should be written to "writep", the "unsent" pointer and "us_len" field are set as shown.

B.3 XIIL code examples

B.3.1 Byte order conversions

To generalize the way in which byte order conversions are handled, XIIL has two function pointers, *r2l_short()* and *r2l_long()*. The two pointers are set by a call to one of two functions when an X client has connected. *r2l_short()* is used to convert 16 bit integers between "remote" and "local" byte order, while *r2l_long()* is used for 32 bit integers.

```
uint16_t (*r2l_short)(uint16_t);
uint32_t (*r2l_long)(uint32_t);

/* This is called if the client uses MSB */
void set_msb(void)
{
  r2l_short = short_msb;
  r2l_long = long_msb;
}
/* This is called if the client uses LSB */
void set_lsb(void)
{
  r2l_short = short_lsb;
  r2l_long = long_lsb;
}
```

The functions pointed to by *short_msb* and *long_msb* depend on the native byte order of the machine that XIIL was compiled for, in the Intel PC case it is Least Significant Byte first, LSB.

```
uint16_t short_msb(uint16_t src)
{
  uint16_t swap;
  uint8_t *a = (uint8_t *)&swap;
  uint8_t *b = (uint8_t *)&src;

  a[0] = b[1];
  a[1] = b[0];

  return swap;
}

uint16_t short_lsb(uint16_t src)
{
  return src;
}

uint32_t long_msb(uint32_t src)
```

```
{
  uint32_t swap;
  uint8_t *a = (uint8_t *)&swap;
  uint8_t *b = (uint8_t *)&src;

  a[0] = b[3];
  a[1] = b[2];
  a[2] = b[1];
  a[3] = b[0];
  return swap;
}

uint32_t long_lsb(uint32_t src)
{
  return src;
}
```

B.3.2 The request ID generator and mapper

Here is the first few lines from the function responsible for adding new request IDs to the mapping database (a hash table):

```
/* Generates new mappingvalues for resource IDs that
 * were generated by the X client.
 */
void request_id_generator(
   long mdb, /* The ID of the current mapping database */
   xReq *pr) /* A pointer to the request */
   {
     switch(pr->reqType) {
       case X_CreateWindow:
       {
         xCreateWindowReq *req =
           (xCreateWindowReq *)pr;

         XIILmap_prepare_rid(
             "wid(gen, CreWin)",/* Some text to help during debugging */
             mdb, /* The ID of the database */
             req->wid); /* The value that should be replaced,
                         * XIILmap_prepare_rid takes care of all
                         * byte conversions.
                         */
       }
       break;

       .
       .
       .
```

Here is a snippet of the request mapper function, a switch block of 1100 lines.

```
void request_mapper(
  long mdb,   /* the mapping database ID */
  xReq *pr) /* a pointer to the request */
  {
    .
    .
    .
  /* Before the values in the request
   * can be mapped, we must make sure
   * they are added to the database first.
   * the request_id_generator() will only
   * add values if they do not already exist.
   */
  request_id_generator(mdb, pr);

  switch(pr->reqType) {
    .
    .
    .
    case X_ChangeProperty: /* this request changes the property
                            * of a window, like the WM_TITLE
                            * for example.
                            */
    {
      xChangePropertyReq *req =
          (xChangePropertyReq *)pr;
      req->window = XIILmap_rid("window(chanpro)",
                                mdb,
                                req->window);
      req->property = XIILmap_atom("property(chanpro)",
                                   mdb,
                                   req->property);
      req->type = XIILmap_atom("type(chanpro)",
                                mdb,
                                req->type);

    }
    .
    .
    .
```

B.4 Control center commands

The control center understands a small number of commands, where each command has a different number of arguments. The commands and their arguments are:

- help
- info
- kill <client number>

- details <client number>
- monitor <client number> <display info>
- quit

The help command will print a small explanatory text to the terminal explaining the different commands and how to use them. The info command will display information about the different clients currently connected. If the administrator would like to have more information about a specific client he or she can use the details command. The kill command will immediately end the specified client's operations and disconnect the X client from the X server. The monitor command will initiate a connection to a secondary X server which will mirror the display of the real X server. The commands are sent from the "terminal" to the "control center" in plain text and the center will parse and analyze the text.

The packets sent between the clients (X loggers) and the control center are defined as follows:

```
/* client->center msgs */
#define XIIL_MSG_DISCONNECTED 1
#define XIIL_MSG_INFORMATION 2
/* center->client msgs */
#define XIIL_MSG_KILLCONNECTION 20
#define XIIL_MSG_REQINFO 21
#define XIIL_MSG_STARTMONITOR 22

/* messages from the control center
 * to the client.
 */
typedef struct {
  char msgid; /* center->client */
  int p1, p2, p3; /* three integers, their use is defined
                   * by the msgid
                   */
} ClientPacket;

/* messages from the client to the
 * control center.
 */
typedef struct {
  char reply_id; /* client->center */
  int reply_length;
  char *reply;
} ClientReply;
```

A *XIIL_MSG_STARTMONITOR* packet has the following structure:

```
  char  msgid    = 22
  int   p1       = n
  int   p2       = <not used>
  int   p3       = <not used>
  char *str      = <n bytes of data>   # A display string, like ":0.0"
```

Appendix C

Glossary

BSD

BSD, Berkeley Software Distribution, is a version of UNIX developed at the University of California, Berkeley. Common variations of BSD are FreeBSD, OpenBSD and NetBSD. These are all open source and free operating systems, much like the famous operating system Linux.

Citrix

Citrix Systems, Inc, has created a system for controlling applications remotely. This is similar to one of the aspects of X Windows. The main differences with X is that it is not an open protocol, and that the the applications controlled via Citrix are unaware of it.

Daemon

A daemon in the UNIX world is a program that runs in the background performing some service. Like a telnet daemon which enables a person to connect to a machine from a remote machine.

Ethereal

The home page for Ethereal states the following: "Ethereal is a free network protocol analyzer for Unix and Windows. It allows you to examine data from a live network or from a capture file on disk. You can interactively browse the capture data, viewing summary and detail information for each packet. Ethereal has several powerful features, including a rich display filter language and the ability to view the reconstructed stream of a TCP session."

OpenSSH

OpenSSH is a free implementation of SSH developed by the OpenBSD project. See SSH and BSD for more information.

Operation code

A number (integer) used to identify a certain operation or command. For example the low level instructions of a CPU.

SSH

SSH, or Secure SHell is an application that allows a user to connect to a remote machine using an encrypted TCP session. It also supports the creation of "tunnels". Using "tunnels" ordinary TCP traffic may be routed via the encrypted connection.

SSH-Tunnel

See SSH.

UNIX domain socket

A UNIX domain socket is a communication method between processes on the same machine. The socket is accessed via special files.

VNC

VNC, or Virtual Network Computing [Rea02], is a system for viewing and interacting with a computer remotely (see Citrix). VNC works by transferring the bitmap representation of a display over a network to another computer. This result is an (almost) identical copy of the original display.

xauth

The xauth program, supplied with the X Consortium's implementation of X Windows, is used to edit and read authorization information stored in a special file. This authorization information is used for authentication when connection to an X server.

The X Consortium

The X Consortium, or the X.org Consortium, is an organization of the Open Group. X.org is "the worldwide consortium empowered with the stewardship and collaborative development of the X Window System technology and standards."

X Windows

X Windows, developed by the X consortium, is a system for window based applications. Microsoft Windows is a similar system. X, however, is designed specifically for centralized applications controlled from remote X enabled terminals.