Leopold-Franzens-University Innsbruck

Institute of Computer Science
Distributed and Parallel Systems

PTP Implementation in SNMP

Bachelor Thesis

supervised by:
Dr. Michael Welzl

Authors:
Robert Binna
Eva Zangerle

Innsbruck, 7th March 2006

Abstract

This bachelor thesis deals with the implementation of the Performance Transparency Protocol (PTP) using the Simple Network Management Protocol (SNMP).

Acknowledgements

Our biggest thanks goes to Michael Welzl, who offered this bachelor thesis. He always was very supportive and ready to answer all our questions. The discussions with him contributed some good ideas to our final result and were very helpful throughout the implementation process.

We also would like to thank Ben Lavender for the help with the interface description and Martin Rabanser for the original PTP implementation.

Contents

Chapter 1

Introduction

1.1 Aims of Implementation

The main aim of this bachelor thesis was the implementation of the functionality of the Performance Transparency Protocol (PTP, Chapter 1.2) using the Simple Network Management Protocol (SNMP, Chapter 1.3).

This functionality is needed because it enables users to retrieve bandwidth information using PTP from a network without having to install PTP on every router in the network. The only prerequisite is that each host along the route has an SNMPv3 client installed. This implementation makes it possible to use PTP in networks without having to apply special patches to the router firmware.

The bandwidth information retrieved via the Performance Transparency Protocol can be used for various applications. One of them is a congestion control mechanism like CADPC (Congestion Avoidance with Distributed Proportional Control, [1]). Such an application should easily be able to retrieve the requested bandwidth information from PTP and should also be able to set certain parameters in order to retrieve relevant and reasonable data.

1.2 Performance Transparency Protocol

The Performance Transparency Protocol was developed by Michael Welzl. Using PTP, a network entity can efficiently request performance parameters from the

network. These parameters are:

- available bandwidth
- bottleneck bandwidth (link with worst performance)
- bit error ratio (number of incorrectly received bits)
- path mtu (maximum transfer unit - largest possible packets that will not be fragmented)

PTP packets sent from the source to the destination are updated by the intermediate routers along the path. This is accomplished by using the Forward Packet Stamping Mode. This means that the routers either just add their information or they compare it with the information already contained in the packet, in which case the router simply overwrites the information in the packet if its information is "worse" (according to the definition in the PTP specification). The packets are either directly sent back to the source or to the next router along the path. Finally the receiver builds a table of router entries and calculates the requested information, which is then sent back to the sender.

1.3 Simple Network Management Protocol

SNMP is a protocol used for the communication between network entities. These entities are either management stations (e.g. consoles) or managed objects (e.g. routers or gateways). SNMP is used to retrieve or manipulate the entries of MIBs (Management Information Bases, see Chapter 1.4).

All the components controlled by a management console need a SNMP agent - a module that is able to communicate with the SNMP manager.

With SNMP, all the parameters regarding network attached devices can be monitored. Doing so, the current state of a network is monitored. SNMP works with both IPv4 and IPv6.

SNMP can be used for:

- Retrieving information from a network attached device, which has to be SNMP capable.

- Manipulating configuration information on a network attached device, which has to be SNMP capable.
- Retrieving a fixed set of information from a SNMP capable device, which is attached to the network.
- Converting and displaying MIB contents and its structure.

1.4 Management Information Base

A MIB is a virtual database consisting of a set of objects. MIBs are used to manage devices in a communications network. The MIB database is structured hierarchically and the different entities within the hierarchy are adressed through object identifiers (OIDs). These entities can be retrieved or modified by SNMP (Simple Network Management Protocol (see Chapter 1.3).

Chapter 2

Implementation

2.1 Design Decisions

2.1.1 Reasons for Choosing C++

The first implementation attempt was in pure C, but it soon became clear that a fully fledged object oriented language was needed to fulfill all the requirements to implement an interface that is easy to use and maintain. Another reason was that it should be quite easy to port the library to other operating systems. Due to the object oriented design the implementation of language bindings is a minor effort.

2.1.2 Callback vs. Threading

The most difficult decision was to choose whether to use callbacks or threads to collect the requested data. The first suggested approach was to code it in the same fashion as the original PTP implementation. In this case it was solved through a combination of multithreading and polling the bandwidth data. For us the most obvious thing to do was to combine the best of both - a scheduled request for the bandwidth data that would invoke a callback function each time the data gets updated. So the user only has to take care about when to start and stop the measurements.

2.2 Traceroute

Traceroute determines the IP adresses of all routers along the path to the destination. For each destination adress, a new thread is created.

To retrieve the IP adresses of the routers along the path to the destination router, UDP (User Datagram Protocol) and ICMP (Internet Control Message Protocol) messages are used.

UDP packages contain a "TTL" - variable (Time To Live). This variable describes the number of hops the packet can take on the network before being discarded. Every time a router receives a UDP packet, the TTL counter gets decremented by one. As soon as the TTL counter reaches zero, the router returns a ICMP_TIMXCEED error message to the dispatcher of the packet. This packet contains the IP of the router which originally sent the error message, which can easily be extracted.

This behaviour can be used to implement a traceroute application. By setting the TTL to one, the IP of the first router along the path can be obtained. Increasing the TTL by one until the destination host is reached finally leads to the complete reconstruction of the path of the packets.

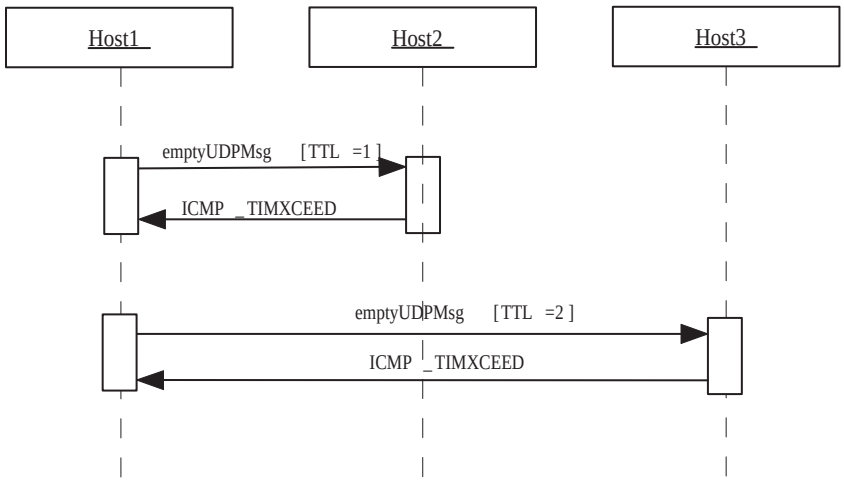

Figure 2.1: Traceroute IP Detection

This implementation of traceroute functionality is threadsafe. For every destination adress a new traceroute thread is created. Threading is done by the ZThreads multithreading and synchronization library [6]. To ensure threadsafety, several Mutex constructs of this library are used.

2.3 Bandwidth Calculation

The first step in the bandwidth calculation process is the execution of a traceroute call. The resulting IP adresses of the routers along the path are stored in a vector.

After having detected the path, the bandwidth of each router has to be detected. Therefore the following connection parameters are retrieved from every router via a SNMP call.

- ifSpeed - nominal bandwidth in bits per second.
- ifInOctets - total number of octets received.
- hrSystemDate - current router date and time.

The following formula describes the calculation of the bandwidth of a host. "inOctets" is the number of bytes received on the current SNMP request. "previousInOctets" stands for the number of bytes received on the previous request. Analogously "check" stands for the date and time as of the last SNMP request and "previousCheck" stands for the date and time of the previous request.

$$BW(bits/sec) = 8 * 1.000.000 * \frac{inOctets - previousInOctets}{check - previousCheck}$$

The calculated bandwidth has to be multiplied with a factor of 1.000.000, because the time unit used by the hrSystemTime timestamp is microseconds and seconds are needed. Additionally the bandwidth has to be multiplied with a factor of 8, because the bandwidth should be stated in bits per second.

Due to the fact that the octets counter may suffer an overflow, another case has to be considered. Namely if inOctets is smaller than previousInOctets, the formula for the bandwidth has to be as follows.

$$BW(bits/sec) = 8 * 1.000.000 * \frac{2^n - previousInOctets + InOctets}{check - previousCheck}$$

The above formula is implemented by subtracting previousInOctets from a 32 bit long 0 and then adding InOctets.

Obviously the first pass of requests does not lead to a result, because there is no information about a previous request available to calculate the bandwith from. Therefore, a "Not enough Information to calculate bandwidth" - exception is thrown.

If the route changes during the bandwidth calculation (IP address of one or more routers along the path changes), the whole bandwidth detection is started all over again. Therefore a new traceroute is executed. Based upon the new traceroute path the bandwidths of all routers along the path are retrieved again. Once the second pass of retrieving bandwidth information is completed, the new and updated bandwidth gets returned. A timeout during a traceroute call or during a SNMP request is handled exactly the same way.

2.4 Class Diagram

Figure 2.2: Class Diagram

2.5 Message Sequence Chart

This message sequence chart shows the execution of a traceroute call and the SNMP requests for a sample path with four hosts included. Host1 sends empty UDP packages to the hosts and receives ICMP Messages containing the other host's IP adresses. In the next step, SNMP requests are sent to the hosts along the path and the requested SNMP values are returned.

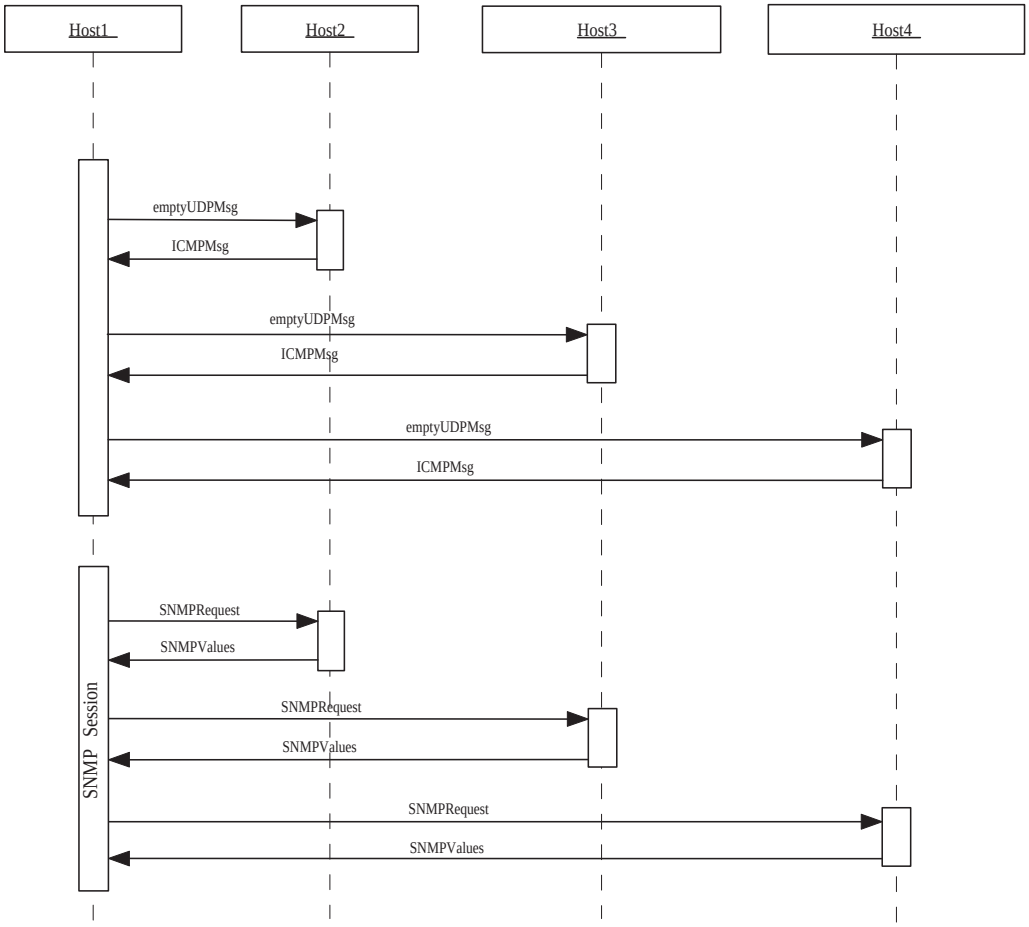

Figure 2.3: Message Sequence Chart

2.6 Daemon

The whole implementation of the PTP functionality is wrapped into three daemons. The main reason for choosing the implementation of a daemon instead of a library was the fact that a daemon implementation makes it possible that only ICMP socket is needed to retrieve the ICMP messages and distinguish between the different messages. Furthermore permissions were needed for the execution of the traceroute.

The implementation consisting of three separate daemons and therefore three separate processes has various advantages. The first advantage is modularity. A part of the software could be changed without affecting the other parts. Another advantage is the configurability of the whole system such that different implementations could be exchanged during runtime. Even though there are three processes, installing the package containing the three daemons is as easy as the installation of just one daemon. According to that, also the dependencies among the processes are captured within the packaging system.

The implemented daemons are:

- traceroute daemon - covers the traceroute functionality
- bandwidth algorithm daemon - contains the bandwidth interval algorithm
- bandwidth retriever daemon - comprises the actual bandwidth retrieval

The interprocess communication between these three daemons happens via Unix Domain Sockets. Figure 2.4 is a schematic diagram of the daemons. The protocol family "PF_UNIX" was used for the interprocess communication. Unix Domain Sockets are named with Unix paths (e.g. /tmp/socket_name). More information about Unix Domain Sockets can be found at the Netzmafia Website [8].

To be able to exchange complex datatypes between the daemons, these objects were serialized. This was done by using the Boost Serialization Library [4]. Also the exceptions were serialized in order to be able to inform the client about any occured exception.

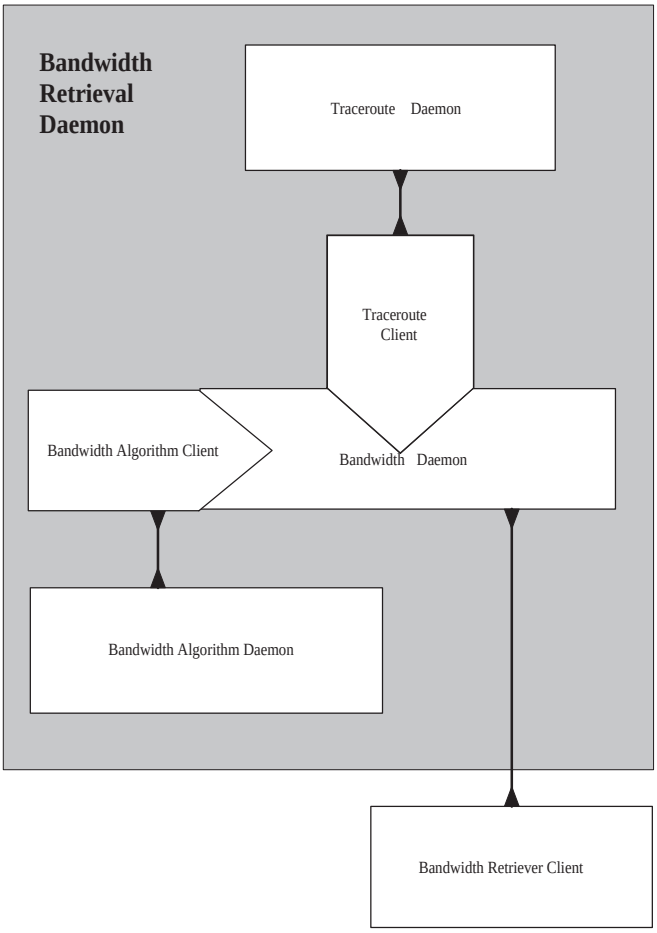

Figure 2.4: Daemon architecture

2.6.1 Message Sequence Charts

The following two message sequence charts show how the three daemons and the client interact. The first message sequence chart shows the interaction without the occurrence of any error. The second message sequence chart shows the interaction when the client does not respond any more. The retriever sends the bandwidth twice and if the client does not respond with an acknowledgement message, the whole bandwidth retrieval process is stopped.

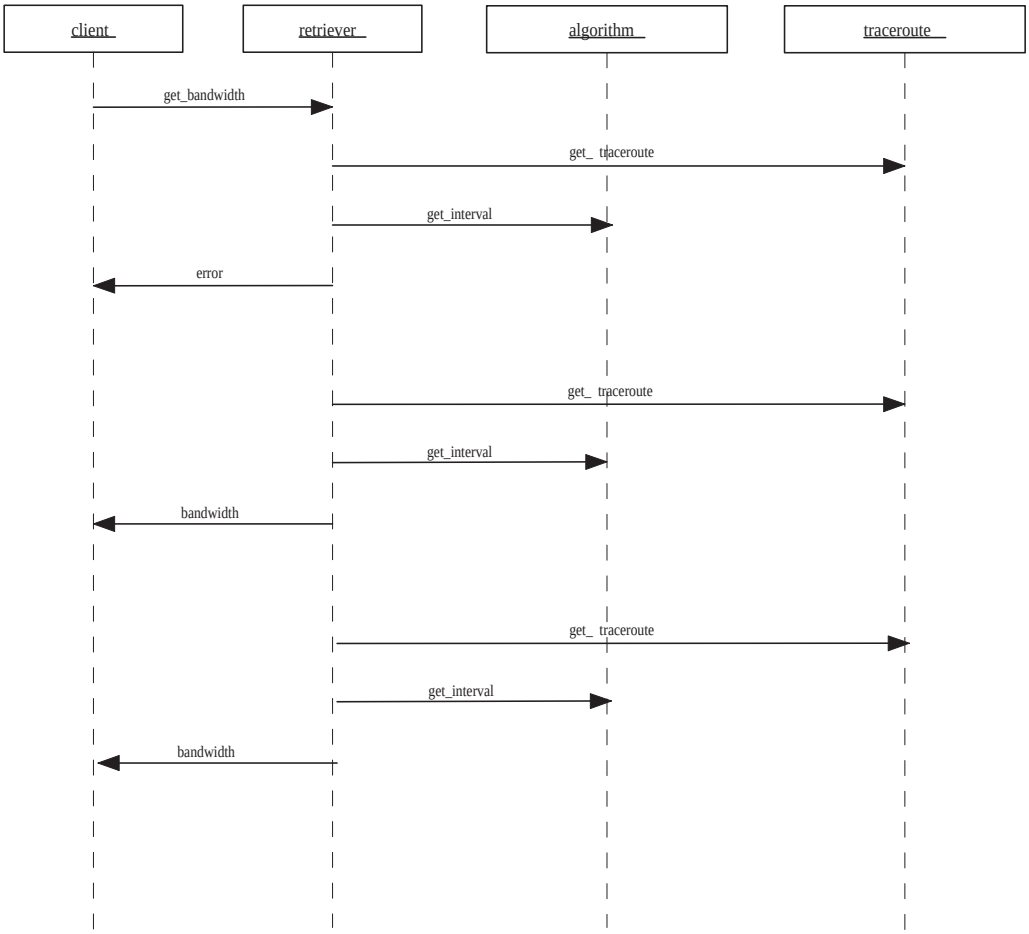

Figure 2.5: Communication between daemons without any error

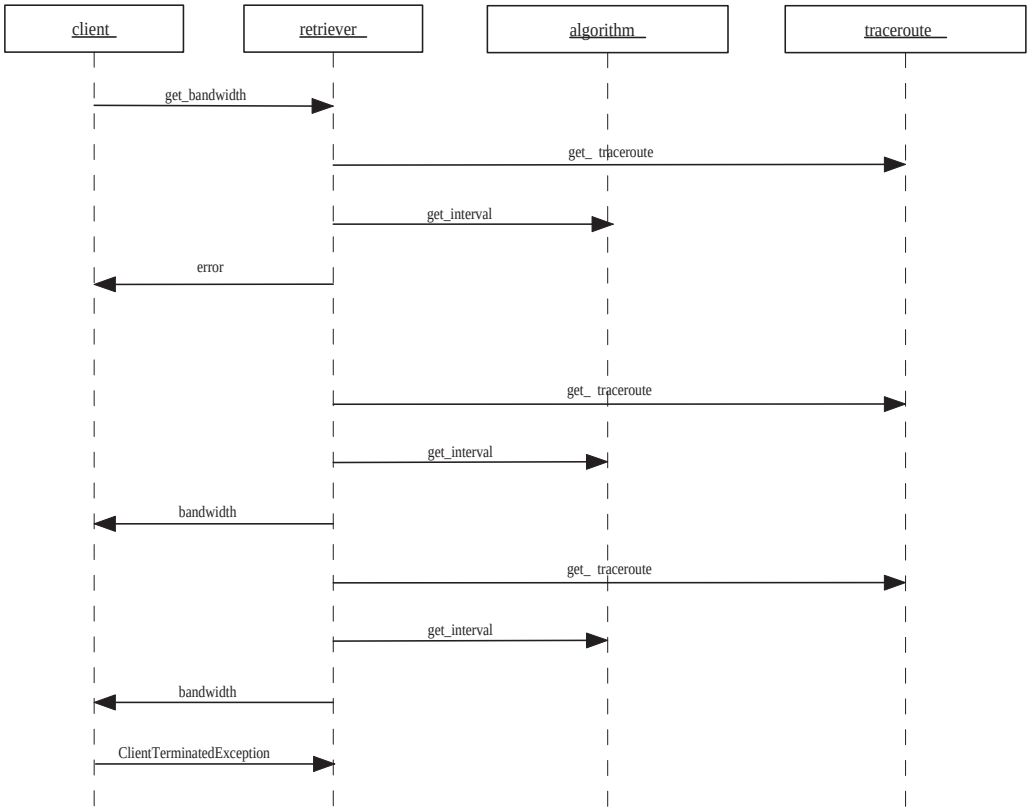

Figure 2.6: Communication between daemons, no response from client

2.7 Programming Environment

2.7.1 Net-SNMP

The Net-SNMP suite contains libraries and tools needed to use the Simple Network Management Protocol. This suite provides command line applications to retrieve information from a certain Management Information Base via single or multiple requests.

Additional information about Net-SNMP can be found at the official website [3].

2.7.2 Boost C++ Libraries

Boost is a free collection of libraries for C++. It e.g. provides shared pointers and several functions for date and time handling, which were used for the implementation. Also the serialization library of Boost was used. A part of the Boost library will be included in the C++ Standard.

Further information about Boost can be found at the official website [4].

2.7.3 SCons

SCons is a software construction tool used for this implementation. It is an alternative to the commonly used "Make" build tool. SCons is based on Python and uses Python scripts as configuration files. It is easily embedded in other applications. Another advantage of SCons is the automatic scanning of files for dependencies, whereas Make stores dependencies statically in a file.

Further information about SCons can be found at the official website [5].

2.7.4 ZThread

ZThread is a platform-independant C++ library for object oriented synchronization and threading.

Additional information about ZThread can be found at the official website [6].

2.8 Exception Handling

The exceptions are specified in the "exceptions.h" - file. In order to be able to directly locate the origin of an exception, the parameters __FILE__,__LINE__ are passed on to the exception class.

To simplify matters, a macro for the creation of exceptions was implemented. Using this macro, a new exception can be created with just one command. All the exceptions used within the implementation inherit from a PTPException. The advantage of this approach is that all exceptions can be catched by this basic exception.

2.9 Problems

One big problem was the implementation of a multithreaded traceroute application. There are many traceroute implementations available, but none of these was threadsafe. Therefore a threadsafe implementation of a traceroute also had to be implemented.

For the multithreaded traceroute, each of the threads creates its own ICMP Socket for receiving the ICMP messages from the routers along the path. These ICMP messages are received because the routers respond to the empty UDP messages that were sent. This method is used to obtain the IP adresses of the routers contained in the ICMP messages. Even though it is possible to create more than one socket it would not make sense, because the system would not know where to dispatch the ICMP message to. The reason for this behaviour is the stateless nature of ICMP.

Another problem occured when sending UDP packages in order to obtain the IP adresses of the routers along the path to the destination router. The checksum of the UDP packets always was calculated incorrectly when using the system command "send" and setting the TTL parameter. Therefore the UDP packets sent to the hosts in the traceroute had to be built within the implementation. This was done by using raw sockets and setting socket options. More information about programming with raw sockets can be found at "Zotteljedis Tipps zur Programmierung mit Raw Sockets" [7].

Chapter 3

Traffic Analysis

Analyzing the network traffic during the bandwidth retrieval process leads to some interesting results. This analysis can be done by executing the command "tcpdump", which prints out the headers of packets on a network interface. The resulting file can be analysed with ethereal, which is a network protocol analyzer.

The test environment consisted of three nodes:

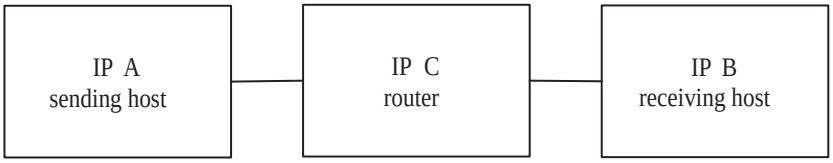

Figure 3.1: Test Environment

This is a list of all packets involved in one bandwidth retrieval process, where IP A requests bandwidth from IP B:

Source	Dest.	Prot.	Info
IP A	IP B	UDP	Source port: 40993 Destination port: 40994
IP C	IP A	ICMP	Time-to-live exceeded (Time to live exceeded in transit)
IP A	IP B	UDP	Source port: 40993 Destination port: 40995
IP B	IP A	ICMP	Destination unreachable (Port unreachable)
IP A	IP C	SNMP	GET
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP C	SNMP	Source port: 32771 Destination port: snmp
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP C	SNMP	Source port: 32771 Destination port: snmp
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	GET
IP B	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	Source port: 32771 Destination port: snmp
IP B	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	Source port: 32771 Destination port: snmp
IP B	IP A	SNMP	Source port: snmp Destination port: 32771

3.1 Traceroute

These are the packets involved in the detection of the route to the destination.

Source	Dest.	Prot.	Info
IP A	IP B	UDP	Source port: 40993 Destination port: 40994
IP C	IP A	ICMP	Time-to-live exceeded (Time to live exceeded in transit)
IP A	IP B	UDP	Source port: 40993 Destination port: 40995
IP B	IP A	ICMP	Destination unreachable (Port unreachable)

As already described in Chapter 2.2, the IP addresses of the routers along the way to the destination are determined by sending out UDP packages. To determine the first router, the TTL of the UDP package is set to one. This can easily be checked by reviewing the detailed information about the first UDP packet sent:

```
Internet Protocol, Src: IP A (IP A), Dst: IP B (IP B)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 40
    Identification: 0x2021 (8225)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 1
    Protocol: UDP (0x11)
    Header checksum: 0x1750 [correct]
        Good: True
        Bad : False
    Source: IP A (IP A)
    Destination: IP B (IP B)
```

The first router - in this case IP C - receives this packet and returns a ICMP_TIMXCEED packet, because the TTL counter has reached zero. After this step, the IP address of the first router has already been obtained. The next step is to send another UDP packet with a TTL of two. This packet forces the second router, which is the receiving host, to return an ICMP message. As this router is already the destination of the traceroute call, the IP addresses of the routers along the way were detected successfully.

3.2 Bandwidth Retrieval

After having determined the routers along the path to the destination, the retrieval of the information required for the bandwidth calculation starts. The following table shows the SNMP packages sent during the information retrieval.

Since three values are required for the bandwidth calculation (ifSpeed, IfInOctets and hrSystemDate), three SNMP request packages are sent to every router along the path. In the table below one can easily see that firstly the information from the first router is requested. After having obtained this information, the requests are sent to the receiving host.

Source	Dest.	Prot.	Info
IP A	IP C	SNMP	GET
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP C	SNMP	Source port: 32771 Destination port: snmp
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP C	SNMP	Source port: 32771 Destination port: snmp
IP C	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	GET
IP B	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	Source port: 32771 Destination port: snmp
IP B	IP A	SNMP	Source port: snmp Destination port: 32771
IP A	IP B	SNMP	Source port: 32771 Destination port: snmp
IP B	IP A	SNMP	Source port: snmp Destination port: 32771

Chapter 4

Installation

The following installation instructions are intended for Fedora systems (Core 4 and upwards).

4.1 Installation of Net-SNMP

The first component that needs to be installed is Net-SNMP. If Net-SNMP is not installed on the computer, run the following commands in root mode, which install Net-SNMP.

```
# yum install net-snmp
# yum install net-snmp-devel
```

If yum is not installed on the machine, please follow the installation instructions on the official websites [3].
The next step is to register Net-SNMP as a service and configure autoload on bootup.

```
# chkconfig --add snmpd
# chkconfig snmpd on
```

Next, a new user has to be added. For the registration of a user the service has to be stopped, if it is already running.

```
# service snmpd stop
```

The following command adds a new SNMPv3 user with a given username and password.

```
# net-snmp-config --create-snmpv3-user -ro -a "password" username
```

Further information on SNMP authentication issues can be found at [9]
The last step is to start the service.

```
# service snmpd start
```

To make sure that the installation process was finished correctly, simply try to execute a SNMP walk. This command retrieves all the information stored in a MIB on localhost, where ¡username¿ and ¡password¿ have to be substituted with the real username and password.

```
# snmpwalk -v 3 -u <username> -l auth -n "" -A <password> localhost
```

4.2 Required Libraries and Tools

Besides Net-SNMP, the Boost libraries are required in order to be able to build the PTP - SNMP implementation. This is done by the following commands.

```
# yum install boost
# yum install boost-devel
```

If yum is not installed on the system, please visit the official websites for further information regarding the installation [4].

4.3 Installation of PTP

The first thing to do is to get the rpm - package. It can be downloaded from the PTP Sourceforge website [10]. After having obtained the rpm file, PTP can easily be installed using the following command in the directory, where the rpm file was stored:

```
$ rpm -ivh installationpackage.rpm
```

where installationpackage.rpm stands for the name of the downloaded rpm file.

4.4 Daemon

The PTP daemon can be started using the following command:

```
$ service ptpd start
```

Analogously the service can be stopped:

```
$ service ptpd stop
```

The following command restarts the daemon:

```
$ service ptpd restart
```

The current status of the daemon can be requested with the following command.

```
$ service ptpd status
```

4.5 Minimal System Requirements

The following system requirements have to be fulfilled in order to make sure that the PTP implementation works correct.

- Fedora Core 4 or higher
- boost version 1.32.0 or higher
- boost-devel version 1.32.0 or higher

4.6 Building RPMs

Building new RPMs (RPM Package Manager) packages can be done using the following instructions. Further information about building RPM files can be found at the official RPM website[11].

For the creation of a RPM file, a .spec file is needed. This file contains the description of the package, compile information for the source and a list of all binaries that have to be installed. The .spec file for the PTP RPM can be found on the source CD or at Sourceforge [10]. When creating a new RPM, this file has to be adapted.

The actual creation can only be executed as root. The next step is to configure a build tree. A build tree consists of the following subdirectories: BUILD, SOURCES, SPECS, RPMS, SRPMS. It can be found at /usr/src/redhat/ in the file system. Copy the spec file to the SPECS directory and put the sources into the SOURCES directory.

Building a RPM file can be done by using the following command:

```
$ rpm -ba specfile.spec
```

The successfull installation can be tested by running bandwidth-retriever "IP address of destination host".

Chapter 5

Interface Description

5.1 Interface

The new and lightweight interface for performance measuring according to PTP contains a very slim interface to retrieve bandwidth measuring data. It contains the class BandwidthRetriever that encapsulates the whole implementation. This class has two methods: *startBandwidthCalculation* and *registerAlgorithm*. After the *startBandwidthCalculation* call, the BandwidthRetriever retrieves bandwidth data and passes them on to the the registered objects. The arguments that are passed on to the method *startBandwidthCalculation* is a Boost shared pointer to a BandwidthReceiver interface.

The parameter for the method *registerAlgorithm* is a Boost shared pointer to an IntervalAlgorithm object. The object implementing BandwidthReceiverInterface is a callback that receives performance data through the call of *setBandwidth* and the object implementing BandwidthIntervalAlgorithm is responsible for the calculation of the time between two meassurements. To obtain the information, *getInterval* can be called with the current measurement information.

5.2 Implementation

The BandwidthRetriever is implemented as a Singleton Object that is responsible for handling connections to multiple hosts and for the administration of the threads that need to do the performance measurements.

Calls to startBandwidthCalculation starts a measurement thread or increases the reference count for the currently running thread. This ensures that multiple connections to the same host will not span of multiple measurement threads. The BandwidthIntervalAlgorithm is implemented as a pseudo Enumeration / Singleton Combination. There is an enumeration representing different algorithms. The enumeration keys are used to identify the different algorithms and retreive them through a factory. For that reason it is possible that the BandwidthRetriever does not need to know anything about the algorithm that was implemented on the one hand, and on the other hand it is possible for the BandwidthRetriever to perform the measurement in a thread and also respond to higher level protocols. Due to the combination with the enumeration an easy identification of the algorithm is possible, and also several connections using the same algorithm without creating duplicated objects.

The setBandwidth method can throw a ClientTerminatedException. If this is the case, the the reference count is decreased. If the reference count is zero, the Bandwidth Measurement is stopped.

5.3 Methods

5.3.1 startBandwidthCalculation

Method:	void startBandwidthCalculation(const boost::shared_ptr <BandwidthReceiver> & p_receiverInterface)
Returnvalue:	void
Parameter:	Boost shared pointer to receiverInterface object
Description:	This method starts the bandwidth calculation for a host that is specified in the BandwidthReceiver object. The algorithm used for the interval detection is specified by the registerAlgorithm method mentioned below.

5.3.2 registerAlgorithm

Method:	static void registerAlgorithm(const std::string & p_algorithmId, const boost::shared_ptr <BandwidthIntervalAlgorithm> & p_algorithm)
Returnvalue:	void
Parameter 1:	string containing the algorithmId
Parameter 2:	Boost shared pointer to BandwidthIntervalAlgorithm object
Description:	This method registers / specifies the algorithm used to detect the interval between bandwidth retrievals.

5.4 Examples

The following example shows how easy bandwidth can be retrieved. The only thing to do is to implement a BandwidthReceiver and to start the bandwidth calculation. For the BandwidthReceiver, the following methods have to be implemented:

- getHost - returns the IP address of the host for which the bandwidth has to be retrieved.
- getAlgorithmIdentifier - returns the ID of the algorithm that is intended to retrieve the interval between the bandwidth requests.
- setBandwidth - this method is responsible for handling bandwidth data. If a ClientTerminatedException is thrown, no more bandwidth data is requested.
- routeChanged - this method allows to react on changes along the route. If a ClientTerminatedException is thrown, no more bandwidth data is requested.
- exceptionOccured - this method is an asynchronous exception handler for exceptions occurring on the daemon side. If a ClientTerminatedException is thrown, no more bandwidth data is requested.

After having implemented these methods, the bandwidth calculation can be started by calling SocketBandwidthRetriever::start (in this case within a main method).

Implementation of the main method:

```
1  #include <iostream>
2
3  #include "TestReceiver.h"
4  #include "SocketBandwidthRetriever.h"
5  #include "PTPConfiguration.h"
6  #include "UnixDomainSocket.h"
7
8  int main(int argc,char** argv)
9  {
10   boost::shared_ptr<BandwidthReceiver> receiver(new
       TestReceiver(argv[1],"ptpAlgorithm"));
11
12   SocketBandwidthRetriever::start(receiver);
13
14   while(true) {
15     ZThread::Thread::sleep(2000);
16   }
17
18   return 0;
19 }
```

Implementation of the BandwidthReceiver:

```
1  #include <iostream>
2
3  #include "TestReceiver.h"
4
5  TestReceiver::TestReceiver(const std::string & p_host,
     const std::string & p_algorithmIdentifier)
6    : m_host(p_host), m_algorithmIdentifier(
       p_algorithmIdentifier)
7  {
8  }
9
10 std::string TestReceiver::getHost() const
11 {
12   return m_host;
13 }
```

```
14 std::string TestReceiver::getAlgorithmIdentifier() const
15 {
16   return m_algorithmIdentifier;
17 }
18
19 void  TestReceiver::setBandwidth(boost::uint64_t ↩
     p_bandwidth) const throw (ClientTerminatedException)
20 {
21   std::cout << "bandwidth :: " << p_bandwidth << " (bits/↩
       s)" << std::endl;
22 }
23
24 void TestReceiver::routeChanged() const throw (↩
     ClientTerminatedException)
25 {
26   std::cout << "route Changed" << std::endl;
27 }
28
29 void TestReceiver::exceptionOccured(const PTPException & ↩
     exception) const throw (ClientTerminatedException)
30 {
31   std::cout << exception.summary() << std::endl;
32 }
33
34 TestReceiver::~TestReceiver()
35 {
36 }
```

5.5 Class Diagram

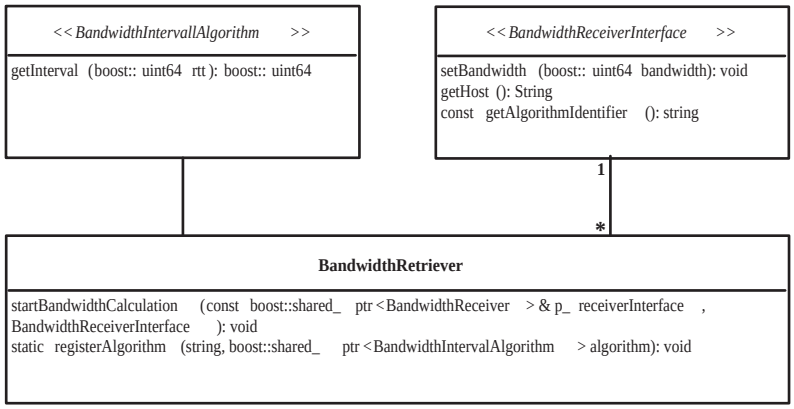

Figure 5.1: Interface Class Diagram

List of Figures

Bibliography

[1] Michael Welzl: *Scalable Performance Signalling and Congestion Avoidance*
Kluwer Academic Publishers, Dordrecht 2003, ISBN 1-4020-7570-7

[2] Bjarne Stroustrup: *Die C++ Programmiersprache*
Addison-Wesley, Boston 2000, ISBN 3-8273-1660-X

[3] *Net-SNMP Application Suite:*
`http://net-snmp.sourceforge.net/`
visited 2005-12-19

[4] *Boost C++ Libraries:*
`http://www.boost.org`
visited 2005-12-19

[5] *SCons - build your software, better:*
`http://www.scons.org`
visited 2005-12-19

[6] *ZThreads - A platform-independent, multithreading and synchronization library for C++:*
`http://zthread.sourceforge.net/`
visited 2005-12-19

[7] *Zotteljedis Tipps zur Programmierung mit Raw Sockets:*
`http://www.zotteljedi.de/doc/raw-socket-tipps.html`
visited 2005-12-18

[8] *Unix Domain Sockets in Linux*
`http://www.netzmafia.de/skripten/server/ThomasSocket2.pdf`
visited 2006-02-15

[9] *Setting up SNMPv3 Users*
`http://www.netadmintools.com/art485.html`
visited 2006-01-07

[10] *PTP Download Website on Sourceforge.net*
`http://www.sourceforge.net/ptpd`

[11] *RPM Package Manager*
`http://www.rpm.org`
visited 2006-02-07

[12] Karsten Guenther: *LaTeX - Das umfassende Handbuch.*
Galileo Press GmbH, Bonn 2004, ISBN 3-89842-510-X