Master thesis in teleinformatics at

the Royal Institute of Technology, Stockholm.

DTM Traffic Analyzer

Martin Christiansson

June 30, 1999

Examiner: Associate Professor Björn Lisper, lisper@it.kth.se
Advisor: Jan Renström, jan.renstrom@netinsight.se

Abstract

Dynamic synchronous Transfer Mode (DTM) is a new technology and the only knowledge of its performance is based on results from simulations. There is no guarantee that these simulations are correct so therefore we need some tool that can collect information about network activities and present it. The problem is that no tools exist for this type of network.

This projects goal has been to create a software based measuring tool for DTM networks. The measuring tool has been focused on measuring fiber utilization and presenting allocated channels since DTM is a circuit switched network.

The report covers the basic principles of DTM and a description of the measuring tools implementation and usage. Resources in the form of processing capacity and memory is a limiting factor in some of the platforms in the DTM network. A solution to that problem is also presented in the report.

Contents

1 DTM Fundamentals **6**

1.1 Overview of current network technologies . . . 6

1.1.1 Demands on future networks . . . 6

1.1.2 Optical fiber capacity . . . 6

1.1.3 Technologies used in high capacity networks . . . 6

1.1.4 What's new in DTM? . . . 7

1.2 DTM overview . . . 7

1.2.1 Introduction . . . 7

1.2.2 Frames and slots . . . 8

1.2.3 Channels and switching . . . 8

1.2.4 Multicast . . . 9

1.2.5 Synchronization . . . 9

1.2.6 Channel priorities . . . 10

2 Problem statement **11**

2.1 Background . . . 11

2.2 Project description . . . 11

3 Earlier work on performance measurement **12**

3.1 Performance tests and traffic monitoring . . . 12

3.2 Measuring on the DTM layers . . . 12

3.2.1 Physical layer . . . 12

3.2.2 DTM channels and control messages . . . 12

3.2.3 Higher layers . . . 13

3.3 Summary . . . 13

4 Net Insight's Implementation of DTM **14**

4.1 Introduction . . . 14

4.2 Platforms . . . 14

4.2.1 Introduction . . . 14

4.2.2 Access devices . . . 14

4.2.3 Switches . . . 14

4.3 DTM channels in greater detail . . . 14

4.3.1 Control channel . . . 14

4.3.2 Data channels . . . 15

4.4 DTM addressing . . . 15

4.5 Management . . . 16

4.6 Operating systems . . . 16

4.6.1 pSOS+ . . . 16

4.6.2 NetBSD . . . 17

5 Implementation overview **18**

5.1 Introduction . . . 18

5.2 Differences from original project description . . . 18

5.2.1 Control channel measurement . . . 18

5.2.2 Data channel measurement . . . 18

5.3 Problems encountered . . . 18

5.4 Implementation network view . . . 19

5.4.1 Event sending . . . 19
5.4.2 Controlling the measurements and retrieving results . . . 20
5.5 Implementation program structure . . . 20
5.5.1 Event sending . . . 20
5.5.2 Controlling the measurements and retrieving results . . . 21

6 Event filter 23
6.1 Introduction . . . 23
6.2 Data structures . . . 23
6.2.1 Top level filter table . . . 23
6.2.2 Event filter struct . . . 24
6.2.3 Measurement . . . 26
6.2.4 Channel information . . . 30
6.3 Functions manipulating filter structures . . . 32
6.3.1 Introduction . . . 32
6.3.2 Functions operating on the filter table . . . 32
6.3.3 Functions operating on the measure table in a filter . . . 33
6.3.4 Functions handling the open queue in a filter . . . 34
6.3.5 Functions handling the closed queue in a filter . . . 35
6.3.6 Functions operating on a measure open queue . . . 35
6.3.7 Functions operating on a measure closed queue . . . 36

7 Sending channel events 37
7.1 Ideas behind choice of event format . . . 37
7.2 Parser process . . . 37
7.3 Event generating process . . . 38
7.4 Event format . . . 38
7.4.1 Conventions used for event format paragraph . . . 38
7.4.2 Creating a filter . . . 38
7.4.3 Opening a channel . . . 39
7.4.4 Closing a channel . . . 39

8 Communication Abstraction 40
8.1 Server side . . . 40
8.1.1 Overview . . . 40
8.1.2 O&M interface . . . 40
8.1.3 Filter creation and removal . . . 42
8.1.4 Event receiving . . . 42
8.1.5 Finding the filter for a node . . . 43
8.1.6 Starting and ending a measurement . . . 43
8.1.7 Retrieving measure result . . . 45
8.2 Client side . . . 46
8.2.1 Overview . . . 46
8.2.2 Communication manager class overview . . . 46
8.2.3 Opening and closing connections . . . 47
8.2.4 Sending a request over the network . . . 48
8.2.5 Data objects . . . 49

9 Client application **52**
9.1 Introduction 52
9.2 Main window 52
9.3 Starting a measure 53
9.4 Viewing measure results 56

10 Summary and future work **58**
10.1 Summary 58
10.2 Future work 59

1 DTM Fundamentals

1.1 Overview of current network technologies

1.1.1 Demands on future networks

If we look back about five years then we will see that most people had not even heard about the Internet. Nowadays it is becoming a natural part of the everyday life. The development will not stop at just "surfing on the net". New demands on services will arise as people get used to having the services that the net can provide today. They will then start to see new possibilities. Video conferencing, data traffic, audio and other types of interactive communication set new demands on real-time support in the networks. The bandwidth demand will also increase dramatically. There will therefore be a need for a scalable and cost efficient network technology.

1.1.2 Optical fiber capacity

The transmission capacity in fibers is increasing rapidly. Most transmission technologies have been designed to maximize the utilization of the transmission capacity of the communication link. Considering the fact that fiber capacity is increasing at a much higher rate than processing speeds. Earlier approaches to focus on the link utilization has created a bottleneck in the nodes. The node has become the main obstacle in increasing the transmission capacity in the networks. A new approach is needed to increase the fiber utilization.

1.1.3 Technologies used in high capacity networks

Circuit switching Network resources are set up and reserved all the way between the sender and receiver before the transmission starts in a traditional circuit switched network. Ordinary telephone networks are a good example of this type of switching. There is no need for any overhead when sending data once the link has been set up. Routing and other processing is only performed when a link is set up or terminated. This gives a number of advantages. We can guarantee a quality of service. This might be a guaranteed bandwidth and a low jitter (constant delay). The main disadvantage is long set up delays. The efficiency when sending small messages and traffic with frequently changing data rate might become very low. It is not well adapted to the bursty computer traffic. Other nodes in the network are also locked out from the used resources when all capacity is allocated even if the capacity is not fully utilized.

Packet switching There is no set up and termination phase when communicating over a packet switched network. The data stream is split up into packets. Each packet contains the information necessary for reaching its destination and to make reassembling of the stream possible. The utilization of the network will be higher than the circuit switched network when we are dealing with bursty computer traffic. The opposite will be true when sending video or other information that has the characteristics of being a steady stream of constant or slow varying data rate.

The path of the packet in the network is not decided at the beginning of the connection as in the case of circuit switched networks. Each router decides were to send the individual packet and the different packets can travel through different paths. The routers will occasionally run out of buffers and some packets will be thrown away. This will lead to resending of packets by higher level protocols. Each packet might take different paths and be stored for varying amounts of time in the routers. These varying delays and resendings of packets will lead to a jitter that will make it unsuitable for some applications. Increasing the speed of these routers will also lead to increasing buffer sizes, thought making this technology less scalable.

It should also be noted that there are two types of packet networks, connection oriented like for instance ATM and connectionless like Ethernet. ATM sets up a logical channel before sending data, but it still uses a statistical multiplexing so there is no real guarantees like in circuit switched networks. Another problem is the fragmentation and overhead in ATM networks. They are not especially well suited for computer traffic like the IP protocol. Improvements of ATM and Ethernet have been introduced to overcome some of its limitations but this adds both overhead in processing of the packets and more complicated management functions.

1.1.4 What's new in DTM?

DTM is basically a circuit switched network that has been designed to overcome the main disadvantage with circuit switching, namely the long set up and termination delays. DTM stands for Dynamic synchronous Transfer Mode. This enables us to use a circuit switching technology to mix steady streams like video on the same network as bursty computer traffic. The possibility of being able to rapidly adapt the bandwidth to variations in data traffic gives us real-time performance and quality of service usually found in circuit switched networks and the flexibility of data rate found in packet switched networks.

Another important aspect of the DTM technology is the switching technology. The general idea is that switching is done in hardware through mapping of one timeslot to another on different ports in the switch. Individual slots do not have to be processed in any way. Their contents are simply copied and this makes it less complicated to implement the switch in hardware. A more detailed description of this will follow in the next section. There is no need for large buffers since it is a circuit switched synchronous network. This keeps the processing in the nodes at a minimum level.

1.2 DTM overview

1.2.1 Introduction

The information in this section is not specific to Net Insight's implementation of DTM. It should be considered as basic knowledge about the basic functionality and principles used in a DTM network [1] [2]. The DTM communication equipment (hardware and software) is divided into something abstract called layers. The lowest layer is the physical medium and the highest is the application using the network. The handling of channels and control signaling is found in the middle layers. The main focus in this project has been the middle layers. Most of the information in this section describes these layers, but some information about the lower layers is included to explain the basic principles of a DTM network.

1.2.2 Frames and slots

DTM is a time division multiplexed network. The smallest unit is a 64-bit time slot. A number of these slots form a frame. The time passed while sending a frame is 125 microseconds (which corresponds to a frame frequency of 8 kHz). When sending at 2 Gbps we get about 3900 slots in one frame. The bandwidth is dependent on the number of slots per frame.

There are two types of slots, namely data and control. There is no restriction on which type a certain slot can be. It is also possible to convert a data slot into a control slot and vice versa. A simple example of this is found in figure 1.

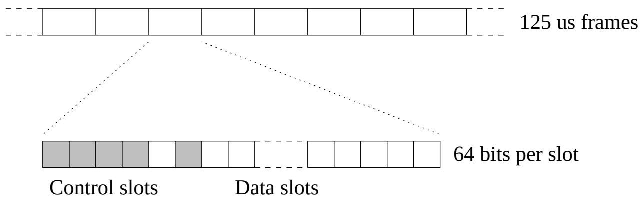

Figure 1: Frames and slots

1.2.3 Channels and switching

Each node gets write access to at least one control slot for each link it is connected to. The allocated slots for sending control data and the slots for receiving control data from other nodes will from now on be referred to as the control channel. Control channel messages are only sent between nodes on the same link. These messages are used to send management information like for instance set up and terminate data channels.

Each node has a pool of free data slots that will be used when setting up a channel. The possibility of asking another node for additional slots exists if the node has too few slots in its own pool. The minimum bandwidth that can be allocated in one channel is 512 kbps (64 bits @ 8 kHz). The capacity is then incremented by 512 kbps for each additional allocated slot.

The number of slots in a data channel that is being switched between two ports will be equal on both ports. The switching of a data channel is done in hardware. It is based on mapping of slotnumbers on the receiving port to the slotnumbers on the sending port. An example of how channel switching works in figure 2 is described in the following list.

1. A allocates slot number 4, 6 and 11 on its own link.
2. A sends an announce message (using the control channel) to S. The message instructs S to set up a channel to B using the slotnumbers 4, 6 and 11 on the link between A and S.
3. S allocates the same number of slots on the link that B is attached to. In this case S allocated slot number 2, 3 and 4.
4. S sends an announce message to B instructing the node to listen on a channel from A using slot 2, 3 and 4.
5. B starts to listen on the slots.

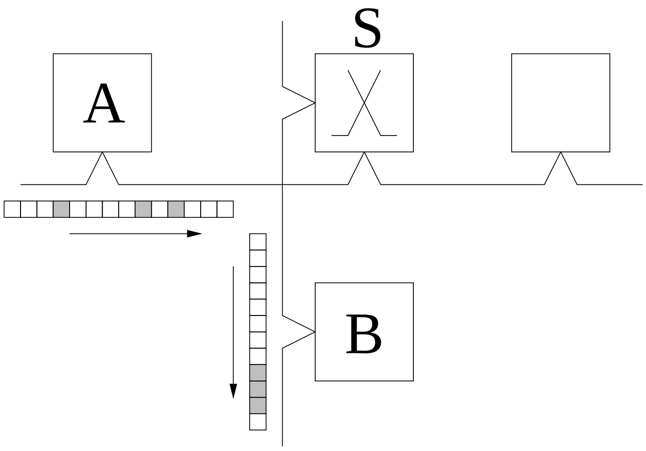

Figure 2: Channels and switching example

6. B send an acknowledge message to S using the control channel.
7. S sends acknowledge back to A after that the acknowledgement message from B is received by S.
8. Transmission on the data channel is started when A receives the acknowledge message.

1.2.4 Multicast

Multicast is used when a node is sending the same message to more then one receiver. The simplest but not the most effective way of doing this is to send several copies of the same message on separate channels to each of the receivers. This is not a good idea if there is a lot of nodes receiving.

Multicast is simply a special case of the data channel. DTM is suitable for the task, since all nodes can instruct the hardware to listen on any slot number. A switch can also set up a channel that is received on one port to transmit on several of the other ports. The set up of a channel will differ from the unicast channel. There will not be any difference after the channel has been set up as far as the switching hardware is concerned.

1.2.5 Synchronization

Synchronization of the DTM network is used in order to minimize delays and reduce the need for buffers in the switches. We only need to store the slots in a buffer during one frame if the ports are synchronized, but it might have to be stored for up to two frames if the synchronization is lost (buffer need is doubled). All nodes use the same 8 kHz frame frequency. One of the nodes of each network link will transmit frame start time slots at the defined frame frequency. We also need to prevent frame drift on different links in the network. Each node chooses one port as the reference frame clock and adjusts the other ports according to the frame clock on this port. The network can then be seen as a tree structure with a master clock at the highest level.

1.2.6 Channel priorities

The circuit switching technology gives us the possibility to guarantee a quality of service. One suggested method (not necessarily the one used in the final products) divides the bandwidth in the DTM network into three priority level.

1. Guaranteed bandwidth
   The channel is set up width a fixed bandwidth. The number of allocated slots will not change until the channel is terminated, regardless of request from other channels.

2. On demand bandwidth
   This gives the channel a guaranteed minimum number of slots and the possibility to allocate more slots given that resources are available.

3. On demand traffic with best effort
   Similar to level 2 but there is some differences. There is no guarantee on a minimum bandwidth and the traffic can be interrupted.

2 Problem statement

2.1 Background

DTM is a new technology and therefore we have no real knowledge about the load the network can cope with. Simulations do exist [2] but measurements on real networks have not been performed. There is no simple way to see how much of an allocated channel, or the control channel, that is actually being used.

2.2 Project description

A DTM network analyzer should be able to measure the load in terms of data and control channel usage. The project will result in a performance monitoring tool for DTM. This tool should consist of two parts. One is included as a part of the software in an access node or a switch, and the other as a client application collecting information from several nodes and switches. The client will present a comprehensive view of the network and form a base for decisions on what to measure. The tool shall be able to measure statistics from the control messages sent and received by the node, e.g. no of control messages to/from one node, utilization of the fiber capacity, utilization of channels, etc. It must be possible to select which measurements to perform and how often to present them. The results will be presented by the client application.

3 Earlier work on performance measurement

3.1 Performance tests and traffic monitoring

There is a difference between testing a network and monitoring it. RFC (Requests for Comments) documents number 1242 [11], 1944 [12] and 2285 [13] contains suggestions concerning how to test a network. The test described in the documents assume that the network isn't used when the test is performed. The performance monitoring tool should only listen and record information about real traffic. This falls into the category of traffic monitoring and not into explicit network testing.

3.2 Measuring on the DTM layers

3.2.1 Physical layer

It is difficult to measure on the physical layer using a software tool. Measuring on the physical layer should preferably be done by some dedicated hardware solution. The monitoring tool will ignore the physical layer and the lowest hardware abstraction layers.

3.2.2 DTM channels and control messages

This is what the monitoring tool will be measuring on. The method to collect this information is specific to DTM and I've not been able to find any existing tool that is suitable to collect data from DTM networks.

Some similarities in how traffic monitoring data is collected by the tcpdump monitoring program (found in most Unix-like operating systems) can be found. The tcpdump filters IP protocol packets using BSD packet filters [15]. A filtering device is opened and associated with a network interface. The incoming packets can for instance be filtered so that only packets sent to a specific destination is kept. Unwanted packets are discarded as soon as possible to save resources. The monitoring tool will also filter information like the packet filter but it will filter information about channel setup and termination and not packets being received.

There is one big difference between BSD packet filters and the monitoring tool. The monitoring tool operates over a network and the packet filters only operates on the local node. It must be possible to set the filtering parameters over the network so that only the relevant information is transfered to the program that is presenting the result.

3.2.3 Higher layers

The DTM channels can transport other types of network traffic. It is for instance possible to connect a number of ethernet networks and make it all look like one network. This functionality is called a bridge. The ethernet packets in a bridge is transported over DTM channels. Several other type of networking protocols can be transported over the DTM network by transporting the protocol inside DTM channels. The monitoring tool will not cover these protocols. It is only targeted towards DTM channels and control signalling.

The CAIDA Measurement Tool Taxonomy [14] is a good source for finding programs for measuring on the efficiency of higher level protocols.

3.3 Summary

It has been difficult to find any useful information about measurements on networks that could be useful when measuring on DTM networks. I guess this is the drawback from creating a completely new protocol.

4 Net Insight's Implementation of DTM

4.1 Introduction

This section will describe some parts of Net Insight's implementation of DTM. I will only deal with areas that are relevant for the measurement application. Some information that the application will need already exists and is reachable through management functions. One example of this is static information about available ports and addresses in a node. More information on management functions is found in section 4.5.

4.2 Platforms

4.2.1 Introduction

The platforms that I'm describing in this section are being used in test installations and in the first generation of products. Later product generations under development will not be mentioned since the monitoring tool isn't targeted towards them.

4.2.2 Access devices

The access device is based on PC technology and equipped with a DTM interface card for connecting to a DTM network and one or more interface cards for other types of communication technologies. Current test installations use access devices with fast ethernet cards and interfaces to bridge traffic between telephone switches (E1/T1).

4.2.3 Switches

The core of the DTM switch is the Internal Switch Module (ISM) which also acts as a motherboard for the system. The ISM has eight slots for interfaces and the controller card (called node controller). The switch only uses DTM interfaces at this time. Interfaces for fast ethernet, bridging of telephone switches and uncompressed video are under development. The node controller consists of a StrongArm CPU and other components needed like for instance RAM memory and flash ROM for booting the system. One of the controllers tasks is to handle setup and termination of channels. The ISM is switching the channels but it only consists of FPGA logic circuitry and can not handle anything more advanced.

4.3 DTM channels in greater detail

4.3.1 Control channel

The control channel is basically a data link bound broadcast channel. The information sent on a control slot will not be switched to another data link. The node hardware is instructed to receive all slots in a frame that corresponds to the other hosts transmit control slots. We want to be able to filter on receiver and sender addresses so that a measurement can be limited to a smaller part of the network. Each node need to keep track of the number of control slots sent to each node on all of the links it is attached to. This information must be accessible through the management interface (management functions are described in section 4.5).

4.3.2 Data channels

When setting up a data channel, one of the hardware ports is instructed to receive on a number of slots given on the form of a linked list with slotnumbers. Another port is instructed to send on another list of slotnumbers. The two ports are then linked by the switching hardware. Access devices handle data channels in a similar way even if they only have one DTM port. The DTM interface card is seen as a one port and the processor as the other. This keeps the differences between the different platforms at a minimum.

The switching is done in hardware and the driver software has no knowledge if a certain slot allocated to a channel is used or not. It is not possible to let the switch count the number of messages passing. That would go against the basic idea of keeping the work needed in the switches at a minimum. The solution will be to let the sending access node count the number of slots sent on each of the channels and store it in the management database (management functions are described in section 4.5).

4.4 DTM addressing

DTM uses 64 bits long addresses. The address is written as pair of hexadecimal digits and leading zeroes should be printed out to make reading simpler. Address prefix is specified as a decimal number representing the length of the fixed part of the address in bits. All possible combinations of the last bits gives us a range of addresses. The result is written on the form <DTM prefix address>-<Prefix number of bits>. Some examples follow.

- Single address:
  01.AA.34.1A.00.32.1B.01

- 256 addresses:
  01.AA.34.1A.00.32.1B.00-56
  01.AA.34.1A.00.32.1B.00 to 01.AA.34.1A.00.32.1B.FF

- 4096 addresses:
  01.AA.34.1A.00.32.00.00-52
  01.AA.34.1A.00.32.10.00 to 01.AA.34.1A.00.32.1F.FF

There are also some addresses ranges that have special purposes.

- Multicast addresses:
  FF.00.00.00.00.00.00.00-8

- Anycast addresses:
  FE.00.00.00.00.00.00.00-8

- Local addresses:
  00.00.00.00.00.00.00.00-16

There is also support for per channel multiplexing below the level of node addresses. DST (DTM Service Type) and DSTI (DTM Service Type Instance) are used for this purpose. This is quite similar to port numbers in the IP protocol.

DST describes what type of traffic that is going to be sent over the channel, and this should be a well-known number for each type of traffic. The DSTI number is used to allow several connections to the same DST in a node. The combination DST/DSTI must be unique in the two nodes that are setting up a channel.

4.5 Management

The management functions in the nodes and switches are found in a module called O&M (operation and maintenance). These functions are used for configuration, fault handling (alarms) and security management. Accounting (billing) and performance measurement will also use this management interface.

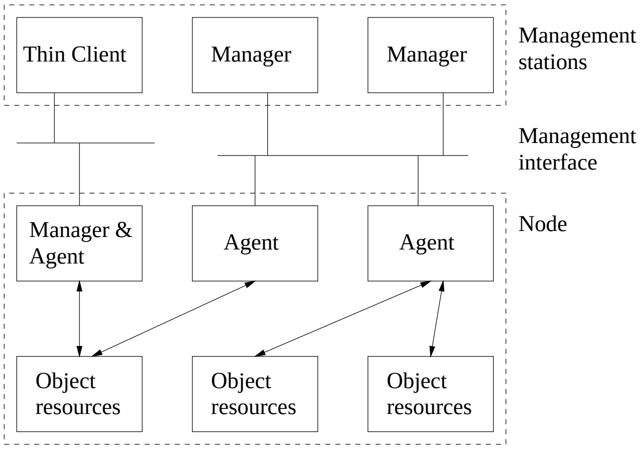

Figure 3: O&M interface

Management interfaces in figure 3 could be one of:

- telnet / rlogin (terminal session, thin client)
- HTTP (www browser, also thin client)
- RPC (Sun remote procedure call)
- SNMP (Simple Network Management Protocol)

Object resources could for instance be routing tables, topology information, information about the node itself or the result from a performance measurement.

4.6 Operating systems

4.6.1 pSOS+

This is the real time operating system used in the switches. The scheduler is priority based and supports time-based and preemptive scheduling. Tasks scheduling policy can be changed at runtime. pSOS+ is a small kernel which takes up between 17 and 40 kb.

The kernel also handles semaphores, events, timers, queues and asynchronous signals. Several subcomponents exist. One example is pRPC+ which contains what is necessary to run remote procedure call (compatible with Sun Microsystems version) [4].

4.6.2 NetBSD

NetBSD is a UNIX-like operating system that is used in the access nodes. It is released under a Berkeley-style license and it is therefore possible to redistribute it quite freely as long as some criteria are meet (for instance naming the authors).

This system is a derivative of the UCB 4.3BSD via the Networking/2 release and 386BSD. Some things have also been taken from 4.4BSD Lite. NetBSD is ported to fifteen different platforms in the formal releases and there are also experimental ports for seven other platforms. The portability of the system is one of the primary focuses of the NetBSD developers [5].

5 Implementation overview

5.1 Introduction

Some parts of the original project description has been changed for various reasons. Section 5.2 describes the how the current implementations project description differs from the original. All the reasons for not following the project description will be presented in the same section. Problems occurred during implementation following the new project description is described in section 5.3.

5.2 Differences from original project description

5.2.1 Control channel measurement

Most of the original description about what to measure on control channels has been changed. Measuring on control channel usage and number of messages to or from a certain node would require quite a lot of modifications to the existing code that handles control signaling. Limited time to finish this project has been one of the reasons for skipping some parts of these measurements. Measuring on data channels is of greater interest and it has been prioritized. Stability and performance of the already existing code that would have to be modified is another reason. Collecting statistical information in the control signaling code might be of reasonable complexity, but designing some mechanism to select which parameters to measure on and to transfer the collected information without disturbing the more important functions is not that easy. The current implementation only presents the static information found in the O&M (operation and maintenance) interface.

5.2.2 Data channel measurement

The hardware lacks any possibility to measure the usage of a channel. My first idea was to instruct the nodes to count the number of slots sent on a channel. It would then be possible to collect these values from the nodes that is the transmitting endpoint on a data channel. This would make the PCI card driver in the access node more complicated and this could create a bottleneck in the access devices. Multiplexing and demultiplexing (mapping of slots to channels) that is performed in this driver is very CPU dependant. I've chosen to skip measuring on channel usage because of this. High throughput is more important than information about channel usage. The current implementation measures on the fiber utilization where the information consists of channel setups and terminations. Important information from the O&M interface is also presented.

5.3 Problems encountered

- Communication method
  Choosing method for communication between the client application and the network nodes running NetBSD and pSOS leaves you with very few alternatives. NetBSD and Linux (that is used on the client side) can use ordinary IP sockets, RPC, SNMP and CORBA. The problem is the lack of support for most of these in pSOS which only handles IP socket and

RPC. I've chosen to use RPC in most cases and TCP/IP sockets for event sending (messages sent when setting up or terminating a channel). All communication is hidden in an abstract layer and it is possible to change the method of communicating in this layer without changing the rest of the application.

- Lack of resources
  Memory is a resource that the switches could run out of. The earlier versions has very little memory and the never might also run out of memory if a lot of information about channel events is stored. The solution for this has been to send events to some other node that can run a measure process for the switch.

- Lack of test equipment
  One of the worst problems has been the lack of test equipment. I've chosen to use a process running on a Linux machine that is generating events to simulate a network node. It is a bit unfortunate but it has been enough to test the basic functionality of the measure application.

- Choosing programming language
  The measure client application is running under Linux. It was possible to use C++ for this part of the project. This application could use a lot of resources and it is not time critical. Using object oriented libraries for user interface and wrapper for system calls has made the code simpler to read. Other platforms where more limited. All previously written code is in ordinary C, which seems quite suitable for real time programs when you want total control of memory usage and the program flow. That was the reason for writing everything running on a network switch or access node in C.

5.4 Implementation network view

5.4.1 Event sending

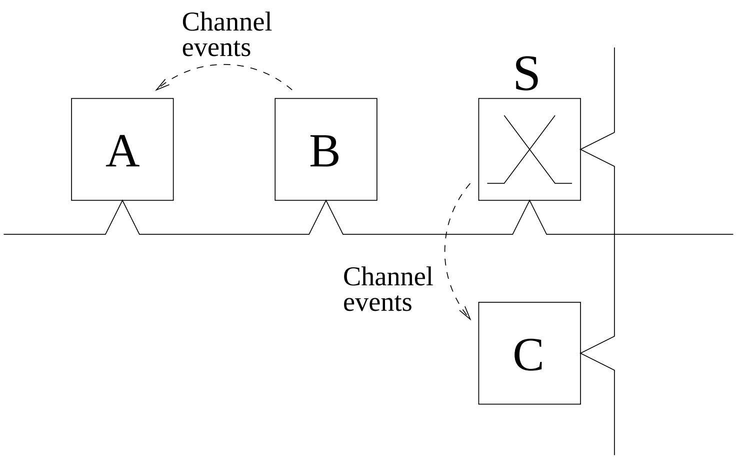

Figure 4: Network view of event sending

The event filter is a process that filters out unwanted information as soon as possible to avoid wasting bandwidth on the network. Figure 4 shows a simple DTM network. Node B and the switch S lacks the resources to filter events and run measurements. Node A runs event filtering and measuring for itself and node B. Node C does the same for itself and the switch. The implementation in this project is sending these events as text strings over a TCP/IP channel.

5.4.2 Controlling the measurements and retrieving results

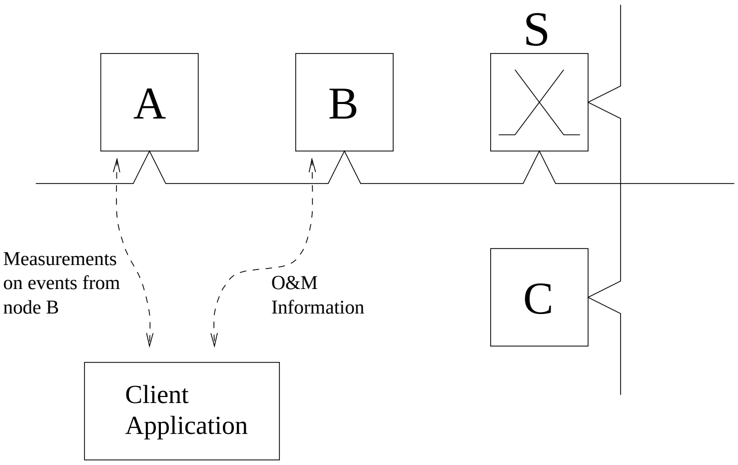

Figure 5: Measuring principle

The client application in figure 5 is measuring on node B. Information from the O&M interface is retrieved directly from the node. Current implementation uses RPC for this purpose, but all communication goes through the abstraction layer. The information is grouped into C++ objects.

Node A is keeping the event filter for node B. The client always sends a request to the node that it is going to measure on asking for the node holding the event filter. The most common respons for a access device will probably be the local node but the switches will probably send the data to some other node.

Measurements are started by specifying what kind of channel information you are interested in. Filtering of events can be done by source and destination address, DST (destination service type) and port number. Several conditions can be used to form a boolean expression. Results from the measurement has to be retrieved periodically within a timeout period specified when the measurement started. There is also a timeout on removing the entire measurement to avoid leaking memory in the filtering process if a client goes down.

5.5 Implementation program structure

5.5.1 Event sending

An event being sent to a filter follows the path in figure 6. The list below describes the dataflow.

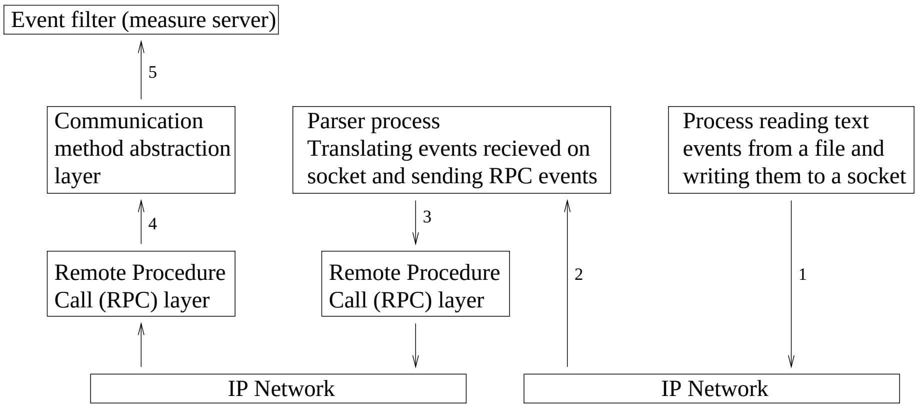

Figure 6: Event sending program path

1. Sending event on socket
   The implementation for this project includes a separate process that reads events in text form from a file. This is only a testbench and the final version should get this information from that channel handling protocol. The file also specifies a delay in seconds between each event. The events are written to a TCP/IP socket.

2. Receiving of text event and parsing
   A parser process is receiving the event. This process is running on the same hosts as the filter. The parser process calls fork() each time a event sending node connects to it so that several nodes can send events to the filter. Abstraction of communication method over the network has not been used in this process. The code is limited in size so there is limited benefit from hiding the RPC mechanism.

3. Sending RPC message
   A RPC call is sent to the filter after parsing the text event and converting the information to an RPC struct.

4. Receiving the RPC message
   The abstraction layer converts the call to one of the filters interface functions.

5. Accessing the filter
   Demultiplexing of events is performed so that it is received by the right filter. Remember that one filtering process might be filtering events from several network nodes.

Event sending will be further described in section 7.

5.5.2 Controlling the measurements and retrieving results

Figure 7 shows the main functional blocks in the measuring clients and the network nodes. Client side is found on the left and the network nodes on the

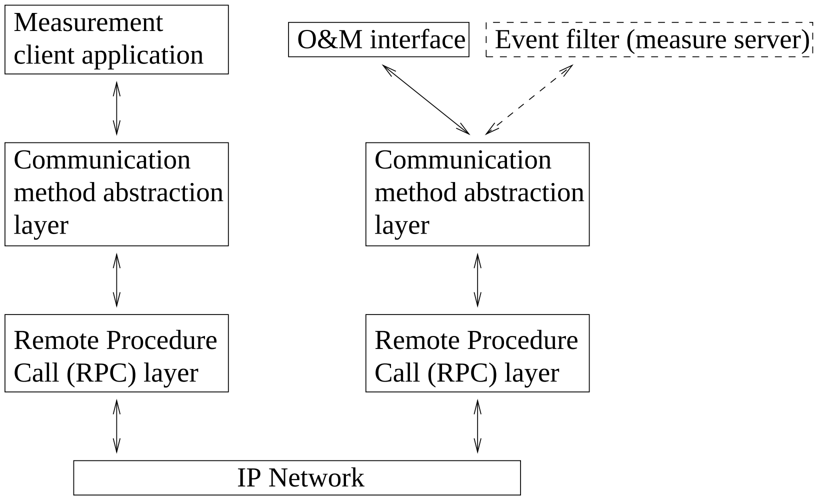

Figure 7: Measure client and network node program structure

right. Dashed lines have been used around the event filter on the network node side. This is because of the fact that some network nodes (most often switches) will have this code disabled and some nodes will be keeping filters for several nodes. The communication abstraction layer on the network node is demultiplexing incoming requests for the O&M interface and all the running filters.

6 Event filter

6.1 Introduction

This section will describe the event filters data structures and programming interface. The programming interface described here are for used for calls within the filter process. Communication methods and programming interface used when contacting the filter from other network nodes will be described in another section. Information in the section about data structures is the most important.

6.2 Data structures

6.2.1 Top level filter table

There should be one instance of the pm_filter_table struct in the filter process. The implementation that I have written is using a global pointer filter_table to find this structure. The pointer is initially set to NULL. Figure 8 shows this top level filter structure.

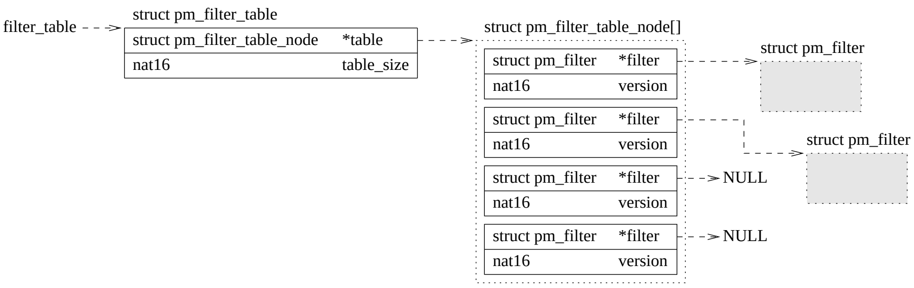

Figure 8: Top level filter table

A pm_filter_table struct is allocated when the first filter is started. The table is allocated with capacity for handling filtering of events from four nodes and the table_size is set to four. It should be enough in most cases. The capacity will be doubled each time we run out of positions in the table. Each pm_filter struct handles channel events from one DTM network switch or access node. A filter id and a filter version is used when a client application wants to use the event filter for a certain network node. The filter id is equal to the position it has in the table. The version number is used to avoid the possibility of clients retrieving data from the wrong node. Version numbers are set by the filter process. The following shows a sequence of calls that would fail without the version number.

1. Client “A” sends a request for filter id and version for the filter handling events from node “B”. The filter returns id and version.

2. Client “A” can start measurements and retrieve data from node “B”.

3. Node “B” chooses to disconnect from the filter process. The filter deallocates all data structures used by the filter, sets the filter pointer in the table to NULL and increases the version number.

4. Node “C” sends a request to the filter process asking it to handle events from the node. The filter process searches through the table and finds a pointer that is NULL. It might be the same position as node “B” used a little while ago.

5. Client “A” tries to start a new measurement from node “B”. It contacts the filter using the old version number. The filter will respond with an error message telling client “A” that the filter doesn’t exist. It would be possible to start a measurement on the wrong node if the version numbers didn’t exist.

It might seem simpler to use a simple linked list of the filters and a 32 bit number to identify them. There is a good reason to avoid this. The setup and removal of the filters is only done a few times but channel events can be received very frequently. The time needed to search a list increases if the number of filters running is increased. Using a filter id as offset in the table means that no searching is done when accessing the filters. The complexity doesn’t increase when running a large number of filters. A simple version number check is enough.

6.2.2 Event filter struct

Each event filter handles the events and measurements for one node on the network. I’ll split this section into three parts that will be describing the pm_filter struct and measurement table, queue holding information on open channels and the queue holding information on closed channels. You can find all data structures used by the filter in figure 12 on page 27.

- pm_filter struct and measurement table

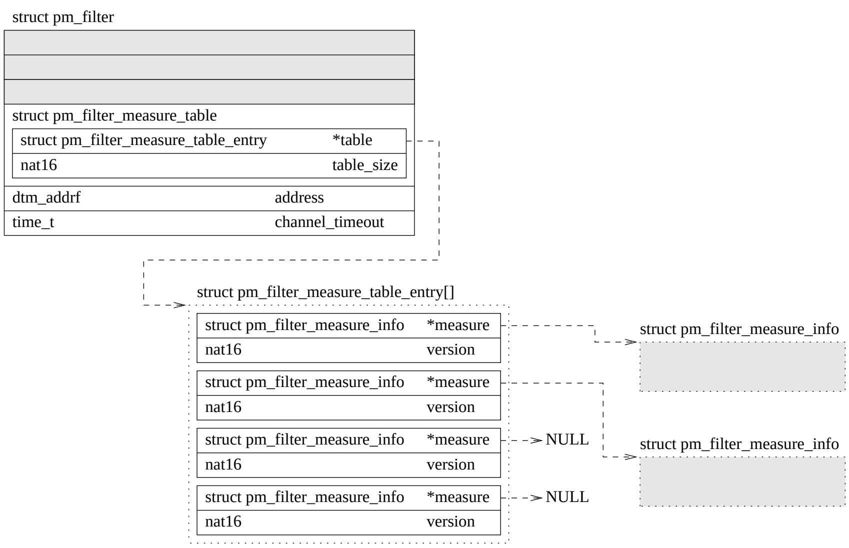

Figure 9: Measurement table in pm_filter struct

Shaded parts of the pm_filter struct in figure 9 will be described later. The address field holds the address of the node that sends events to the filter. The channel_timeout stores the number of seconds that information about a closed channel will be kept if nobody has subscribed to that type of events. Some similarities to the filter table in section 6.2.1 is found when looking at the table pointer. The table_size is set to four and a table of the same size is allocated when the filter structure is created. This tables capacity is doubled each time we want to start a new measurement and the number of active measurements is equal to the table size. Version numbers are used for reasons similar to those in the top level filter table. Complexity in searching for the right measurement doesn't increase when the number of measurements increase. The following shows a simple example that would fail if no version numbers were used.

1. A client initiates a measurement on a node.
2. The client retrieves measure results.
3. Each measurement keeps a timeout timer and the client fails to retrieve results before this time has passed. The measurement is removed and the version number is increased.
4. A new measurement is initiated by another client. It might just happen that the same measure id is used.
5. The first client tries to retrieve more results but gets an error message that informs the client that the measurement doesn't exist.

- Open channels queue

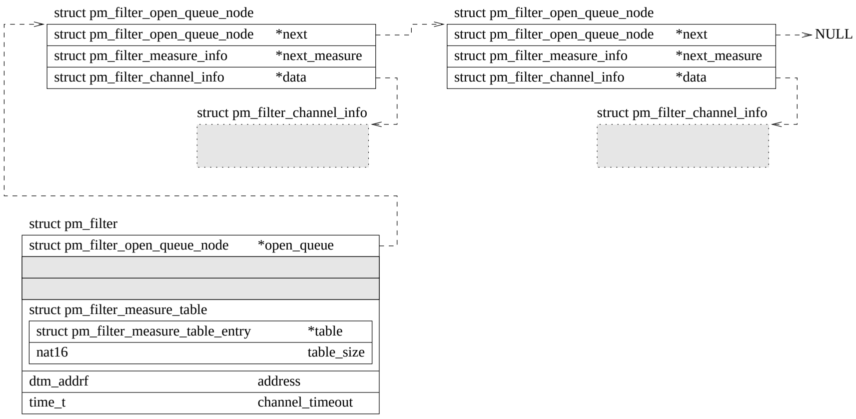

Figure 10: Open channel queue in pm_filter struct

Figure 10 shows the open channel queue. The pm_filter_channel_info struct will be described in section 6.2.4. Some knowledge about the struct is needed at this point. A unique key number is assigned to each open channel even that is sent from a node in the network when a channel is opened. This number is used when the close channel event is received to

match the events. It also has a reference counter that keeps track of how many nodes that is using the struct. The channel info struct is removed when the reference count reaches zero. All other field are information about the channel.
The open queue is a simple linked list and new nodes are allways inserted at the beginning. The next_measure pointer in the pm_filter_open_queue_node struct will be described further in section 6.2.3. That pointer is used to speed up removal of all references (data pointers to pm_channel_info struct) to the channel info struct pointed to by data field.

- Closed channels queue

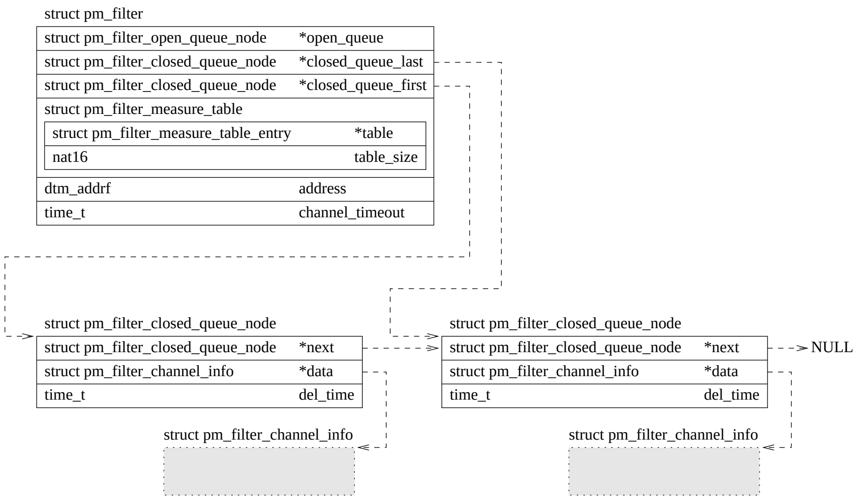

Figure 11: Closed channel queue in pm_filter struct

This queue (shown in figure 11) is also a simple linked list but a pointer to the last element is kept in the pm_filter_struct. Every channel that is closed is inserted as the last element in this list. The del_time field is set to the current time plus the channel_timeout period in the pm_filter struct. We will now get a list sorted on del_time. The smallest value is always found at the first element since newer elements are inserted at the end. Testing if timeout period has passed for the channel becomes a simple operation. Start at the first element and remove all nodes until one node with del_time set to a higher value than the current time is found. The channel_timeout is chosen when the measurement is started.

6.2.3 Measurement

This section will also be split up into three parts. The measure has a list of nodes that acts as a test function to establish if an event should be included in the measurement result queues. The result is held in a queue with open channel information and a queue with information on closed channels. Figure 18 on page 31 shows a complete view of all data structures in a measurement.

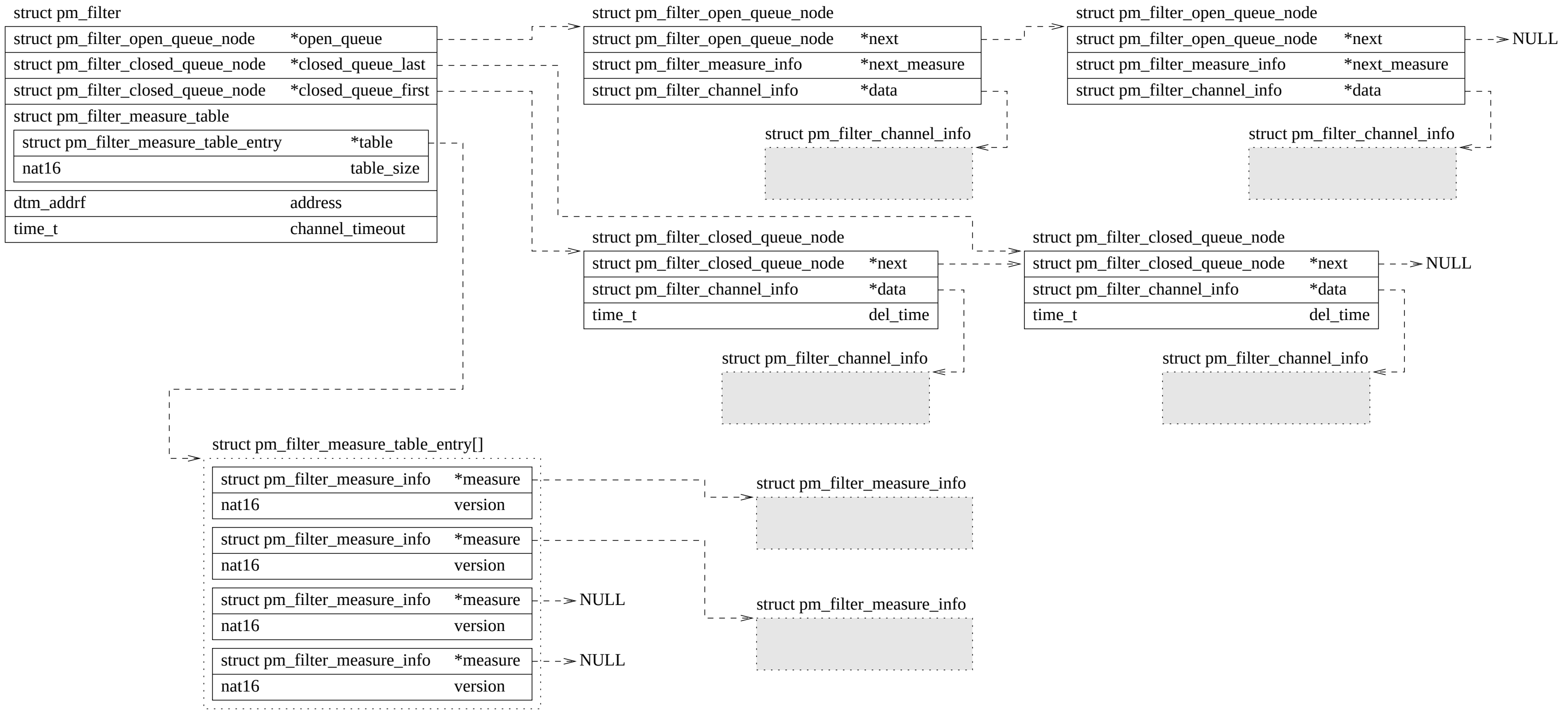

Figure 12: Data structures used by an event filter (section 6.2.2).

- Test function list

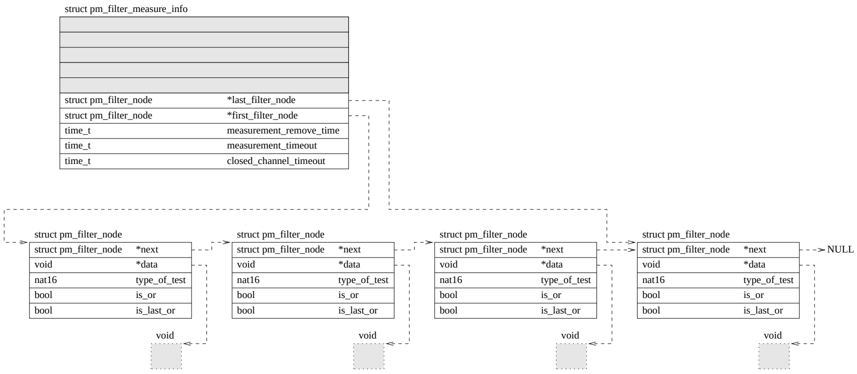

Figure 13: Test function list

Every incoming event to a filter is passed through every measurements test function list. This list is shown in figure 13. The channel is added to the measurements open channel queue if the test is successful. Figure 14 shows a closeup on the pm_filter_node structure.

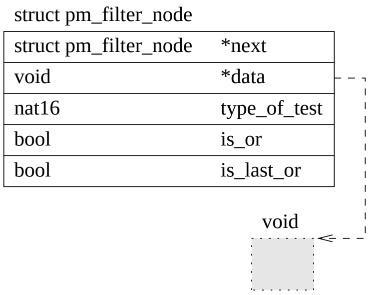

Figure 14: pm_filter_node struct

It's not obvious how this struct is used. The next pointer is simple and the data type depends on the type of test used in the node. The other fields are a bit more complicated. Consider that you would like to measure all traffic from node A to node B or C using DST (destination service type) equal to 3. A boolean expression for this should be something like (source == A) * ((destination == B) + (destination == C)) * (DST == 3). The first node would have type_of_test set to indicate an address equal test. is_or is set to FALSE since it has an logic AND relationship to the next node in the list. The last field (is_last_or) is not used when is_or is set to false. The second node has the same type_of_test, but the relationship to the next node is a logic OR so the is_or field is set TRUE. The third node differs on one of the fields. Testing if destination is equal to address of node C is the last test in the OR expression. This is why is_last_or is set to TRUE in the third node but set to FALSE in the second. The last node performs a DST value equal test and it has the is_or field set to FALSE.

- Open channel queue

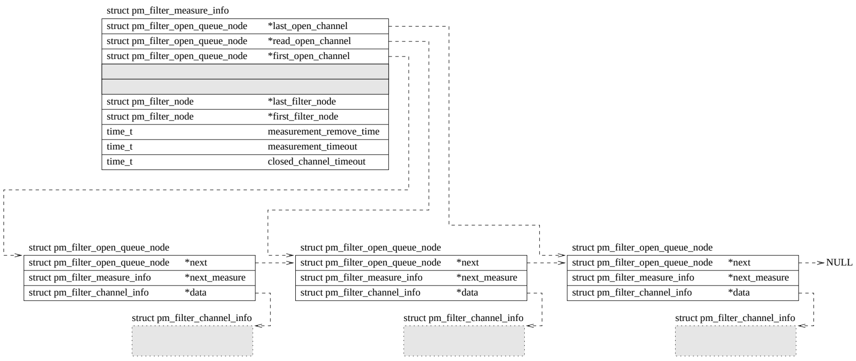

Figure 15: open channel queue in pm_filter_measure_info struct

Figure 15 shows a simple example of a open queue in a measure struct. The nodes in this list are of the same type as the one in the filter open channel queue. There are three pointers to the list. This is needed to ensure that the same measure result (on open channels) isn't sent to the client application more than one time. Each open channel event that passes successfully through the filter function is inserted at the end of this list. The read_open_channel is pointing to the next node to be read by the client application. Reading results from this queue will fail if this pointer is set to NULL.
It is now time to explain the usage of the next_measure pointer. Consider three measurements that have a specific pm_channel_info_struct as data in one of the open queue nodes. Only the nodes of interest is included in figure 16 to keep it as simple as possible.

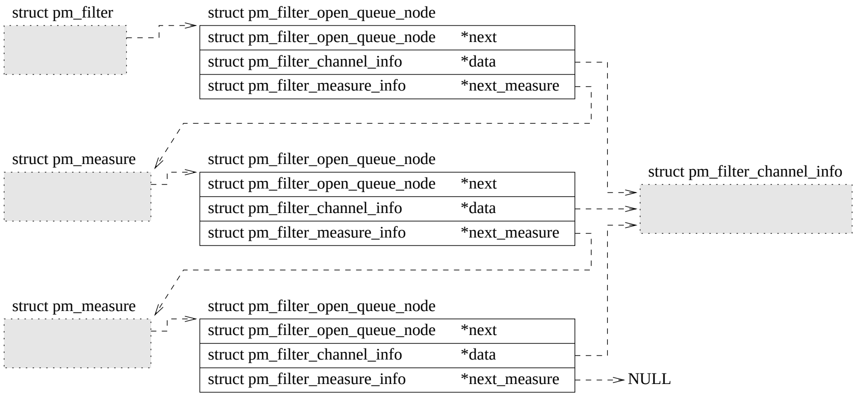

Figure 16: Usage of next measure pointer

The open channel queue in the filter and measurements has to be searched each time a close channel event is sent to the filter. The next_measure pointer is used to avoid searching through measures were the channel doesn't exist. We still need to search for the right node in the open channel queues of the measures but the number of measures to search is kept at a minimum.

- Closed channel queue

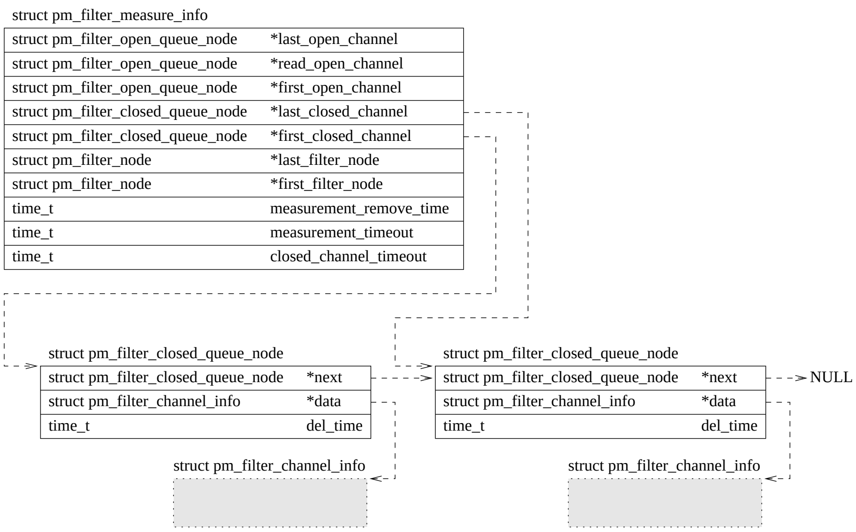

Figure 17: closed channel queue in pm_filter_measure_info struct

This queue is almost identical to the closed channel queue in the pm_filter structure (section 6.2.2). Figure 17 shows this structure. New channels are inserted at the last position and the del_time (removal time) is set to the current time plus the value of the closed_channel_timeout in the pm_filter_measure_info struct. Timeout test is exactly the same. Starting at the beginning of the list and removing every node which has a del_time smaller than the current time until one test fails. Each node that has been read by the client application is removed.

There are to fields in the pm_filter_measure_info struct that need some describing. Timeout test for the measurement is done by comparing measurement_remove_time to the current time. The measurement is removed if remove time has passed. The remove time field is updated to current time plus the value of the field measure_timeout on each access.

6.2.4 Channel information

This structure is shown in figure 19 and it is the simplest one. The slotlist contains one entry for each fragment (sequentially allocated slots) in the channel. All fields except key and ref_count are information received by open and close channel events. The key value matches an earlier received open channel event

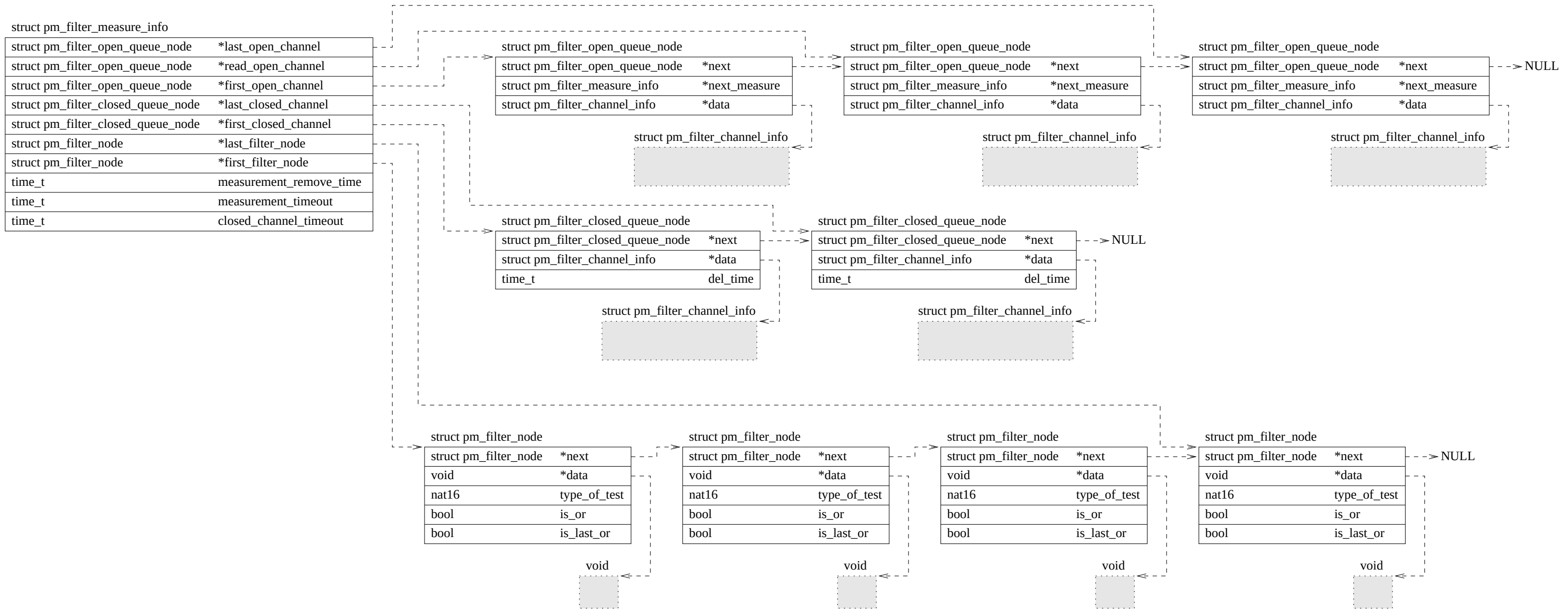

Figure 18: Data structures used by a measurement (section 6.2.3).

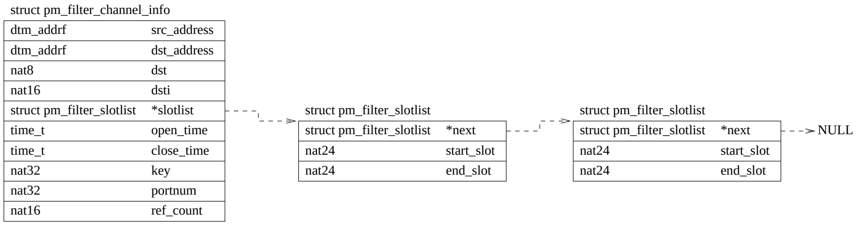

Figure 19: pm_filter_channel_info struct

to a later received close channel event. A open channel info struct retrieved by the client application will be received again once the channel has been closed. The key number is needed by the client application to identify the second copy. Counting of the number of nodes referring to the node is performed so that it should be clear when the channel info struct should be deallocated. The ref_count is decreased when the queue node referring to it is removed and the channel info struct is deallocated if it reaches zero.

6.3 Functions manipulating filter structures

6.3.1 Introduction

This section is not interesting unless you want to modify the communication interface that interacts with the filter and it is not needed for understanding the rest of the report. It can safely be skipped in most cases. Next section starts at page 37. The functions described in this section should allways be used when manipulating the data structures. There should not be any reason to manipulate them directly. It would increase the risk of programming errors. Every function in the implementation will not be described because some of them are only usefull when called from the functions described here. Do not try to use any undocumented instruction.

6.3.2 Functions operating on the filter table

```
pm_filter_table *
pm_filter_table_create(void)
```

Creates a new top level filter table with capacity for four filters. The capacity is automatically increased if needed when the functions described in this section are the only ones that are used to manipulate the structure. The return value is a pointer to the created struct if the function succeeded or NULL if it failed.

```
signed long int
pm_filter_table_add_node(
          pm_filter_table *table,
          dtm_addrf       *address,
          time_t          timeout)
```

Add a new filter to the filter table for events sent from a node. The arguments are the address of the node generating the events and the default closed channel timeout for removal. The return value on success is a filter id or negative if the function failed.

```
signed long int
pm_filter_table_get_version(
          pm_filter_table *table,
          nat16           id)
```

This function returns the current version number for an existing filter id in the filter table. The return value is negative if the function fails.

```
signed long int
pm_filter_table_find_by_addrf(
          pm_filter_table *table,
          dtm_addrf       *address)
```

The filter id is returned if a filter for a network node with the given address exists in the filter table. Return value is negative if it function fails.

```
pm_filter *
pm_filter_table_get_filter(
          pm_filter_table *table,
          nat16           id,
          nat16           version)
```

This function returns the address for a filter in the table if a filter with the given filter id and version exist. Return value is NULL on failure.

```
signed long int
pm_filter_table_remove(
          pm_filter_table *table,
          nat16           id,
          nat16           version)
```

Remove a filter with the given id and version from the table. Return value is zero on success and negative on failure.

6.3.3 Functions operating on the measure table in a filter

```
signed long int
pm_filter_measure_table_new_measure(
          pm_filter_measure_table *measure_table,
          time_t                  measure_timeout,
          time_t                  channel_timeout,
          pm_filter_node          *filter_list)
```

This function is used to create a new pm_measure_info_struct and add it to the measure_table. The measure_timeout is the timeout for removal of the measurement and channel_timeout is the timeout for removal of information about a closed channel. A measurement needs to have a test function and this is given by the filter_list argument which is a linked list of pm_filter_node structs that forms the conditional test. Return value is the id of the new measure or a negative value if it failed.

```
signed long int
pm_filter_measure_table_get_version(
          pm_filter_measure_table *measure_table,
          nat16                   measure_id)
```

Return value from this function is the current version number for the measurement with the given id or a negative value if it fails.

```
pm_filter_measure_info *
pm_filter_measure_table_get_measure(
          pm_filter_measure_table *measure_table,
          nat16                   measure_id,
          nat16                   measure_version)
```

Returns a pointer to the measure struct with the given id and version number. NULL is returned on error.

```
void
pm_filter_measure_table_timeout_test(
          pm_filter *filter)
```

This function tests every measure in a filter to see if any channel references or entire measurements should be removed.

```
void
pm_filter_measure_table_remove(
          pm_filter *filter,
          nat16     measure_id,
          nat16     measure_version)
```

Removes one measurement from the filter.

```
void
pm_filter_measure_table_remove_all(
          pm_filter *filter)
```

Removes every measurement from a filter.

6.3.4 Functions handling the open queue in a filter

```
bool
pm_filter_open_channel_add(
          pm_filter              *filter,
          pm_filter_channel_info *channel)
```

This function will insert a reference to the channel into the filters open channel queue. Return value is true on success or false on failure.

```
bool
pm_filter_open_channel_move_to_closed(
          pm_filter    *filter,
          unsigned int key,
          time_t       close_time)
```

This function is called when a channel is closed and it should be moved from the open queue to the closed queue. The correct channel struct is found using the key value and the close time is written to the struct. The channel is also moved from the open queue to the closed queue in all measurements having a reference to this channel. Return value is true if function succeeded.

```
void
pm_filter_open_channel_remove_all(
          pm_filter *filter)
```

Every entry in the open queue is removed when this function is called.

6.3.5 Functions handling the closed queue in a filter

```
bool
pm_filter_closed_channel_add(
          pm_filter              *filter,
          pm_filter_channel_info *channel)
```

This function adds a reference to a channel struct to the closed channel queue. Return value is true if the call succeeded.

```
void
pm_filter_closed_channel_timeout_test(
          pm_filter *filter)
```

Test if any of the references in the closed queue has timed out and remove them if it is the case.

```
void
pm_filter_closed_channel_remove_all(
          pm_filter *filter)
```

Remove every reference in the queue.

6.3.6 Functions operating on a measure open queue

```
bool
pm_filter_measure_open_channel_add(
          pm_filter_measure_info *measure,
          pm_filter_measure_info *next_measure,
          pm_filter_channel_info *channel)
```

Insert a new reference to a channel into the measurements open channel queue. Read description about the next measure pointer in the text about the open channel queue on page 29. Return value is true if reference is inserted successfully.

```
pm_filter_channel_info *
pm_filter_measure_open_channel_read(
          pm_filter_measure_info *measure)
```

Returns a pointer no a new channel info struct in the open queue or NULL if there isn't any available.

6.3.7 Functions operating on a measure closed queue

```
pm_filter_channel_info *
pm_filter_measure_closed_channel_read(
          pm_filter_measure_info *measure)
```

Remove the first channel reference from the closed queue and return a pointer to it. NULL is returned if no reference exist or if an error occurred.

7 Sending channel events

7.1 Ideas behind choice of event format

One might argue that it would be faster to pack the data into a struct and send it to the event filter. This implementation uses text strings as event format. A complete specification of the event format follows in section 7.4. I've chosen to use text strings based on the following considerations.

- Transparent to method of transport
  The event can be for instance be sent over a TCP/IP channel (used in current implementation) or sent to a syslog daemon with remote logging enabled. Log messages can be filtered and directed to a named pipe by the syslog daemon. A named pipe is a fifo (first in first out queue) that has a name in the filesystem. The event parser can then listen to the pipe and convert the event to RPC calls.

- Transparent to endian issues
  Different computer architectures with different endian format [6] still handles the text event in the same way.

7.2 Parser process

This process parses through the received events and calls the filter using RPC. The process is actually a main process that forks a child process each time an event generating node connects to it. The behavior of the main process is described in the following pseudo code.

```
Initialize a socket that listens on a specific port
Enter infinite loop
    Wait for connection from event generating client
    Create child process
        On failure: Write error message to socket
    Close the new socket
```

Each of the client process behavior is described in the following lines of pseudo code.

```
Close the port that the main process is listening on
Initialize data structures
Wait for a "New Filter" event
Try to create the filter
    On failure: Write error message to socket and exit
Loop until input socket is closed by other node
    Wait until an open or close channel event is received
    Convert event to an RPC call and send it to the event filter
Remove the filter from the filter table
```

7.3 Event generating process

The IP address of the machine running the parser process is given to this process as the first command line option. Current version only supports IP address in number format. It isn't capable of resolving names. The code found here should be integrated into the channel setup and removal code in the DTM protocol. Following pseudo code describes the testbench.

```
Parse the first command line option to find IP address
Try to connect to the server
    On failure: Print error message and exit
Loop until EOF on stdin
    Read "event" from stdin
    Send the "event" to the server
    Read "delay" from stdin
    Delay execution for "delay" number of seconds
Close the socket to the server
```

One part of the code was not included. A child process is started after a successful connection to a server. This child process will print error messages that is sent by the server. That process will run an infinite loop waiting for data to read and forwarding everything to standard out. The events and delay values should be separated by a newline.

7.4 Event format

7.4.1 Conventions used for event format paragraph

Each field of the event is separated by a space (ascii value 0x20) and the fields will be described as <<variable name>>. The variable name is the value stored in the field. Grouping of fields is done by enclosing them with brackets (for example [<<field 1>> <<field 2>>]). Three dots means any number of the preceding field or group. Any other character is part of the event string. The following sections will describe the different event formats.

7.4.2 Creating a filter

Event format:

```
NF <<Version>> <<DTM Address>> E
```

Description:

Current version is 1 and no other versions exist at this moment. It has been included to make it simpler to modify these events. Version number should be changed when new events is added or old ones modified so that the parser process knows the capabilities of the clients. Keeping backward compability in the future will be simpler by specifying this field from the beginning.

DTM address is the address of the node that is generating the events (opening and closing channels). It would have been possible to see who connected to the socket of the parser process but it is likely that the node is using a DTM address different from the one on the IP network. This is why the address is sent.

Error messages:

```
FS FORK FAIL E
```

Calling fork() to start a new child process for this client failed.

```
FS FILTER INIT FAIL E
```

Connecting to filtering process failed.

```
FS MEMORY ALLOC FAIL E
```

The server is out of memory.

```
FS VERSION NOT SUPPORTED E
```

The server is running an old version that can't handle these events.

7.4.3 Opening a channel

Event format:

```
OC <<Source Address>> <<Destination Address>>
<<Number of Slotlist Fragments>>
[<<Start Slot>>-<<End Slot>>,]...[<<Start Slot>>-<<End Slot>>]
<<DST>> <<DSTI>> <<Open Time>> <<Port>> <Key>> E
```

Description:
Source and destination address is the DTM addresses for the transmitting and receiving nodes on the DTM network.

Each channel consists of a number of slots. These slots are grouped into one or more fragments. A channel using slots 1, 2, 4, 7 and 8 has three fragments, and these are 1-2, 4-4 and 7-8. Fragments are groups of consecutive slots. Each fragment sent has the form <<Start Slot>>-<<End Slot>> and the fragments are separated by a comma.

DST and DSTI values identify the type of service using the channel. These values were described in section 4.4 on page 15.

Open time is the return value from the time() function called in the switch or access device that generated the event.

The port number is simply the physical port in the switch or access device that is transmitting or receiving the channel.

Key is a unique number that matches open and close channel events.

7.4.4 Closing a channel

Event Format:

```
RC <<Key>> <<Close Time>> E
```

Description:
Key value is the same as the one sent with the open channel event. Close time is the return value from the time() function called in the node that generated the event.

8 Communication Abstraction

8.1 Server side

8.1.1 Overview

All calls over the network follow the client/server model. The server side of the communication abstraction consists of a number of functions (written in C) that controls the measurements or returns result and static information. All the information these functions need is given as a number of arguments to the function and all result values is stored in a single struct.

The current implementation of the server is running as an RPC process that is translating the incoming RPC calls to the corresponding function calls in this abstraction layer. Multiple threads accessing these functions won't work at the moment. The data structures and the functions used for filtering are not thread safe. This is one of the most important things to improve but I've had to consider it as future work due to limited time. The next sections will specify the interface and the struct used to return data. Section 8.2 on page 46 describes the client side abstraction layer. Jump to that section if you don't want to read about the smaller details.

8.1.2 O&M interface

These functions is used to retrieve values that are found in the O&M interface (section 4.5) under the entry ".registry.dtm.0.dlm.topology". There are only three error messages that need to bee considered when using these functions. Every struct that returns the result has one field named "error" that contains one of these values.

PM_COM_EXEC_OK Everything worked as expected. Returned values are only valid if "error" field is set to this value.

PM_COM_OAM_IF_FAIL Something didn't work when trying to contact the servers O&M interface.

PM_COM_OAM_NO_VALUE The server reported that no value existed for the requested entry.

Each of the following paragraphs names is the same as their names relative to ".registry.dtm.0.dlm.topology" in the O&M interface. The variables x, y and z are variables sent to the functions as arguments. Text on the form <x> should be replaced with the value of x to find the correct entry in the O&M interface.

.local_node

```
void get_local_node_info(pm_local_node_info *)
```

The data structure pointed to by the first argument is used to store the result. All other fields store information found under the .local_node struct in the O&M interface.

.local_node.port_list.<x>.

```
void get_node_port_info(pm_node_port_info *,
                        unsigned int      x)
```

Information about one of the local ports is returned with this call. The variable x is set to the port number (it is in the range of zero to total number of ports minus one). Error field in the return struct (first argument) has the same function as earlier.

.nr_of_links

```
void get_nr_of_links_info(pm_nr_of_links_info *)
```

The values returned in the struct are simply the error value and the total number of links. Several ports can be used in a link (for example a double ring or bus).

links.<x>.

```
void get_link_info(pm_link_info *,
                   unsigned int x)
```

This function returns information about a link who's number is equal to x. The link is divided into something called loops. This is used to find the correct interface to transmit on to reach a certain node on the link. It is not necessary to know all the details about this. The definition is a bit unclear at the moment so I've decided to skip this in the report. Every node on a link can be reach any other node on the same link regardless of which loops the are found in.

.links.<x>.loops.<y>.

```
void get_link_loop_info(pm_link_loop_info *,
                        unsigned int      x,
                        unsigned int      y)
```

Here we find out the number of nodes in the loop with number y under the link with number x. The values stored in the struct is only the error value and the number of nodes.

.links.<x>.loops.<y>.nodes.<z>.

```
void get_link_loop_node_info(pm_link_loop_node_info *,
                             unsigned int           x,
                             unsigned int           y,
                             unsigned int           z)
```

Information about a node on one of the links will be stored int the return struct. You can safely ignore the loops.

8.1.3 Filter creation and removal

Starting a filter

```
void pm_start_filter(pm_start_filter_info *,
                     dtm_addrf *address,
                     time_t channel_timeout)
```

This function initiates a new filter for a node using dtm address and closed channel timeout from the second and third arguments. Results from the call is stored in the pm_start_measure_info struct. The struct is declared as follows.

```
struct pm_start_filter_info
{
  pm_com_errno error;
  unsigned int filter_id;
  unsigned int filter_version;
};
```

The filter_id and filter_version fields are only valid if error is set to PM_COM_EXEC_OK. An error occurring while calling the function will result in error field being set to PM_COM_COULD_NOT_CREATE_FILTER.

Removal of a filter

```
void pm_remove_filter(pm_remove_filter_info *,
                      unsigned int filter_id,
                      unsigned int filter_version)
```

Tries to remove the filter with the given filter_id and filter_version. The pm_remove_filter_info struct is declared as follows.

```
struct pm_remove_filter_info
{
  pm_com_errno error;
};
```

The error field is set to PM_COM_FILTER_DOES_NOT_EXIST if the filter didn't exist, or PM_COM_EXEC_OK if it was successfully removed.

8.1.4 Event receiving

Open channel event

```
void pm_open_channel_event(
          pm_filter_channel_info *channel,
          unsigned int           filter_id,
          unsigned int           filter_version)
```

This call will add a reference to the channel onto the filters open channel queue and tests will be performed to see if it matches any measurement. It will be added to the measurements open queues if it matches. An event where the filter_id and filter_version fields doesn't match an existing filter will silently be discarded.

Close channel event

```
void pm_close_channel_event(
        unsigned long int key,
        time_t            close_time,
        unsigned int      filter_id,
        unsigned int      filter_version)
```

The filters open queue will be searched for a matching key, close_time will be written to a found channel and the channel will be moved to the close channel queue for the filter and any measure holding a reference to it. The event will be discarded if no filter matching filter_id and filter_version is found or if no channel matching key value is found in the filter.

8.1.5 Finding the filter for a node

Any kind of request concerning measurements on a node can only be done after that we have found out what node on the network that is filtering its event log.

```
void get_event_log_node(pm_event_log_node *)
```

This call retrieves the address holding a nodes event log. Starting a measure must be done in these steps.

1. Contact the node you want to measure activities on.
2. Call get_event_log_node() to find out the address of the node holding the event filter for the first node.
3. Contact the node that is running the event filter and start a measure on the first node.

The declaration of the pm_event_log_node struct used to save return values from the function call is found on the next lines.

```
struct pm_event_log_node
{
  pm_com_errno error;
  dtm_addrf    address;
};
```

The address field is set to the address of the node running the filter if the error value is set to PM_COM_EXEC_OK. The error field might also be set to PM_COM_FILTER_NO_LOG_NODE in which case the address type is set to DTM_ADDRTYPE_NOADDR. Address lookup is always done in this way even for nodes running the event filter on localhost.

8.1.6 Starting and ending a measurement

Starting a measure

```
void pm_start_new_measure(
          pm_start_new_measure_info *measure_info,
          time_t                    measure_timeout,
          time_t                    channel_timeout,
          dtm_addrf                 *address,
          pm_filter_node            *filter_list)
```

Result values from this call is stored in pm_start_new_measure_info struct. Each close channel event that matches this measure will be kept for channel_timeout number of seconds and the measurement will be removed if the client doesn't contact it within a period of measure_timeout number of seconds. The address field is set to the node generating the events we want to measure on. The result struct looks as follows.

```
struct pm_start_new_measure_info
{
  pm_com_errno error;
  nat16        filter_id;
  nat16        filter_version;
  nat16        measure_id;
  nat16        measure_version;
};
```

Filter and measure identification is needed when retrieving the results and when ending the measure. The error field can be set to the following values.

PM_COND_EXEC_OK Set if the function executed successfully.

PM_COM_NO_LOG_FOR_REQ_NODE Set if the filter isn't running an event filter for the requested address.

PM_COM_COULD_NOT_GET_FILTER Set if some error occurred while trying to find the a filter that should exist. This should not happen unless there is a bug in the event filter code.

PM_COM_FAIL_CREATE_MEASURE Set if filter was retrieved but an error occurred while trying to add a measure to the filter. Memory allocation failure might cause this.

There is one last field in the pm_start_new_measure struct to describe.

```
struct pm_filter_node
{
  struct pm_filter_node *next;

  void    *data;
  nat16   type_of_test;
  bool    is_or;
  bool    is_last_or;
};
```

A list of these nodes describing the expression to evaluate for each event should be given when the measure is started. This list was earlier described in section 6.2.3 on page 26. No further description will be given here.

Ending a measure

```
void pm_end_measure(
          pm_end_measure_info *info,
          unsigned int        filter_id,
          unsigned int        filter_version,
          unsigned int        measure_id,
          unsigned int        measure_version)
```

The measure will be removed from the filter if filter and measure identification and matches an existing measure. Result is stored in the pm_end_measure_info struct.

```
struct pm_end_measure_info
{
  pm_com_errno error;
};
```

One part in this function call could be improved. Return value is PM_COM_EXEC_OK if the filter is found even if the measure id and measure version doesn't match. The error field is set to PM_COM_COULD_NOT_GET_FILTER if the filter isn't found.

8.1.7 Retrieving measure result

```
void pm_fetch_measure_result(
          pm_fetch_measure_res *info,
          unsigned int         filter_id,
          unsigned int         filter_version,
          unsigned int         measure_id,
          unsigned int         measure_version)
```

The function first tests the filter and measure identification and version numbers. Result values are stored in the pm_fetch_measure_res struct.

```
struct pm_fetch_measure_res
{
  pm_com_errno                        error;
  struct pm_fetch_measure_channel_res *channels_list;
  unsigned int                        num_of_channels;
};
```

The following values can be found in the error field.

PM_COM_ECXEC_OK Set if everything worked and the other fields in the result struct are valid.

PM_COM_COULD_NOT_GET_FILTER Set if filter id and version didn't match an existing filter.

PM_COM_COULD_NOT_GET_MEASURE Set if the filter was found but the measure id and version didn't match any existing measure.

PM_COM_FETCH_RESULT_CHANNEL_LOSS Set if there was a problem in allocating memory. Some information has probably been lost if this value is returned.

Number of channels shows how many items we have in the linked list pointed to by the channels_list field in the struct.

```
struct pm_fetch_measure_channel_res
{
  struct pm_fetch_measure_channel_res *next;
  struct pm_filter_channel_info       *channel;
};
```

The channel that the list node is referring to is of the type described in section 6.2.4 starting at page 30.

8.2 Client side

8.2.1 Overview

The client side of the abstraction layer is completely different from the server side. Using C++ as language is probably the most obvious difference. All communication from a measuring application to nodes on the network (retrieving O&M information and controlling measurements) goes through this layer. State information about connections to servers is kept even if a stateless communication method is used. This makes it simpler to use stateful connection types in the future. The interface to higher layers of the code will not have to change even if we add new methods for communication. It would also be possible to run several methods concurrently. There is one instance of the pm_com_clnt class in the client application process. This object will keep track of connections to servers and manage sending of requests. The pm_com_clnt object will be called the communication manager. Requests is an object who's class is inherited from the pm_request_data class. Some knowledge of UML (Unified Modeling Language) [7] will be needed to understand the following sections.

8.2.2 Communication manager class overview

Figure 20 shows the objects that are used in the communication manager. Note that the pm_com_clnt object is the only one that is visible to the higher layers. The other objects are only used by the pm_com_clnt object. Each of the classes will be described in the following paragraphs.

pm_addr_transl A DTM address can be a IP address but it might also be a MAC or 64 bit DTM address. There is also the possibility that a node in the DTM network might have a serial port running SLIP/PPP or an ethernet interface on the node controller that is using an IP address different from the DTM address. The purpose of this class is to translate the DTM address to a IP address string that can be used by the RPC calls. The objects constructor will read addresses from the file ~/.dtm_to_ip_address_table where each line should consist of the DTM address string followed by the IP address string. A table of all known DTM addresses can also be retrieved from this object (used for selecting measure parameters).

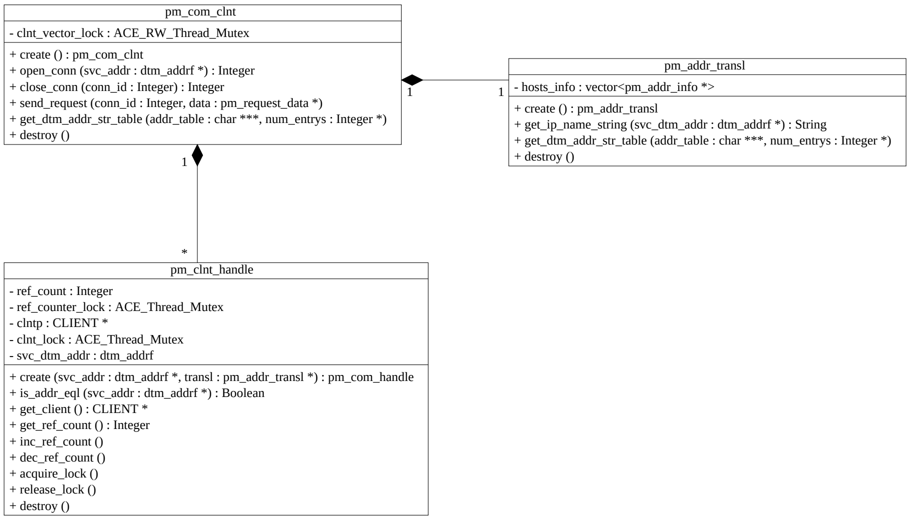

Figure 20: Communication manager class diagram

pm_clnt_handle This object is responsible for keeping information needed for the communication with a network node. It also keeps track of how many “users” that has opened a link to this node. The number of “users” is kept in the reference counter. Connection info can be removed and communication resources deallocated when all “users” has closed the connections. A communication manager can have any number of these objects (one for each node).

pm_com_clnt Users of the communication manager can only see this object. The object has an address translation object and one handle for each network node that has an open communication link.

8.2.3 Opening and closing connections

Opening a connection A call to the open_conn function in the communication manager will result in the following sequence of events.

1. Compare the address fields of all existing open connection handles. Increase the reference count in the handle and return its descriptor number if the address field matches the one requested (first argument to open_conn function).

2. Create new handle for this connection using the address translator object to find the correct address. Initiates the reference counter to one and opens the connection. The descriptor number for this newly created handle is returned.

Future work on the translator object could include returning possible communication methods along with the address. The handle should also be aware of the communication method. This would make it possible to concurrently handle several methods of communication.

Closing a connection The following sequence will be executed when the close_conn function in the manager is called.

1. Make sure that the descriptor number is a valid one. Return immediately if it isn't found.

2. Decrease the reference counter in the handle corresponding to the descriptor number.

3. Remove the handle and close connection to the handles remote node if the reference count has reached zero.

8.2.4 Sending a request over the network

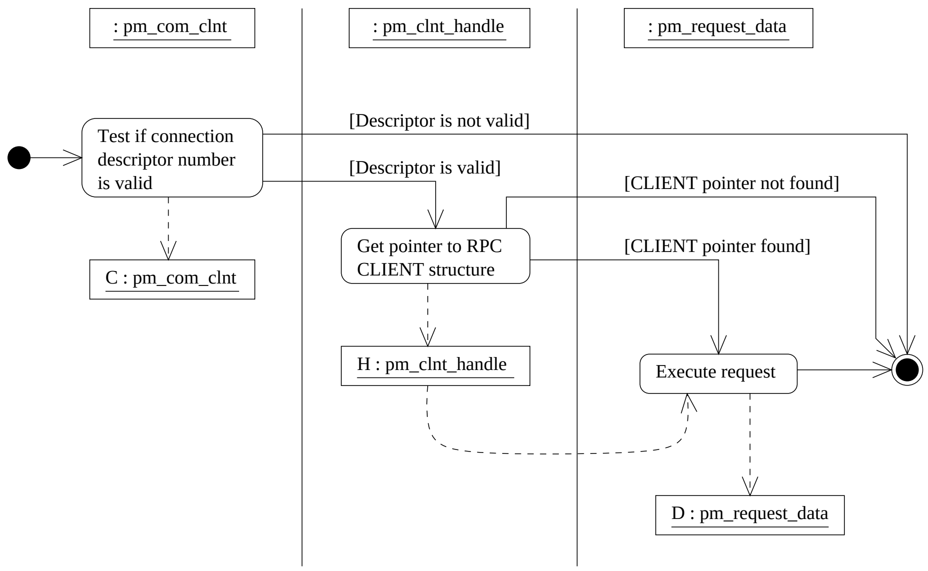

Figure 21: Sending a request using the manager object

Figure 21 shows what happens when send_request in the communication manager is called. The pm_request_data object is responsible for the actual RPC call and it contains the result from the request. It will be further described in section 8.2.5. The three main groups it the activity diagram is described in the following list.

1. The pm_com_clnt tests if the descriptor number (argument to send_request function) matches a handle and returns if it isn't valid.

2. The handle is called to retrieve the CLIENT pointer (RPC connection information). Return if the pointer is NULL.

3. The second argument to the send_request function is an object inherited from the pm_request_data class. This class has a virtual function called execute_request, and this function is the only part that is related to RPC in the object (the function executes the RPC call). Execute request is

called with the CLIENT pointer from the handle. Local failures will result in error values in pm_request_data object.

Observe that even if the data object isn't transparent to RPC, we still have transparency when using the object from higher layers. New virtual functions would have to be added to the data objects if other communication methods should be supported. The communication manager could then choose one of these depending of communication method used for a specific node. The process calling send_request doesn't know which of the methods in the data object that were used. Following code example retrieves the number of ports (interfaces) in a node with address 1a.f2.45.65.24.d2.8b.01. Simple error checking and reporting has been replaced by comments.

```
pm_com_clnt          com_manager;
pm_request_node_info node_info;

int        conn_id;
dtm_addrf node_addr;   // Contains DTM address:
                       //   1a.f2.45.65.24.d2.8b.01

conn_id = com_manager.open_conn(&node_addr);
// Test conn_id, report error if less than zero

com_manager.send_request(conn_id, &node_info);
if(node_info.get_error() == PM_COM_EXEC_OK)
  {
    // Everything worked ok
    cout << "Number of ports: "
         << node_info.get_number_of_ports() << endl;
  }
else
  {
    // Report error (network call failed)
  }
```

There is nothing in the above code that is dependent on RPC. We can safely extend the communication manager with another communication method. All data classes and the communication manager would have to be modified but the rest of the application would be unaware of that the new method is used.

8.2.5 Data objects

Error reporting The base class (pm_request_data) has a protected variable called "error". It contains the result from the last send_request() call to the communication manager with this object as second argument and it is set to PM_COM_NOT_CALLED by the constructor. A call to the get_error() function will return this variable and it should be PM_COM_EXEC_OK if the call was successfull.

Getting the received values There can be any number of access functions in each object. Figure 22 shows the class hierarchy for the request data objects.

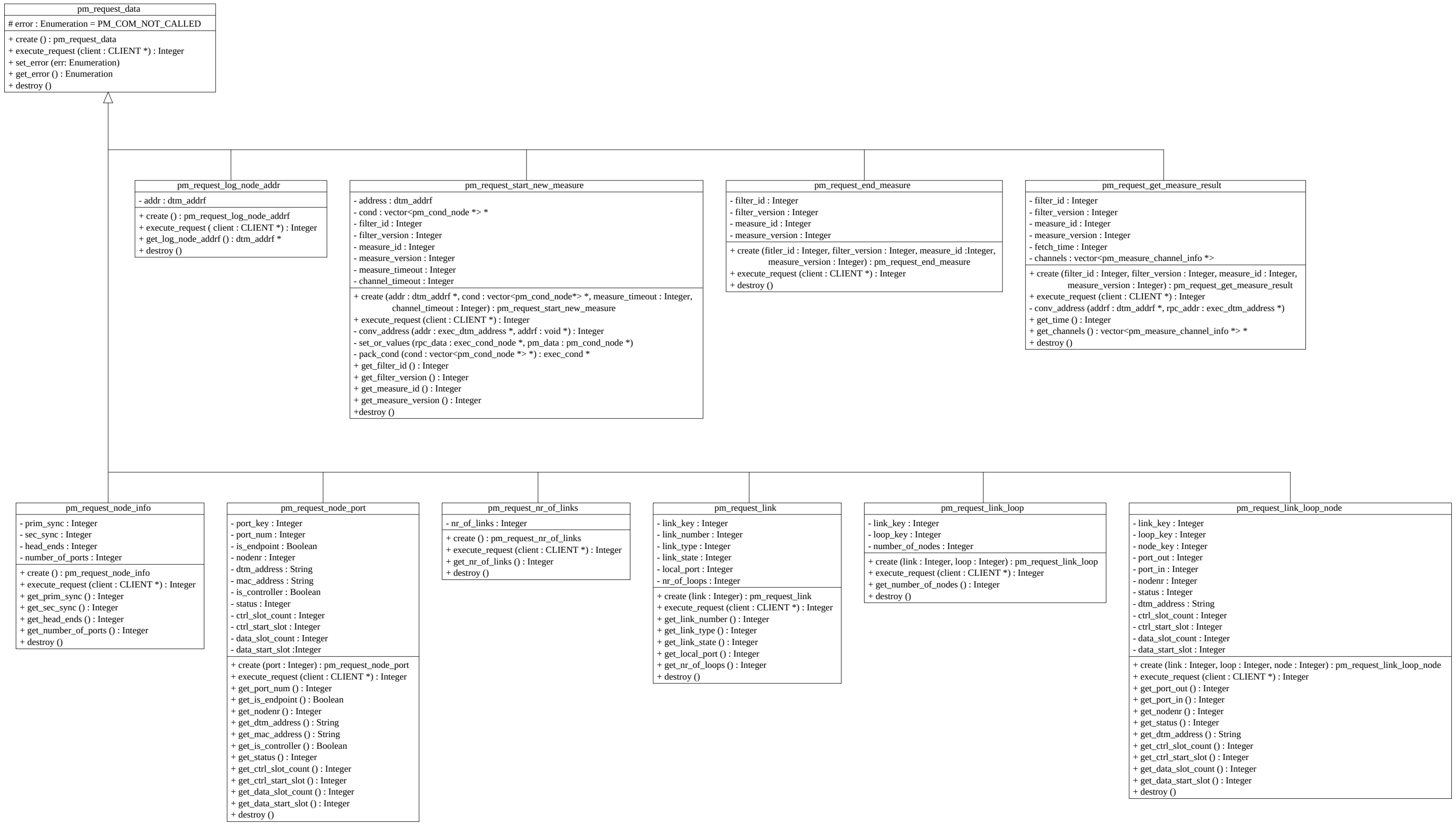

Figure 22: Class diagram for pm_request_data and inherited classes

The top object is the base class. Classes that is used for controlling measures and retrieving results are found on the middle and the ones used to access the O&M interface is found at the bottom of the diagram. The most common case is that these access functions return valid data only if the get_error() call returned PM_COM_EXEC_OK, but there are some classes that work differently.

Mapping of objects to server side function calls There is no need to further document the objects purpose since they directly map to a server side call. Table 1 on page 51 shows which calls the differrent objects are using on the server side. Note that the pm_request_log_node_addr class is misssing a function on the server side. It is currently using one of the other functions to resolve the DTM address of the server which means that the implementation assumes that the event filter is running on the same host. An entry in the O&M interface should be added to show where to find the event filter and the pm_request_log_node_addr class should use this entry. The "Page" column shows where to find descriptions of the corresponding server side functions.

Client side class	Server side function	Page
pm_request_data	<< Never Used >>	
pm_request_log_node_address	<< Not implemented >>	
pm_request_start_new_measure	pm_start_new_measure	43
pm_request_end_measure	pm_end_measure	45
pm_request_get_measure_result	pm_fetch_measure_result	45
pm_request_node_info	get_local_node_info	40
pm_request_node_port	get_node_port_info	41
pm_request_nr_of_links	get_nr_of_links_info	41
pm_request_link	get_link_info	41
pm_request_link_loop	get_link_loop_info	41
pm_request_link_loop_node	get_link_loop_node_info	41

Table 1: Mapping of functions between client and server

9 Client application

9.1 Introduction

The client application is running under Linux and could probably be used under any UNIX-type operating system with minor modifications. It should be seen as a testbench for the event filter, measure control and communication methods. The application will probably be the first part to replace. X-windows programming has been simplified by the help of Gtk– [9]. Gtk– is a C++ wrapper for the GIMP Toolkit (GTK+) [8]. I've had some problems with the wrapper. It has improved a lot since the start of this project but there are some functionality that doesn't work perfectly yet. Most of the problems occurred when trying to dynamically allocate and deallocate widgets. A widget can for instance be a button, window, scrolled list or any other object you want to put on the screen. One example is the measure result win. It was not possible to deallocate the window (it caused the program to exit with a segmentation fault). Some strange behavior in the application is caused by these problems. Future versions of the client application would benefit from being written with a cross-platform toolkit (to make the program run under Microsoft's Windows without rewriting the code) like for instance the Qt toolkit.

9.2 Main window

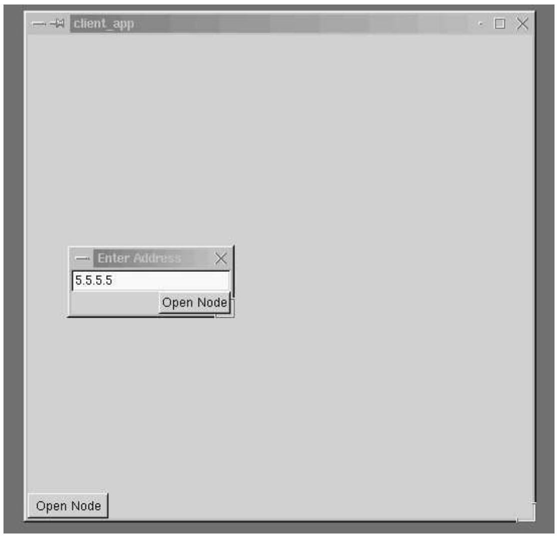

Figure 23: Client application connecting to a network node.

Figure 23 shows the process of connecting to a node on the network. There is only a single button in the main window when the application is started. The Child window appears when the "Open Node" button in the main window is clicked. Address of the node should be typed into the text entry row in the child window and the "Open Node" button should be pressed after that. Acceptable DTM address types are IPv4, MAC, and 64 bit DTM.

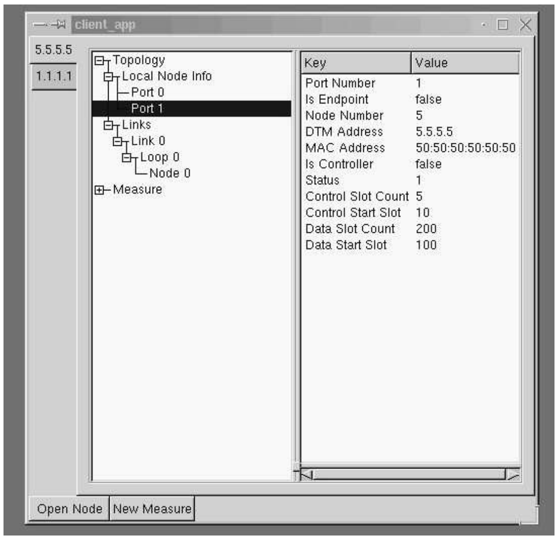

Figure 24: Client applications main window.

The main window in figure 24 has a notebook widget [9] where each page holds information and measurement for one opened node. There is a tree structure in the left part of the notebook page. Topology item and all items under it contains the information in the O&M interface that is found under the entry ".registry.dtm.0.dlm.topology". The measure item holds one entry for each measurement and the items have a label that is the same as the measurements name. Clicking on an item under topology will show all values found under that entry in the right list. The variables name is found in the "Key" column and The value in the "Value" column. The list on the right will change to one column named "Description" and the measures description is printed if an entry in the measure tree is selected.

9.3 Starting a measure

Figure 25 shows the first out of five dialog windows that appear when the "New Measure" button in the main window (figure 24) is clicked. The description field is optional but the a name has to be entered. Clicking Next without typing a name won't work.

The next page (shown in figure 26) chooses which ports to include in the measure. Default is to accept any port. You can also choose to only include those moved to the selected list or to skip those in the selected list. There is one more detail that has to be implemented here. Current implementation lists 12 ports regardless of how many ports that exist in the node. A pm_request_node_info request should be used to find the correct number of ports.

Types of services to include in the result is chosen in the third page (figure 27). The dialog works in the same way as the choosing of ports. Worth to notice is that this implementation assumes eight bit DST (Destination Service Type) values. This is true in the current implementation but it will change in future versions of the channel protocol and that will require this page to be rewritten.

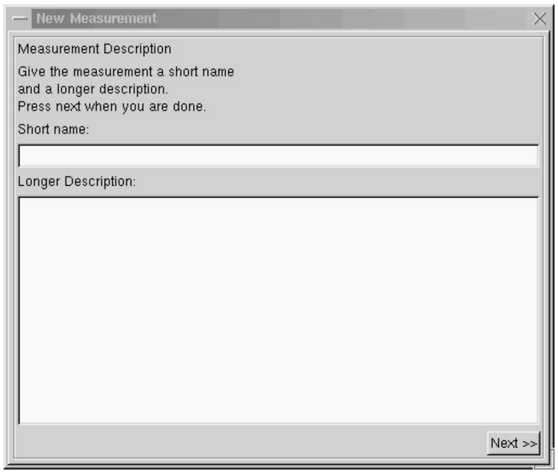

Figure 25: Start new measure dialog (page 1).

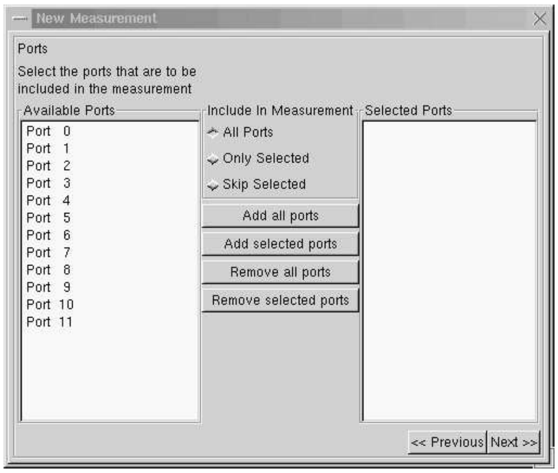

Figure 26: Start new measure dialog (page 2).

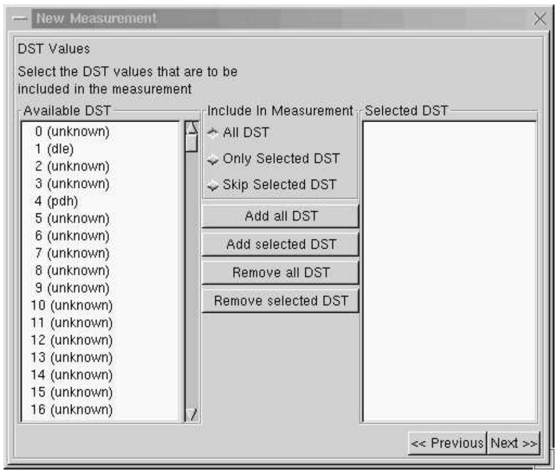

Figure 27: Start new measure dialog (page 3).

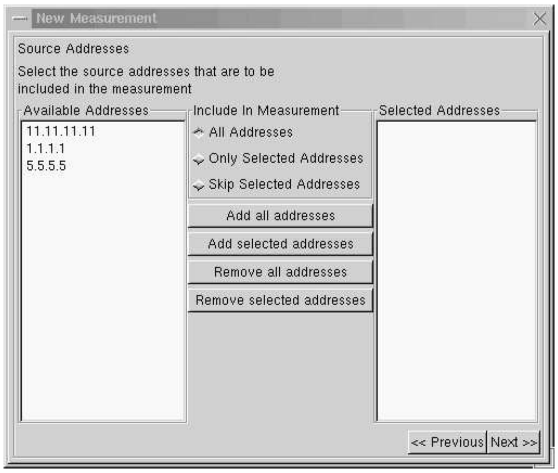

Figure 28: Start new measure dialog (page 4).

Figure 28 needs no further explanation. It is used to select the source addresses to include, and it works like the port and DST pages. The available list shows all addresses known to the address translator object (section 8.2.2 on page 46).

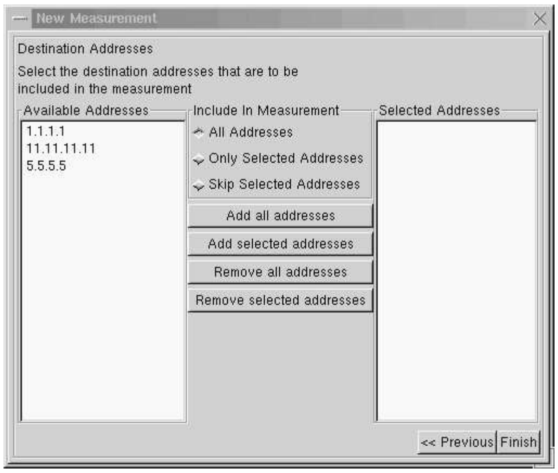

Figure 29: Start new measure dialog (last page).

The last window (figure 29) chooses which destination addresses to include in the result. It works like the source address dialog. The event filter will be contacted and a measure started when the “finish” button is pressed. An entry with a label equal to the measures name (from the first page in figure 25) is added to the measure tree in the main window and periodic collection of results is started.

9.4 Viewing measure results

The measure result windows will appear when doubleclicking on a measure in the main window. Figure 30 shows three measure result windows where “dst_3” is recording events with service type equal to tree, “dst” is recording events where the destination address is equal to “5.5.5.5” and “src” is recording events with source address “5.5.5.5”. Update is done automatically every second so the “Update” button is of questionable use right now. Removing a measure is done by clicking on the window managers “X” in the upper right corner of the window. This will remove the measure from the event filter and stop the periodic retrieval of results. It will not deallocate all memory used by the measurement in the client application and it will not remove the measure entry in the main window due to the problems with Gtk– mentioned in section 9.1.

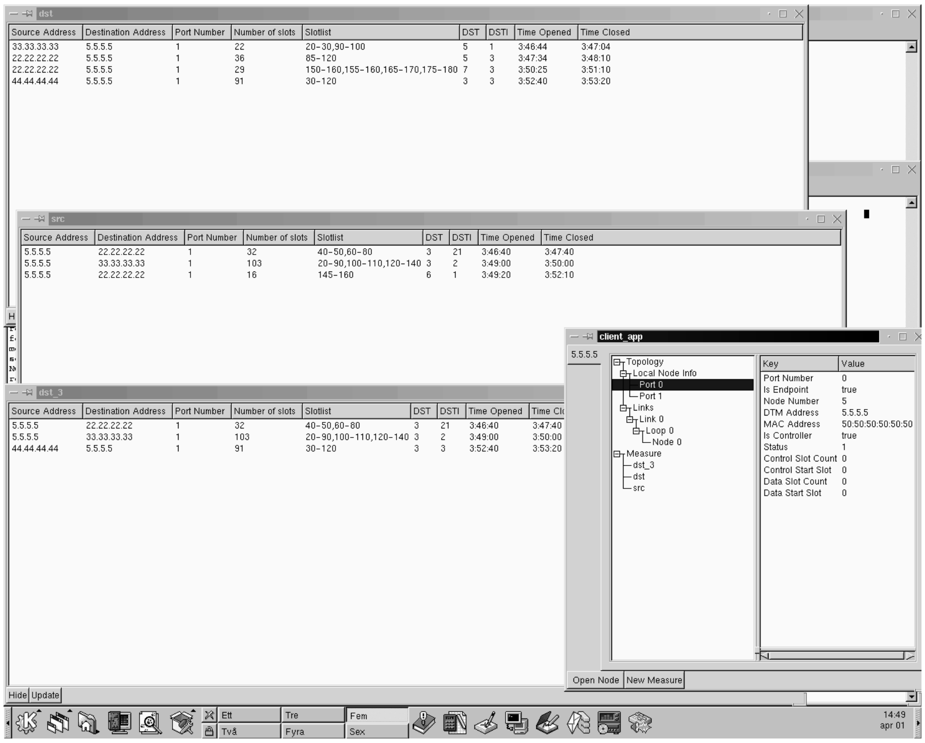

Figure 30: Main window and three measure result windows.

10 Summary and future work

10.1 Summary

This project has been focused on finding a method to measure activities on a DTM network. Every type of measure specified in the original project description has not been implemented. Control channel measuring is one example of this. The basic measuring functions for data channels are implemented but not tested to any greater extent.

Timeouts have been implemented in the event filter, but they have not been tested. The event filter seems to be stable and it returns the correct result when sending a number of events and when there are several measures running at the same time.

Sending the events as text strings over TCP/IP has simplified the debugging and provided a simple way of connecting to the filter. The events could also be sent using syslog or any other method since they have the time of the event recorded. There is no need to deliver the event immediately after it occurred.

RPC was chosen for communication with the node. It was the only method available on all the platforms. An abstraction layer removed dependencies to RPC so that it will be possible to add new methods without rewriting everything. C++ was chosen as language for the client interface and C for the server.

The client application was written and tested under Linux. It should be simple to port it to other Unix-type operating systems. The language chosen for this application was C++ and the Gtk– library was used for X-windows programming. There have been some problems with Gtk–. It should be noted that the library has improved during the time I've worked on this project, but I still have some problems which have limited the application´s functionality. The application should be seen as a testbench.

Tests of the measurement tools have not been as extensive as described in the original project description. The modifications to existing software that would be needed to generate the events have not been done. It was not possible for me to modify the code and more important work has prohibited others from assisting me. Basic functionality has been tested. More complex filtering functions and heavier loads (received events) have not been tested.

10.2 Future work

One thing that has to be done is the addition of event sending code in the channel handling protocol in the DTM network nodes. The result of this project is unusable if it isn't done. Perhaps there should be a additional event format specified for raw data to save the processing time for converting between data and text.

Next part on the list would be the client application. This program needs to be modified or rewritten. First some investigations should be done concerning choice of library for the graphical user interface, how to chose what parameters to measure on and how to present the result. The GUI library should preferably be cross platform.

The event filter needs some modifications. Channel information structs that time out are silently discarded. There should be some notification to the measure client. Test functions are needs to have more functionality. Current implementation is suitable to logical expressions of the type ((A"or" B) "and" (C "or" D)). Problem arises if you want to have functions similar to ((A "and" B) "or" (C "and" D)).

Most parts of the current implementation have not been designed to handle multiple threads. There is also a need to investigate how the communication methods (for instance RPC) libraries react to multithreaded programs. The same goes for the graphical user interface libraries. Most of them are not thread safe for performance reasons.

Security and resource limits have been ignored in this implementation. There should be some kind of access control. The simplest way would be to have a password for starting a measure, but more complex access control could give a user the possibility to study his own traffic without being able to study other users´ traffic. Limiting resources on a user or measure basis might be useful if some measure is considered more important than the other.

References

[1] Christer Bohm, "The DTM Protocol - Design and Implementation", Licentiate Thesis, Telecommunication System Laboratory, Department of Teleinformatics, Royal Institute of Technology, Sweden, February 1994.

[2] Per Lindgren, "A Multi-Channel Network Architecture Based on Fast Circuit Switching", A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Technology, Telecommunication System Laboratory, Department of Teleinformatics, Royal Institute of Technology, Sweden, May 1996.

[3] Net Insight AB, "DTM Basics and Solutions", second edition, information about Net Insights DTM implementation, January 1999.

[4] Integrated Systems Inc. (ISI, company behind pSOS+ real-time OS). "Embedded Systems Software and Design Automation Solutions". Online. Available: http://www.isi.com/. 29 January 1999.

[5] NetBSD project. "The NetBSD Foundation, Inc.". Information about the NetBSD operating system. Online. Available: http://www.netbsd.org/. 29 January 1999.

[6] Steven Zucker, "Endianness in Solaris", Paper focusing on programming-related aspects of endianness. Sun Microsystems, Inc. 1998. Online. Available: http://www.sun.com/realitycheck/endian.pdf. 9 April 1999.

[7] Sinan Si Alhir, "UML in a nutshell", Quick reference to the Unified Modeling Language (UML) version 1.1, O'Reilly & Assosiates, Inc., September 1998, ISBN: 1-56592-448-7

[8] GTK+ Documentation. The GIMP Toolkit (Graphical User Interface Toolkit for X-Windows). Online. Available: http://www.gtk.org/. 23 April 1999.

[9] Gtk– Documentation. C++ wrapper for the GIMP Toolkit (GTK+). Online. Available: http://lazy.ton.tut.fi/terop/iki/gtk/. 23 April 1999.

[10] Troll Tech AS. Documentation about the Qt cross platform GUI (Graphical User Interface) toolkit. Online. Available: http://www.troll.no/. 23 April 1999.

[11] S. Bradner, "Benchmarking Terminology for Network Interconnection Devices", Network Working Group, Request for Comments: 1242, Harward University, July 1991. Online. Available: ftp://ftp.funet.fi/pub/netinfo/rfc/rfc1242.txt. 30 April 1999.

[12] S. Bradner, J. McQuaid, "Benchmark Methodology for Network Interconnected Devices", Network Working Group, Request for Comments: 1944, Harward University and Bay Networks, May 1996. Online. Available: ftp://ftp.funet.fi/pub/netinfo/rfc/rfc1944.txt. 30 April 1999.

[13] R. Mandeville, "Benchmarking Terminology for LAN Switching Devices", Network Working Group, Request for Comments: 2285, European Network Laboratories, February 1998. Online Available: ftp://ftp.funet.fi/pub/netinfo/rfc/rfc2285.txt. 30 April 1999.

[14] Cooperative Association for Internet Data Analysis (CAIDA), "CAIDA Measurement Tool Taxonomy", Overview of Internet and TCP/IP performance measurement tools and efforts. October 1998. Online. Available: http://www.caida.org/Tools/taxonomy.html. 30 April 1999.

[15] Gary R. Wright, W. Richard Stevens, "TCP/IP Illustrated, Volume 2", TCP/IP Implementation descriptions, Addison-Wesley Publishing Company, June 1995, ISBN 0-201-63354-X.