Design and Implementation of a DTM Network Simulator

Johan Larsson, Thomas Eriksson
Master’s Thesis:
Design and Implementation of a DTM Network Simulator

Johan Larsson          Thomas Eriksson
johan@ludd.luth.se     thomas@ludd.luth.se

Dept. of Computer Science and Electrical Engineering
Luleå University of Technology, Sweden
Net Insight AB Stockholm, Sweden

21st May 2001

Supervisor: Björn Fahlcr, Net Insight AB, Sweden
Examiner: Lars-Åke Larzon, Luleå University of Technology, Sweden
Abstract

When developing networking protocols there exists a need to test the reliability and correctness of them. This can be done using ordinary hardware, but often tests are performed using a simulator.

The aim of this thesis was to develop a simulator for Net Insight AB, to be used when testing their DTM protocols. DTM or Dynamic synchronous Transfer Mode is a networking technique based on circuit switching.

A simulator that simulates DTM hardware via a hardware interface has been designed and implemented. It runs on Linux and supports simulation of different kinds of errors, for example, fibre breaks and hardware resets. Furthermore, it can be run separately on one computer or distributed on several computers.
Preface

This Master’s thesis was done at Net Insight AB during the first six months of year 2000 by Johan Larsson and Thomas Eriksson. There are a number of persons we would like to thank, who have given us support and insightful comments on our work. These persons are:

- Lars-Åke Larzon, examiner, CDT - Luleå University of Technology.
- Björn Fahlér, supervisor at Net Insight AB.
- Erik Brage, Stephane Tessier, Martin Christiansson, Joakim Jäderberg, Jonas Thor, all members of the protocol group at Net Insight AB.
- Families and friends.

Stockholm, August 2000
Johan Larsson and Thomas Eriksson
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Why use a simulator?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Outline of this document</td>
<td>2</td>
</tr>
<tr>
<td>2 Theory</td>
<td>3</td>
</tr>
<tr>
<td>2.1 DTM (Dynamic synchronous Transfer Mode)</td>
<td>3</td>
</tr>
<tr>
<td>2.2 DXL (DTM Switching Layer)</td>
<td>5</td>
</tr>
<tr>
<td>2.3 SUPV</td>
<td>5</td>
</tr>
<tr>
<td>3 Design</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Design goals</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Important design decisions</td>
<td>8</td>
</tr>
<tr>
<td>3.3 Overall design structure</td>
<td>9</td>
</tr>
<tr>
<td>3.4 The Controller</td>
<td>10</td>
</tr>
<tr>
<td>3.4.1 Controller</td>
<td>11</td>
</tr>
<tr>
<td>3.4.2 TimeInfo</td>
<td>11</td>
</tr>
<tr>
<td>3.5 The Node Process</td>
<td>12</td>
</tr>
<tr>
<td>3.5.1 Node Process</td>
<td>12</td>
</tr>
<tr>
<td>3.5.2 DXL (DTM Switching Layer)</td>
<td>13</td>
</tr>
<tr>
<td>3.5.3 SimDXLManager</td>
<td>14</td>
</tr>
<tr>
<td>3.6 The Net Core</td>
<td>15</td>
</tr>
<tr>
<td>3.6.1 NetProcess</td>
<td>15</td>
</tr>
<tr>
<td>3.6.2 NodeUnit</td>
<td>15</td>
</tr>
<tr>
<td>3.6.3 HEAL</td>
<td>15</td>
</tr>
</tbody>
</table>
3.6.4 Channel .................................................. 15
3.6.5 Board ................................................... 16
3.6.6 InterfaceBase ......................................... 16
3.6.7 NodeController ....................................... 16
3.6.8 Interface ............................................... 16
3.6.9 Slot ..................................................... 16
3.6.10 Events ................................................ 16
3.6.11 EventCreator ....................................... 19
3.6.12 EventHandler ....................................... 19
3.6.13 EventHistory ....................................... 19
3.6.14 Config ............................................... 19
3.6.15 Log .................................................. 19
3.7 The configuration file .................................. 20
3.8 The communication protocol .......................... 20
  3.8.1 SimPacket .......................................... 21
  3.8.2 SimPacketQueue .................................... 21
  3.8.3 CommHandler ....................................... 21
4 Implementation .......................................... 23
  4.1 The Node Process ..................................... 23
    4.1.1 The Node Process main method ................. 23
    4.1.2 DxI_main ......................................... 24
    4.1.3 DXL .............................................. 24
    4.1.4 SimDXLManager ................................ 24
  4.2 The Net Core .......................................... 24
    4.2.1 The NetProcess main loop ..................... 24
    4.2.2 Event handling ................................. 26
    4.2.3 Data sending algorithm ....................... 28
5 Conclusions .............................................. 33
A Word list .................................................. 37
B Configuration file ....................................... 39
C Bison grammar .......................................... 41
Chapter 1

Introduction

Net Insight AB [9] is a company that develops and markets broadband network products based on the Dynamic synchronous Transfer Mode technology (DTM). All their products use a DTM protocol stack, developed by a special group at Net Insight. To test the functionality of their different protocols, the group wanted a simulator platform on which they could run them. It should simulate the hardware of a DTM node, have support for introducing simulated errors and be able to perform logging of messages sent by the protocols to the platform.

The objective of this thesis was to design and implement such a simulator for Net Insight AB.

1.1 Why use a simulator?

It is important to test the reliability and correctness of software produced in development of networking protocols. One way to do this is to build a test laboratory with real hardware and test the protocols in that environment. However, this is both expensive and space demanding. Another way to test the protocols is to build a simulator which simulates the underlying structure that they run on. This provides a more cost efficient solution that does not require any special hardware.

Some of the benefits with a simulator are described below:

- Support for easier configuration of the network, that is, there is no need to connect fibres to create a certain topology. To do this a configuration file is simply modified. In the same way, it is also possible to add a new interface or board without actually inserting them into your hardware.

- If logging is performed when running the simulator, an error can be detected, which leads to that modification of software is carried out. It is then possible to validate the correctness of the modifications by running the simulator again.
For the simulator developed in this project, the platform used is Linux, which has some powerful tools for profiling and memory leakage detection. These tools can not be used when running the stack on the real hardware because the operating system used for it, is not supported by the tools.

As mentioned above, hardware is space demanding, which implies that if you want to simulate networks with large number of nodes it would not be feasible. It is however with a simulator.

1.2 Outline of this document

This thesis is outlined as follows. Chapter two gives the theoretical background necessary to understand this thesis. In Chapter three, the design of the simulator is presented. Chapter four and five deals with some important parts of the implementation. And finally, in chapter six, some concluding remarks are given and some possible future improvements are proposed.
Chapter 2

Theory

This chapter describes the theory behind the DTM networking technique. Section 2.1 describes the DTM technique in a wide perspective. Section 2.2 handles the DTM Switching Layer. Finally section 2.3 briefly describes the Supervisor (SUPV), part of Net Insight’s DTM system.

2.1 DTM (Dynamic synchronous Transfer Mode)

**DTM** (Dynamic synchronous Transfer Mode) [7], is a networking technique based on circuit switching. It is designed for a medium with the capacity shared by all connected nodes. DTM can be built in several different datalink topologies such as point-to-point, ring, dual-ring and dual-bus, see figure 2.2. A DTM network can be expanded using switch nodes to interconnect several links.

An important part of DTM is the channel concept, where a channel is a set of time slots with a sender and an arbitrary number of receivers. A channel has constant delay and guaranteed capacity for the transport, two of DTM’s most important properties. The total capacity of a DTM link is divided into frames, DTM frames, see figure 2.1. These frames have a frame frequency of 8kHz, and each frame is divided into 64-bit time slots. This gives a DTM...
channel a granularity of 512kbps. The number of slots in each frame varies depending on the capacity of the link. Channel set-up is fast and the capacity of a channel can easily be changed later. More of the DTM Network system can be found in [2].

DTM can be used to carry several different kinds of media, for example telephony, video and data traffic. For video and telephony DTM is especially suited because of its constant delay and in-order quality of service with guaranteed bandwidth.

More information on DTM and Net Insight’s DTM system can be found at Net Insight’s Knowledge Bank [10].
2.2 DXL (DTM Switching Layer)

As described in [4], DXL can be seen as the glue between the DTM protocol stack and the switching hardware in a DTM node. It provides four different interfaces. The first, DTM Switch Interface (DXI), is used for channel control, for instance, to create channels and reserve bandwidth on them. The second, DTM Channel Adaption Interface (DCAI), handles data send and receive operations. The third, Hardware Equipment Abstraction Layer (HEAL), is the interface used to interact with the DTM hardware. The final interface, DTM node Synchronisation Control interface (DSYNC), is used when performing operations on sync sources. These four interfaces are more thoroughly described below.

- **DXI** [5], provides methods for control of DTM channels. It consists of interface functions and common data-structures for this.

  The interfaces that the DXI controls are physical interfaces. An interface can either be a link interface or a leaf interface. A link interface is an interface connecting DTM nodes. A leaf interface is an interface where DTM channels terminate.

  A DTM channel can have one RX interface and several TX interfaces. An RX interface is the interface in a node over which data is received into the node, and a TX interface is an interface over which data is sent out from the node. A special interface in the node is the Node Controller, it is the interface the software uses to send and receive data. Any interface in the node can either be a TX or an RX interface.

- **DCAI** [11] is the part of the DXL that does the data handling. In DCAI there are methods such as `dcai_add_sender`, which adds a sender identified by a certain `cmi` to a channel. A `cmi` (channel multiplexer identifier) is used to enable multiple senders and receivers on a single channel. `Dcai_send`, is another method in DCAI, and is used to send data to a specific channel and `cmi`.

- **HEAL**, as described in [13], is used to manage and control the hardware, such as system boards and their interfaces used for receiving and transmitting data. **HEAL** is also where device drivers to handle specific boards can be loaded. This may or may not involve software loading depending on the system architecture.

- **DSYNC** is used to enable DTM network synchronisation support. It operates locally on a node and provide, for example, get and set methods for which sync source to use. In a DTM network it is important that all nodes have a synchronised DTM frame rate [3]. A sync source can either be **Hold Over** (internal clock), **External Clock Source** or a **DTM interface**. In [12] more of DSYNC is handled.

### 2.3 SUPV

The **SUPV** is a part of Net Insight’s DTM system. It is used to keep control of the different parts of the system. In a large system it is desirable to have a coherent way of controlling and dispatching the different phases in the system. The control is given to the SUPV at system
startup by a call to `supv_start` from `main`. After that, the SUPV dispatches the different startup phases. The system is now ready to run and give the control to the parts of the system that requests it.

SUPV uses something called blocks. These blocks belong to different parts of the DTM system. For example, there exist a DXL block, which is a container for information specific to the DXL. The blocks are created calls from SUPV to different previously defined methods.
Chapter 3

Design

This chapter describes the design of the simulator. Section 3.1 presents the goals of the design. Section 3.2 discusses some important design decisions. Section 3.3 describes the overall structure of the design. Section 3.4, 3.5 and 3.6 handles the design of the Controller, the Node Process and the Net Core respectively. In section 3.7, the design of the configuration file is explained, and finally in section 3.8, the communication protocol design is discussed.

3.1 Design goals

When this work started, some demands on the design were specified. The demands are listed and explained in this section.

- General
  Two important demands on the simulator are the operating system to use and what different topologies it should manage in its first version. The operating system that should be used is Linux [8], simply because Net Insight has bought a testing tool that works only under Linux. The topologies are point-to-point, dual-bus, ring and dual-ring, see figure 2.2.

- Nodes
  A simulated node must be able to have several interfaces. It should be possible to have one upstream fibre and one downstream fibre connected to each interface. It should also be possible to have the interfaces partially or totally disconnected.
  The node should also be developed in such a way that it is possible to link with C code, this is because the protocol stack is implemented in C and will run together with the simulator.
3 Design

• Events

Other demands on the simulator is that pre-configuration of events and manual instant addition of events should be supported. Only a few events has to be included in the first version of the software, but introduction of all the specified events in later versions of the simulator should be taken into consideration when designing it.

These are the specified events:

– Endian variation, that is, it should possible be to run the simulator on architectures with different byte orders.
– Network delays.
– Data filtering, for example, bit errors.
– Fibre cuts.
– Access board errors, for example errors that occur in case of power failure.
– Node errors, for example a hardware reset.
– Introduction of new fibres between nodes.
– DTM interfaces with different capacity and line coding.
– Saving of events into file for later execution.

• Logging

It is important that the simulator supports logging of events. Both errors introduced and ordinary commands executed should be logged. Examples of these kinds of events are a fibre cut event and channel creation event. Another important logging feature is that it should be possible to synchronise the logging done in the protocol stack against logging performed in the simulator. The final demand on the logging functionality is support for filtering out log messages originating from any given node.

3.2 Important design decisions

The design decisions made during the lifetime of a project are many, but some have a greater impact on the result than others and are of course more important because of this. These important design decisions are the focus of this section.

Originally, the plan was to use some free simulator environment and add functionality on top of it. A study of a simulation kernel called WARPED [14] was carried out. It was decided however, that the kernel was unnecessarily complex to use. Therefore, a decision was taken to develop the simulator from scratch. This alternative meant that the design process would be more flexible, because it was not dependent on already implemented structures.

The next thing to decide on was the overall structure of the simulator. Two main approaches were considered. The first was to keep the information needed to simulate a DTM node in a single process, and then interconnect such nodes as in a real network. The second
3.3 Overall design structure

The approach was to have some kind of central process that nodes connect to for holding information. The topology information of the network is then located in the information process and not distributed as in the former approach.

Using the first approach described above may seem as the most logical solution, but the second approach makes many things easier, for example, no major effort need to be put into synchronising the nodes. Each connected node is simply handled in a first come first served manner by the central information process. Another thing to consider is the task of introducing errors, which is easier to do in a centralised solution.

The chosen structure was the centralised. However, there was one more design decision to be made about this kind of structure. The question was whether to keep data structures in the node and simply have the topology information in the central information process or to put as much information as possible in the central information process. The latter alternative was chosen, because it meant less inconsistency problems to solve than in the former.

Another important choice in the design was the decision to use TCP/IP sockets for the communication between nodes and the central information process. This made it possible to run nodes on more than one computer, which might be desirable when simulating a larger number of nodes. It also meant that different byte orders could be checked because nodes could be run on different hardware platforms.

The choice of programming languages was easy. Nodes ought to be implemented in C because the protocol stack should be able to link together with them. In the information process, the structures needed would be easily and well expressed by an object-oriented design. For this reason, C++ was chosen for that part of the simulator.

3.3 Overall design structure

This section briefly describes the overall structure of the simulator. This is done so that the reader will be familiar with the main parts of the design when reading the more detailed descriptions in later sections.

In figure 3.1, the main parts of the design are shown. There are three parts, the Controller, the Net Core and the Node Process.

The simulator is built so it can be run distributed on several computers. To support this, binaries that do the start-up are executed on each computer. The binaries started are the Controllers.

The broken lines in figure 3.1, symbolise the start-up that is performed by Controller one and Controller two. The Controllers do the start-up and keep a virtual time vector for each machine, which the Node Processes fetch via a socket.

The Node Processes in the figure can be seen as the DXL modified so that it communicates with the Net Core rather than hardware. This communication is symbolised by the lines drawn between the Node Processes and the Net Core. A Node Process is lightweight in means of states concerning hardware because all such states are located in the Net Core. This means that all commands introduced via the DXL is sent down to the Net Core, which then modifies its internal structures and replies to the sending Node Process. The Node Processes is the part
of the simulator where the protocol stack may be integrated.

The *Net Core* is much more advanced than the *Node Processes*. Each node is represented by special data structures, the structures are configured via a configuration file and the *Node Processes* do not know which node they represent until connected to the *Net Core*. It also has functionality for performing logging of events and introducing simulated errors. The simulated errors can be of any kind, there is for example one kind of error that simulates a broken fibre between two nodes.

### 3.4 The Controller

This section describes the *Controller* and the *TimeInfo*. In figure 3.2 the design for these parts can be seen.
3.4 The Controller

3.4.1 Controller

The Controller is the first part of the simulator that starts. It reads the configuration file, launches the correct number of Node Processes and the Net Core if supposed to. It also holds the Virtual Time, which is discussed in section 3.4.2. The Node Processes and the Net Core are connected to the Controller through sockets.

3.4.2 TimeInfo

TimeInfo is used to make it easy to keep track of commands sent between the Node Processes and the Net Core. It consists of two parts, a TimeVector which is described below and a TimeVal. The TimeVal is a C type called struct timeval, that holds the real time when a TimeInfo was created.

The Virtual Time is a counter increased every time an event is sent in the machine. A TimeVector in each Node Process and in the Net Core is used to hold the Virtual Time for the different machines. It is updated every time a Node Process or the Net Core receives or sends an event.

The TimeVector is used for synchronising the logged events sent in the simulator. And it is also used when a simulation is re-run, so that the events will be done in the correct order and the simulator is in the correct state. The TimeVector is an important part of the simulator and is very useful if the simulator is distributed on several machines.

If there are two machines in a simulation, there will be two Virtual Times and the TimeVector will have two elements. Each time a Node Process or the Net Core sends an event they will get a new sequence number from the Controller via the TimeInfo, and update their TimeVectors.

<table>
<thead>
<tr>
<th>Event</th>
<th>Before send</th>
<th>After send</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node 1</td>
<td>Node 2</td>
</tr>
<tr>
<td>Node 1 -&gt; Net</td>
<td>&lt;0, 0&gt; &lt;0, 0&gt; &lt;0, 0&gt;</td>
<td>&lt;1, 0&gt;</td>
</tr>
<tr>
<td>Node 2 -&gt; Net</td>
<td>&lt;1, 0&gt; &lt;0, 0&gt; &lt;1, 0&gt;</td>
<td>&lt;0, 1&gt;</td>
</tr>
<tr>
<td>Net -&gt; Node 1</td>
<td>&lt;1, 0&gt; &lt;0, 1&gt; &lt;1, 1&gt;</td>
<td>&lt;2, 1&gt;</td>
</tr>
<tr>
<td>Node 2 -&gt; Net</td>
<td>&lt;2, 1&gt; &lt;0, 1&gt; &lt;2, 1&gt;</td>
<td>&lt;0, 2&gt;</td>
</tr>
<tr>
<td>Node 1 -&gt; Net</td>
<td>&lt;2, 1&gt; &lt;0, 2&gt; &lt;2, 2&gt;</td>
<td>&lt;3, 1&gt;</td>
</tr>
<tr>
<td>Net -&gt; Node 2</td>
<td>&lt;3, 1&gt; &lt;0, 2&gt; &lt;3, 2&gt;</td>
<td>&lt;4, 2&gt;</td>
</tr>
</tbody>
</table>

Table 3.1: TimeVector example

An example of how the TimeVector is used is shown in table 3.1. The Nodes are Node Processes and the Net is the Net Core. Node 1 and the Net run on one machine and Node 2 runs on another. Every send and receive task is completed before the next one is handled.
3.5 The Node Process

This section presents and explains the Node Process part of the simulator. In the first subsection the Node Process is described. The second subsection describes the DXL, and finally in the last subsection the other parts of the Node Process are described. In figure 3.3 the design of the Node is shown, and in figure 3.4 the DXL can be seen.

3.5.1 Node Process

Node Process is the part of the simulator that simulates a real node, with a complete DXL interface. There can be any number of Node Processes in the simulator. Each Node Process talks to the Net Core via a socket on which it will send commands. These commands have been called through the DXL interface. The Node Process will also receive commands and callbacks from the Net Core. For example, if Node Process should shutdown or if it receive a data packet sent to it via DCAI, it will receive a corresponding command or a callback for each case.

The protocol stack will be compiled together with the rest of the Node Process. This can be done because the DXL interface used is exactly identical to the one used in a real DTM node.
3.5 The Node Process

3.5.2 DXL (DTM Switching Layer)

The DXL is a very important part of the simulator because the protocol stack uses it as the interface when, for example, controlling the hardware, setting up channels and sending data.

The DXL consists of four different parts, DXI, DCAI, HEAL and DSYNC. These parts are explained more thoroughly in section 2.2. The different parts of the DXL can be seen in figure 3.4.
3.5.3 SimDXLManager

*SimDXLManager* is used to hold the variables that are used in different parts of the *Node Process*. The design of the SimDXLManager can be seen in figure 3.4. It is also handled in section 4.1.4.

![Diagram of SimDXLManager](image_url)

Figure 3.5: Net Core - Design
3.6 The Net Core

In this section, the classes of the Net Core are presented and explained. Most classes, except the events can be found in figure 3.5.

3.6.1 NetProcess

NetProcess is the main class of this design, and it handles two kinds of basic event classes. The first is used for events representing simulated errors introduced manually or via the configuration file. The second kind is used for encapsulating the different commands received from Node Processes. These two event classes are called ErrorEvent and CommandEvent, respectively.

The events are interpreted by NetProcess, which then changes the underlying data structures representing the DTM network according to the currently interpreted ErrorEvent or CommandEvent.

This class is also responsible for all communication with the Node Processes and initialisation of the simulator at start-up. Furthermore, NetProcess has the main loop of the Net Core. Its relations to other classes can be seen in figure 3.5.

3.6.2 NodeUnit

The NodeUnit class is the structure on which NetProcess perform operations that can be performed on a real node. It has a number of Boards, one NodeController, one HEAL and a number of Channels.

There are as many NodeUnits as there are Node Processes, that is, each Node Process maps to one NodeUnit. One could say that a NodeUnit represents the DXL in the Net Core. Almost all methods found in the real DXL can be called as methods on this class.

It is possible to set errors on a NodeUnit. Errors are received from NetProcess and are added and removed dynamically.

3.6.3 HEAL

HEAL is a storage class for event and alarm receivers, the receivers are used when an event or alarm should be sent to a Node Process. Events and alarms are sent to indicate that something has happen on a certain switch, board or interface. The receivers contain information necessary when sending the event or alarm to a Node Process.

3.6.4 Channel

The Channel class represents a DTM channel. It holds the channel’s RX interface and a list of its TX interfaces. To each interface a slotlist is associated, the size of the slotlist is the same for all interfaces and determines the bandwidth of the Channel. Channel has a list of
cmi senders and a list of cmi receivers. The cmis are used to enable more than one receiver and sender on a channel. Channels are added and created dynamically in NodeUnit.

3.6.5 Board

Board is the class representing a board in a real node. It can have a number of Interfaces connected to it. The number is configured via the configuration file. This class handles errors in a way similar to the one found in NodeUnit.

3.6.6 InterfaceBase

InterfaceBase is the abstract base class of NodeController and Interface. It has methods for doing slot manipulations, used for setting up Channels and more. It is also possible to get the interface id and the bandwidth left on the TX and RX interface, respectively.

3.6.7 NodeController

The NodeController is the representation of the interface in a real node that is not connected to any fibres. It is on the NodeController the software in a node runs. It can be seen as the interface used to send data to or receive data from the protocol stack.

3.6.8 Interface

The Interface class represents a physical DTM interface. It has a number of slots that can be allocated for use by Channels and each Interface has knowledge of which NodeUnit, Board and Interface its upstream fibre is connected to. It is necessary to have this information when sending data over the network. The Interface has an error-handling scheme similar to the ones of NodeUnit and Board.

3.6.9 Slot

A Slot is an abstraction of a DTM slot in a real node. It can be in a number of states, which are used to keep track of what switching mode a Slot is in. The class holds a reference to the channel that sends data on it. This reference is necessary because we need to do some backward lookup when we find out which nodes should receive data when sending data over the simulated network.

3.6.10 Events

When one has different types of command formats, it might be a good idea to treat them in a common way. In Net Core commands from the Node Processes are represented by SimPackets and simulated errors as strings. To handle these different types in a big loop
would not be a good solution and therefore a solution based on events has been chosen. The idea is simple, all kinds of command formats are converted to one format, the event format. However, events can be in different classes, but the basic structure is the same. EventCreator creates the events, and then some of them are stored for later execution in EventHandler, while others are executed immediately. One kind of events, ErrorEvents, can be saved to file using EventHistory. The different event classes are discussed and explained below, EventCreator, EventHandler and EventHistory will be treated later in this chapter.

![Event Class Relations](image)

Figure 3.6: Events - Class relations

- **Event**

  Event is the abstract base class of ErrorEvent and CommandEvent. Common members and operations for basic events are defined in this class. There are currently two such members. The first member holds the time when the event is supposed to be executed. The second holds the type of event, that is, ErrorEvent or CommandEvent. See figure 3.6 for the event class relations.

- **ErrorEvents**

  ErrorEvent is the base class for the events associated with different kinds of errors that can be introduced during execution. The class contains the string used for creating the event and the type of ErrorEvent. The string is currently publicly accessible but the ErrorEvent can be retrieved by a method call. Below the now available error event classes are described.

  - **FibreErrorEvent**

    This class is an event used for simulating a fibre cut in a network and inherits ErrorEvent. The FibreErrorEvent class has three members. The first two are used to reference the Interface to set the error on, one board and one interface identifier. The third member states at what time to remove the error. This event
will result in a fibre error being set on an Interface. FibreErrorEvent should be applied on the RX interface, that is, the downstream interface.

- FibreErrorReleaseEvent
  When a FibreErrorEvent is executed, it generates another event for the removal of the error it introduced. This type of event is called FibreErrorReleaseEvent.

- HardwareResetEvent
  To simulate a hardware reset of a node, HardwareResetEvent is used. At the execution of this event, a message is sent to the Node Process to prepare a reset. When the Node Process receives this message, it sends an acknowledgement to the Net Core and starts waiting for a restart message. When the Net Core receives the acknowledgement from the Node Process, it resets all data structures related to the Node Process and sends a restart message to the it. Upon receipt of the message the Node Process restarts itself and continues the interaction with the Net Core.

- SoftwareStallEvent
  If the software stalls in a node, the hardware may still be able to switch data. To simulate this case, the SoftwareStallEvent is used. When the event is executed, the Net Core sends a message to the Node Process containing the amount of time to stall the node. The Node Process stalls the node and then sends an acknowledgement to the Net Core. Net Core then resets the data structures for the Node Process and restarts it in the same way as in the HardwareResetEvent case.

- CommandEvent
  CommandEvent is the super class of events used for holding commands received from the Node Processes. It has three members; timing info, from which node the command origins and a session id. All three members are received from the calling Node Process. More specialised events that inherit CommandEvent hold the specific data for those events. The different command event classes are described below. However, the DXL commands have been grouped together and are described under DXL Events.

  - DXL Events
    This section covers all methods in the DXL, that is, commands from DXI, DCAI, HEAL and DSYNC [5, 11, 13, 12] received by the Net Core and transformed to events. To write about all methods in the DXL is not necessary because they have common properties, which are described in this section.
    Each event holds the arguments of the DXL method corresponding to it. The events are processed by NetProcess and information held by the events is used to change the node data structures of the Net Core.

  - LogEvent
    LogEvent is the carrier of log messages received from the Node Processes by the Net Core. It also carries node id and time info from the sending Node Process. When the event is received, it is executed and the received message, node id and time info is written to file by the Log.
3.6 The Net Core

- HardwareResetNodeReplyEvent
  This event represents the acknowledgement on the first reset message sent to the Net Core when executing a HardwareResetEvent, discussed above.

- SoftwareStallNodeReplyEvent
  SoftwareStallNodeReplyEvent represents the acknowledgement on the stall message sent to a Node Process when executing a SoftwareStallEvent, which was discussed above.

3.6.11 EventCreator

EventCreator is responsible for creating all kinds of events. This class has two ways of creating events. In the first way, it calls the method createEvent with a string, and an ErrorEvent will be created. The second way, createEvent is called with a SimPacket sent from any of the Node Processes, at present means that a CommandEvent or a LogEvent will be created. In the future, it might be possible to receive some other kind of events from the Node Processes.

3.6.12 EventHandler

EventHandler keeps track of when and in what order events should be executed. Before the start of a simulation, events are added to the EventHandler in which they are sorted in order of execution. By calling a start method its internal time will be set to the value received from the standard C library function gettimeofday. Events will be ready for execution when EventHandler’s internal time is equal to its execution time.

3.6.13 EventHistory

EventHistory is a class that saves Events. The type of Events saved are those that has been introduced manually or via the configuration file, that is, ErrorEvents. They are saved because it should be possible to repeat the same sequence of events in following executions.

3.6.14 Config

Config reads the network configuration from file and creates structures that describe the network. Furthermore, it reads and stores the configured errors if any. NetProcess fetch information from Config when creating its structures of nodes, boards, interfaces etc. An example configuration file can be found in appendix B.

3.6.15 Log

To make it possible to follow which and in what order commands have been executed Log is used to save this information about commands to file. The information written to file is the id
of the Node Process the command relate to, what origin the command has, time information and a text message.

### 3.7 The configuration file

To make it easy to configure the simulator, a common configuration file is used for the Net Core and the Controllers.

In appendix B, an example configuration file can be found. It has three sections, the first section, the controller section, contains the file names of the node binaries to be started by the controllers and the location of the Net Core. The second section describes the network, a specified network can have a number of nodes, and each node can have a number of boards with a number of interfaces attached to them. To define the connections between nodes, each interface defines the upstream interface to which it should be connected. Nodes, boards and interfaces all have a configuration field, which specify their characteristics. The last section is used to specify the different kinds of errors that should be introduced during execution of the simulator.

The configuration file is scanned and parsed by code generated from Flex [6] and Bison [1], respectively. The scanner and parser structures are defined in two text files. C code are then generated from these files and linked together with the rest of the source. The configuration file is not compiled into the source but can be changed dynamically. The scanner tries to match given patterns against the text in the configuration file and sends tokens to the parser when a pattern is found. The parser identifies token patterns and for each pattern found, code can be executed. To identify the patterns a Bison grammar is specified, it can be seen in appendix C.

The Bison grammar for the configuration file is built up by a number of rules, each defining a part of the configuration file. Some rules are recursive. Recursion is necessary if it is desirable to have, for example, different numbers of interfaces.

A rule consists of two types of symbols, non-terminal and terminal. The non-terminal symbols are the lower-case strings and can be thought of as a name of a rule. Terminal symbols are returned from the scanner function and can be one of two types. The first type, called a token, is the upper-case version of a string in the config file. For example, if the scanner reads 'Interf ace' it returns an INTERFACE token to the Bison parser. The second type is the 'character literal tokens', which are single characters that separate tokens. For more information on the Bison grammar, see [1].

### 3.8 The communication protocol

This section describes SimPacket, SimPacketQueue and CommHandler. In figure 3.7 the different parts of the Communication Protocol can be seen.
3.8 The communication protocol

A SimPacket is a container for the arguments and data that are sent in a call via the DXL interface. A SimPacket also knows the nodeId, which identifies from which Node Process the SimPacket originated. It also holds a taskId so that when a Node Process gets a callback it knows for which command it was referenced.

A TimeInfo is also included in the header of the SimPacket. The TimeInfo is used for logging and backtracking of commands. Information on the TimeInfo can be found in section 3.4.2.

SimPacket also has different specialised methods, which are used for copy complex data types to and from a SimPacket’s payload.

3.8.2 SimPacketQueue

SimPacketQueue is a FIFO queue. It is used for holding incoming SimPackets sent from the Net Core when a Node Process is busy sending to the Net Core.

3.8.3 CommHandler

The CommHandler is used to establish connections and to send and receive data. It has, for example, a sendSimPacket operation and a readSimPacket operation. These operations are used to send and receive SimPackets.

The methods supplied by the CommHandler are designed to make operations, for example send and receive operations, easy to use. These operations would, if the CommHandler did not exist, have been dependent on the standard C library, which can be quite complex and therefore desirable to avoid.

The procedure to send and receive a SimPacket can be seen in figures 3.8 and 3.9. In the latter figure a Node Process tries to send a SimPacket at the same time the Net Core begins to send. The CommHandler in Node Process then has to queue the SimPacket sent from the Net Core onto the SimPacketQueue before it sends its own SimPacket.
Figure 3.8: Send algorithm - CommHandler

Figure 3.9: Send algorithm - Collision detect - CommHandler
If a Node Process and the Net Core tries to send at the same time.
Chapter 4
Implementation

This chapter describes the most important parts of the Node Process and Net Core implementations. First, implementation details specific to the Node Process are described. Second, the Net Core is handled.

4.1 The Node Process

This section describes the implementation details of the Node Process. In section 3.5 the design of the Node Process is handled.

4.1.1 The Node Process main method

The main method of Node Process is in the Node (figure 3.3). At startup the Node Process connects to the Controller and the Net Core. The Node Process can also be restarted with a system exec call from dxl_main. If this is done, all the necessary connections are already open and the Node Process does not have to do this again.

At startup the SimDXLManager (see section 4.1.4) is also created and the different variables in SimDXLManager are set to values specific to the Node Process, variables like nodeId, thisMachine etc. The SimDXLManager design and purpose are explained in section 3.5.3.

The SUPV’s block, a block is an information container for specific modules in the system, is also created at startup. See section 2.3 for more information about SUPV. Finally a call to superv_start is done, and then the SUPV has all the control of the Node Process.

The SUPV was already implemented, so it is just a part of the system that the simulator uses to be compatible with Net Insight’s DTM current system.
4.1.2 Dxl_main

In *dxl_main* there are methods to handle DXL block commands that is called by the *SUPV*, take care of a node restart, execute SimPackets from the SimPacket queue and handle commands called from the *Net Core*. The calls from *SUPV* can for example be a *LOCAL_INIT* call, which is used to initialise the *DXL*, or a *TRAFFIC* call, when the *DXL* is started.

The handling of calls from the *Net Core* is done by a method called *simDXLHandleFd*. This method is called every time the *SUPV* notices an activity on one of the file descriptors registered in it by the *DXL*. These calls are often callbacks from different commands sent via the *DXL* in the *Node Process* to the *Net Core*. Callbacks are methods called whenever something special has happened, for example, when incoming data has been received and should be delivered to the registered receiver.

4.1.3 DXL

The *DXL* is simply a wrapper for the *DXL* interface. The easiest way to see it is that a command is called in the *DXL* and it then sends the command in a SimPacket to the *Net Core*. The call are then returned with a correct return value, which usually comes in a reply from the *Net Core*.

4.1.4 SimDXLManager

The *SimDXLManager* is a container for global variables such as *nodeId*, *netSocket*, *controllerSocket* and *supvp*. *SimDXLManager* consists of a struct *SimDXLManagerStruct* that holds all these variables. There is also *create, delete, start, stop and print* methods available for managing the *SimDXLManager*.

4.2 The Net Core

This section describes the implementation details of the *Net Core*. In section 3.6 the design of the *Net Core* is handled. First the NetProcess main loop is handled. Second, the different ways to handle events are presented. Finally, the algorithm used when sending data between nodes is explained.

4.2.1 The NetProcess main loop

In the main function of the *Net Core*, a *NetProcess* object is created. At creation time this object reads the configuration file and creates a number of objects necessary for the *Net Core* to function. At this time, no *Node Processes* are connected to the *Net Core*.

After creation, the main function calls the setup method on the *NetProcess* object. Here all *Node Processes* connect to the *Net Core*, and when this is done the method returns. Now
4.2 The Net Core

the run method is called on the NetProcess object. This is the actual main loop of the Net Core. The course of events in the loop can be seen in figure 4.1.

In the run method, a call to the EventHandler is made to start the timer used to activate ErrorEvents at the right time. Messages are then sent to each registered Node Process. Now a sequence is repeated: first, the time to next ErrorEvent is fetched. Second, a call to select is made, the select function returns when there is data on a socket or when the time to next event has passed. If there were data on one or more sockets, each socket is handled by a call to handleFd. Situations when this function is called are described below. After this, ErrorEvents are handled, which is also described below.

Figure 4.1: NetProcess main loop
4.2.2 Event handling

The NetProcess can handle Events in three different ways. First, it can handle them by receiving CommandEvents from a Node Process. Second, it can handle the case when a user wants to manually execute an ErrorEvent, and in this case, the event is received from a socket from a telnet session. Finally, the case when an ErrorEvent has been scheduled to execute at a certain time via the configuration file. These three cases will be explained more thoroughly in the following subsections.

![Figure 4.2: CommandEvent from a Node Process](image)

- CommandEvents from a Node Process

Figure 4.2 describes the behaviour of three actors when they handle a CommandEvent. The first actor, Node Process, is a separate process while the two other actors NetProcess and NodeUnit execute in the same process. Node Process initiates the command sequence by sending a simDXLFunctionCmd via a socket connection. NetProcess handles the data by calling the method handleFd that reads a SimPacket from the socket. After that, a call to createEvent is made. The result from the call is a pointer to an Event. This Event pointer is used when calling processEvent. In processEvent the Event pointer is type casted to its actual type, for example DxiChannelCreateEvent. The correct NodeUnit object is then found and a method corresponding to the event type, simDxiFunction, is called on the object. ProcessEvent then sends the method return value back to the Node Process.
4.2 The Net Core

- **Manual ErrorEvents**
  
  To introduce ErrorEvents manually, a telnet connection can be established to the Net Core and the commands are sent to it as text. Figure 4.3 shows the command sequence performed when data is present at the telnet socket.

  Incoming sockets are handled by handleFd. If the incoming socket is the one used when receiving manual ErrorEvents the special function handleManualFd is called. A call to createEvent creates the ErrorEvent. Now processEvent is called with the created event, here the ErrorEvent is added to a NodeUnit. An example that more thoroughly explains the addition of an ErrorEvent can be found below.

- **Pre-scheduled ErrorEvents**

  The handling of pre-scheduled ErrorEvents is done in the main loop of the NetProcess, see Figure 4.1. In the figure the handling of the ErrorEvents starts at the call to putEventsOnExecQueue, this call makes it possible to fetch all ErrorEvents ready for execution. This is done by calling getEvent to get the event and then executing it by calling processEvent with it. These two operations are carried out until no more ErrorEvents are scheduled for execution.

  An example of how the addition of an ErrorEvent is done after the call to processEvent can be seen in figure 4.4. First, the NodeUnit to set the error on is fetched, then a
4.2.3 Data sending algorithm

In a real DTM network data is sent between nodes as bits over a fibre. In the solution implemented in the simulator, data is passed between nodes by using a recursive data-sending algorithm. It is recursive because this was the easiest way to implement it. Of course, in a later version of the program, another algorithm could be implemented. This section will first give some background to make it easier for the reader to understand the algorithm, then it will be explained in detail.

---

Figure 4.4: Example of an ErrorEvent addition

Board and finally an Interface. The ErrorEvent is then set on the Interface by calling the method setError. After this, an ErrorEvent to remove the added event after a certain time is added to the EventHandler.
4.2 The Net Core

- Background

In figure 4.5 a simple DTM network is depicted, two rings interconnected to each other by a switch. If Node N1 wants to send data to N4, a unicast channel is first established from N1 to N4 via N3. The channel set-up is explained by describing what happens in each node.

- N1
  Node N1 sets up a channel from the interface NC out on interface I1, if data is sent from the NC it will now be sent out on the fibre ring.

- N3
  Here data has to be switched between interface I1 and I2. This is done by setting up a channel between these interfaces. Data on the ring will now be switched via I1 and I2 to the other ring.

- N4
  In this node, data should be received. This is done by setting up a channel from interface I1 to the interface NC. Data on the ring will now be sent up to the node.
Data can now be sent from node N1 and be received in node N4. Later in this section this set-up will be used to explain how data is sent over the network.

It is necessary to know a bit about the data structures used in the Net Core to understand the algorithm. In figure 4.6, the data structures for node N1, N3 and N4 are shown. As can be seen, each Interface holds information on which NodeUnit, Board and Interface it is connected to, hence it is possible to retrieve next Interface in a network. There is also a data structure in each Interface, by which it is possible to fetch the correct channel for the data stream coming from the ring into the interface.

```
1:1_next_if = 3:1:1
3:1.2_next_if = 4:1:1
```

Figure 4.6: Send data structures

- **The Algorithm**

The main parts of the algorithm is shown in figure 4.7, please refer to this figure and figure 4.6 when reading the algorithm description.

When transmitting data from N1 the correct NodeUnit is first found and the method simDcaiSend is called on it. This call returns information on which Interfaces to send data on. The call to sendToNode is only executed if a loopback situation is present, and then data is sent to the Node Process that executed simDcaiSend.

After this, a call to the method distributeData, which is the method that may be called recursively, is performed. A list with references to which NodeUnit, Board and Interface the current NodeUnit is connected, is passed in the call. Now each element in the list is used to send data to the NodeUnits that should receive data, in this example the switch, NodeUnit N3. By calling getAssociatedChannel on this NodeUnit, the channel necessary for retrieving the outgoing Interfaces is fetched. This is the channel held by the Interface in figure 4.6. Calling getSendVector with this channel gives us a similar list as before, this call also tells whether to let data bypass the interface I1 or not. This is not the case here, because the NodeUnits N1 and N2 is not set up to receive data.

Now a call to distributeData, recursively with the retrieved list, is performed. The call now works on N4, and data is received in the NodeUnit.
4.2 The Net Core

Figure 4.7: Data send algorithm
Chapter 5

Conclusions

In this thesis, the demands on, the background on and the goals for our thesis have been described. The reader has also been given a short introduction to the DTM technique. Important design decisions made during the design were discussed. The design and chosen parts of the implementation have been presented.

The constructed simulator is, when writing this paper, practically finished. There is some testing left, which has not been done yet and some implementation that could not be done because some specifications were unclear. The goals specified when this work started have been reached as planned.

During this work, not many problems have arisen. Some specifications changed during the work, and some were not completely finished. These problems however, depend on that Net Insight is in the process of developing new products and should be accounted for in the time schedule.

In perspective, it probably would have been a good idea to reduce the implementation part of the thesis. This way the thesis would have been finished a bit earlier.

Of course, there are always improvements and extensions to be made, for example, it should be possible to simulate other kinds of errors than today. Some kind of primitive that makes it possible for the stack to synchronise commands sent to the DXL on when a certain error has occurred in the simulator would also be a useful feature.

Hopefully, the simulator will serve as a very good support when testing the DTM protocol stack and make it even better than today.
Bibliography

[1] Bison, the YACC-compatible Parser Generator,


    INTERNAL DOCUMENT, Document NID 73 version 0.2.0, Mars 1999.

    INTERNAL DOCUMENT, Document NID359 version 0.1, December 1999.

    INTERNAL DOCUMENT, Document NID328 version 0.6, November 1999.

[6] Flex, a fast lexical analyser generator,

    INTERNAL DOCUMENT, Document NID34 version 0.10, January 1998.


[10] Net Insight Knowledge Bank,

    INTERNAL DOCUMENT, Document NID386 Rev 0.5.0, Mars 2000.

[12] Rangne, Göran “DTM Node Synchronisation Control
    Interface Specification - DSYNC”,
    INTERNAL DOCUMENT, Document NID479 Rev 0.1.0, Mars 2000.

[13] Roth, Björn. “DTM hardware equipment abstraction layer (HEAL) specification”,
    INTERNAL DOCUMENT, Document NID401 version 0.4, February 2000.
[14] WARPED, a Time Warped simulation kernel,
    <URL: http://www.ececs.uc.edu/paw/warped/>. 

Appendix A

Word list

- **DTM**
  Dynamic synchronous Transfer Mode is a networking technique based on circuit switching.

- **DXL**
  DTM Switching Layer (DXL) can be seen as the glue between the DTM protocol stack and the switching hardware in a DTM Anode. It provides four different interfaces, DCAI, HEAL, DXI and DSYNC.

- **DCAI**
  The DTM Channel Adaption Interface (DCAI), handles data send and receive operations in a DTM node.

- **HEAL**
  Hardware Equipment Abstraction Layer (HEAL), is used to interact with the DTM hardware in a DTM node.

- **DXI**
  DTM Switch Interface (DXI), is used for channel control, for instance, to create channels and reserve bandwidth on them.

- **DSYNC**
  DTM node Synchronisation Control interface (DSYNC), is used when performing operations on sync sources in a DTM node.

- **Rx interface**
  The interface in a DTM node where data is received into a channel.

- **Tx interface**
  The interface on which data is sent out from a channel.
• **Channel**
  The virtual connection between sender and receiver.

• **Slot**
  A timeslot, the smallest part of the transmission part in the DTM system. Channels consist of slots. The size of a slot is 64 bits.

• **cmi**
  An id used to multiplex several senders and receivers on one channel.

• **SUPV**
  A part of the DTM system that keeps control of the different parts of the system.

• **Bison**
  A program that generates source code that parse for example configuration files.

• **Flex**
  A program that generates a scanner, can for example be used together with Bison to parse a configuration file.
Appendix B

Configuration file

[CORROLLER]
NetProcess = computername:7500;
NetStarter = 0;
Controller 0 = 0;
Controller 1 = 1;
Controller 2 = 1;
[END_CORROLLER]

[NET_CONFIG]
Net {
  Node 1 {
    Config {
      NodeController {
        rx_num_slots = 100
        tx_num_slots = 100
      }
      Board 1 {
        Config {
          maxNumOfIfffs = 2
          version = 1
          type = 1
          async = 1
        }
        Interface 1 {
          Config {
            rx_num_slots = 100
            tx_num_slots = 100
            async = 0
            downstream_if = 2:1:1
          }
        }
        Interface 2 {
          Config {
            rx_num_slots = 100
            tx_num_slots = 100
            async = 1
            downstream_if = 1:2:3
          }
        }
      }
    }
  }
}
Board 2 {
    Config {
        maxNumOfffs = 2
        version = 1
        type = 1
        async = 1
    }
    Interface 3 {
        Config {
            rx_num_slots = 100
            tx_num_slots = 100
            async = 1
            downstream_if = 1:1:1
        }
    }
}

Node 2 {
    Config {
    }
    NodeController {
        rx_num_slots = 100
        tx_num_slots = 100
    }
    Board 1 {
        Config {
            maxNumOfffs = 2
            version = 1
            type = 1
            async = 1
        }
        Interface 1 {
            Config {
                rx_num_slots = 100
                tx_num_slots = 100
                async = 1
                downstream_if = 1:1:2
            }
        }
    }
}

[ERROR_CONFIG]
Interface 1:1:1 = IF_FIBER_ERROR:1:30;

[END_ERROR_CONFIG]
Appendix C

Bison grammar

cfg: BEGIN_CONTROLLER controllerinfo
    BEGIN_NET_CONFIG netstmt
    BEGIN_ERROR_CONFIG errorsequence
    
controllerinfo: NET_PROCESS '=' STRING ':' VAL ';'
    NET_STARTER '=' VAL ';' controllersequence ';'

controllersequence: controllerstmt
    | controllersequence ';' controllerstmt

controllerstmt: CONTROLLER VAL '=' VAL

netstmt: filenamesequence ';' NET '{' nodesequence '}' '}'

filenamesequence: filenamestmt
    | filenamesequence ';' filenamestmt

filenamestmt: logfilename
    | eventhandlerfilename

logfilename: LOG_FILE '=' STRING

eventhandlerfilename: EVENT_HISTORY_FILE '=' STRING

nodesequence: nodestmt
    | nodesequence '}' nodestmt

nodestmt: NODE
    | VAL '{' nodeconfigstmt nodecontrollerstmt boardsequence '}'

nodeconfigstmt: CONFIG '{' nodedatasequence '}'

nodedatasequence: /* empty statement*/

nodecontrollerstmt: NODE_CONTROLLER
    VAL '{' nodecontrollerdatasequence '}'

nodecontrollerdatasequence: RX_NUM_SLOTS '=' VAL
    TX_NUM_SLOTS '=' VAL

boardsequence: boardstmt
    | boardsequence '}' boardstmt

boardstmt: BOARD
    VAL '{' boardconfigstmt interfacesequence '}'

boardconfigstmt: CONFIG '{' boarddatasequence '}'

boarddatasequence: MAX_NUM_OF_IFS '=' VAL
    VERSION '=' VAL
    TYPE '=' VAL
    ASYNC '=' VAL

interfacesequence: interfacestmt
    | interfacesequence '}' interfacestmt

interfacestmt: INTERFACE
    VAL '{' interfaceconfigstmt

interfaceconfigstmt: CONFIG '{' interfacedatasequence '}'

interfacedatasequence: RX_NUM_SLOTS '=' VAL
    TX_NUM_SLOTS '=' VAL
    ASYNC '=' VAL
    downstreamstmt

upstreamstmt: DOWNSTREAM_IF '='
    | DOWNSTREAM_IF '=' VAL ':' VAL ':' VAL

/*Error statements start here*/
errorsequence: /* empty */
    | errorsequence1
errorsequence1: errorstmt
    | errorsequence1 errorstmt

errorstmt: STRING ';'