Data Transmission in the Antares Data Acquisition System

Master's Thesis

Joris van Rantwijk
joris@deadlock.et.tudelft.nl

March 2002

Supervision:

D.H.J. Epema (TU Delft)

W.P.J. Heubers (NIKHEF)

R. van Dantzig (NIKHEF)

Delft University of Technology
Faculty of Information Technology and Systems
Parallel and Distributed Systems Group

National Institute for Nuclear Physics
and High Energy Physics, Amsterdam

Afstudeergegevens

Spreker :	Joris van Rantwijk `<joris@deadlock.et.tudelft.nl>`
Titel :	Data Transmission in the Antares Data Acquisition System
Datum :	woensdag 10 april 2002 om 10:30
Plaats :	Faculteit ITS, Mekelweg 4 (Delft), zaal HB09.130/140

Opleiding :	Technische Informatica
Specialisatie:	Parallelle en Gedistribueerde Systemen
Opdrachtgever :	Nationaal Instituut voor Kernfysica en Hoge Energie Fysica (NIKHEF)
Periode :	mei 2001 – februari 2002

Afstudeercommissie:
prof.dr.ir. H.J. Sips (TU Delft)
ir.dr. D.H.J. Epema (TU Delft)
ing. W.P.J. Heubers (NIKHEF)
dr. R. van Dantzig (NIKHEF)

Samenvatting:
Het doel van het ANTARES project is het waarnemen van hoog energetische neutrino's. Hiervoor wordt een detector gebouwd op de bodem van de Middellandse Zee, op een diepte van 2,4 km. Deze detector zal enorme hoeveelheden digitale gegevens produceren – ongeveer 1 GByte per seconde. Om deze gegevens te verwerken, worden ze via een netwerk naar de kust getransporteerd en daar verdeeld over een groep computers. Het is noodzakelijk om het patroon van data transmissies zo te organiseren dat de beschikbare netwerk capaciteit efficiënt wordt benut. In het kader van dit onderzoek is daarvoor een methode ontwikkeld. Een simulatie van het netwerk wordt gebruikt om het effect van deze methode te beoordelen. De software die verantwoordelijk is voor het ontvangen van gegevens aan de kust, is ontworpen en gedeeltelijk geïmplementeerd. De werking wordt getest in een proefopstelling van computers in een netwerk. Aan de hand van deze tests wordt geschat hoeveel computers nodig zijn voor de verwerking van de detector gegevens.

Abstract:
The goal of the ANTARES project is to look for high energy neutrinos. This is done by means of a detector, which is being built on the bottom of the Mediterranean Sea at a depth of 2.4 km. This detector will produce very large amounts of digital data – around 1 GByte per second. In order to process these data, they are transported through a network to shore. There, they are distributed over a cluster of computers. The pattern of data transmissions must be organized such that the available network bandwidth is used efficiently. In this report, we describe a method to achieve this. A simulation of the network has been used to evaluate our method. Software for receiving data on-shore has been designed and partially implemented. The funcionality and performance of this software has been evaluated in a test environment with networked computers. Based on these tests, we estimate the number of computers needed to process the detector data.

Preface

This thesis describes the work I have done for my graduation from Delft University of Technology in the field of computer science. For one year, I have worked on the data acquisition system of the ANTARES detector. This work was done in the computer technology group at NIKHEF, under the supervision of Wim Heubers and René van Dantzig. From the TU Delft, my work was supervised by Dick Epema of the Parallel and Distributed Systems group.

I greatly enjoyed working at a physics institute. The mysterious theories of particles and forces have always fascinated me. Now, from discussions with physicists and engineers, I finally have come to understand these things a little better.

I would like to thank my supervisors for their guidance and support. Furthermore, I would like to thank the people of the ANTARES group, the computer group and others at NIKHEF for their cooperation and their pleasant company.

Joris van Rantwijk
March 2002

Contents

Chapter 1

Introduction

ANTARES is a deep-sea detector for high energy neutrinos [2]. This detector is currently being built on the bottom of the Mediterranean Sea. The main components of the detector are photomultiplier tubes – sensitive light detectors – which are able to record light flashes resulting from neutrino interactions. A *data acquisition system* (DAQ) is responsible for recording the signals from the photomultipliers. The DAQ transports, processes and stores the resulting data such that they become accessible to physicists. This is not a simple task, since the data production rate of the detector is high and all data must be processed in real time. A network is used to transport data from under the sea to a computer farm on the shore. The data are filtered by computers in the farm; interesting parts are stored and everything else is discarded. Eventually, the resulting data can be used to *reconstruct* the tracks of neutrinos through the detector.

An international collaboration of scientific institutes is working on the design and construction of the detector. The contribution of NIKHEF consists of the development of a data acquisition system and software for event reconstruction. This summer (2002), a first part of the detector will be deployed. The full detector should be completed in 2005.

My assignment in this project consists of two parts: The first goal is to study the data traffic in the DAQ network by means of a simulation model. This work should prove that the system can work. It may also disclose pitfalls that haven't been noticed yet. The second part is to make a (partial) real implementation of the network traffic code, and to measure its performance. This should prove that the system works also in practice.

This thesis is organized as follows: Chapter 2 gives an overview of the ANTARES project with an emphasis on the DAQ system. In Chapter 3, we present a number of methods for data transmission through the DAQ network. We make a qualitative comparison of the performance, stability and scalability of these methods. This analysis is taken one step further in Chapter 4. A simulation of the traffic pattern in the DAQ network provides a basis for a quantitative comparison of transmission methods. Chapter 5 describes the design and implementation of a component in the *data filter* software. This component is responsible for receiving data from the network and for interaction with other DAQ software. A conclusion, with a summary of the results from the previous chapters, is given in Chapter 6.

Two more aspects of the DAQ network were studied in detail. Since these topics are not directly related to the goals of this thesis, we include them here as appendices. Appendix A describes an experiment with fast network transmission on an embedded system running Linux. In Appendix B, we develop a method to determine the buffer size of network switches.

The work described in this thesis is a continuation of a separate research assignment [1]. That first assignment focussed on the technical aspects of data traffic in a network, listing various options for transmission methods and buffering policies. Our present discussion of transmission methods in Chapter 3 is derived from it. The new work for this thesis consists of the network simulations and the implementation of the DataFilter code.

Chapter 2

Overview of the Antares project

In this chapter, we introduce the ANTARES project. The concepts of the data acquisition system are explained. We describe the components of the detector and the software needed for control and data processing. Many of these concepts are needed again in later parts of this thesis. More details can be found in the ANTARES technical design report [3] and on the website [2].

2.1 Looking for high energy neutrinos

Neutrinos are elementary particles which have no charge and little or no mass. They almost never interact with other matter but go straight through everything without being noticed. Lots of neutrinos are produced continuously by nuclear reactions, for example in the sun, in other stars and in nuclear reactors. Massive numbers of them are floating through the universe, possibly about 500 neutrinos per cm^3 [4].

Since neutrinos are so reluctant to interact, they can travel over very long distances through the universe without being obstructed or deflected. They can go straight through stars and dust clouds which block most ways of looking at the universe. If we could detect those neutrinos and measure their direction, we can use them to “look” back to their source, which may be a turbulent cosmic object. This is the primary goal of ANTARES: detecting cosmic neutrinos (i.e. the ones from far outside our galaxy) in order to “look at” the universe. So ANTARES is not just a detector but also a *telescope*. We look in particular for neutrinos with energies above 10 GeV[1].

Detection principle

Because neutrinos rarely interact, they are difficult to detect. After all, in order to be detected, there must be some interaction between the particle and the detector. While neutrino interactions are rare, they are not impossible. A neutrino might react with another particle on its path, producing other kinds of particles which are easier to see than the neutrino itself. With ANTARES, we mainly look for reactions in which a *muon* is produced.

[1] Giga-electronvolt; 1 electronvolt equals $1.60 \cdot 10^{-19}$ Joule

Being a charged particle, a muon produces a sort of bluish light when it travels faster than light through a medium (water for example). This is called the Čerenkov effect. The light flashes can be recorded with very sensitive light detectors. From the locations and precise timing of the light flashes, we can then *reconstruct* the track followed by the muon. The muon moves in nearly the same direction as the original neutrino. That gives us the direction in the universe where the neutrino source must be.

Detector location

Since neutrino interactions are rare, the detector must be quite large to still have a reasonable chance of detecting neutrinos. It must be filled with water to utilise the Čerenkov effect. It must also be very dark to properly see the light flashes.

The ANTARES solution is to build the detector on the bottom of the Mediterranean Sea, 40 km offshore from Toulon (France), at a depth of 2.4 km. The effective detector area is about 0.1 km^2 and it is about 300 m high. The sea itself is a very cheap supply of water for such a large detector. No daylight penetrates at this depth. It seems that all we need to do is put a lot of sensitive light detectors in the sea in order to see neutrinos.

Building a detector in the sea has some disadvantages as well. The sea water contains a little bit of radioactive potassium. The decay of these potassium atoms causes tiny Čerenkov light pulses, just like the muons but with less intensity. Since these decay reactions happen very frequently, our light detectors will record a background of noise pulses at about 50 (!) kHz. Some algae and deep sea creatures can emit light; this is called bioluminescence. Measurements indicate that they mostly do this in bursts of a few seconds during which a small part of our detector will be unusable for muon detection.

2.2 Detector components

About 900 photomultiplier tubes will be employed to detect the Čerenkov light flashes. A photomultiplier is a very sensitive light detector capable of detecting light pulses of just a few photons; it works a bit like a reversed television tube. Each tube is contained in a protective glass sphere called an *optical module* (OM). These OMs are attached to vertical cables – *strings* – to keep them in place.

The OMs are connected to the strings in groups of three. Such groups are called *floors* (because the OMs are at the same height in the string). Each floor also contains a *local control module* (LCM) with a PowerPC-based embedded computer. It contains sampler chips to digitize the signals from the photomultipliers. A 100 Mbit Ethernet interface is used for data transmission towards the shore. The total number of LCMs is 300.

A group of 5 floors within a string is called a *sector*. One of the LCMs in a sector contains a little extra functionality and becomes a *master local control module* (MLCM). It has an embedded Ethernet switch which is used to combine the 100 Mbit channels from the 5 LCMs into a single 1 Gbit channel to the shore station.

A complete string consists of 6 sectors (30 floors) and a *string control module* (SCM). It is about 350 meter long, fastened to the seabed on the bottom and held up straight by a buoy at the top. The strings contain fiber cables for optical Ethernet transmission, as well as copper wires for electrical power. In the SCM, the gigabit Ethernet channels from the 6 MLCMs are multiplexed into a single optical fiber using *dense wavelength division multiplexing* (DWDM). The full detector will consist of 10 strings. A junction box forms the interface between the strings and the 40 km long *shore cable*.

Sector line

In the summer of 2002, a short prototype string will be deployed to test the integration of the various hardware and software components. The string consists of only one sector: 1 MLCM, 5 LCMs, 15 OMs and instruments for calibration and positioning. This so called *sector line* will be used for real data-taking; algorithms for filtering and reconstruction will be tested with these data.

2.3 Data Acquisition

The data acquisition (DAQ) system is responsible for reading and digitising the output signals of the photomultipliers and for transporting, processing and storing the resulting data.

The off-shore part of the DAQ system has already been mentioned; it consists of the LCMs, MLCMs, SCMs and fiber connections between them. We now turn to the on-shore DAQ components.

Network topology

The LCM processors in the detector are connected to the shore through a network based on Ethernet. This network is used to send data from the detector to shore and to send control commands and configuration data from shore to the detector. All components are linked by full duplex fiber cables. A separate *clock system* is used for signals which must be accurately synchronized.

Figure 2.1 shows the design of the DAQ network. Each LCM has a 100 Mbit network interface. Switches inside the MLCMs group these into 1 Gbit links to shore. The 6 links from each string are DWDM multiplexed to reduce the number of fibers between the detector and the shore. This means that different channels send light with different colours through a single fiber. At the receiving side, the colours are separated and go to different receivers. This multiplexing step is transparent to the data-link level; each of the 60 MLCMs has an independent link to the shore station.

In the shore station, a large Ethernet switch connects the 1 Gbit MLCM links to a *processing farm*. Each *farm node* is also connected to this switch through a Gbit network interface card.

Data filtering

Together, the optical modules produce too much data to keep all of it. Most of these data arise from background noise due to radioactive decay and bioluminescence; only a tiny fraction is

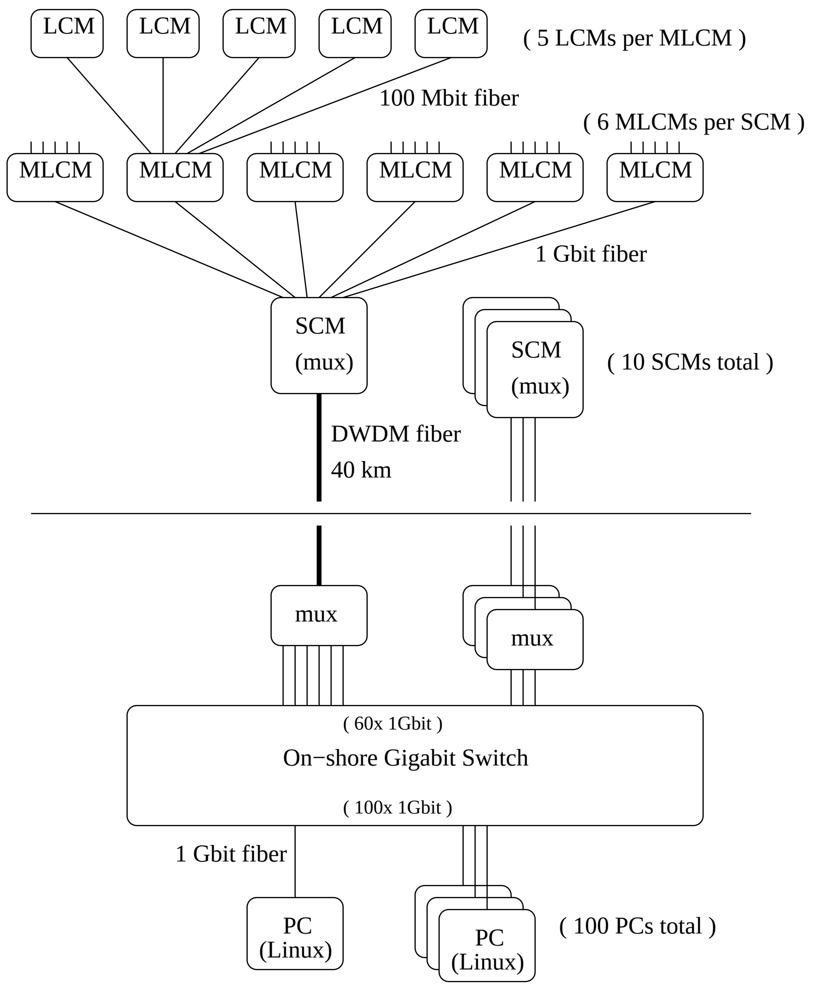

Figure 2.1: Network topology

related to muon/neutrino interactions. We need to separate the background from the real signals to reduce the total data rate. Only the relevant data are stored, the rest can be discarded.

The key distinction between background noise and muon signals is that a muon produces related hits in different parts of the detector, while background hits are random and uncorrelated. An algorithm has been developed which searches for *correlations* between hits in different parts of the detector. When a cluster of related hits is found, the corresponding data are grouped into an *event*. Only these event data are kept and all other data are discarded. Experiments with simulated data show that the overall data rate can be reduced by a factor of thousand with this method. Details of the filter program are discussed in Chapter 5.

Time slices

Even a very fast computer can not filter the data from the full detector in real time; the data rates are just too high. Our solution is to divide the work over a farm of computers such that each computer works on a subset of the data.

In order to find correlations between different parts of the detector, each processor needs access to data from all parts of the detector. This is achieved by splitting up the data in the time domain. We define *time slices* with a fixed duration, probably around 10 ms. Each slice is assigned to a processor in the farm. This processor receives all data corresponding to this slice, allowing it to look for hit correlations through the full detector. Subsequent slices are assigned to different processors, such that each processor can take more than 10 ms to complete the filter algorithm. After finishing with a slice, a processor is ready to receive another slice.

The division into time slices starts in the LCM software. For each slice, the LCM creates a set of *data frames*. Multiple frames are needed to separate data from different OMs and different types of data. The set of frames from one LCM for one particular timeslice is called a *slice fragment*. The LCM sends the frames to the farm node that handles this particular slice. This node receives slice fragments from all LCMs in the detector and merges these fragments into a complete slice. The slice is then processed with the filter algorithm.

There must be a mechanism to decide which farm node handles a specific time slice. This could be done in a dynamic fashion: assign each slice to the node which has the least amount of work to do. This has the advantage of *load balancing*. But it requires a central process which monitors the workload of all farm nodes, decides about slice assignments and informs the LCMs about its decisions (the LCMs must know the destination of their data frames).
Since the amount of processing per slice is dominated by the level of background noise, which is nearly constant, there is no real need for explicit load balancing in the farm. We decided to avoid the complexity of dynamic slice assignments and to use static assignments instead. Slice number s will be handled by farm node number $n = s \textbf{ mod } N$, i.e. the remainder when dividing the slice number by the total number of nodes in the farm. This results in a round-robin assignment where first all nodes get one slice, then all nodes get a second slice, and so on.

Data storage

The output of the filter programs is sent to a *data writer* system, which writes the data to persistent storage. This could be a removable harddisk, or a magnetic tape.

Eventually, these data will be analyzed with reconstruction software to search for evidence of neutrino interactions. This process is not considered a part of the data acquisition system. It will not be done in real time.

Data rates

The digitizing chips in the LCMs detect pulses in the OM signals. For each pulse, the exact time and amplitude are recorded in a 6-byte structure. This is called a single photo-electron (SPE) hit.

The chip may decide that a pulse is particularly large or long (for example due to a series of light flashes in close succession). It can then record a waveform hit: a sequence of 128 samples of the signal amplitude at a rate of 1 GHz. Such hits take 263 or 519 bytes, depending on the number of channels sampled.

Assuming a background SPE rate of 70 kHz per OM, we expect a data rate of about 3 MByte/s per LCM. This is 3 OMs $\times$ 70 kHz $\times$ 6 bytes = 1.3 MByte/s for SPEs plus about the same amount for waveforms. With timeslices of 10 ms, this translates to 3 MByte/s $\times$ 0.010 s = 30 KByte per timeslice per LCM. The full 10-string detector will then produce 300 $\times$ 30 KByte = 9 MByte per timeslice.

In Chapters 3 and 4, we look at methods to efficiently transport these data through the network.

2.4 DAQ software

Several components of the DAQ system are controlled by software. In this section, we discuss some aspects of the global software design.
The programs are written in C, C++ and Java. They fall into the following categories:

- Programs for data processing and data storage in the farm.
- Embedded software to control off-shore components: the LCMs and SCMs. The LCM software handles data transmission as well as control of the detector instruments.
- A system for overall control of the DAQ system and interaction with the user/operator. We call this the *RunControl*.
- Software for online monitoring of the detector systems. Statistics from various processes are gathered, displayed to the operator and stored in the database.
- A database with information for configuration and calibration, and information recorded during data-taking runs. The database is implemented in Oracle. It is accessible through JDBC[2].

2.4.1 Operating systems

Linux

The on-shore part of the DAQ sofware is based on GNU/Linux[3]. This was a very natural decision: Linux is cheap and reliable, and there is much knowledge about it in the physics community. The availability of source-code makes it possible to investigate problems and a wealth of information is available on the Internet.

Linux has full support for TCP/IP networking, multi-tasking and dual-processor systems. Many parameters can be changed to match our needs; for example, we raise the limit for shared memory allocation and for network buffers in our test farm. Much professional software exists for Linux, including the Oracle database system and a Java environment.

[2]Java Database Connectivity; an interface to access any database system from Java

[3] Linux is a free operating system kernel based on Unix/POSIX; see `http://www.linux.org/`. GNU provides a free set of Unix-like system tools and libraries; see `http://www.gnu.org/`.

VxWorks

VxWorks [5] is a real-time operating system for embedded applications. It is commercially available from WindRiver Systems. This operating system will run on the LCM processors in the DAQ system.

The system is POSIX compliant, meaning that programs written for Unix should be able to work with minor adjustments. Networking is supported through a complete TCP/IP stack. Since it was designed for embedded platforms, VxWorks can be much smaller and faster than Unix/Linux. This is important because – to keep powerconsumption low – the LCM has limited processing power; its PowerPC chip runs at just 80 MHz.

Running VxWorks on the LCMs has some important disadvantages as well. An expensive run-time license from WindRiver is needed for each of the LCMs in the DAQ system. Moreover, ANTARES software developers are quite familiar with Unix, but not with VxWorks. Writing software for VxWorks therefore requires new expertise. Some software (such as the ControlHost library, see next section) will be used both on PCs and on LCMs. This implies that the software must be maintained for Linux as well as for VxWorks, which requires an extra effort.

Some experiments were done with Linux on the LCM processor to see if this could replace VxWorks. The results were negative, primarily because the performance of the Linux TCP/IP stack was disappointing.

Zero-copy network transmission

Communication primitives are typically implemented in the kernel of the operating system. In order to send a message, the application issues a *system call* to the `sendmsg` routine in the kernel, passing a pointer to the message buffer.

The usual (BSD `socket` based) interface to the networking system requires that the kernel make a copy of the message contents to a new buffer. This copying requires additional time and memory bandwidth, resulting in a lower transmission throughput. It is possible in principle to avoid this overhead by using a *zero-copy* interface to the networking system. However, this requires special support in the operating system, the device drivers and the application software. We experimented with zero-copy transmission under Linux, and compared the throughput with the normal `socket` interface. Details of these measurements are given in Appendix A.

The VxWorks operating system has a special built-in interface for zero-copy networking. The LCM software will use this interface to transmit data frames.

2.4.2 Run Control

The RunControl is the main user interface to the DAQ system. It is implemented in Java, and will run on an on-shore Linux system. It is responsible for coordinating the activities of other processes in the DAQ system.

The processes in the DAQ system are modelled as a set of concurrent state machines. Figure

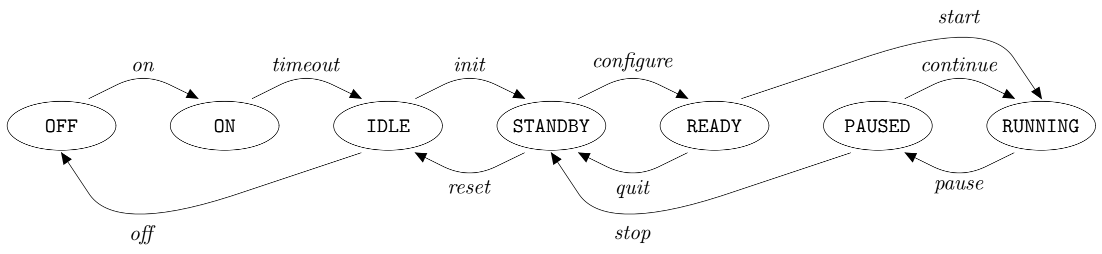

Figure 2.2: States and transitions of the DAQ system

2.2 shows the possible states and the allowed transitions between them. It is the job of the RunControl to make sure that all processes are in the same state by synchronizing state transitions.

The DAQ state machine is described in the CHSM[4] meta language [6]. Such descriptions can be automatically translated into source code for C++ or Java by the CHSM meta compiler. The resulting code defines an object-type representing the state machine. State transitions can be triggered by calling the methods of this object. Each transition may have an associated *semantic action*: a fragment of code in the target language (C++ or Java) which is executed when the transition occurs.
Unfortunately, the objects produced by CHSM are very complicated to work with. Furthermore, they use some advanced features of C++ which has caused problems with certain compiler versions. Since the DAQ state machine is conceptually very simple, it seems that the efforts of using CHSM are not justified. A custom implementation of the DAQ state machine would have been much easier to work with.

The user interface is based on the Java Swing toolkit. It presents an overview of all active processes and their current states. State transitions can be initiated by the operator by clicking buttons. It is foreseen that the RunControl will also be able to autonomously initiate certain transitions when no manual operator is present, for example to stop a run when the disk is full.

Clients communicate with RunControl through ControlHost (discussed below). The RunControl initiates transitions by broadcasting a command message. Active clients respond by changing their state according to the command and sending a response message to report their new state. The RunControl waits for these answers; clients which fail to respond are marked 'inactive'. On some transitions (INIT and CONFIGURE), the command message contains extra data needed by the clients.

2.4.3 Control Host

ControlHost [7, 8] is a message passing system implemented on top of TCP/IP. A ControlHost *process space* consists of a special process called the `dispatcher` and a number of application processes called *clients*. Each client maintains a TCP connection with the `dispatcher`.

[4] Concurrent Hierarchical State Machine

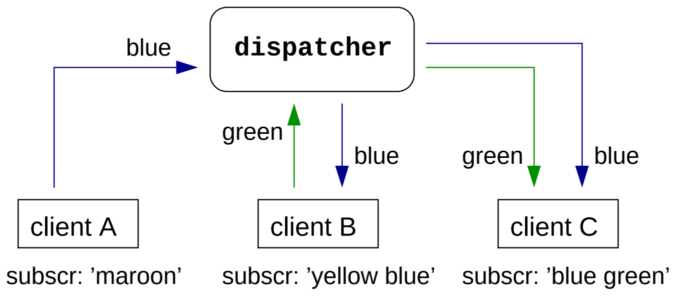

Figure 2.3: Example of message routing in ControlHost

A message consists of a *tag* and a *data* part. The tag must be a short string but the data may be of arbitrary length and structure. Clients may send messages by calling a `PutFullData` function, specifying the tag and the data block for the message. The message is then transferred to the `dispatcher` via TCP. Clients may *subscribe* to a specific set of tags. Messages received by the `dispatcher` are forwarded to just those clients that have subscribed to the message's tag. The clients periodically make calls to `CheckHead` and `GetData` functions to receive the messages forwarded to them.

There are several advantages to ControlHost over plain TCP/IP. It allows easy and flexible communication among a group of processes. Individual processes don't need to know each-other's addresses; they don't even need to known about each other's existence. All they need is the address of the `dispatcher` and a common mapping between tags and message types.

Application in the DAQ system

In the ANTARES DAQ system, ControlHost is used for all network traffic including commands, control data and detector data. In some cases, this results in great flexibility. For example, a central `dispatcher` is used for RunControl commands, message logging and configuration. The DAQ software components initially only need the address of this `dispatcher`. When components are added, modified or moved to different hosts, no changes are needed in other programs.
A log viewer client may be started and stopped at any time while the system is running. Active components simply send their messages to the `dispatcher`; they don't care whether someone is listening to them.

However, the transmission of detector data through ControlHost causes some problems. The total data rate of a 10-string detector is too much for any single system. It is thus not possible to use a single `dispatcher` for all detector data. To work around the above problem, separate `dispatcher`s will run on each of the farm nodes. All LCMs subscribe to all of these `dispatcher`s and transmit data directly to the right node. A rewrite of the ControlHost client library [8] was needed since the original system didn't support simultaneous connections to multiple `dispatcher`s.

Transmission through ControlHost is inherently less efficient than normal TCP/IP networking since all data must be sent twice: first to the `dispatcher` and then again to the destination process. When the `dispatcher` and receiving client run on the same computer, a *shared* *memory* mechanism may be used to reduce this overhead. Even then, network calls are used to notify the client of new messages. The shared memory mechanism also adds a new source of overhead and complexity because the `dispatcher` needs to keep messages in memory until the clients are finished with them.

Chapter 3

Data transmission method

In this chapter, we describe the data transmission method used to move data from the LCM devices to the farm PCs. We present alternative solutions and explain their advantages or problems. Our focus is on two aspects: the network transport protocol and the strategy for data buffering.

3.1 Network transport protocol

The low-level design of the DAQ network is based on Ethernet technology and the Internet Protocol (IP). These choices are motivated by the wide availability of hardware, software and expertise for IP/Ethernet.

The transport layer of a network protocol stack provides the application with a set of communication primitives: routines for sending and receiving messages. The IP stack specifies two separate transport protocols: the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

Both protocols provide primitives for sending and receiving messages, but with different semantics. The TCP protocol transports a data stream after establishing a logical connection between the communication partners, while UDP transports individual messages. An application can use the protocol which best fulfills its communication requirements. The two protocols may safely be used together in a single network without interference.

3.1.1 TCP

In order to send or receive messages with TCP, it is necessary to first establish a *connection* with the communication partner (or *peer*). Once established, the connection can be used to send messages in both directions until it is explicitly closed by the application. TCP guarantees *reliable* delivery of all messages sent through it, even if packets are occasionally lost or damaged in the hardware layer. Messages are delivered in the same order as they were transmitted, but message boundaries are not preserved; TCP may split or join messages to optimize network usage. This is why TCP is said to be a *stream-oriented* service.

The reliability of TCP is implemented by having the receiver acknowledge incoming data, and making the sender *retransmit* data which are lost or damaged. Complex algorithms are used to decide when retransmission is necessary, based on the information in acknowledgements and on timing. Additional algorithms implement *flow control* which avoids flooding a slow receiver, and *congestion control* which helps to avoid congestion in the switching network.

Application in the DAQ system

At the start of a run, each LCM creates TCP connections to each of the PCs. The transmission of a data frame is done by sending it as a message over the connection to the destination PC.

Each PC is connected to each of the LCMs, and it continuously monitors all these connections. As soon as data arrive, it does a receive operation on the connection to retrieve the data. Since message boundaries are not preserved by TCP, the PC must look at the message headers to know how many bytes it must receive to get a complete message.

3.1.2 UDP

With UDP, individual messages are handled independently of each other. UDP is a stateless protocol; it is not necessary to establish a connection before sending or receiving messages. The delivery of UDP messages is not reliable; delivery of messages is not guaranteed and no error is reported when delivery of a message fails. The send primitive creates one or more network packets to contain the UDP message and sends them to the destination process. If any packets are lost or damaged underway, the message will not be delivered. A sequence of transmitted messages is not necessarily delivered in order. However, message boundaries are always preserved: a message is either delivered completely, or it is not delivered at all.

Possible application in the DAQ system

At the start of a run, all LCMs and PCs create a single socket each. One socket can handle the communication with many hosts, since no connections need to be established; the destination address is specified for each message separately.

A data frame is transmitted as a single message to the destination PC. Each PC continuously monitors its socket to see if messages are coming in. When data are available, a complete frame can be retrieved with a single receive operation.

Notes

There are three major reasons why a UDP message may be lost:

- The underlying physical network is not perfect. A packet may be damaged by a bit-error on the physical level.
- When a switch receives multiple packets for the same destination, it can forward only one of these packets immediately; the rest must be temporarily stored in a buffer. When

its buffers are full, the packets will be dropped. This condition is caused by network congestion.

- A large burst of transmitted or received UDP messages may overflow internal buffers in the operating system of the LCM or PC. Subsequent packets will then be dropped.

Due to the absence of congestion control mechanisms, the use of UDP as transport protocol in the ANTARES DAQ system would imply that we **must** find a way to explicitly avoid synchronous transmissions from all LCMs to a single PC. If we wouldn't, most of the messages would be lost due to overflowing switch buffers. Methods to avoid this are discussed in the next section.

Even if the network traffic is tuned to avoid overflowing any packet buffers, there would still be a slight loss of data due to bit-errors. This effect could be reduced by implementing a limited form of retransmission: if the receiver finds that a small fraction of the data didn't arrive, it sends a retransmission request to the transmitter. Of course, the retransmissions themselves may suffer from bit-errors, but that would happen even less frequently.

3.1.3 Advantages of TCP

Advantages of TCP over UDP:

- Transmission is reliable. Automatic retransmission recovers from data corruption, and ensures that all messages are correctly delivered. This is important, not only to recover from bit errors in physical channels, but also to recover from data loss caused by buffer overruns in operating systems or in network hardware.
- The flow control algorithm avoids overflow of the receiver's buffers.
- The congestion control algorithm reduces packet loss due to network congestion. This is very important with respect to the synchronous transmission of data from all LCMs to a single PC.
- Most existing network applications have been written for TCP.

Disadvantages of TCP with respect to UDP:

- Complex algorithms are used, which incur some overhead per message. This is important because the LCMs have little processing power.
- Each host must manage a large number of connections simultaneously.
- A connection may be aborted in the unlikely event of many transmission errors. This would result in persistent loss of data until the application explicitly re-establishes the connection. UDP is a stateless protocol and would as such never cause persistent errors.
- The timing of actual TCP packet transmissions is subject to complex protocols. With UDP, packets would be transmitted immediately after execution of the send primitive. This makes it easier to analyze the resulting network traffic.

The ANTARES DAQ group decided that the reliability of TCP outweighs the simplicity of UDP. With appropriate buffering strategies, it is likely that TCP will provide fully lossless data transmission while UDP would most likely lose at least a small fraction of data.

3.2 Buffering policy

It is tempting to design the LCMs such that they transmit each piece of data as soon as they have it available. There seems to be no point in keeping data in the front-end of the detector longer than strictly necessary. However, immediate transmission is not necessarily the best option. It may be better to keep data in a buffer for a while and send it at a suitable moment. The process of keeping data and sending them later is what we call *buffering*, and the policy for deciding whether a message should be delayed and for how long is what we call the *buffering policy*.

In this section, we show that data buffering is essential for the operation of the DAQ network. We consider two options: buffering in the TCP layer (which we will call *implicit* buffering because it is transparently handled in the transport layer) or buffering through a delayed transmission algorithm in the application code (which we denote as *explicit* buffering).

3.2.1 The need for data buffering

The requirement that all frames from a single timeslice be sent to a single PC, implies that data must be buffered one way or another. We prove this by calculating the dataflow for the case that no buffering is done at all, and comparing the result with the available network bandwidth [1]. At the end of a timeslice, all LCMs send the corresponding slice fragments (at 30 KByte per LCM) to the destination PC. This transmission must complete within 10 ms (before the end of the next timeslice). Without buffering, all fragments arrive at the PC simultaneously at a total rate of 300×30 KByte $\times \frac{1}{10\text{ms}} = 900$ MByte/s. The PC's Gigabit Ethernet link has a theoretical limit of only 120 MByte/s, hence the no-buffering case is infeasible.
One way around this is by keeping part of the data inside the LCMs, and transmitting it at a later moment.

3.2.2 Buffering in the transport layer

The TCP protocol is capable of implicitly buffering outgoing data. It uses the flow control and congestion control algorithms to decide when to send more data. Therefore TCP should in principle be able to handle our buffering requirements without extra modifications. This would remove the need to implement an explicit buffering policy.

However, as demonstrated in Chapter 4, TCP's flow/congestion control algorithms are not adequate for the ANTARES DAQ situation. The following factors contribute to this problem:

- TCP flow control was designed to manage a single, large flow between two hosts. But in the DAQ, the flow from a single LCM to a single PC is relatively small; in the order of 30 KByte/s. The PC will probably never need to activate flow control.
- TCP congestion control was designed to make the protocol stable in large networks with uncorrelated data streams and a gradually changing level of congestion. It is not clear that this will work properly in the DAQ, where data streams are highly correlated (all LCMs send at the same time) and congestion develops instantaneously. Oscillations in the congestion control algorithm may make the problem worse.

The semantics of TCP guarantee that no data are ever lost in the transport layer. Instead, TCP will block the data stream when the network cannot keep up with the incoming data rate. This would eventually force a loss of data on a higher level when the system runs out of memory space.

Implementation

Since TCP handles all buffering implicitly, no special provisions are needed in the LCM or the PC.

We assume that the LCM's program contains a function `handle-timeslice()`, which is called whenever a timeslice has ended and the corresponding data are ready for transmission. This function could be implemented according to the following pseudocode fragment:

```
procedure handle-timeslice(timeslice-nr, timeslice-data)
  dest ← timeslice-nr mod nr-of-consumers
  sendmsg(consumer-socket[dest], timeslice-data)
end procedure
```

This program determines the PC which will handle this timeslice by taking the timeslice number modulo the total number of PCs. It then invokes the `sendmsg` function to send the data through the TCP socket corresponding to that PC.

The PC will constantly monitor its connections with the LCMs and receive data when they arrive. As soon as all fragments for the next timeslice are read-in, it can provide the slice to the `DataFilter` algorithm. In order to recover from network failures, a timeout mechanism may be implemented which ensures that the processing algorithm is started even if not all data are received yet. Data which arrive too late are simply ignored in this case.

Receiving and processing must be overlapped in time, so that the next timeslice can be received while a previous timeslice is still being processed. This could be achieved by implementing the receiver in a separate process, taking advantage of the multi-tasking facilities of Linux.

3.2.3 Explicit buffering

An advantage of explicit buffering is that the buffering policy is under our own control, so that we can design it to be stable for the DAQ traffic pattern. This approach is sometimes referred to in the literature as *traffic shaping* [9].

Requirements for a buffering policy

Since the network bandwidth of a single PC is about 10 times that of a single LCM, congestion will be avoided completely provided that no more than about 10 LCMs send to the same PC at the same time. This is the key concept of the buffering policy we will describe in this section. To assure that oversubscription of the PC's network bandwidth is avoided, we must satisfy the following requirement:

R1 No more than a given number of LCMs may send to the same PC at the same time.

One way to achieve this, is to have the PCs send request messages to the LCMs. A LCM doesn't send data as soon as it is available, but waits until it receives a request from the PC. Each PC sends request messages to the LCMs in a cyclic fashion. By limiting the number of outstanding requests, the PC can guarantee to avoid oversubscription on its receiving channel. This algorithm changes the data transmission from a sender-driven process into a receiver-driven process. The timing of data transmissions is not anymore determined by the availability of data in the LCMs, but rather by the availability of processing resources in the PC. We have effectively turned the process upside-down (or inside-out if you like).

While this algorithm can easily be made to satisfy **R1**, it has an important drawback. It is now possible that an LCM receives several requests from different PCs at the same time. The LCMs do not have sufficient network bandwidth to satisfy multiple PCs, so this is an example of oversubscription of the LCM's transmission channel. On the other hand, it may happen that an LCM does not receive any requests for some time, in which case it cannot transmit any data and is forced to waste its network bandwidth.

We conclude that the simple request-algorithm doesn't solve the oversubscription problem, but merely moves it to another part of the network. We add two more requirements to avoid this mistake:

R2 An LCM must not be required to send data to more than one PC during a timeslice.

R3 The total average throughput must not be limited by the buffering policy; therefore each link in the network should be used as efficiently as possible. Since the LCM links are the tightest bottleneck, each LCM should be allowed to send continuously at full speed.

3.3 Delayed transmission

All three requirements can be satisfied with a technique called *barrel shifting* [9]. It is based on the synchronisation of the network flows to a global notion of time slots. A variant of this technique, suitable for the DAQ system, was introduced in [1] and further discussed in [10]. It has been adopted by the ANTARES DAQ group and will be implemented in the LCM software.

3.3.1 Concept

Each LCM is assigned a *delay parameter*, which denotes the number of time slices that the LCM should wait before sending its data. This comes down to maintaining a queue of buffered slice fragments with a length equal to the delay parameter. Each time a new timeslice ends, its fragment is stored on the queue; the oldest fragment is taken from the queue and transmitted. By assigning different delay values to different LCMs, the arrival of data frames at the PC is spread in time.

This process is illustrated in Figure 3.1. For simplicity, we assume that there are only three LCMs and three PCs. LCM-1 has a delay parameter of 0, while LCM-2 is set at 1 and LCM-3

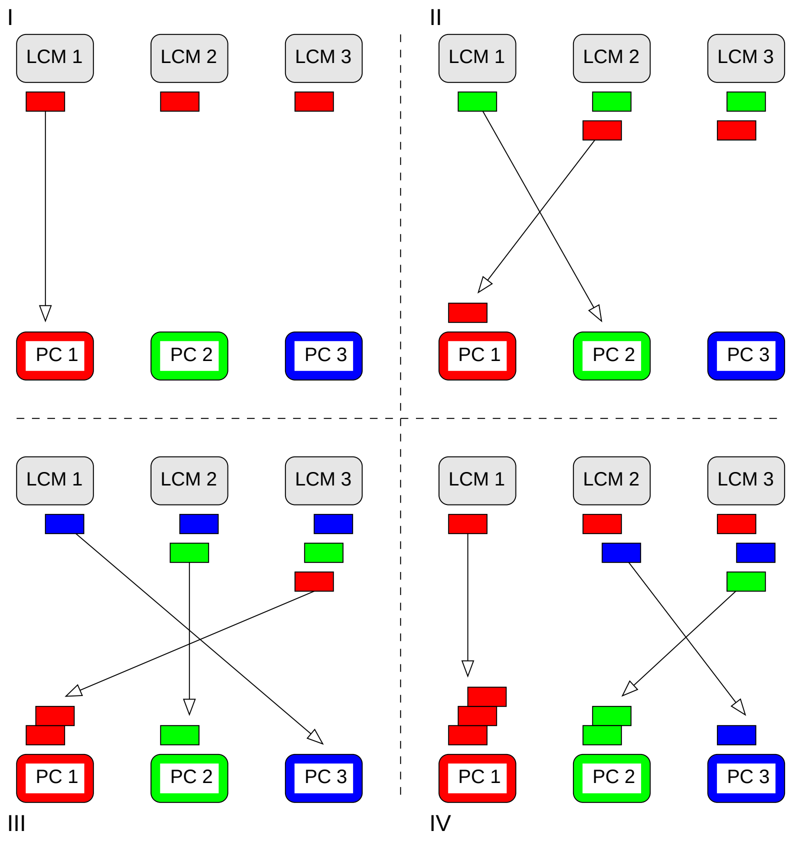

Figure 3.1: Delayed transmission

at 2. This means that LCM-1 does not queue its data at all, but rather sends fragments when they become available. The other LCMs delay all data transmissions by means of a queue. A stable state is reached in picture III, where the first *cycle* has been completed. From that point on, the queue lengths will remain constant.

As can be seen from the figure, once a stable state is reached, the following conditions continue to hold:

- Each PC receives one fragment per timeslice. Therefore, requirement **R1** is met.
- Each LCM sends one fragment per timeslice. This satisfies **R2**.
- The queue lengths remain constant. This means that the system can continue to function for an unlimited period without saturating buffers. There is no need to impose a limit on the throughput to achieve this stability, so we also meet **R3**.

This looks very promising. We should now find out whether it is feasible to implement this mechanism in the DAQ system.

3.3.2 Application to the DAQ system

In order to apply this policy in the DAQ system, it must be scaled up to the actual number of LCMs and PCs, and it must be adapted to the case of an unequal number of LCMs and

PCs. The up-scaling is quite easy. We simply take the n LCMs and assign them increasing delay values ranging from 0 up to $n-1$.

This is not appropriate in case there are fewer PCs than LCMs (like in the ANTARES DAQ). For m PCs, a full cycle lasts only m timeslices. It makes no sense to delay a transmission for m or more timeslices since this transmission would then overlap with transmissions to the same destination belonging to a later cycle. The solution is to assign a single delay value not to one, but to a small group of (let's say k) LCMs. This modification preserves all properties except that a PC now receives data from k LCMs during the same timeslice. The maximum delay value (largest queue length) drops to $\frac{n}{k}$. For $n = 300$ LCMs and $m = 100$ PCs, the value of k should be at least $\frac{n}{m} = 3$. It may also be larger, as long as the PCs can handle the increased receive rate. This has the advantage of lowering the maximum delay value, thereby lowering the buffering requirements for the LCMs.

The required synchronisation between the data streams follows automatically from the global notion of timeslices. The transition to a next timeslice happens at the same time in all LCMs. This process is driven by the accurately synchronous clock signal. Since the timing of network transmissions in this scheme is based on the availability of timeslice data, it is indirectly driven by this same clock signal.

Implementation

The LCM part of the buffering policy can be implemented as a modified version of the `handle-timeslice()` function.

```
procedure handle-timeslice(timeslice-nr, timeslice-data)
  delay ← producer-index div $k$
  add-to-queue(timeslice-data)
  if queue-len > delay then
    ndata ← get-from-queue()
    nts ← timeslice-nr − delay
    ndest ← nts mod nr-of-consumers
    sendmsg(consumer-socket[ndest], ndata)
  end if
end procedure
```

This algorithm first calculates the delay parameter as a function of the identity of the LCM and the value of k (the number of LCMs with equal delay). Then, instead of sending the most recent data fragment, it adds it to the tail of the delay queue. As we have seen, this queue should always contain exactly *delay* fragments. If the new fragment makes the queue too large (as will always be the case except during the startup phase), the least recent fragment must be taken out and transmitted. The destination is determined according to the original timeslice number (*nts*) of the fragment; the `sendmsg` is used to start the transmission.

As before, the PC constantly monitors its connections with the LCMs. The buffering policy is fully handled by the LCM, so the PC requires no special support. If a timeout mechanism is used, the time limit may have to be adjusted to the extra delay imposed by the explicit buffering, but this delay can never be more than one cycle (about 100×10 ms = 1 sec).

Due to the time spreading of data arrivals, it may take a longer time to receive all data from a specific timeslice. Overlapped receiving and processing in the PC will be required to avoid wasting time in the receive loop.

The amount of buffer memory needed for buffering, can be calculated from the length of the delay queue. The maximum possible delay is equal to one cycle (1 second for 100 PCs and 10 ms timeslices). Multiplication by the average data rate, yields the average memory requirement: 1 sec $\times$ 3 MByte/sec = 3 MByte. This is quite small compared to the total of 64 MByte available.

3.3.3 Variable data rates

Barrel shifting policies in general are best suited to traffic patterns where all fragments are equal in size. This is not the case in the ANTARES DAQ; as a result of bioluminescence, some slice fragments are much larger than most others. Problems appear when a fragment is too large to transmit in a single 10 ms time window.

We can deal with this in three ways (or possibly a combination):

- By modifying the delayed transmission policy, we can make it suitable for variable sized data blocks. More specifically, we could allow LCMs to run behind on the transmission schedule. This breaks the strict limitation on the number of LCMs transmitting to the same PC. However, assuming that peak data rates occur infrequently and in uncorrelated random patterns, it is very unlikely that a PC is affected by a large number of LCMs running behind.
- If the transmission of a fragment has not completed after 10 ms, we can pause its transmission, push the remainder of the fragment back on the queue and continue the transmission when we come across this PC again in the next cycle. This breaks the limitation on the length of the queue. If an LCM is affected by high data rates for a long period of time, it will run out of buffer memory. Some modifications in the PC may be needed to implement this option, since it must be able to wait for more than one cycle to fully receive all slice fragments.
- We can choose a fixed limit for the size of a fragment. Larger fragments are either dropped or truncated before transmission. The loss of data is an important disadvantage of this method. However, the data acquired during peak data rates is of minor importance for physics, because the front-end electronics is already dropping part of the data in such situations.

It has not yet been decided which method to apply to the ANTARES DAQ system. In this report, we implicitly use the first method: peak data rates cause the corresponding LCM to fall behind temporarily.

Chapter 4

Network traffic simulation

A simulation model of the DAQ network is used to evaluate the performance of different transmission methods. This analysis will help to select the best transmission method, and enables us to search for optimal values of parameters. Also, it may reveal unsuspected problems and help to avoid those problems in the implementation of the real DAQ system.

In this chapter, we describe the architecture, models and parameters of our simulation. We discuss the validity of our models and the underlying assumptions. Results are given for varying model parameters. These are used to optimize the transmission parameters, and to estimate the maximum throughput of the DAQ network.

4.1 Simulation techniques

There are three general techniques for systems performance evaluation: measurement, analytical modelling and simulation [11]. Analytical modelling is not used because it is difficult to describe the dynamic, state-oriented aspects of TCP with it. Direct measurements are not possible since the full system does not yet exist. Simulation offers the opportunity to perform measurements on a virtual system. This makes it easy to compare results for different system parameters and workloads.

A number of different simulation techniques exist, including Monte-Carlo simulations which are used to model probabilistic phenomena, and discrete-event simulations which keep track of a global system state changing in time. For computer network models, the discrete-event approach is most suitable.

Discrete event simulation

In a discrete event simulation, all objects have a discrete *state* which can change due to the occurrence of *events* [11]. The word 'discrete' is used to emphasize that a change of state always takes place at a single, well defined point in time. It doesn't mean that the time values or state variables must be of a discrete type. In fact, our network simulations use a continuous time scale.

A scheduler inside the simulator maintains a queue of events, ordered by timestamp. The first event is selected, removed from the queue and *executed*, i.e. the state is modified to reflect the situation after the occurrence of the selected event. Often, the execution of an event *causes* a number of new events to be scheduled at a future point in time. For example, the transmission of a packet causes the reception of the packet at the other side of the link some milliseconds later. In this case, the new events are simply added to the scheduler queue with appropriate timestamp values (*now* plus a few milliseconds). They will be executed when their time comes, that is, when all events with earlier timestamps have been processed.

4.2 The network simulator NS-2

"NS is a discrete event simulator targeted at networking research. NS provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks." [12]

For the DAQ simulations, we use version 2.1b8, released in June 2001. A patch is applied to modify the TCP protocol and to add a custom UDP implementation.

4.2.1 NS-2 architecture

The simulator package is written in C++ and OTcl [13] (an object oriented script language). It works under Unix/Linux; source code is freely available.

The C++ part contains a number of components which constitute a basic simulation framework. It also provides mechanisms for describing a network topology, constructing data packets and routing these packets through the network. The simulator program is produced by linking the C++ simulator code with an OTcl interpreter.

The main loop of the simulator program consists of the event scheduler. The scheduler repeatedly selects the oldest pending event and invokes the corresponding event handler routine. New events may be added to the pending set by the handler routine. The data structure of the pending event set must efficiently implement insertion of new events and extraction of the oldest event. NS-2 supports a number of scheduler implementations based on different data structures for the pending event set, including a linked list and a *heap*. The default scheduler uses the more sophisticated *calendar queue* structure [14]. Insertion and extraction operations on this structure run in O(1) time on average, making the simulator run faster than with the alternative queue structures.

Random numbers are needed for modelling stochastic processes (such as bioluminescence). A *pseudo-random number generator* is a deterministic algorithm that produces numbers which 'appear to be random'. NS-2 provides a random number generator based on the multiplicative linear congruential method.

4.2.2 Model definition

A simulation model consists of a program in the OTcl language. The program creates and manipulates simulator objects to construct the static component of the simulated environment. It then schedules a series of events (for example, packet transmissions). Finally, it invokes the scheduler of the simulator object to start the simulation.

An example of a very simple model is shown below. It describes a small, DAQ-like network with TCP data transmissions.

```
# Initialize simulator object
set ns [new Simulator]

# Create nodes for an LCM, a MLCM, a PC and the Gigabit switch
set lcm0 [$ns node]
set mlcm0 [$ns node]
set pc0 [$ns node]
set gigaswitch [$ns node]

# Connect the nodes through links
$ns duplex-link $lcm0 $mlcm0 100Mb 50us DropTail
$ns duplex-link $mlcm0 $gigaswitch 1Gb 300us DropTail
$ns duplex-link $gigaswitch $pc0 1Gb 50us DropTail

# Create TCP agents, set PC agent in listen state.
set lcm0_tcp [new Agent/TCP/FullTcp]
set pc0_tcp [new Agent/TCP/FullTcp]
$ns attach-agent $lcm0 $lcm0_tcp
$ns attach-agent $pc0 $pc0_tcp
$ns connect $lcm0_tcp $pc0_tcp
$pc0_tcp listen

# Schedule data transmission
$ns at 1.0 "$lcm0_tcp send 10000"
$ns at 2.0 "$lcm0_tcp send 10000"

# Schedule end of simulation
$ns at 5.0 "exit 0"

# Start simulation
$ns run
```

4.2.3 Report generation

In order to do something useful with a simulator, it must produce some form of output describing the process inside the simulation. There are two methods to produce output from an NS-2 simulation:

- The simulator has built-in provisions for generating a trace file. Such a trace file includes a description of the simulated network topology, and a list of all packet transmission and reception events with timestamps.
- Custom handlers for certain events can be installed by the model script. These handlers can use Tcl statements to print log records.

The trace file provides information at the level of individual packets. When the simulation has completed, it can be visualized using a separate program called NAM. All necessary information is read from the trace file. The network topology is displayed as a graph; packet traversal is visualized as little dots moving along the lines. The program can also draw histograms of link loads and packet drop rates.

Quantitative performance characteristics can be obtained by analyzing the trace file. However, since we're dealing with several concurrent (but distinct) data streams, it would be difficult to draw conclusions from packet level events. We are mainly interested in what happens at the application level: how long it takes before data become available, which amount of data is ultimately lost, etc. To obtain this information, we would need to reconstruct the relations between multiple packets. This is difficult, and we therefore decide to use a different approach.

We disable the simulator trace file, and collect information through explicit logging statements in the model description. When an application object sends or receives a data fragment, it calls a procedure to write a timestamped message to the log file. After the simulation is completed, this log file is parsed with scripts written in Perl and AWK.

4.2.4 NS-2 objects

Simulator objects are implemented as *classes* in either C++ or OTcl. The simulation script creates instances of these objects, and calls methods to configure and connect them. A large number of standard network components are readily available in NS-2. The most important are:

- **Nodes** represent end stations (computers), as well as hubs and switches. A node is a very generic object; it doesn't include a detailed model of a network interface or network switch. Issues such as maximum bus transfer rates, backplane capacity, packet queueing are thus not taken into account.

- **Links**, either simplex or duplex. A link has parameters for bandwidth, delay, and queueing model. In the real world data links do not have queues, but in NS-2, transmission queues are implemented in the link instead of in the transmitter node where they are physically located. Additionally, a link may have an error model which specifies a certain packet loss rate. Default links are perfectly lossless.

- **Agents** are abstract entities that implement transport protocols. An agent is always *bound* to a node, and may be *connected* to a peer agent. NS-2 provides a UDP agent and a variety of TCP agents, including a restricted transmission-only model called `TCP` and the reasonably complete full-duplex model `FullTcp`.

Experiments with TCP in congested networks revealed a bug in the implementation of `FullTcp`. While setting up a connection with its peer, the agent loses data that the application tries to send; something that should never happen with TCP. We located the problem in the `FullTcp` implementation, and modified the code to fix the bug in our case. A question regarding this bug was posted to the NS-2 mailing list; this will probably be fixed in a future release of NS-2.

The builtin UDP agent in NS-2 was found to be too restricted for our measurements. We designed a new, more detailed simulation of the UDP protocol called `JUdp`[1]. This agent is implemented as C++ class and linked with the rest of the simulator.

4.3 DAQ network model

An OTcl script was written to model the ANTARES DAQ network in NS-2. Some details of the model can be controlled by a set of parameters. By varying the parameter values, we can deduce their effect on the performance of the system.

A list of parameters and their values is shown in Table 4.1.

4.3.1 Topology

The network topology consists of nodes and links. A node is created for each of the LCMs, MLCMs, farm PCs and for the gigabit switch.

The number of LCMs and PCs (detector scale) is configurable. The number of PCs is chosen equal to the number of LCMs divided by three. All nodes are connected by duplex links with a suitable propagation delay. For the LCM-to-MLCM and the Switch-to-PC links, the delay is set to 50 μs; the Detector-to-Shore link is set to 300 μs (corresponding to approximately 50 km).

The nominal network bandwidths for LCM and PC are 100 Mbit and 1 Gbit respectively. But the processors and software will not be able to handle the full link speed. This aspect is incorporated in the model by selecting a lower simulated link speed.

Each link has a drop-tail queueing model (when the queue is full, all newcoming packets are dropped). We know that the Ethernet transmission queue of Linux systems has a default length of 100 packets. The actual size of the packet buffers inside the MLCM and Gigabit switches is not known, but simple experiments suggest a size of around 100 packets (see Appendix B). We therefore set the lengths of all drop-tail queues to 100 packets.

An error model can be specified for the 50 km cable to account for bit errors in the physical layer and the DWDM hardware.

4.3.2 Data transmission

Data are transmitted through the network with either the TCP or the UDP protocol. In the case of TCP, a set of `FullTcp` agents is created and bound to the LCM and PC nodes such that each of the LCMs has a TCP connection to each of the PCs. In the case of UDP, a `JUdp` agent is created for each of the LCM and PC nodes.

By default, all LCMs send their data to the same PC at the same time. Delayed transmission can be enabled to modify the transmission pattern as described in Section 3.3. The number of simultaneously sending LCMs (the k parameter) is configurable.

[1]Joris's UDP agent

Topology	
Detector scale	1 sector; 2 strings; 10 strings
LCM network bandwidth	50 Mbit/s
PC network bandwidth	500 Mbit/s
Switch output buffers	100 packets
Bit error rate on 50km cable	0; 10^{-8}
Data transmission	
Transport protocol	TCP; UDP
Segment size	1460 bytes
Delayed transmission	off; on with $k = 8$, 15 or 20
Workload	
Timeslice length	10 ms
Data rate	30 000 bytes/fragment
Bioluminiscence	none; 5%; 5% with cutoff
Biolum: data rate	150 000 bytes/fragment
Biolum: burst duration	random, uniform $0 \ldots 3s$

Table 4.1: Model parameters. Values separated by semicolons have been used in successive runs.

4.3.3 Workload

Transmission of slice fragments is scheduled as it would occur in the DAQ system: every 10 ms, each of the LCMs sends a fragment to one of the PCs. Timeslices are assigned to PCs according to a static round robin schedule: the first timeslice to the first PC, the second slice to the second PC etc. When we're out of PCs, the next slice goes to the first PC again.

When bioluminescence simulation is disabled, each slice fragment has a fixed size (statistical fluctuations are small and therefore ignored here). The default of 30 000 bytes per LCM per time slice corresponds approximately to a 70 kHz singles rate [3]. Larger fragment sizes can be selected to see at which point the system breaks down.

With bioluminescence simulation enabled, we use a stochastic model as a rough approximation of the bioluminescence phenomenon. Parameters define the fragment size during bursts, the average duration of a burst and the average time fraction spent in burst mode. In our model, a burst always affects all three OMs in a floor, but bursts in different floors or different strings are independent. During a simulated burst, the fragment size of the affected LCM is set to 150 000 bytes (maximum output rate of 6 ARS chips [3]) instead of the normal fragment size. At the start of a timeslice, each LCM has a fixed probability to start a bioluminescence burst. This results in a geometric distribution of the time intervals between successive bursts on an LCM. The length of a single burst is drawn from a uniform distribution between 0.0 and 3.0 seconds. The probability of starting a burst is selected such that any LCM is expected to spend 5% of the total time in burst mode; this is in agreement with site measurements [15]. When bioluminescence is enabled, we decrease the normal data fragment size to 25 000 bytes to remove the contribution of bursts which are now explicitly modelled.

It has been proposed to suppress data transmission by an LCM when the hit rate exceeds some preset value. This *cutoff* principle would avoid transmitting large amounts of meaningless data during bioluminescence bursts. It has been implemented in the simulation as follows: when an LCM goes into burst mode, only the first 4 and final 4 slices of the burst are transmitted. Transmission is suppressed during the remainder of the burst. The first and final few slices represent the period during which the counting rate is affected by the burst, but is still below the cutoff point.

The simulation starts in an idle state (no data transmitted, no connections established yet). We then apply the transmission workload for a period of 60 simulated seconds. A relaxation period of 10 simulated seconds follows, during which delayed fragments may still arrive at their destinations. The simulation is then terminated and results are collected.

4.3.4 Performance metrics

For each slice fragment, the time of transmission and time of arrival are reported in the simulation output. When the simulation is completed, this output is analyzed and a number of performance metrics are computed from it.

For network performance evaluation, it seems natural to choose *throughput* as a metric. This is, however, not appropriate for our situation because both the data quantity and the amount of time in which it must be transfered are specified as model parameters. Instead, we attempt to measure *how well* the system is coping with the specified workload. Eventually, we can obtain the maximum throughput by running a series of simulations with increasing workload.

With the TCP protocol, we measure the *network delay* time (the elapsed time from transmission by the LCM until reception by the PC) as metric for the goodness of the system. Under congestion, delays will increase as a result of retransmissions and non-optimal throttling by TCP. Additionally, we measure the *fraction of retransmitted segments.* These two metrics provide a simple method to decide whether the system still handles the workload. In a healthy system, network delay should be low, dominated by signal propagation time and time for store-and-forwarding in the switches. The fraction of retransmitted segments should be close to zero.

With the UDP protocol, we simply measure *data loss* (the fraction of slice fragments which is lost or damaged). The data loss metric corresponds directly to the actual fraction of physics data which would be lost in the corresponding DAQ system. Like the retransmission metric, this should be zero or close to zero.

4.3.5 Model validity

The reliability of simulation results depends on the validity and correctness of the model. These can be checked by comparing simulation results to real measurements.

The real detector network does not yet exist, and is therefore unavailable for model validation. However, it is possible to construct a scaled down (1 sector) prototype which can be used to validate the 1-sector models. This would also provide confidence in the larger models, assuming that the model remains valid under scaling.

Some factors in the DAQ network are not yet understood. For example, the effective network bandwidth of the LCMs and PCs is still uncertain; the exact buffer length and queueing model

of the Ethernet switches are unknown.[2] The corresponding parameters in our simulation model are probably not exactly right.
Some factors from the DAQ system were deliberately left out of the simulation model, because they are thought to be insignificant and too complicated to implement. Examples are: data processing overhead in the LCMs and PCs, backplane capacity of the switches, other data streams on the network (for detector control and logging).

There exists a large set of subtly different implementations of TCP, often extended with experimental techniques to improve retransmission and/or congestion characteristics. In addition, most TCPs have many tunable parameters with different default settings in different implementations. The TCP implementation in NS-2 is probably not the same as that in Linux or VxWorks. A few bugs and inaccuracies in the simulator's TCP agent were discovered while debugging the model. The TCP behaviour observed in the simulation is therefore not quantitatively reliable.

4.4 Simulation results

4.4.1 TCP results

We begin by visualizing the network delay metric in a plot for a single run of the simulation. Each timeslice has a mark on the horizontal axis. On the vertical axis, we set the network delay in seconds. For each slice fragment received, a data point is drawn in the plot. These plots provide a quick overview of the transmission process.

Direct transmission

In the sector simulation (Fig. 4.1), all slices are delayed by the same amount of time, approximately corresponding to the time it takes to transmit the data. We conclude that no congestion is present in this case.

In the 10 string case (Fig. 4.2), some fragments are delayed by more than 40 seconds, which is a very unnatural situation for a TCP connection. We conclude that the network simply cannot handle this much data and starts behaving erratically.

Plot points from a single TCP connection (a single LCM-PC pair) are organized in diagonal patterns. This indicates the nearly concurrent arrival of a group of data frames through a connection after resolving some communication problem. The banding effects are a result of the deterministic behaviour of TCP; each band represents a set of connections from which a certain number of successive packets have been lost. It seems that the system reaches a stable state after about 50 seconds (at slice 5000), but that can not be concluded. Any frames which have not been received before the end of the simulation, are not shown in the plot so there may be even higher bands which have 'fallen off the diagram'.

[2] A study of the queueing model of Ethernet switches is presented in Appendix B

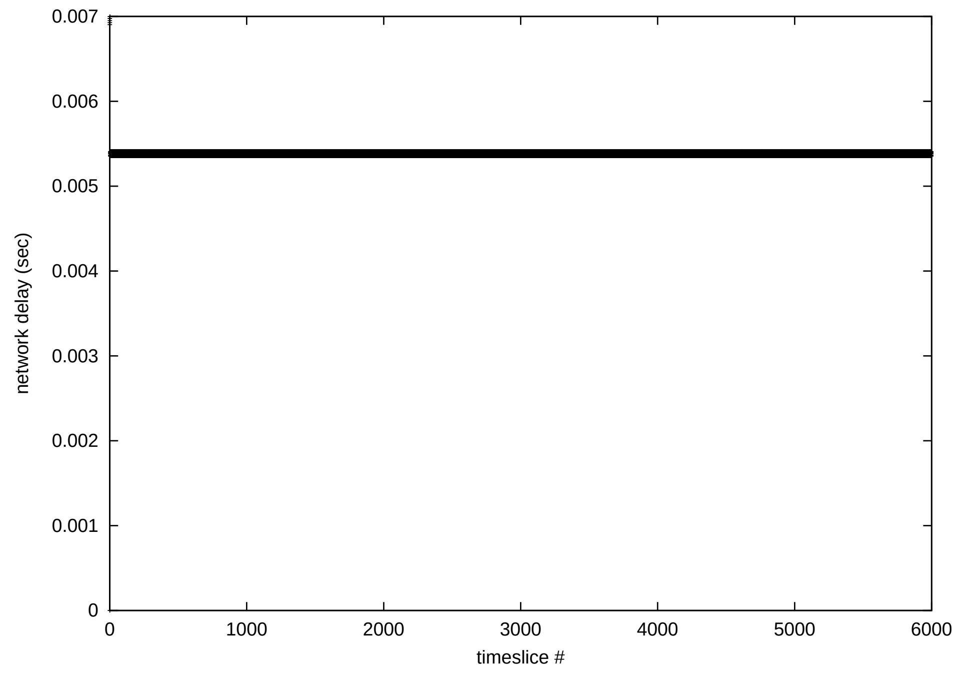

Figure 4.1: Delay plot for 1 sector, TCP, direct transmission

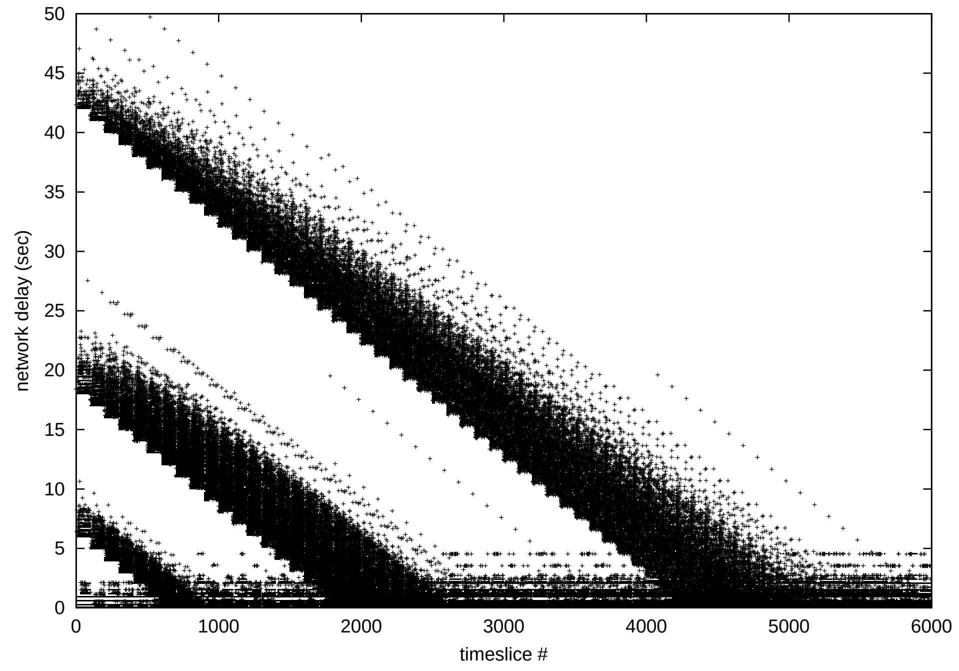

Figure 4.2: Delay plot for full detector, TCP, direct transmission

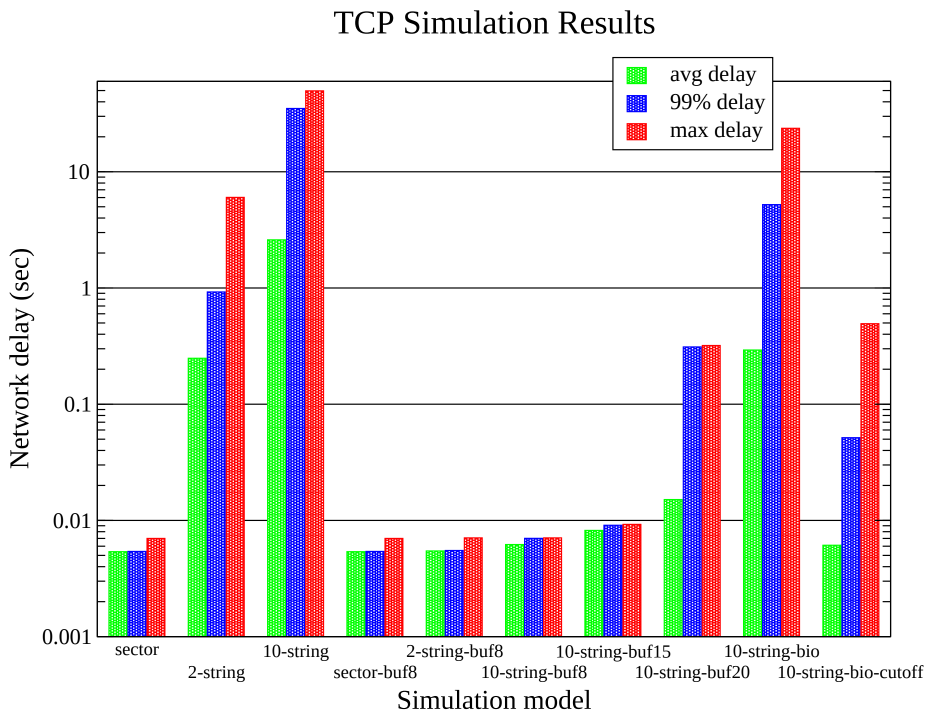

Figure 4.3: Comparing performance for TCP models on a logarithmic scale

model	**avg delay (s)**	**99% delay (s)**	**max delay (s)**	**retrans (%)**
direct transmission				
sector	0.005	0.005	0.006	0.0
2-string	0.248	0.924	6.021	1.2
10-string	2.597	35.069	49.724	0.8
delayed transmission				
sector $k = 8$	0.005	0.005	0.006	0.0
2-string $k = 8$	0.005	0.005	0.007	0.0
10-string $k = 8$	0.006	0.007	0.007	0.0
10-string $k = 15$	0.008	0.009	0.009	0.0
10-string $k = 20$	0.015	0.310	0.319	$\ll 0.1$
delayed transmission ($k = 8$)				
10-string bio	0.292	5.213	23.661	0.9
10-string bio-cutoff	0.006	0.051	0.492	$\ll 0.1$
10-string 50k	0.009	0.010	0.010	0.0
10-string 60k	0.015	0.018	0.022	0.0
10-string 65k	2.550	13.614	57.313	~ 13
10-string bio-cutoff +errors	0.006	0.051	6.007	$\ll 0.1$

Table 4.2: Comparing performance for TCP models

Delayed transmission

In Figure 4.3, we compare the delay characteristics of a number of different models. We compute the average and maximum delay over all slices. Furthermore, we compute the 99 percentile delay, which corresponds to the time needed to receive 99% of the data (and accept a 1% loss). Note that the extra delay due to delayed transmission (see Section 3.3) is not included in the *network delay* values. Table 4.2 shows the same values, together with the retransmission statistics.

It is obvious that the performance of the direct transmission models degrades as the detector gets larger. With delayed transmission however, the system performs well even for the full size detector. The delay for a 10-string detector is now hardly larger than that for a single sector. This demonstrates that explicit buffering is an important requirement.

The method works fine with the k parameter (number of simultaneously sending LCMs) set to 8 or 15. For $k = 20$ there is some additional delay and some retransmissions, presumably caused by emerging congestion. These results are not unexpected; each LCM produces about 3 MByte/s and each PC has a network bandwidth of about 60 MByte/s, so 20 simultaneous LCMs is the point where things should start to go wrong. To be on the safe side, we select $k = 8$ for further experiments.

Bioluminescence

Bursts of bioluminescence activity are started at random moments, according to a simple stochastic model. It is easy to make a mistake during the implementation of such a model, causing it to behave in a non-random or biased fashion. To eliminate any serious mistakes, we analyze the generated pattern of bursts. For each timeslice in the simulation, we count the number of LCMs that are in burst mode. A histogram of these data is shown in Fig. 4.4. As specified in the model, on average 5 % of the detector (15 out of 300 LCMs) is in burst mode. This indicates that the burst model works properly.

Performance of the system degrades seriously under bioluminescence, even with the delayed transmission technique (Fig. 4.3). This is no surprise: an LCM in burst mode produces more data than it can transmit, causing it to run more and more behind on the schedule imposed by delayed transmission. The result is congestion and packet loss, showing itself in the form of retransmissions and network delay.

The proposed cutoff mechanism for peak data rates reduces this problem successfully. A combination of peak rate cutoff with delayed transmission provides reasonable performance.

Maximum throughput

It is important to know how the system reacts to a sustained data rate which is higher than expected. We therefore simulate a number of runs with increased data volumes per fragment. Instead of 30 000 bytes, we attempt to transmit 50 000, 60 000 and 65 000 bytes per fragment. Bioluminescence is not explicitly modelled in these runs.

As shown in Table 4.2, all delay values increase slightly for the 50k and 60k models when compared to the normal 30k model. This is a normal result of the longer transmission times due to larger data fragments. The retransmission count is zero, so there is still no packet loss.

At 65k per fragment, the system performs suddenly much worse. Since LCMs have a 50 Mbit/s uplink, they can't transmit more than about $50/8 = 6.25$ MByte/s $\Rightarrow$ 64 KByte/fragment. Data is produced faster than it can be transmitted and transmission queues will keep growing. The delayed transmission algorithm allocates fixed time intervals to fragment transmissions. These intervals are too short for the data volumes in this case; the system will run more and more behind on the delayed transmission schedule. Massive packet loss and retransmissions are the result.

We conclude that the maximum throughput of the DAQ network with delayed transmission is close to the bandwidth of the LCMs. When the data rate exceeds this maximum, even the delayed transmission policy fails and performance drops sharply.

Transmission errors

Until now, we assumed that all network links are perfectly error free on the physical level. This seems like a reasonable approximation. Bit error rates in fiber optical cables are usually extremely low, such that we can safely ignore them. This may change when a link is damaged or works less well than expected for some reason.

To find out how the DAQ system reacts to this, we include an error model in the simulation. Each link in the 50 km cable (MLCM-to-PC connections) is set to have a bit error rate of 10^{-8} (it randomly and independently damages one out of 10^8 bits). A fiber optical link wouldn't exhibit such error rates unless it is in a really bad condition. With this error model included, we re-run the simulation of the 10-string detector with delayed transmission and peak rate cutoffs. The result (Table 4.2, bottom row) is that the average-case performance is not affected, but the worst-case delay increases considerably.

4.4.2 UDP results

The performance results for UDP are summarized in Table 4.3. Network delay is always low, because UDP by itself can't buffer or retransmit data like TCP. Congestion will now show itself directly as loss of data.

With direct transmission, even the 2-string model loses almost all data (as expected). Delayed transmission eliminates this data loss completely.

The effects of delayed transmission are partly defeated by bioluminescence bursts. For this reason, data loss reappears when we take bioluminescence into account. Peak rate cutoff reduces this loss, but doesn't eliminate it completely. This demonstrates that a simple UDP protocol without error correction is inadequate unless a continuous loss of approximately 1% of the data is accepted.

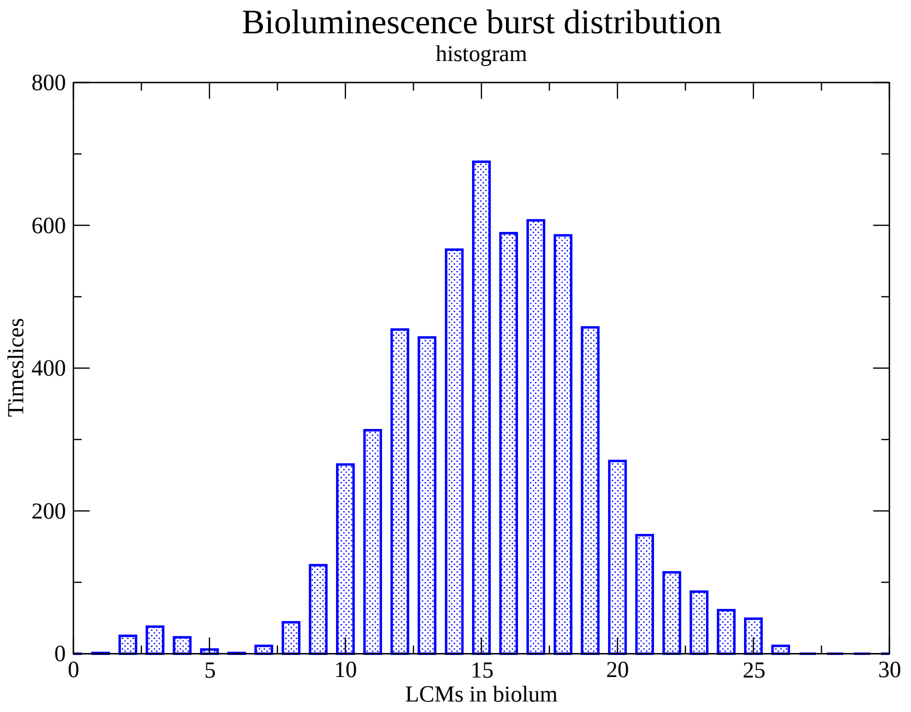

Figure 4.4: Distribution of the number LCMs in bioluminescence mode

model	**max delay (s)**	**data loss (%)**
direct transmission		
sector	0.005	0.00
2-string	0.007	92.51
10-string	0.007	98.50
delayed transmission ($k = 8$)		
sector	0.005	0.00
2-string	0.005	0.00
10-string	0.005	0.00
delayed transmission ($k = 8$)		
10-string bio	0.019	5.00
10-string bio-cutoff	0.019	0.77
10-string bio-cutoff +errors	0.019	0.98

Table 4.3: Comparing data performance for UDP models

Chapter 5

Data filter prototype

5.1 Introduction

The purpose of the ANTARES DataFilter software is to reduce the volume of the physics data by approximately a factor of thousand. Input to the filter consists of raw data frames from the LCMs. The output of the filter algorithm is stored on persistent media by the DataWriter system. A farm of Linux PCs takes care of the filtering task. Each processor in the farm runs an identical copy of the DataFilter software.

The most essential part of the DataFilter is the algorithm[1] that decides which data are kept. This algorithm needs to be both very fast (to filter all data with limited computing resources) and very accurate (to avoid dropping important physics data). We will not discuss the details of this algorithm here, but treat it like a black box. We concentrate instead on the software environment and data flow of the filter process.

The DataFilter has interfaces with other parts of the DAQ system: the LCMs and the DataWriter, but also the RunControl and online monitoring. We design the part of the DataFilter which implements these interfaces. A sketch of the *data flow* and the *control flow* is presented. We implement a control framework for the DataFilter which allows it to operate correctly in the DAQ system.

5.2 Requirements

Functionality

R1 The DataFilter program must accept raw data from the LCMs, process these data with the filtering algorithm and send filtered data to the DataWriter.

R2 In the TDR [3], it is mentioned that all raw data could be archived for a short period in response to an external trigger signal. This trigger may be invoked, for example when other projects[2] detect an interesting cosmic event such as a *gamma-ray burst* (GRB).

[1] developed at NIKHEF by P. Payre from the Centre de Physique des Particules de Marseille.
[2] such as the GRB Coordinates Network; `http://gcn.gsfc.nasa.gov/`.

The design of the DataFilter program must support this feature. The trigger signal may arrive several (10 to 30) seconds after the start of the burst, so the recent history of raw data must always be held in memory.

Software context

R3 The DataFilter software must run on the Linux x86 platform. Dependencies on external libraries, helper applications etc. should be avoided to allow running the filter in a minimal Linux configuration.

R4 The software must be prepared to run on a *processor farm*. Multiple copies of the DataFilter must be able to run independently on different *nodes* in the farm.

R5 The software should be prepared to work efficiently on multi-processor systems.

R6 The DataFilter must interact with the raw data interface of the LCMs. Raw data are delivered to the filter using ControlHost, through a local `dispatcher` process running on the farm node itself. The timing of data transmissions is as discussed in Section 3.3 (this is handled by the LCM software).

R7 The DataFilter must interact with the interface of the DataWriter. Filtered data are transported to the DataWriter system using ControlHost, through a dedicated `dispatcher` process running on the DataWriter node.

R8 Fragments of raw data may be needed by the *online monitoring system* for quality control and statistics. There must be a way for the online monitoring system to selectively collect raw data.

R9 The DataFilter must interact with the RunControl. The RunControl initiates state transitions, sends commands and configuration data and monitors the state of the DAQ system. Communication with the RunControl is based on ControlHost through a central `dispatcher` process. This interface must be implemented as a state machine in the CHSM meta language.

Performance

R10 Each filter process must be able to accept raw data at an average rate of 10 MByte/s (1 GByte/s divided by 100 processes). The rate of incoming data will exhibit peaks as a result of clustering of the data transfers from multiple LCMs.

R11 Each filter process must be able to output filtered data at an average rate of 20 KByte/s.

R12 The filter algorithm must reduce the data rate by a factor of thousand on average.

5.3 Design

The required functionality of the DataFilter is implemented in two separate programs. The `dfilter` program handles the main task of filtering the data (**R1**). An additional `burstsaver` program can be used to save raw data for the GRB feature (**R2**). Both programs will run on each of the farm nodes, but they are completely independent. In this report, we focus on the `dfilter` program. Design and implementation of the `burstsaver` is left as a future project.

A running `dfilter` program consists of a single process. No internal multi-threading or multi-processing is done; it would make the software much more complex without improving performance. If dual-processor systems are used (**R5**), it is possible to use both processors for filtering by running two copies of the `dfilter` program on each farm node.

The data filter program is divided into two parts: one part handles communication, control and data management, while the other part contains the actual filtering algorithm. A simple *interface* between these parts allows separate development. This makes it easy to merge the separately developed algorithm with the control framework.

5.3.1 Data flow

Figure 5.1 is a diagram of the data flow in the DAQ system. Each node in the filter farm runs the `dispatcher`, `dfilter` and `burstsaver` programs. The LCMs send raw data to the `dispatcher`s on the farm nodes. The `dfilter` and `burstsaver` each receive a copy of these data. After running the filter algorithm, the `dfilter` sends filtered events to the DataWriter system. The `burstsaver` normally does little more than keeping the last few seconds of data in memory. When a GRB trigger signal arrives, it starts writing raw data to a local harddisk on the farm node. Afterwards, these data are read back and sent to the DataWriter.

It is foreseen that the *online monitoring* system also has access to raw and filtered data in order to do data quality checks and statistics. These data paths are shown as dashed arrows in the diagram. When the monitoring system needs to *tap* data from a certain point, it connects to the corresponding dispatcher and subscribes to the needed message tags.

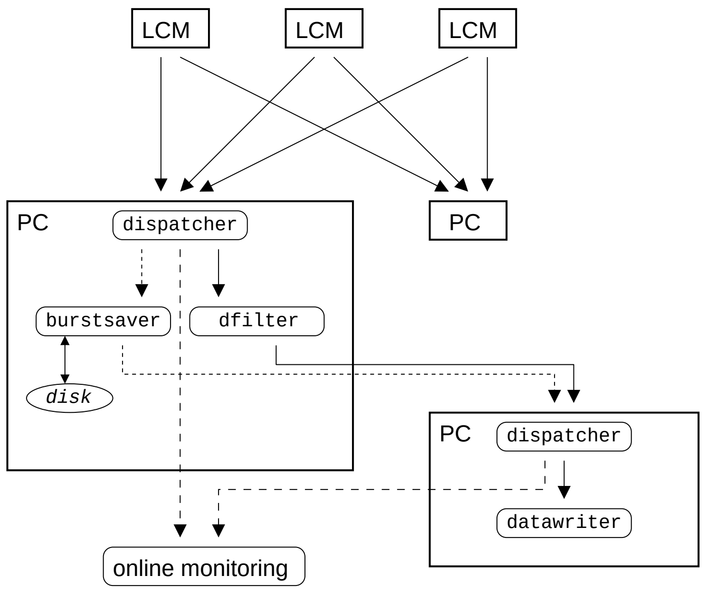

Figure 5.1: Data flow in the data filter farm

Although not shown in the diagram, the `dfilter` processes are connected to an additional central `dispatcher` process for communication with the RunControl.

5.3.2 Control flow

The activities of the `dfilter` program depend on its interaction with other parts of the system. The work of the filter algorithm can continue only while raw data are available from the `dispatcher`. When the program runs out of data temporarily, it needs to wait until new data arrive. At the same time however, the program must continue to respond to commands from the RunControl.

To implement such behaviour, it is natural to use an *event driven* control structure. The program has a *main loop* in which it waits until something interesting (an event[3]) happens, acts to deal with the event, then waits until the next event.

Fig. 5.2 depicts the main loop of the `dfilter` program. First, all pending commands from the RunControl are processed and any incoming data frames are stored on a queue. If the queue contains enough data frames, the filter algorithm is invoked on these frames (this takes about 1 second); otherwise a `select` function is called to suspend the program until new data (or RunControl messages) are available.

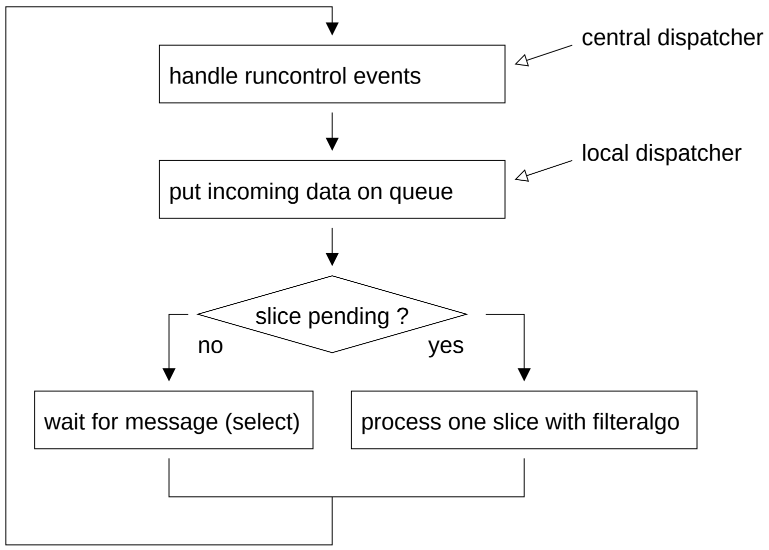
handle runcontrol events
central dispatcher
put incoming data on queue
local dispatcher
slice pending ?
no
yes
wait for message (select)
process one slice with filteralgo

Figure 5.2: Main loop of the `dfilter` program

5.3.3 Managing incoming data

The filter algorithm works on a slice-by-slice basis. It needs all frames from a single timeslice as its input. All LCMs in the system must contribute to this set of frames by sending their own portion of the slice.

[3]not to be confused with a neutrino event from the detector

The `dfilter` receives each frame as an independent ControlHost message. Received frames must be stored in memory until a slice is complete; only then can it be passed to the filter algorithm for processing.

Queue structure

It is not possible to receive data frames strictly slice-by-slice, moving to the next slice only when the current slice is completed. Some LCMs may already start sending frames for a next slice while others haven't sent their current frames yet. The delayed transmission (see Section 3.3) method adds to this effect. A more complicated structure, which can store frames for multiple slices, is thus required.

The `dfilter` program maintains a list of *active slices*: slices for which at least one frame has been received but which have not yet been processed by the filter algorithm. Each active slice has its own list of received frames.

Immediately after receiving a new frame message, the `dfilter` examines the *frame header* [16]. The information in the header is used to decide to which slice the frame belongs. This slice is looked up in the active-slice-list, and the frame is added to it. If the slice was not in the list (the new frame is the first of its slice), a new entry is created in the list.

As soon as the oldest (least recent) active slice is complete, it is removed from the list and processed by the filter algorithm.

Avoiding stalls and overflows

Due to bugs, hardware failure or network congestion, some frames may be received with a large delay or never received at all. We would like the system to continue as well as possible in this case, starting the filter algorithm on a subset of the slice if needed. This is done in `dfilter` by means of a *timeout* concept. If a frame is still not received some fixed time after it was expected, we don't wait for it any longer but start the filter algorithm on the incomplete slice.

The difficult part is to decide *when* the frame should have been received. The `dfilter` is not coupled to the realtime detector clock system, so it doesn't know exactly when to expect a certain data frame. We solve this by relying on the synchronisation of the data sources. There is normally a fixed time distance between adjacent slices: 1 second, assuming 10 ms slices and 100 filter nodes. If we choose the timeout to be 5 seconds, we then know that a frame is too late if it has not been received before the first frame of the 5th next slice. In other words: as soon as the number of active slices exceeds 5, we know that the oldest slice is timed out and we can start processing it.

If an incomplete slice has been processed, it may happen that its missing frames arrive at a later time. These frames can't be related to the incomplete slice anymore since the incomplete slice has already left the active list. So our queueing process would tend to create a new active slice for them. This is not a good idea however; there would probably just be a few delayed frames going into this slice and filtering it would be a waste of resources. In order to detect and resolve this situation, we keep the *timestamp* of the most recent slice which was removed from the active list. Any subsequently received frames for this slice (or for older slices) are discarded immediately. This ensures that the filter algorithm processes slices in timestamp order, and that each slice is processed only once.

It may be that the filter algorithm is systematically too slow to keep up with the incoming data rate. If, for example, a new slice is received every second while the filter algorithm takes 1.1 seconds to process a slice, the list of pending slices will tend to grow longer and longer.

If more and more pending frames are stored, we risk running out of memory. This must be avoided because it would block the delivery of new data, causing stalls in the dataflow everywhere in the system. We therefore keep track of the number of pending frames and their total size. When a certain limit is exceeded, we simply discard some of the pending frames. This is an emergency action which should never happen during normal runs.

5.3.4 Interface with RunControl

Communication with the RunControl is done by means of ControlHost messages through a central `dispatcher` process. Each command from the RunControl triggers a *state transition* in the `dfilter` program. After each transition, a confirmation is sent back to the RunControl.

The states which have been defined for the DataFilter software are shown in Fig. 5.3. When started, the program automatically enters the IDLE state. From then on, it will only change states when told to do so by the RunControl. An OFF transition causes the programs to exit. Some transition commands carry additional information for the client programs. For example, the CONFIGURE transition carries a set of configuration parameters, and the START transition carries a unique *run number* to identify the run.

Actual recording and processing of data will only take place in the RUNNING state. However, the DataFilter software does not distinguish between the RUNNING and PAUSED states. In both states, it will accept and process any incoming data. Since there won't be any incoming data during the PAUSED state, the net result is that the filter will be idle during PAUSED.

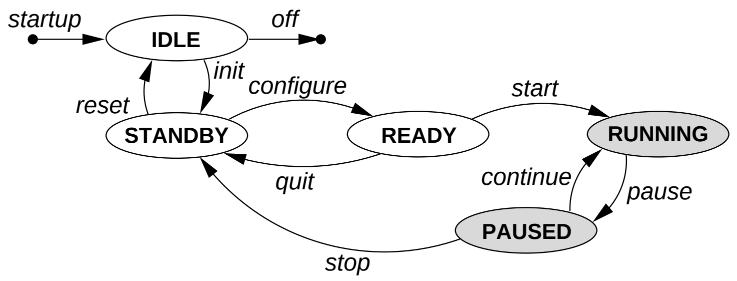

Figure 5.3: Data filter state transitions; grey states are active.

5.4 Implementation

The `dfilter` program is written in C++ and consists of two parts: the filter algorithm and the interaction with the DAQ system. These parts communicate through a simple function-call interface. Both parts are implemented in separate object files and then linked together to form a single executable.

Filter algorithm

An implementation of the filter algorithm was already available when we started our work on the control framework. This was a stand-alone C++ program, capable of reading and processing data slice-by-slice. We transformed this program into a module which can be linked to other programs. The `main()` function was removed and replaced by exported functions which may be called from the control framework. The routines for I/O were removed, and replaced by access to a data structure which is shared between both program parts.

The resulting module implements an interface with three functions. A definition of these functions is given below. The `trigger_init` function is responsible for preparing the filter algorithm. It is called before the start of each run, and takes a list of configuration parameters. Its counterpart is the `trigger_cleanup` function, which is called after the end of each run to free any resources claimed during initialization. The real work is done by `trigger_slice`, which is called once for each slice. Its parameter of type `DFilterSlice` contains the data frames corresponding to this slice in the form of a list of pointers to the data buffers involved.

```
void trigger_init(int argc, const char * const * argv);
void trigger_slice(const class DFilterSlice *slice);
void trigger_cleanup(void);
```

Interaction with DAQ system

The control framework interacts with the DAQ system and calls the functions of the filter algorithm when needed. It makes ControlHost connections to three `dispatcher`s: the central DAQ `dispatcher` for RunControl and message logging, a local `dispatcher` to receive data frames and the `dispatcher` on the DataWriter node to send output from the filter code.

A main loop continuously looks for incoming messages from the central and local `dispatcher`s. Messages from the RunControl are handled by a state machine. This machine is an implementation of the diagram in Fig. 5.3 written in CHSM and C++. Some state transitions cause the execution of an additional code fragment. For example, the CONFIGURE transition invokes a function which decodes the list of parameters and calls `trigger_init`.

Messages from the local `dispatcher` are handled by putting the data frames on a queue of pending frames. When a complete slice is on the queue (or when a timeout condition occurs), the slice is removed from the queue and the `trigger_slice` function is invoked on it.

High performance aspects

The DataFilter is a high performance computing application. Since we want to do a lot of filter work with as few resources as possible, it is important to make efficient use of the available computing power.

The DataFilter needs to receive large amounts of data through a network connection of limited bandwidth. Assuming an average data rate of 9 MByte/s and an effective network bandwidth of 80 MByte/s, $9/80 = 11\%$ of the time is spent on receiving data. Network receive operations

are, for the major part, done in the hardware of the network interface using DMA[4]; the CPU is hardly involved in this. It would thus be a waste of time to wait until all frames for a slice are received before starting to process the slice.
The multi-tasking facilities of the Linux operating system make it easy to *overlay networking and processing.* While the `dfilter` is processing one slice, the `dispatcher` can already receive the next slice in the background (using some of the CPU time, but not very much).

Shared memory is used to avoid unnecessary copying of data. Normally, data passing through the ControlHost system is sent and received twice: once from the sending application to the `dispatcher` and once from the `dispatcher` to the receiving application. In the case of the DataFilter, the second transmission would be a waste of time. Since the `dispatcher` and the receiving `dfilter` process are always on the same node (Fig 5.1), transmission through the network is not needed. For this case, a shared region of memory is created which can be accessed by both the `dispatcher` and the `dfilter`. Received data are stored in this region by the `dispatcher`, and can be used directly by the `dfilter`.

Symmetric multi-processing (SMP) is a technique to build low-end parallel computers. Two or more CPUs of the same type are placed in a single computer. The processors share most system resources (memory, interrupts, I/O bus). The operating system assigns a different process to each of the processors, so that multiple processes can be running at the same time. In principle, this means that a 2-CPU SMP system can be essentially twice as fast as a 1-CPU system with the same processor type. Whether this speedup is really achieved depends on the type of application. A single process can use only one processor at a time; it can't be split-up and divided over multiple processors. So in periods that there is only one process active in the system, only one processor is used and the other one will be idle. Even if multiple processes are ready to run at any time, the speedup may be lower than expected. Since the processors share resources and memory, heavy resource usage in one process will cause delays in other processes.
There are two approaches to using 2-CPU SMP systems for the DataFilter. One is to run two `dfilter` processes on each of the farm nodes. This makes it possible to divide the work evenly between the two processors, resulting in good *load balancing.* The other approach is to run only one `dfilter` process, leaving the other processor free for the `dispatcher` and the operating system. This is an asymmetric workload (it is unlikely that the `dfilter` and `dispatcher` take exactly the same amount of CPU time) so it will not make optimal use of processing power.

5.5 Tests

A small PC farm has been prepared to test the DAQ software. We use this farm to test the functionality and performance of the DataFilter software.

[4]Direct Memory Access: a technique for data exchange between devices and the system memory without intervention of the CPU.

5.5.1 Farm setup

The configuration of the test farm reflects the situation as foreseen for the *sector line* prototype (see Section 2.2).

Hardware

- 5 *producer* nodes: dual-Pentium III 933 MHz, 1 GByte RAM,
  3Com 3C905C 100 Mbit Ethernet adapter
- 3 *consumer* nodes: dual-Pentium III 933 MHz, 1 GByte RAM,
  3Com 3C985 (acenic) 1 Gbit Ethernet adapter
- 1 farm server: Pentium III 930 MHz, 256 MByte RAM,
  two 3Com 3C905C 100 Mbit Ethernet adapters
- 1 Gbit Ethernet switch: 3Com 3C17702 Switch 4900 SX 12-port
- 1 100 Mbit Ethernet switch: 3Com 3C16980A Switch 3300 24-port,
  with 3C16975 1000baseSX module

Configuration

All farm nodes, as well as the farm server, run the Linux 2.4.x OS kernel. The farm nodes are configured as diskless systems (although they do have a large harddisk for data storage). They download their bootimages, programs and datafiles through NFS[5] from the farm server.

The eight farm PCs are connected in a private network as shown in Fig 5.4. The farm server connects to both the farm network and the regular NIKHEF network. This makes it possible to reach the farm from a normal workstation.

Sector line model

The sector line will consist of five LCMs (each with three OMs), one MLCM switch and a DWDM link to shore. In the shore station, a small PC network will filter and store the incoming data.

In our farm setup, we use the five 100 Mbit nodes to simulate the LCMs by running a data generation program. This program produces Gaussian noise in the same data format as an LCM device. The MLCM switch is modelled by our 100 Mbit switch. Its uplink to the Gbit switch represents the DWDM link. The propagation delay in the 40km cable is currently not taken into account. The three 1 Gbit nodes are used to run the DataFilter and DataWriter software. RunControl and logging is done from the farm server. ControlHost `dispatchers` are running on the farm server and on the three Gbit nodes.

[5]Network File System

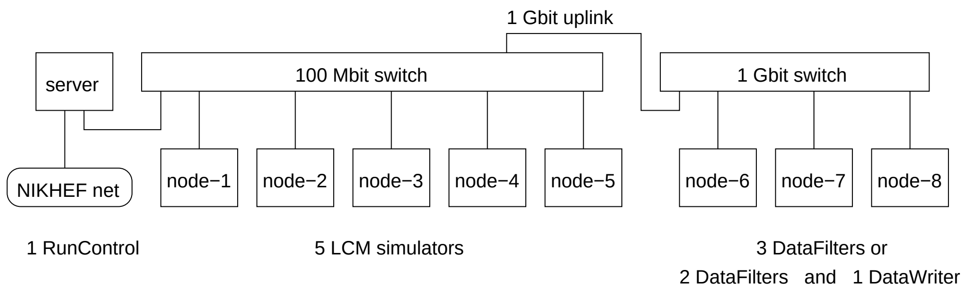

Figure 5.4: Test farm setup

5.5.2 Functionality tests

In parallel with the DataFilter, four other parts of the DAQ software were developed at NIKHEF: the ControlHost system, the RunControl system, the DataWriter and a data generator. These programs need to work closely together when exchanging data and commands. Our experimentation with the farm was the first occasion where the combined functionality of these programs was tested.

We confirmed that the farm software interoperates correctly by manually giving state transition commands to the RunControl. Log messages and other activity from the client programs showed that the clients correctly follow the RunControl commands. Randomly killing a few client programs showed that the system is to some extent capable of recovering from error conditions. The stability of the system was tested by leaving it in the RUNNING state for several hours, and by automatically issuing a large number of state transitions.

5.5.3 Performance measurements

The performance of the DataFilter software is measured in a number of different configurations. The results of these measurements may be used to guide optimization of the filter algorithm, selection of the final farm hardware and selection of the final configuration for the DataFilter software.

The goal of these first measurements is to answer the following questions:

- What is the maximum input rate which can be handled by the DataFilter ?
- Can the efficiency be enhanced through the use of *shared memory* ?
- How does the use of SMP dual-processor systems affect the performance ?

An indication for the performance of the software is obtained by measuring the CPU usage of the various program parts; a program which uses little CPU time can potentially reach a higher throughput before saturating the available CPU resources. The maximum input data rate is determined by increasing the rate in steps until the system can no longer keep up.

Test results for shared-memory/socket communication				
system configuration	single-CPU system		dual-CPU system	
ControlHost mechanism	sharedmem	socket	sharedmem	socket
CPU time for `dfilter`				
total per slice	17.4 ms	17.5 ms	18.7 ms	18.9 ms
total in % of real time	57 %	58 %	62 %	62 %
filter algorithm per slice	17.2 ms	17.2 ms	18.1 ms	17.6 ms
CPU time for `dispatcher`				
per slice	2.3 ms	2.9 ms	2.2 ms	2.5 ms
in % of real time	8 %	10 %	8 %	8 %

Table 5.1: Test results for shared-memory/socket-based communication

Shared memory

We compare CPU usage for shared memory and network communication, both for 1-CPU and 2-CPU nodes. In these tests, 5 data generators produced simulated data with a hit rate of 70 kHz and a timeslice length of 10 ms (as stated in the TDR [3]). No waveform date are produced, only SPE events. The timeslices are divided over three `dfilter` processes, so that each filter process has 30 ms available to process a slice. A complete slice is typically about 60 KByte, so the dataflow into each filter process is 2 MByte/s.

Results of measurements are given in Table 5.1. We calculated the actual CPU time used (as reported by the OS) per time slice and as a percentage of the real time.

A single `dfilter` process runs slightly slower on a 2-CPU node than on a 1-CPU node. This is presumably a result of inefficient scheduling (processes move between the two processors) and overhead due to contention for memory access. Shared memory makes the `dispatcher` a bit faster but has little effect on the overall time used by `dfilter`. On a 2-CPU node, shared memory seems to slow down the filter algorithm. Typically, the `dfilter` will run on one processor and the `dispatcher` on the other one. This situation may suffer from cache coherency overhead (when one processor modifies the contents of the shared memory, the other processor must update its cache).

Dual-processor systems

We now compare a 1-CPU node running one `dfilter` process against a 2-CPU node running one process and a 2-CPU node running two `dfilter`s. In these tests, only two farm nodes are used for filtering. The data production rate remains the same, so the workload of the nodes is higher. In the cases of one filter per node, the slices are divided over two filters in total, so each filter may take 20 ms (2 × 10 ms) to process a slice. In the case of two filters per node, the slices are divided over four filters in total, so each filter may take 40 ms to process a slice. However, two filter processes must share a single node during these 40 ms. We also changed the input event rate to find the point where the filters can no longer keep up with the incoming data.

Results are shown in Table 5.2. The run with 1-CPU nodes failed; the input rate was too high for the filter processes and much of the data was discarded. The performance values are

Test results for single/dual-CPU systems			
system configuration	single-CPU	dual-CPU	dual-CPU
`dfilter` processes per node	1	1	2
time between slices for a filter	20 ms	20 ms	40 ms
CPU time for `dfilter` at 70 kHz input			
total per slice	16.9 ms [4]	19.0 ms	19.1 ms
total in % of real time	51 %	94 %	47 %
filter algorithm per slice	20.2 ms	18.5 ms	18.3 ms
CPU time for `dispatcher` at 70 kHz input			
per slice	7.5 ms [4]	2.1 ms	2.8 ms
in % of real time	36 %	11 %	14 %
data flow through `dispatcher` at 70 kHz	3 MByte/s	3 MByte/s	6 MByte/s
maximum input rate without overflow	50 kHz	70 kHz	100 kHz

Table 5.2: Test result for single/dual-CPU systems

re-calculated to account for the data they did process, but these results cannot be compared with other test results. For 2-CPU nodes, there is little difference between running one or two filters per node at 70 kHz. However, the one-filter configuration cannot keep up with an input rate above 70 kHz since, together with the `dispatcher` it already needs 94 % of the available CPU time. The two-filter configuration splits the work evenly over both processors and continues to work correctly up to a 100 kHz input rate.

Conclusion

A farm with three `dfilter`s on 2-CPU nodes can easily handle the output of five LCMs generating 70 kHz SPE data. This confirms that the present configuration can be used for data filtering in the *sector-line test setup.*

In our present setup, two 2-CPU nodes are just enough to handle all SPE data. The full detector will consist of 300 LCMs, which produce wavefrom hits in addition to the SPE hits. Assuming that the workload of the filters scales linearly with the size of the detector, the full detector would need at least $2 \times 300/5 = 120$ 2-CPU nodes; slightly more than the 100 nodes originally planned. This may be compensated by using faster CPUs or by optimizing the algorithms.

The use of shared memory makes very little difference for the overall performance of the data filter software; certainly much less than we had expected. We could consider changing the shared memory mechanism to make it faster. Alternatively, we could abandon the shared memory approach and focus our efforts on other issues.

A 2-CPU node can reach a significantly higher throughput than a 1-CPU node. This holds even when only a single `dfilter` process runs on each node, since the `dispatcher` can use the second processor. When two `dfilter` processes are started on a 2-CPU node, it can reach almost the same throughput as two 1-CPU nodes. A farm with 2-CPU nodes may then be less expensive since the number of nodes can be lower. This should be taken into account when selecting a farm configuration for the full 10-string detector.

[4] Filter program cannot keep up with input rate and discards data; this value is calculated per input slice, not per processed slice.

Chapter 6

Conclusions

Three different aspects of the ANTARES project were discussed in this thesis. We formulate our conclusions for each of these aspects.

Data transmission method

Physics data are sent from LCM devices to farm nodes according to a timeslice pattern. Two important points are the selection of a transport layer protocol, and of a buffering policy. We discussed two possible transport protocols: TCP and UDP. TCP was chosen for the DAQ system because it offers reliable data transfers.

We stressed the importance of buffering in order to avoid bottlenecks in the transmission process. The buffering policy inside TCP seems inadequate for the DAQ traffic pattern. We therefore developed an explicit buffering policy at the level of the LCM software. This policy, called *delayed transmission*, is based on the principle of barrel shifting. It is efficient and robust, except when dealing with peaks in data rates. Several extensions are possible to deal with fluctuating data rates; if the peaks are short, no modifications may be necessary at all.

The ANTARES DAQ group has adopted the delayed transmission technique for implementation in the LCM software.

Network traffic simulation

We constructed a simulation model of the DAQ network. This model was used to predict the network performance of the DAQ system for a number of scenarios.

The results indicate that TCP data transmission without explicit buffering is not capable of handling the workload of the 10-string detector. With delayed transmission enabled, the system handles a 10-string workload just as well as a 1-sector workload. This is a strong argument in support of the delayed transmission policy.
The value $k = 8$ for the number of simultaneous sending LCMs was found to give good results. Data rate fluctuations due to bioluminescence cause a significant decrease in performance, even with the delayed transmission algorithm. This problem can be sufficiently reduced by implementing the proposed mechanism for data cutoff during peak rates.

Direct transmission with UDP loses almost all data. This loss is eliminated by delayed transmission, but reappears when bioluminescence is taken into account. A simple UDP protocol without error correction is therefore inadequate, unless a continuous loss of approximately 1% of the data is accepted.

Data filter prototype

A data filter program has been designed and implemented. This program is a combination of the existing filter algorithm and a new control framework. The control framework handles the interaction with other components of the DAQ system and the management of the data queue. Command messages and raw data are transferred through ControlHost.

Tests have shown that the filter program cooperates correctly with the RunControl, the DataWriter and the simulated LCMs. The performance of the filter program was measured in a number of different configurations. We discovered that a farm of three dual-processor nodes is sufficient to filter sector line data with 70 kHz SPE hits. Assuming that the workload of the filter scales linearly with the detector size, a farm of 120 dual-processor systems of the presently used type is required to filter data from a 10-string detector. When properly configured, a dual-processor system can reach almost the same throughput as two single-processor systems.

Appendix A

Experiments with zero-copy UDP transmission under Linux

A.1 Introduction

The off-shore part of the ANTARES DAQ will be controlled by embedded boards with a Motorola MPC860 PowerPC. The design of the DAQ calls for fast transmission of large amounts of data over Ethernet. To this end, the PowerPC boards are equipped with a 100 Mbit Ethernet interface, connected to the processor's on-chip Fast Ethernet Controller (FEC). The performance of this interface as we have measured it under Linux, is well below the theoretical limit of 100 Mbit/sec. It is believed that the speed of the CPU and the bandwidth of the memory bus are a bottleneck for the throughput of the FEC.

Experiments in Saclay with the VxWorks OS on the MPC860 indicate that the use of a zero-copy interface to the network hardware can greatly enhance the throughput. Unfortunately, a zero-copy interface is currently not available in the standard Linux kernel. To be able to measure the performance impact of zero-copying under Linux, we implemented an ad-hoc zero-copy interface for UDP transmission. We also modified the low-level FEC driver to implement the Linux scatter/gather interface. These modifications are restricted to transmission of UDP/IP datagrams.

A.2 Copying

In most OSs, the standard implementation of single datagram transmission proceeds as follows (see Fig. A.1):

1. The application prepares a buffer in memory, containing the data to be transmitted. It then invokes a system call, passing a pointer to the message buffer and an indication of the message length.

2. The kernel allocates a new buffer in a protected address space, copies the contents of the application buffer into this new buffer and adds protocol headers and checksums.

The packet is then passed to the low-level network driver, which adds the buffer in its transmission queue.

3. The kernel returns from the system call, **without guarantee** that the transmission has completed. The application can continue its work, for example by filling the message buffer with new data.

4. Some time later, the network hardware fetches the contents of the kernel buffer through DMA, and transmits it to the network. It then invokes an interrupt to inform the kernel that the buffer may be released.

The extra copy to a new buffer in step 3 imposes a performance penalty. This penalty can be significant, especially on hardware where memory access is expensive (like our PowerPC board).

Why then, does the kernel make this copy? Copying the data is done for four reasons:

- After the system call returns, the application is free to use the data buffer for other purposes (a new data packet for example). This would not be possible without copying, since in general, the network hardware fetches the data after the return of the syscall.

- During copying, the kernel also calculates a checksum over the buffer contents. This checksum is stored in the protocol header, and can be used by the receiver to detect

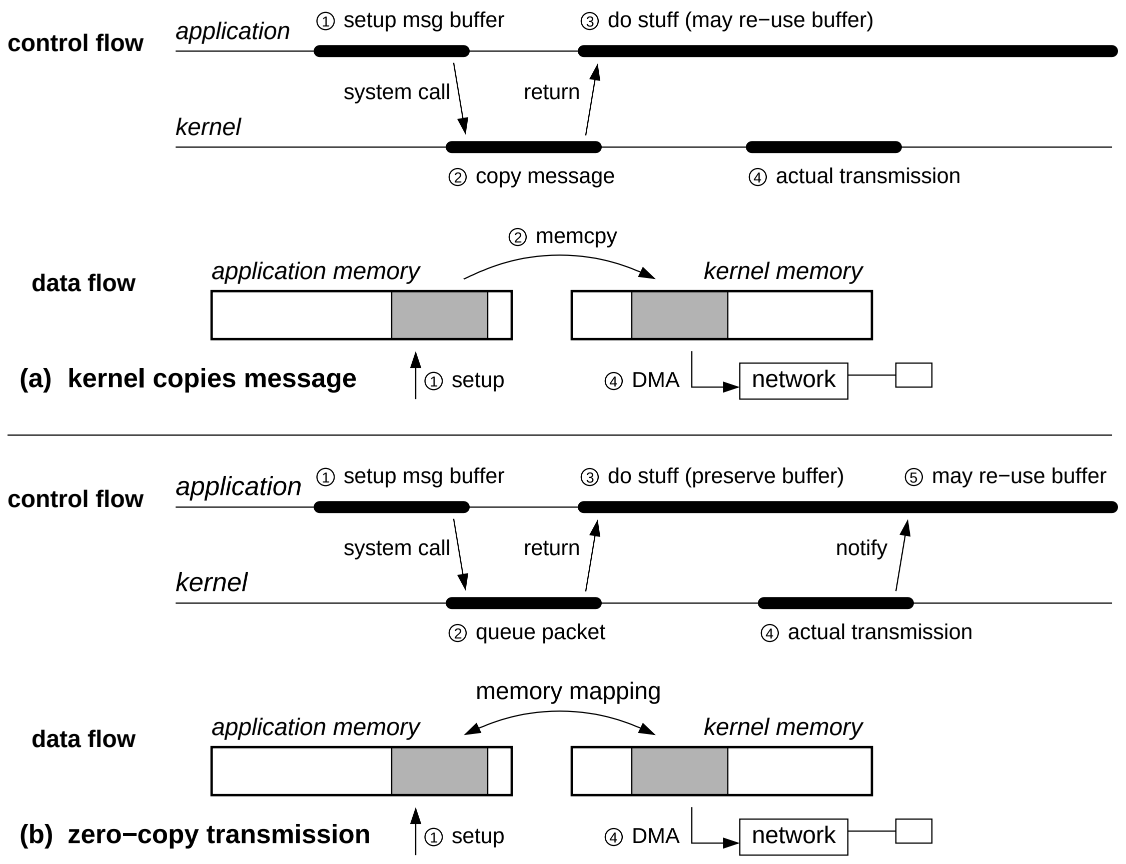

Figure A.1: Network transmission (**a**) with copying, (**b**) zero-copy

corrupted datagrams. Even without copying, the kernel would still need to go through the buffer contents just to calculate the checksum. Since memory bandwidth is the most important performance factor, this would take almost as much time as copying.

- Network cards often require the packet data to be in a contiguous region of physical memory to facilitate DMA transfers. However, memory locations that seem contiguous to an application may be far apart in physical memory due to mappings in the virtual memory system. The application's message buffer is therefore in general not a contiguous region of physical memory. The copy buffer allocated by the kernel is guaranteed to be contiguous.

- Some reserved room is needed before the start of the data to store protocol headers. The application programmer in general can't or doesn't want to deal with this.

A.3 Eliminating the copy

We implemented an ad-hoc interface to provide zero-copy UDP transmission under Linux. This is done in the form of a Linux 2.4.4 kernel module which adds a character device driver to the system. An application can then call this driver to transmit a datagram; the normal `socket`-based interface is not used.

To allow eliminating the copy, we needed to find a different way of addressing the points which are normally achieved by copying. The first two points are easy: the semantics of our ad-hoc interface simply don't guarantee that the buffer can be reused after the system call. Instead, the application should preserve the buffer contents until it has transmitted some fixed number of other buffers.

We completely avoid checksum calculation by leaving out the UDP checksum in the protocol header. This might reduce the reliability of the protocol. However, the checksum at the Ethernet level alone should suffice to catch most damaged packets. Modern high-speed network interface cards often have facilities to calculate IP checksums in hardware, allowing a nicer solution to the checksum problem. Unfortunately, the PowerPC FEC has no such provisions.

We developed two separate methods to deal with the other two points: either by mapping a kernel buffer in application memory, or through the network driver's scatter/gather support.

A.3.1 The mmap method

After opening the device driver, the application invokes the `mmap` syscall to map a kernel buffer into application memory space. Since this buffer is allocated by the kernel, it is guaranteed to be contiguous in physical memory.

The application writes data packets directly into the mapped buffer, taking the responsibility of reserving sufficient room before and after the data. Transmission of a packet is requested through an `ioctl` syscall.

This method is much less flexible than the standard `socket`-based interface: the application is forced to put its data in a pre-arranged area and to leave room before and after each packet. Consequently, the dividing of data into packets must take place in an early stage, complicating the design of the application.

A.3.2 The kiobuf method

Modern network interface cards often support scatter/gather DMA transactions. It removes the need to store the packet in a single contiguous range of memory. Instead, a packet may consist of multiple fragments, which are scattered over physical memory. During packet transmission, the network hardware *gathers* the packet contents from the various memory locations.

Recent Linux kernels (2.4.4) allow fragmented network packets **provided that** both the network hardware and the low-level network driver support scatter/gather DMA. Unfortunately, the Linux 2.4.4 version of the FEC driver doesn't support scatter/gather. However, inspection of the driver code and documentation from Motorola indicated that the MPC860 hardware can handle scatter/gather through its regular buffer descriptor mechanism. We modified the FEC driver code to take advantage of this mechanism and implemented the fragmented-packet-interface.

This feature eliminates the problem with non-contiguous application buffers, allowing us to map the application buffer to kernel memory space instead of the other way around. It also removes the need to explicitly reserve room for protocol headers: these headers can comfortably be stored in an additional first fragment. The application is now free to store the packet contents wherever it wants. Transmission of a packet is requested through a `write` syscall. The device driver then uses the new `kiobuf` interface in the Linux kernel to map the application buffer into kernel memory space.

A.4 Performance

We measured the performance of our extensions on an RPX CLLF board equipped with a Motorola MPC860T PowerPC processor running at 50Mhz with a 50Mhz bus, 16 MB DRAM and a QS6612 100baseT Ethernet interface. The operating system consisted of the standard Linux 2.4.4 kernel with the runtime environment provided by the MontaVista Hardhat development kit. To work around a known bug in the MPC860T data cache, we enabled the *CPU6 Silicon Errata* option in the kernel configuration.

Our test setup consisted of the PowerPC board, connected to a Intel-based Linux PC through 100Mbit UTP. The performance of the network interface was measured by transmitting series of UDP datagrams from the PowerPC to the Intel PC. Socked-based transmission was done through the `ttcp` program. In all cases, we used the `ttcp` program on the Intel PC to receive the datagrams and to obtain the performance results presented in Table A.1. A series of 10,000 datagrams were transmitted in each measurement. All measurements were repeated 3 times and the median values were selected.

interface	**checksums**	**dgram size**	**KByte/sec** [1]	**Mbit/sec** [2]
standard socket		1400	3157	25.9
standard socket		8192	4334	35.5
copy extension [3]	NO	1400	4251	34.8
copy extension [3]	YES	1400	3801	31.1
mmap extension	NO	1400	6872	56.3
mmap extension	YES	1400	5120	41.9
kiobuf extension	NO	1400	4758	39.0

Table A.1: Transmission performance results

A.5 Discussion

Our best result (the `mmap` extension without checksums) shows a factor 2 performance enhancement when compared to the standard socket interface with the same datagram size. However, it should be noted that the extension interface is severely less flexible than the standard socket interface, especially on the following points: no UDP checksums (reduced reliability); limited to UDP (no TCP, no streaming, no automated retransmission); application data buffers must reside in a fixed address region; the application is responsible for reserving room at the head and tail of the buffer.

In its current form, this zero-copy interface doesn't seem suitable for employment in the ANTARES DAQ system. However, our results clearly demonstrate (1) that the MPC860 hardware is capable of delivering a much better throughput than indicated by previous measurements and (2) that copying of data buffers on the MPC860 imposes a significant performance penalty and should be avoided if possible.

[1]Values obtained from the receiving `ttcp` program.

[2]Calculated as Mbit = KByte * 1024 * 8 / 1,000,000; this is not exactly right: it leaves out the IP and Ethernet protocol headers.

[3]Copies the data to a kernel buffer but otherwise uses the same extension interface.

Appendix B

Switch buffer length measurement

B.1 Packet buffering in a switch

An Ethernet *switch* is a device for interconnecting network nodes. Each of the nodes is physically connected (by a cable) to a *port* on the switch. For each incoming packet, the switch examines the destination address and then forwards the packet to the appropriate port; i.e. the port to which the destination device is connected.

When multiple sources are sending packets to the same destination at the same time, the switch will be in trouble. It is asked to forward multiple packets to a single output port, but only one packet at a time can actually be transmitted on this port. This situation is commonly called *oversubscription* of an output port. The usual solution is to transmit one packet directly, and keep the other packet temporarily in a memory buffer. The buffered packet is transmitted as soon as transmission of the first packet has completed (Fig. B.1).

This solution works quite well in most situations. However, when multiple sources produce a continuous stream of packets for a single destination, the switch will eventually run out of buffer memory. When that happens, the switch has no other choice than to discard newcoming packets until there is room in the buffer again.

A switch's sensitivity to the oversubscription problem depends on the amount of buffer memory available. For some applications (like the ANTARES DAQ simulations), it is important to understand the buffer behaviour of the switches. Unfortunately, this information is usually not available from hardware vendors and manufacturers.

In this chapter, we develop a method for measuring the packet buffer size of a switch. We use this method to measure the packet buffer size of the Gigabit Ethernet switch used in the ANTARES test-farm.

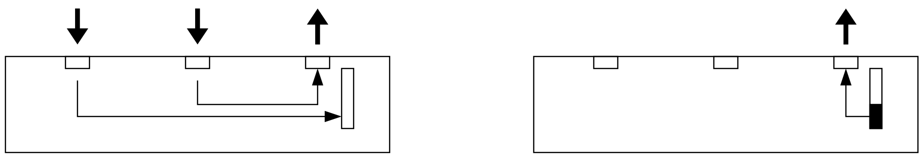

Figure B.1: Packet buffering in a switch

B.2 Method

We attempt to measure the size of the output buffers in an Ethernet switch by flooding it with packets. The output of the switch is analyzed to determine which packets were dropped. From this information, we deduce the maximum number of packets that the switch can store in its buffers.

It is assumed that each output port is equipped with a static amount of buffer space. Most modern switches (so called *output-buffered switches*) are known to use this strategy. It then follows that the buffering behaviour of separate output ports is independent.

It is possible that the switch can store a fixed *number of bytes* in its buffers, such that the total size in bytes of all buffered packets cannot exceed this limit. It may also be the case that the switch can store a fixed *number of packets* in its buffers, such that the number of buffered packets cannot exceed this limit. We distinguish these cases by performing the measurement with large flooding packets as well as with smaller packets.

Method 1: Flooding with multiple senders

Our first method is based on oversubscription of the switch output. Multiple senders are transmitting packets to a single receiver at full line speed. The switch can't forward packets to the receiver faster than line speed and is forced to store packets in buffer memory. At some point, the buffer is full and the switch must drop a packet. This drop is detected by the receiver and used to estimate the buffer size.

We assume s senders, transmitting at full line speed, and 1 receiver, receiving at full line speed. The switch has room for b packets in its buffer. Each packet contains a sequence number giving its position in the packet stream. All senders start transmission at the same time at sequence number zero.

The incoming flow to the switch is s times the outgoing flow from the switch. At some point, the switch has received k packets from each sender, so $s \times k$ packets in total. No more than k packets have been transmitted to the receiver, so the buffer should contain at least $(s-1) \times k$ packets. As soon as $(s-1) \times k \geq b$, the switch is forced to drop at least one packet.

The receiver can analyze the stream of packets it receives from the switch, and register the first missing sequence number (from any sender). If the first missing packet has sequence number k, it follows that the switch could successfully store at least $(s-1) \times k$ packets in its buffer. This provides a lower bound for the packet buffer size.

This method looks solid. However, it relies on the fact that the senders transmit at full line speed. A low-end Linux PC is unfortunately not fast enough to transmit at the full 1 Gbit/s rate. We could compensate for this in our calculations, provided that we know the exact transmission rate of the senders. But the transmission rate is difficult to measure reliably, and may be subject to fluctuations due to scheduling in the operating system.

Another requirement is that all senders start transmitting at exactly the same time. The receiver can verify this by looking at the relative position of the first packets from different senders in the stream of received packets. We checked that this synchronisation works quite well in practice; the distance between first packets is in the order of five packets or less.

Method 2: Flow control

The second method is based on a feature in the Ethernet standard: full duplex flow control [17]. A device with a full duplex Ethernet link may transmit a special *pause frame* to its link partner, requesting that the remote device temporarily stops transmitting.

This feature is normally used when a network device runs out of buffer space. For example, a switch may request a PAUSE on its input ports when an output buffer is filling up. A network card may invoke flow control when its receive buffer ring gets full. The PAUSE operation is handled transparently by the hardware (or firmware) of the network devices.

We abuse the flow control feature to measure the switch buffer length. Just before the sender starts transmitting, the receiver sends a PAUSE request to shut the output port of the switch. Since the switch is unable to forward packets to the receiver, it must either store or drop all incoming packets. After the end of the transmission period, the PAUSE request expires and the switch transmits all buffered packets to the receiver. The receiver counts all incoming packets. The resulting number is a lower bound for the switch buffer size.

This method requires that the switch respond correctly to our PAUSE frame, and that the PAUSE period lasts until after the sender finishes its transmission. PAUSE operation of the switch is verified at the receiver by timing the interval between start of transmission and reception of the first packet. When flow control is enabled, this time goes from 3 ms up to 33 ms (about equal to the requested PAUSE period). The receiver also checks that no frames are transmitted by the sender after the flow control period by looking at the sequence numbers of received packets.

Two variants of this method, with multiple senders and multiple receivers respectively, are tried to verify some of our assumptions about the switch. With multiple senders, the total received packet count should be the same as with a single sender (an input-buffered switch would behave differently). With multiple receivers, each of the receivers' packet counts should be the same as with a single receiver (sharing buffers across output ports would cause different results).

B.3 Equipment and configuration

switch under test
3Com Gigabit Ethernet switch: 3C17702 Switch 4900 SX 12-port

transmitting PC nodes
Dual Pentium III 933 MHz, 265 KByte cache, 1 GByte SDRAM PC133,
VIA Apollo Pro133A
3Com 3C985 (acenic) Gigabit Ethernet adapter, flowcontrol disabled
Linux 2.4.14-pre3 single-processor kernel

receiving PC nodes
as transmitting PC nodes but with Ethernet flowcontrol enabled

B.4 Experiments

Two C programs (named, of course, `send` and `recv`) were written to implement the tests described above. The UDP/IP protocol is used for packet transmission. Flow control is implemented by creating pause frames in software and transmitting them through the low level packet interface. Synchronisation between hosts is achieved by broadcasting a *trigger* packet.

All measurements are repeated five times and the best result is selected. Buffer sizes in bytes are calculated as $(s + 28 + 18) \times p$, where s is the packet length in bytes and p is the buffer length in packets. The extra 28 bytes are due to the UDP/IP header; the 18 bytes are for the Ethernet header and CRC. Results are shown in Table B.1.

Method 1: Flooding with two transmitters

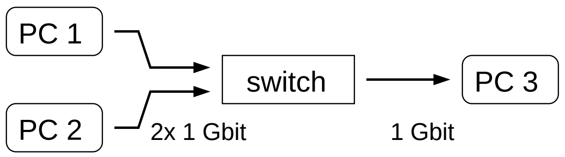

Figure B.2: Experiment with two transmitters

Two transmitting PCs and one receiving PC are connected to the switch. The receiving PC broadcasts a packet to trigger the transmitters. On reception of the trigger packet, the transmitting PCs start flooding the receiving PC with 1000 packets. The receiver analyses incoming packets and registers the sequence number of the first missing packet from either transmitter. This first missing packet's number approximates the buffer length.

Method 2a: Flow control with one transmitter

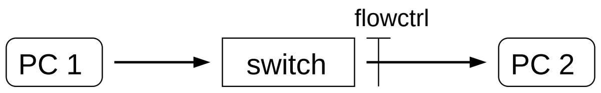

Figure B.3: Experiment with flow control

One transmitting PC and one receiving PC are connected to the switch. Immediately after sending a trigger packet, the receiver sends a flow control frame to the switch, forcing it to inhibit transmission for a period of $\sim$ 30 ms. On reception of the trigger packet, the transmitter starts flooding the receiver with 1000 packets. After the flow control pause, the receiver counts the number of packets received. This packet count approximates the buffer length.

Method 2b: Flow control with two transmitters

Two transmitting PCs and one receiving PC are connected to the switch. Immediately after sending a trigger packet, the receiver sends a flow control frame to the switch, forcing it to inhibit transmission for a period of $\sim$ 30 ms. On reception of the trigger packet, both transmitter start flooding the receiver with 1000 packets. After the flow control pause, the receiver counts the number of packets received from each of the transmitters. The sum of these packet counts approximates the buffer length.

Method 2c: Flow control with two receivers

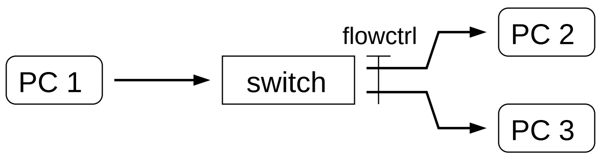

Figure B.4: Experiment with two receivers and flow control

One transmitting PC and two receiving PCs are connected to the switch. After sending a trigger packet, both receivers send a flow control frame to the switch, forcing it to inhibit transmission for a period of $\sim$ 30 ms. On reception of the trigger packets, the transmitter starts flooding both receivers with 500 packets each. After the flow control pause, the receivers count the number of packets received. The packet counts approximate the buffer length.

B.5 Conclusion

The flooding method is not reliable because it assumes that both senders transmit at full line speed, which is not correct as we know. Results for this method fluctuate, and differ from results based on flow control. We conclude that the flooding method does not produce reliable results.

The foundation of the flow control method is simple and seems correct. We have checked that the mechanisms used in this method (flow control, sender synchronisation) actually work in practice. Results from the flow control method are consistent between repeated measurements and between different experiments. We are convinced that this method produces a reliable estimate of the buffer size in the switch

The experiments with large- and small packets result in a different number of packets, but an almost equal number of bytes in the buffer. We deduce that the switch has room for a fixed number of bytes, not a fixed number of packets.
Experiments with flow control and different numbers of senders or receivers all produce the same results. We conclude that the switch is indeed output-buffered and that each output port has a fixed amount of buffer space allocated to it.

Flooding with two transmitters	
first packet received after trigger	3 ms
size of flood packets	1460 bytes
first dropped	83 84 107 **113**
estimated buffer size	113 packets ⇒ 166 KByte
size of flood packets	365 bytes
first dropped	**625** 616 595 614 610
estimated buffer size	625 packets ⇒ 250 KByte
Flow control with one transmitter	
first packet received after trigger	33 ms
size of flood packets	1460 bytes
received packet count	**74** 72 72 73 74
estimated buffer size	74 packets ⇒ 108 KByte
size of flood packets	365 bytes
received packet count	**253** 247 250 251 250
estimated buffer size	253 packets ⇒ 100 KByte
Flow control with two transmitters	
size of flood packets	1460 bytes
received packet count	**38+36** 37+36 38+36 37+36 38+33
estimated buffer size	74 packets ⇒ 108 KByte
size of flood packets	365 bytes
received packet count	**131+120** 130+118 122+115 124+124 127+120
estimated buffer size	251 packets ⇒ 99 KByte
Flow control with two receivers	
size of flood packets	1460 bytes
received packet count	71,71 **73**,72 72,72 72,72 73,72
estimated buffer size	73 packets ⇒ 107 KByte
size of flood packets	365 bytes
received packet count	247,245 254,244 250,242 243,**256** 250,247
estimated buffer size	256 packets ⇒ 101 KByte

Table B.1: Measurement results

The switch under test has a buffer size of approximately 108 KByte per port, corresponding to 74 full-size Ethernet packets. Since only one switch was tested, our results may be specific for the chosen switch model. But we expect that other switches would produce values in the same order of magnitude. This means that the value of 100 packets for the switch output buffer in the network simulations (Chapter 4) was a reasonable approximation.

Bibliography

[1] Joris van Rantwijk, "Dataflow Mechanisms for the Antares Data Acquisition System" (May 2001); `http://deadlock.et.tudelft.nl/~joris/antares/`

[2] *Antares Website*; `http://antares.in2p3.fr/`

[3] *Technical Design Report of the Antares 0.1 km^2 project*, version 1.0, Antares Internal Document (Jun 2001); Chapter 4: Readout, trigger and DAQ.

[4] Jonathan Allday, *Quarks, Leptons and the Big Bang*, IOP Publishing (1998).

[5] Wind River Systems, *VxWorks Website*; `http://www.windriver.com/products/html/vxwks5x.html`

[6] Paul Jay Lucas, *An object-oriented language system for implementing concurrent, hierarchical, finite state machines*, Master's Thesis, University of Illinois (1993).

[7] *ControlHost: Distributed Data Handling Package*, version 2.2, CASPUR Consortium (Mar 1995).

[8] Ruud van Wijk, *CHOO library manual* (2001); `http://www.nikhef.nl/~ruud/HTML/choo.html`

[9] M.F. Letheren, "Switching techniques in data acquisition systems for future experiments", *1995 CERN School of Computing*, pp 245-275 (1995).

[10] Joris van Rantwijk, *Data buffering in the Antares DAQ system*, Antares Internal Note Antares-Soft/2001-011 (Nov 2001).

[11] Raj Jain, *The Art of Computer Systems Performance Analysis*, Wiley (1991).

[12] *The Network Simulator – ns-2* `http://www.isi.edu/nsnam/ns/`

[13] *MIT Object Tcl* `http://www.isi.edu/nsnam/otcl/`

[14] Randy Brown, "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem", *Communications of the ACM*, vol 31, no 10, pp 1220-1227 (Oct 1988).

[15] N. Palanque-Delabrouille, *Optical background measurements*, Antares Internal Note Antares-Site/1998-002 (Jun 1998).

[16] S. Anvar, *Antares DAQ Format Proposal* v4, Antares Internal Document (Nov 2001).

[17] *IEEE 802.3 International Standard for Local Area Networks*, 2000 Edition; Annex 31B MAC Control PAUSE operation, pp 1472-1481 (2000).