NETWORK TIME SYNCHRONISATION IN REAL TIME DISTRIBUTED COMPUTING SYSTEMS

by

Kenneth Thomas Crellin

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science (Applied Science)

Electrical Engineering Department
University of Cape Town

1998

Approved by _____
Chairperson of Supervisory Committee

_____

_____

_____

Program Authorized
to Offer Degree _____

Date _____

Declaration

I declare that this thesis is my own work. Where collaboration with other people has taken place the parties are indicated in the acknowledgements or references as appropriate.

This work is being submitted for the Master of Applied Science degree at the University of Cape Town, department of Electrical Engineering. It has not been submitted to any othe ther degree or examination.

Signed by candidate

K.T Crellin _____

Date _____

Index Terms: real time protocols, system integration, time synchronization, distributed systems, fiber optic networks, fault tolerant systems, survivable services, network time protocol, xpress transport protocol.

University of Cape Town
Abstract

NETWORK TIME SYNCHRONISATION IN REAL TIME DISTRIBUTED COMPUTING SYSTEMS

by Kenneth Thomas Crellin

Chairperson of the Supervisory Committee: Mr. M. J. Ventura
Department of Electrical Engineering

In the past, network clock synchronization has been sufficient for the needs of traditional distributed systems, for such purposes as maintaining Network File Systems, enabling Internet mail services and supporting other applications that require a degree of clock synchronization. Increasingly real time systems are requiring high degrees of time synchronization. Where this is required, the common approach up until now has been to distribute the clock to each processor by means of hardware (e.g. GPS and cesium clocks) or to distribute time by means of an additional dedicated timing network. Whilst this has proved successful for real time systems, the use of present day high speed networks with definable quality of service from the protocol layers has lead to the possibility of using the existing data network to distribute the time. This thesis demonstrates that by using system integration and implementation of commercial off the shelf (COTS) products it is possible to distribute and coordinate the time of the computer time clocks to microsecond range. Thus providing close enough synchronization to support real time systems whilst avoiding the additional time, infrastructure and money needed to build and maintain a specialized timing network.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The author wishes to thank C^2I^2 Systems (Pty) Ltd. for the use of their research laboratory, most of the work done in this thesis was done while employed in their Kenilworth offices. Without the help of my colleagues there the task would have been far more difficult. I would especially like to thank Mr. Manie Steyn and Mr. Etienne de Villiers for their advice and support during the development of the software stack, Mr. Conrad Vermeulen for the initial porting of the Xpress Transport Protocol and Mr. Johan Kannemeyer for running the tests of the software implementation. Without Dr. Richard Young's work on real time distributed architecture and protocol strategies this thesis would not have been possible.

I would like to thank Dr. Andrew Hutchison of the Computer Science Department at UCT for his advice and time spent proofreading the technical aspects of this thesis. Thanks go to Mr. John Fenton of Mentat Inc. (developers of the STREAMS based XTP), Mr. Axel Fischer of SysKonnect DatenSysteme (developers of the STREAMS based SCO PCI FDDI device driver), Mr. Stephen Porter of Wind River Systems (developers of the VxWorks Real Time Operating System) and Mr. Harlan Stenn and Prof. David Mills from the University of Delaware, (distributors of the standard distribution NTP) for tireless email support. Mr. Greg Seibert of LynxOs provided help when testing NTP on the LynxOs operating system for which I am grateful. Thanks especially to Ms. Lara Smith for the thorough proof reading.

Lastly, I would like to thank my supervisor Mr. M.J. Ventura for guidance and assistance during the last two years.

ABBREVIATIONS

AAL	ATM Adaption Layer
ATM	Asynchronous Transfer Mode
BSD	Berkeley Standard Distribution (Unix version)
COTS	Commercial Off The Shelf
DLL	Data Link Layer
DLPI	Data Link Provider Interface
DTS	Digital Time service
FDDI	Fiber Distributed Data Interface
FSF	Free Software Foundation
GPS	Global Positioning System
GNU	GNU's Not Unix (Unix Tools)
IMS	Information Management System
ISR	Interrupt Service Routine
LAN	Local Area Network
LLC	Logical Link Control
MAC	Media Access Control
MBARI	Monterey Bay Aquatic Research Institute
MBII	Multibus II
MIB	Management Information Block

NTS	Network Time Services
NTP	Network Time Protocol
NIC	Network Information Card
ODT	Open Desk Top
OS	Operating System
OSI	Open Systems Interconnection
PCS	Probabilistic Clock Synchronization
POSIX	Portable Operating System Interface eXtension
SBA	Synchronous Bandwidth Allocator
SCO	Santa Cruz Operation
SNMP	Simple Network Management Protocol
SVR4	System Five Release Four (Unix version)
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
VFO	Variable Frequency Oscillator
VLAN	Vetronics LAN
XTP	Xpress Transport Protocol
WRS	Wind River Systems
WWW	World Wide Web

Chapter 1

1. INTRODUCTION

Network time synchronization is required for coordinating the time on real time mission critical systems. The needs of modern distributed computing require offsets of the time between nodes in a network to be in the order of the low milliseconds. This allows high-speed computer networks to distribute and coordinate the processing of tasks while maintaining time and sequence integrity.

This is important to ensure that nodes do not act upon "stale data" (data that should be disregarded if not processed before a bounded time period).

> *"The correctness of the computation depends not only on the logical correctness but also on the time at which the results are produced" [SH194]*

The issues related to distributed time services have become more important in the last decade and have now reached the stage where the network is truly the computer, with various computers capable of working cooperatively to deliver a specified result. Timing and synchronization of distributed services is often crucial to the success of the system in the present day. Tasks traditionally accomplished by a single server and mainframes have now become client-server type applications, utilizing multiple computers operating over a LAN to accomplish the same tasks in a more effective manner. The drop in prices of hardware and the rapid increases in technology have lead to fast and efficient distributed workstations that can now do the job of the mainframe. They cost a fraction and provide greater survivability through means of geographic dispersion

and greater reliability through redundancy (should one fail, the processing can be shared among the remaining stations). The problem that is posed is the temporal coordination of this distributed architecture. As O'Donoghue and Marlow [DON96] note:

> *"As more processors are added, the time synchronization infrastructure must be scaled to meet these new requirements. Experience has shown that techniques which adequately synchronized two computers are not necessarily capable of synchronizing dozens of computers."*

Parallel and distributed computing, with client-server relationships, is becoming more blurred, with both server and client capable of interchangeable roles within an integrated solution (e.g. a time server may also be a file transfer client). Tasks that were once the domain of parallel processing, using a single multiprocessor computer and a parallel bus, can now be accomplished over a high speed network. Thus as technology progresses, tasks that would have taken place over a bus type of architecture are now taking place across a network. Essentially the LAN is becoming the bus. This leads to demands for "bus type" clock synchronization across the LAN. The major hurdle facing this new networked computer is the issue of synchronized time clocks. In the single server system time synchronization was not a problem, as it would only have one clock from which to source the time. O'Donoghue and Marlow [DON96] state that this "common understanding of time" is "vital", while Suri et al. [SUR94] describe a global time base as a prerequisite for a distributed real time system. From the above it can be seen that the need exists for closer and more stringent coherence of the time clocks of the different computers, with bounded response time for the time stamps requested of the distributed nodes.

"Evolving applications including sensor correlation, data extraction, and distributed data bases require all participating processors to have a single consistent understanding of time." – O'Donoghue and Marlow [DON96]

Examples of such systems can be found in many leading edge technology enterprises. The results from this research will be applicable to any system where information and messages become stale very quickly. Timing services have grown from the initial stage of maintaining network file systems. They now encompass Internet wide global time, and other loosely coordinated services to hard real time distributed database updates, air traffic control and electronic warfare. This thesis investigates this last area in particular, where time coordination in the low microseconds is required.

This thesis can be divided into three main sections as is shown in Figure 1. Each section in turn can be divided into components. The diagram clearly illustrates how the main sections are grouped, and how the various chapters relate to one another.

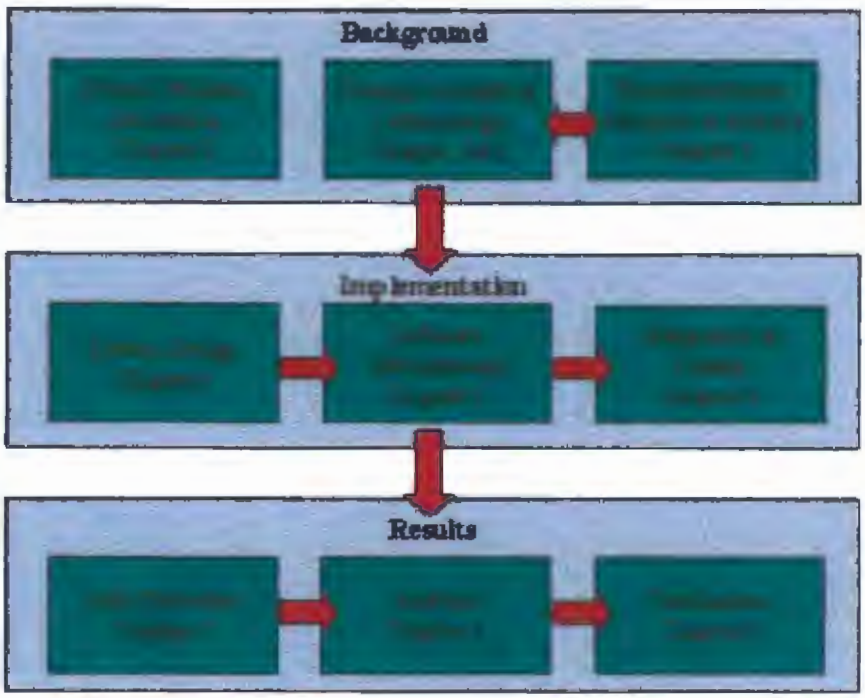


Figure 1: Thesis Overview

Chapters 1 to 3 lay the foundation of vital background information. This includes comparing the notion of distributing time with distributing clocks as well as an examination of the systems requirements for hard real time distribution of bounded time. The next three chapters describe the proposal and implementation of a system integration of commercial off the shelf (COTS) products to accomplish time distribution. The final chapters demonstrate how analysis of the data is done and describe conclusions reached from the analysis. Two sets of references have been included, the first relating to academic articles referenced and the other relating to the World Wide Web referenced sites, both set of references are used in the thesis.

This thesis contributes to the doctoral thesis of Young [YOU96] by providing verification, validation and implementation of his proposals, while exposing the pitfalls of open systems implementation.

Chapter 2

2. PROBLEM DESCRIPTION

2.1. Introduction

Chapter 1 reveals the general need for network time synchronization. This chapter discusses the specific requirement for a distributed time stamping protocol and details the specific bounded time requirements of this study. The requirements for redundancy, fault tolerance and scalability are introduced as they are relevant to the requirements for expansion and future upgradability. The nature of the network is described and examples of industrial applications are listed, with a brief overview of each describing the applicability of the research in this thesis. The terms and concepts that are used in the network time synchronization literature are introduced as these are required for the discussion of the current technologies and proposed solutions in later chapters.

2.2. Statement of the Problem

A method for synchronizing time across an isolated LAN must be devised with the time deemed to be *synchronized* should the offsets between the master clock and the slave clock differ by not more than 250μs. The system should report a timestamp as *unsynchronized* should it exceed this figure. In Figure 2 the Master Clock is labeled as having the correct time while the other Slave Clocks are labeled with their offset from the Master Clock (the extreme offsets tolerated are listed). From this example we can see two subsets. One set with a clock time

125µs ahead of the master and a second subset with clocks 125µs behind the master. Thus the variance across the network will be 250µs at most.

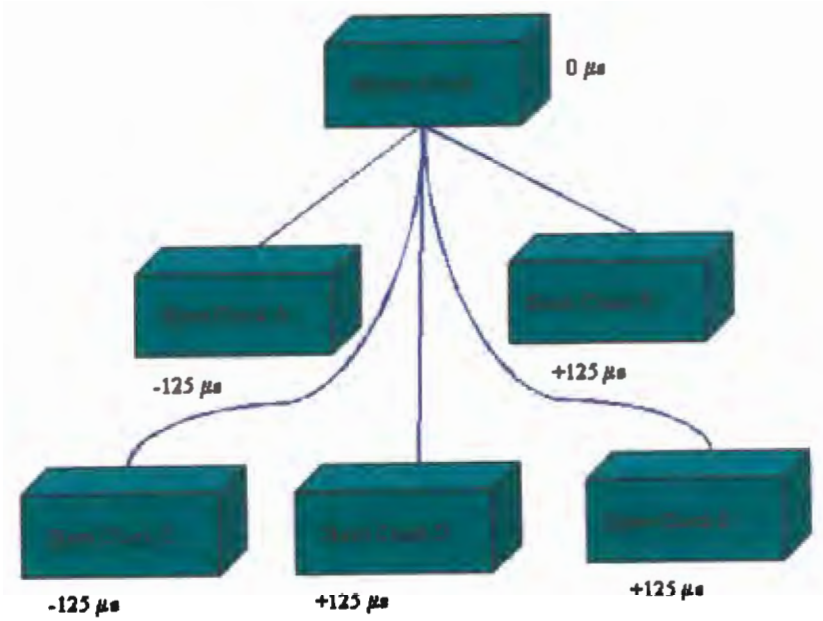


Figure 2: Network Synchronization

The different network nodes should be able to work individually and nodes may sporadically leave the network and rejoin at any time. The nodes leaving the network should be able to continue running the time synchronization software so that upon rejoining they may synchronize within a minimum length of time. The notion of *time masters* and *time slaves* must be reconfigurable.

To summarize, the following redundancy and reconfiguration requirements are placed upon the implementation of the time synchronization system:

- Network path reconfiguration: quickly reconfigure the paths used to exchange time information (i.e. must handle certain underlying network failure of cables and connectors)

- Alternative reserve clocks: upon failure of a referenced time source used for synchronization alternate time sources should be found and utilized.

- External time reference: the system must be configurable to use an external time reference (e.g. GPS or LORAN-C)

Use will be made of the current real time distributed architecture (as described in Chapter 4) as set forth in Young [YOU92][YOU96]. The synchronization requirements imply that the following services are provided, clock coordination, clock access and time management [DON96].

2.3. Purpose of Study

The needs of the computer industry for time stamping of a fine granularity are described by means of the examples below. Without the concept of a *"standard network time"* the following projects cannot be completed. These examples are taken from current proposals which are being investigated at C^2I^2 Systems (Pty) Ltd. and which will make use of the work accomplished in this thesis. These fall within the subset of problems that deal with mission critical real time synchronization with low microsecond tolerances as outlined in Chapter 1.

2.3.1. Information Management System (IMS)

"The IMS is in full scale development phase for a ship board network, based on SAFENET [SAF94], that will handle time critical command and control messages, network time stamping, multimedia streams, background file transfer

and data distribution from many sources to many destinations. The IMS architecture supports unicast, broadcast and reliable multicast features. IMS is also gaining interest in mobile tactical real-time systems as well as in mobile and semi mobile air surveillance radar systems" [IMS98] brochure. The IMS offers latency control, message level priorities, multimedia support, bandwidth allocation, multicast capabilities, determinism and reliability.

The IMS makes use of a partially homogenous distributed computing environment. A FDDI LAN links autonomous Multibus II (MBII) racks of computers needing to maintain accurate system time synchronization (250µs between all the computers) i.e. computers cannot range more than ± 125µs from the designated master clock. Owing to the mission critical nature of the IMS, no erratic behaviour from this norm is tolerable. These clocks are used to co-ordinate and regulate the general maintenance and battle preparedness of the system. Applications housed on host processor boards in the MBII racks use the time synchronization provided by the IMS Network Information Card (NIC) to gain time stamps for data to be processed.

2.3.2. Vetronics LAN (VLAN)

The proposed VLAN tests both the temperature and speed of the FDDI network to its limit. Conditions are more stringent than those for the IMS, although both systems carry similar information. The VLAN requires accurate time stamping with time clocks differing by less than 50µs. Space is limited therefore it is not possible to use additional hardware to generate a LAN wide pulse. Apart from the defense applications of this technology it could be utilized in long haul trucking, for example, to increase safety and control, (specifically with regard to brake control systems and fuel management). An Antilock Braking System could be implemented on the horse and trailer, provided the time synchronization between braking nodes is sufficient.

2.3.3. Long Train Project

Locomotives on long trains interspersed between carriages often spend much of their time expending energy in competition with each other. While one is increasing power on a hill, another further ahead is braking as the train descends the hill. Time information and the timely response of appropriate information helps to minimize and possibly avoid these situations. Important data regarding acceleration, strain and temperature must be acquired on trains of up to 10km long. Time synchronization between the computers on the engines would provide accurate time stamping and prevent stale information being acted upon, whilst resulting in the more efficient co-ordination of resources to optimize performance of the train. Mecalc (Pty) Ltd. and Spoornet are currently doing research in this area.

2.4. Timing Concepts and Terminology

Before it is possible to summarize the current state of time synchronization the following terms related to the synchronization problem must be introduced. They are presented here to allow for the common understanding of the terms, as they will be used later in this thesis.

The implementation discussed in Chapters 4, 5 and 6 is based on the concept of using COTS products and integrating them to provide a suitable solution. The major issue related to the integration of the products is called *porting*. This involves the rebuilding of software designed on one platform/operating system to another. Porting involves understanding the workings of the code and the relationship between the code and its host operating system. The porting may involve a quick change to a few header files or an extensive rewrite of the system. Generally the closer the program to the presentation/application layers of OSI network model, the easier the port.

The time keeping *clock*, which plays a role in the normal time of day served by a computer will be referred to as the clock, this is to avoid confusion with the bus clock. Where another clock is referred to it will be qualified by its location and function.

The following terms relate to the clock itself and its relation to the master clock. The *phase* is the time value of a clock at an instant while the *precision* is the smallest incremental value of which the clock is capable. *Stability* is the measure of how well the local clock maintains a constant *frequency* – this being the rate at which the clock increments.

The slave is presumed to have an *offset* from the master, which is the difference between the clock reading of the master and slave at an instant. This *offset* is measured in terms of *skew*, which represents the *frequency* difference (first derivative of the *offset* with time) between two clocks. The accuracy of the local clock depends on how close the local clock is to the reference clock in terms of *phase* and *frequency*. The slave clock is liable to *drift*, a variation on *skew* with time (second derivative of *offset* with time). In laymen's terms this can be seen as momentum of the local clock i.e. "how it is inclined to move away from the reference clock". The *jitter* is random variance that cannot be accounted for in readings. Jitter represents the latencies in hardware, software and network delays.

Chapter 3

3. TIME DISTRIBUTION SOLUTIONS IN PRACTICE

3.1. Introduction

The overview of the scope of the problem to be dealt with in this thesis given in Chapter 2 leads to an evaluation of the time and clock synchronization on networks at present.

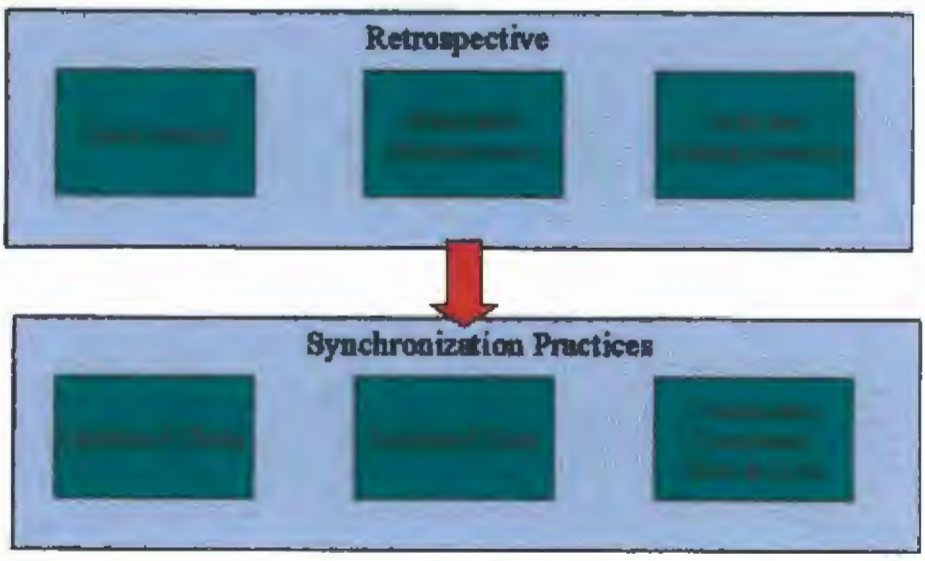


Figure 3: Synchronization Overview

This chapter presents the differences between time distribution and clock distribution. The fundamental principles involved in distributing time are also

discussed, these include clock latencies, offsets and error calculation. The three main methods of time distribution are critically reviewed for their incorporation into the integrated solution proposed in Chapter 4. The proposed solution needed (according to the requirements of the problem specification, as set out in Chapter 2) includes this knowledge in its formulation.

3.2. Retrospective

3.2.1. Basic Clock Services

Most computers have some kind of time clock. Usually this is a quartz oscillator and a hardware counter that causes an interrupt every number of milliseconds, depending on the clock rate, this interrupt may vary from 16ms to 1ms intervals. When this interrupt occurs a value is added to the variable representing the clock time. This variable is known as a *tick*. The value for the clock can be read and set by application level programs. Once the value has been set the clock will increase by a tick at every interrupt. [MIL98]

The systems time value can then be used for various time related programming tasks (e.g. read a file every 10 seconds). As long as the time granularity required is sufficient for the programming task, the task will function normally. With the advent of distributed systems the concept of a shared notion of the time has become important. Should a program wish to synchronize events between computers, the idea of the current time needs to be negotiated between these computers. The negotiation process can be accomplished in many ways. The following paragraphs will outline the past, current and future perspectives for the synchronization of time between distributed tasks.

3.2.2. Mainframe/Multiprocessor Environment

The mainframe/multi processor environment is characterized by the sharing of a common time clock. As a result, the issue of time synchronization is not relevant

in terms of distributed computing. All processes operate within the same architectural bus and thus they have common access to the clock. Should the tasks require synchronization then the call to the common clock will suffice.

This can be seen as the most accurate notion of clock synchronization – the trivial case.

3.2.3. Dedicated Network Solutions

Where computers are distributed in geographically proximity, a centralized clock can be used to distribute the time via a dedicated network using a periodic pulse. This is often done by means of a point to point network. as depicted in Figure 4 [DON96]. This electronic pulse can seek to synchronize the connected computers in bounded time that is, time between pulses. This form of inter computer timing has no relation to time of day and serves exclusively to time synchronized tasks. It is devoid of time management. Problems with dedicated network solutions are the lack of clock coordination and they are often expensive to install and maintain. Each computer still maintains its own time of day identity and tasks are synchronized according to the network pulse. This method does not cater for redundancy or fault tolerance. Once the clock line is broken, the timing of the processes is in jeopardy. This model was only suitable for networks with few computers and negligible propagation delay.

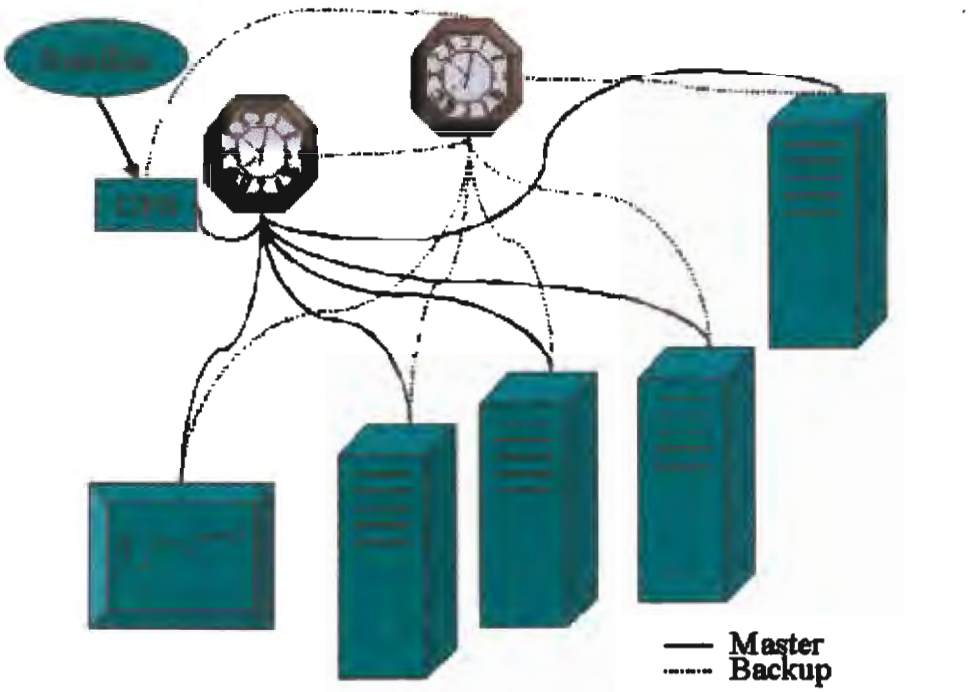


Figure 4: Dedicated Timing Network

An improvement on this method is the similar use of a dedicated clock attached to a GPS system such that the dedicated network now distributes the time to the computers involved, not just a synchronization pulse. The time word from the GPS master clock is written into a register on the client, which the client then reads. While this method now caters for time of day it has all the shortcomings described above. Figure 4 depicts the crossover of the methods with a GPS receiver synchronizing the clock. The clock has one backup and all the nodes are connected point to point to the two clocks.

3.2.4. Distributed Clocks

GPS can be used to distribute the time clocks on a set of computers.

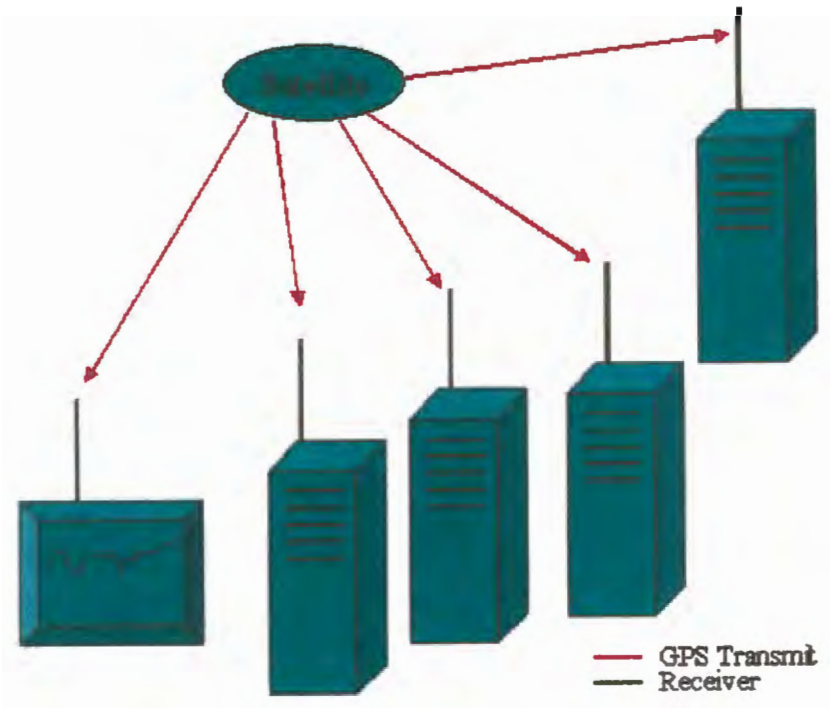


Figure 5: GPS Time Distribution

The installation of compatible GPS receivers which serve as clocks on each node ensure that the clocks all maintain a common notion of time within the bounds of the GPS specification [STA96]. The time bound will be twice the margin for error on the GPS. Although this functions well it consumes space. Specialized equipment is needed to compensate for this, making it expensive. If the GPS fails, the clock on the computer is rendered obsolete and the other computers cannot synchronize with that machine. The local clock on the unsynchronized machine will tend to its own time path depending on the conditions of the local

oscillator, and there is no restriction placed on the rogue clock, as no local clock training has taken place.

3.3. Current Perspective

3.3.1. Dedicated Network Solutions

An upgrade on the GPS method with redundancy and fault tolerance of the server, is provided by the introduction of a backup clock [DON96]. This allows for the failure of the clock, but will not allow for the autonomous operation of the various nodes in the network.

The embedding of the clock into a computer as a general purpose computer device falls into this class of solution. A GPS receiver may reside inside the computer but still make use of the dedicated timing network to distribute the time.

Hardware schemes which used dedicated hardware at each node that coordinate with all other nodes have been developed, most notably by Smith [SMI81] and Shin [SHI84]. The method is limited owing to the assumption of a fully connected network of clocks and negligible propagation delay. In small systems this is not a problem but when dealing with larger systems these assumptions are difficult to fulfill [SHI94]. Shin managed to devise a clustering scheme to limit the number of interconnections. However, the use of specialized hardware still precluded its use from large distributed systems.

3.3.2. Distributed Time

The most popular method for current purposes is to distribute the time whilst maintaining a degree of local time keeping. This is often done by requesting time from a server and then using this time to set the local clock after taking cognizance of the relevant delays. Digital Time Service [DEC89] and the

Network Time Protocol [MIL92] are examples of this. The method for calculating these delays is what makes the difference between the major time synchronization protocols below. This method makes use of the data network provided in the LAN and is thus subject to the problems that normal data experience across the LAN. Packet collisions, random delays based on traffic and the latencies within the hardware and software stacks that take the time information to and from the time server to the time client all play a role in delaying the timestamps.

3.3.2.1. Timing Errors

The above approach leads to errors in the distribution and the local computer's idea of time. These errors fall into two categories, namely those that can be accounted for and built into the equation and those that are of a random nature. Mills [MIL96] expresses these elements of time synchronization in the following terms:

$$T(t) = T(t_0) + R(t - t_0) + D/2t^2 (t - t_0)^2 + x(t) \qquad [1]$$

where t is the current time, T is the time offset at the last measurement update t_0, R is the frequency offset and D is the drift due to aging since the last reading. The first three terms include systematic offsets that can be corrected, whilst the fourth contains random variations that cannot be corrected. The components of the network delays and software and hardware latencies can be divided into correctable components and random variances (jitter) that cannot be corrected. Thus, the protocol aims to minimize the error in $T(t)$ by using the known factors in the delays and minimizing the random variances. The value T(t) include the constant propagation delay while the x(t) is the uncertainty due to jitter in the network and in the measurement and reading process.

3.3.2.2. Error Correction

Mills goes on to point out that the frequency spectrum of x(t) can be useful in examining the measurement errors and unexpected synchronization effects. By measuring the offsets between two clocks at two or more time comparisons the error in frequency comparisons can be ascribed to different phases in the synchronization. Offset measurements taken over a long period of time are likely to show crystal aging, physical vibration, temperature changes and fluctuations in power supply voltage. Mills states that the mean of this frequency error is no larger than the difference of the time offsets divided by the time between measurements. By decreasing the time between measurements and increasing the number of comparisons, this data can be examined at the depth required. Mills [MIL98] states :

> *Analysis of quartz-resonator stabilized oscillators show that errors are a function of the averaging time, which in turn depends on the interval between corrections. At correction intervals less than a few hundred seconds, errors are dominated by jitter, while, at intervals greater than this, errors are dominated by wander.*

In Mills [MIL94] a series of experiments shows that a poll interval of 16s relegates the error region to almost entirely jitter. From this poll interval and lesser the jitter should be the only factor influencing the clock stability. As will be demonstrated in a later chapter this insight will provide us with reason for using the system integration model with a compatible time protocol, the architecture designed by Young[YOU96] provides an ideal platform on which to lower this jitter considerably. An attempt will be made to place a ceiling on the jitter so that this too can be accounted for in the time synchronization.

3.4. Current Protocols

Distributed time using the data network is advocated by the following main contributors in the field. The methods used in the different protocols, shown in Figure 6, were reviewed in order to determine the extent to which they would match the requirements set out in Chapter 2. The aim of this exercise being to understand the current work in the field and to select a suitable protocol for use.

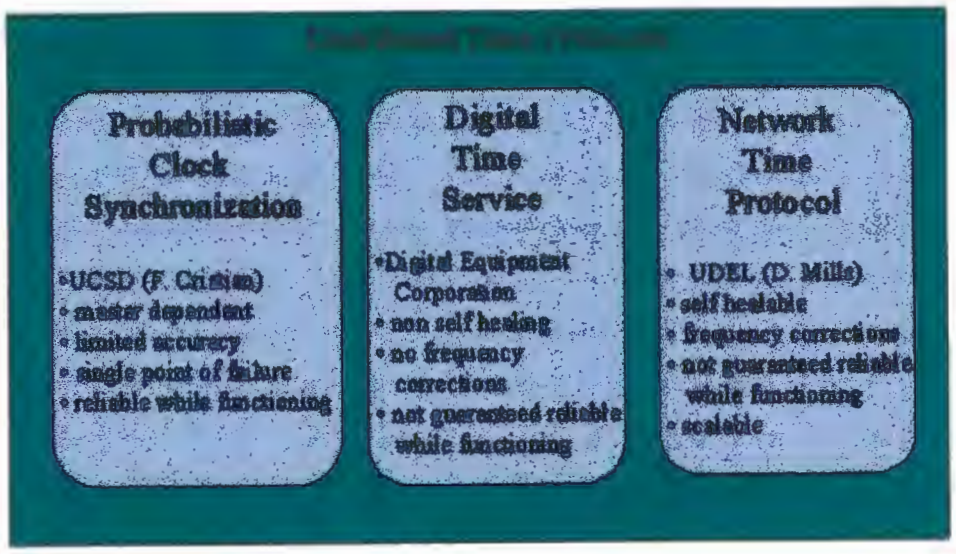


Figure 6: Distributed Time Protocols

3.4.1. Probabilistic Clock Synchronization (PCS)

Cristian [CRI89][CRI94], devised a method for time distribution allowing for a predefined fault tolerance. At startup time the accuracy of the time needed is set and all round trip delays exceeding twice this delay are discarded i.e. we tolerate all those packets received before the tolerance time is reached. This allows for bounding the time stamp once received, as the time stamps received are within

the tolerable limits. If seen in the light of equation [1] this would be limiting the ceiling of the first term of the equation.

If we take this argument to the extreme, we would need to decrease the tolerable window in order to increase the accuracy of the timestamp. The possibility exists that such a situation would leave no packets in the tolerable window. The margins required by the examples listed in the second chapter, would cause great time stamp loss. The required rate of survivable packets would limit the accuracy obtainable, as an increase in accuracy automatically implies a decrease in tolerance, which in turn implies an increase in time stamp loss. Thus we need to reach a compromise between survivable time stamps and accuracy required. The accuracy would ultimately be determined by the network round trip speed, which would limit the accuracy achievable.

These issues would lead to PCS not being acceptable to our areas of investigation. In summary the PCS would :

- not allow autonomous client operation
- place a limit on the accuracy achievable
- not allow for the time master to leave the network

3.4.2. Digital Time Service (DTS)

DTS was developed by the Digital Equipment Corporation[DEC89], in order to disseminate time across a network. It reads a server clock and then transmits that reading to the client and adjusts the local clock. DTS places heavy emphasis on configuration management and correctness within managed networks and would appear to be a closer fit to the requirements. However, DTS does not allow for frequency errors or filtering of the timing noise [MIL98]. According to the formula [1] this would amount to only evaluating the first term.

This leads to DTS being unable to remove residual errors owing to errors in the system frequency. Thus the DTS will seek to minimize the time offset while not changing the frequency offset. This would lead to no correction of clock drift and a computer removed from the network will have a clock that runs completely freely. Thus the resynchronization would be difficult should monotonicity be important. The need for a managed network would limit self-healing and fault tolerance. The areas under investigation would thus need much additional functionality added to the DTS including the above and a future requirement for the use of an external time source.

The DTS was deemed inappropriate for the following reasons:

- required network maintenance
- non self healing or fault tolerance
- external time reference problems
- no frequency corrections possible

3.4.3. Network Time Protocol (NTP)

The Network Time Protocol developed by Mills[MIL92], has been in existence for more than a decade and is now in version 4. The local clock is used as the building block for time services. This is then enhanced by providing a software clock that is updated and controlled by the responses of local nodes servers. NTP uses a *hierarchy* of servers to provide redundancy and fault tolerance in the servers. This hierarchy forms a *synchronization network* in which each level (stratum) is described as a *synchronization subnet*. The time served by these servers is used to train the local clock to adjust its phase and frequency to that of the server. According to the formula [1] above this would be the first and second term of the equation. This allows for the NTP aware computers to converge on an agreed

network time, whilst preparing for possible network outages, and ensuring timely recovery.

The NTP was originally designed with heterogeneous networks in mind and has functionality for reference clocks (e.g. GPS , LORAN-C and WWV as well as the local oscillator). Thus, should the external time source be needed there are a range of methods already supported. The training of the client node will allow for the node departing from the network to rejoin it with minimal problems. As opposed to DTS, while it is separated from the network the drift of the local clock is controlled by the historical drift it displayed while part of the network. The NTP software clock is capable of keeping resolution to 2^{-32} binary fractions of a second (about 212 pico seconds) and 2^{32} seconds (about 136 years) thus ensuring that the accuracy of the clock is well within our requirements.

The Network Time Protocol is the de facto industry standard with over 100 000 installations [MIL95] and an ongoing development plan. It is supported by its own newsgroup, World Wide Web (WWW) pages, File Transfer Protocol (FTP) site and dedicated mailing list. The project is well documented and is part of the Internet Engineering Task Group (Networks), NTP is presented in RFC 1305, RFC-1119, RFC-1059, RFC-958 [MIL92].

The Network Time Protocol will satisfy the requirements for the clock coordination and clock management as set out in Chapter 2. Lacking the ability to provide the clock access routines it is of itself not an entire solution, nor is it possible to get the accuracy required without additional intervention. The NTP will allow for the requirements for autonomous, redundant and robust performance, while a custom application layer could provide for the clock access routines. As NTP was chosen as the basis for time synchronization, it will be discussed in detail in Chapter 4. Attention will be given describing the additional

protocol choices which provide NTP with the components required to ensure that the target accuracy is reached.

Chapter 4

4. SYSTEM DESIGN

4.1. Introduction

The presentation of the major contributions to time synchronization in Chapter 3 showed that no singular system was suitable on it's own. The NTP model came closest to meeting the requirements needed on which to build a reliable service. In this chapter the integration model is described, the real time distributed architecture is discussed in detail and the protocols, operating systems and hardware are introduced that will be used to enable the building of a low offset time synchronization system. The use of "off the shelf" software and industry standards are discussed and the notion of "future proofing" is introduced. A custom software add-in, the Network Time Services software suite, is introduced and its relation to the underlying protocol stack is discussed, enabling the construction of the protocol stack as presented in Chapter 5.

4.2. System Integration Model

Mills [MIL95] states that for a correction interval of less than "a few hundred seconds", clock offsets are dominated by jitter. The possibility of achieving a suitable solution to this would thus lie in the ability to minimize the jitter. Within the bounds of the protocol strategies and architecture described in Young [YOU96] the components required to develop a time stamping protocol suite that will seek to minimize this jitter are examined. Each of the options chosen will be detailed and compared with the available choices. The reason for the choices will be highlighted.

A modular approach to the design of the components was taken so that an improvement in any of the components in the protocol stack would feed back to an improvement in the offsets achieved between the servers and the clients. By integrating the various components needed in such a manner that they maintain an open systems interface, a path is left for future upgrades to each of the individual components with minimum interference in the layers lying above and below the updated layer. This "future proofing" is used to allow for the timely integration of improvements while only needing to verify the upper levels of the protocol stack to ensure error free integration.

4.3. Real Time Distributed Architecture

In his master and doctoral theses Young [YOU92][YOU96] describes the architecture used in this thesis. He addresses the requirements for data communication protocols and data services in mission critical real time distributed systems implementation strategies. By using the International Standards Organization (ISO) Open System Interconnection (OSI) reference model and supplementing it with appropriate technologies and protocols, a robust LAN capable of delivering "dependable, closed-loop, real-time control" [YOU96] is devised. Understanding the protocols and hardware suggested by Young allows for a software implementation that can form a "gestalt" – more than the sum of its parts. By introducing determinism into each layer of the protocol stack, the timing protocol was bound within a lower margin than previously possible. The bounds were eventually determined by the residual jitter in the system that cannot be accounted for by the real time LAN. The theory is explained in relation to Young's LAN profile designed for real time. The recommendations that are relevant to the time synchronization are included. For a full list of his recommendations please refer to [YOU96].

<table>
<tr><th>Layer No.</th><th>ISO OSI Layers</th><th colspan="4">Real-Time LAN Profile</th></tr>
<tr><td rowspan="2">9</td><td rowspan="2">Application Process*</td><td colspan="4">Software Application Tasks</td></tr>
<tr><td colspan="2">Network Management Services</td><td colspan="2">Timestamping Services</td></tr>
<tr><td rowspan="2">8</td><td rowspan="2">Operating System*</td><td colspan="4">POSIX Real-Time Operating System Extension</td></tr>
<tr><td colspan="4">Real-Time Operating System</td></tr>
<tr><td>7</td><td>Application</td><td>Built-in Test Services</td><td>Network Time Service</td><td rowspan="3">File Transfer Services</td><td rowspan="3">Application Interface Services</td></tr>
<tr><td>6</td><td>Presentation</td><td></td><td>Network Time Protocol</td></tr>
<tr><td>5</td><td>Session</td><td></td><td></td></tr>
<tr><td>4</td><td>Transport</td><td colspan="2">UDP</td><td colspan="2">XTP</td></tr>
<tr><td>3</td><td>Network</td><td colspan="4">IP</td></tr>
<tr><td rowspan="3">2</td><td rowspan="3">Data Link</td><td colspan="4">SNAP</td></tr>
<tr><td colspan="4">IEEE LLC Type I Protocol</td></tr>
<tr><td colspan="2">ANSI FDDI SMT Protocol</td><td colspan="2">ANSI FDDI MAC Protocol</td></tr>
<tr><td>1</td><td>Physical</td><td colspan="4">ANSI FDDI PHY Protocol</td></tr>
<tr><td>0</td><td>Cable layer*</td><td colspan="4">Multimode Fiber cable Plant</td></tr>
<tr><td colspan="6">Note: Layers marked with an asterisk (*) fall outside the ISO OSI 7 layer Model</td></tr>
</table>

Figure 7: Real-Time LAN Profile

"To support real-time, mission critical systems, the LAN profile is required to include protocols capable of real-time performance at each layer" [YOU96]

The application resides on a Central Processing Unit (CPU) which communicates with the Network Interface Card (NIC) via a Parallel Backplane Bus (PBB).

4.3.1. Cable Layer

The cable layer transmits the electronic pulses between connected nodes. This is a non OSI defined layer, with the medium of transport falling outside the OSI scope, yet cabling is needed for an integrated solution. Young[YOU96] stresses the following for the real time LAN

- able to accommodate present and future bandwidth requirements
- meets the geographic range requirement
- meets mass and size requirements
- cost effective
- reliable
- supportable

Fiber optic components were chosen for the cabling, as investigations into fly by light and next generation warships had noted that fiber optic cabling provided the best combination of robustness vs. size and mass, bandwidth, affordability and support.

4.3.2. Physical Layer

At this layer the binary signals are encoded onto the physical cable media. The primary focus of this layer is how well the raw bits can be transmitted across the communication channel. The optimization of the mechanical, electrical and procedural interfaces to the cable layer is the most important factor.

The chief attributes sought are:

- enough transitions to extract signaling synchronization (clock recovery)
- provides for low level error detection
- provide efficient use of physical mediums bandwidth capability.

FDDI and Fiber Channel are selected as options owing to their efficient use of the mediums bandwidth.

4.3.3. Data Link Layer Protocol

The Data Link Layer (DLL) is responsible for rendering the transported raw bits in a fashion that appears free of error and is divided into Data Link Service Units (DLSU or data frames) which are transmitted sequentially. The DLL also processes acknowledgement frames to the sender. Thus the DLL must solve the problems inherent in damaged, lost or duplicated frames. This layer is divided into two parts that of Media Access Control (MAC) and Logical Link Control (LCC)

4.3.3.1. Media Access Control

This layer is fundamental to the ability of the network to support real time deterministic transfer. Young [YOU96] summarizes the important issues as:

- support for present and future performance requirements
- supports data transfer determinism
- makes efficient use of the physical layer
- support for future functionality (e.g. multicast addressing)
- stable and cost effective

- appropriate network topologies supported

FDDI is described as an international standard that satisfies all the above requirements when coupled with a synchronous and asynchronous message transfer mode.

4.3.3.2. Logical Link Control (LLC)

The Logical Link Control (LLC) layer is responsible for the formatting and disassembly of the DLSU's into packets. The LLC allows for connection oriented and connectionless services The definition of LLC Type 4 by the IEEE states that it offers:

- reliable sequenced delivery
- reliable non sequenced delivery
- non reliable sequenced delivery
- segmentation/re-assembly
- quality of service
- multicast
- multiple logical connections between Service Access Point pairs

Young points out that "these capabilities indicate a close resemblance to the services offered by XTP", as both are aimed at providing the requirements needed to ensure the delivery of real-time data. XTP is detailed in the Transport Protocol Decision later in this chapter.

4.3.4. Network Layer

The network layer controls the operation of the subnet. It provides packet routing and flow control. The network layer determines the service provided to the transport layer (virtual circuit connection vs. non-guaranteed datagram

service). The network layer also incorporates the accounting information for determining packet count.

Young's assessment of the network layer makes the following points with regards to an optimal network layer candidate:

- Meets present and future performance requirements (routing, addressing, flow control)
- Provides connectionless service
- Provides for priority routing

He concludes that Internet Protocol (IP) and Connectionless Network protocol (CLNP) meet most of these requirements, while IP is seen as preferable.

4.3.5. Transport Layer

The transport layer is responsible for accepting data from the session layer above it, dividing it into smaller units (if needed), passing these to the network layer below and ensuring the correct delivery to the recipient. Thus it is responsible for end-to-end data flow control. The issues of end-to-end error control flow and rate control are regarded as critical to real time systems[YOU96]. The requirements are described as:

- Meets present and future performance requirements (latency, jitter, throughput)
- Meets present and future functional requirements (e.g. dataflow control)
- Provides priority scheme
- Provides multicast services

Young compares TCP, ISO TP4 and XTP for the above requirements (amongst others). The use of XTP is recommended for real-time while TCP is recommend for connectivity.

4.3.6. Network Time Services

A timing system that circumvents the problems of latency and jitter within the network is described. The Network Time Services, compatible with Survivable Adaptable Fiber Optic Embedded Network (SAFENET) standard [SAF94], this provides for the use of time services to an accuracy of 1 ms normally with time of nanosecond range being optional. At the point of completion of his thesis Young proposed the use of NTP via FDDI as was shown experimentally by the author on the C^2I^2 Systems prototypes.

4.3.7. Summary

While Young provides a sound basis of protocols on which to place time synchronization model using distributed time, each of the areas described allow for the introduction of jitter.

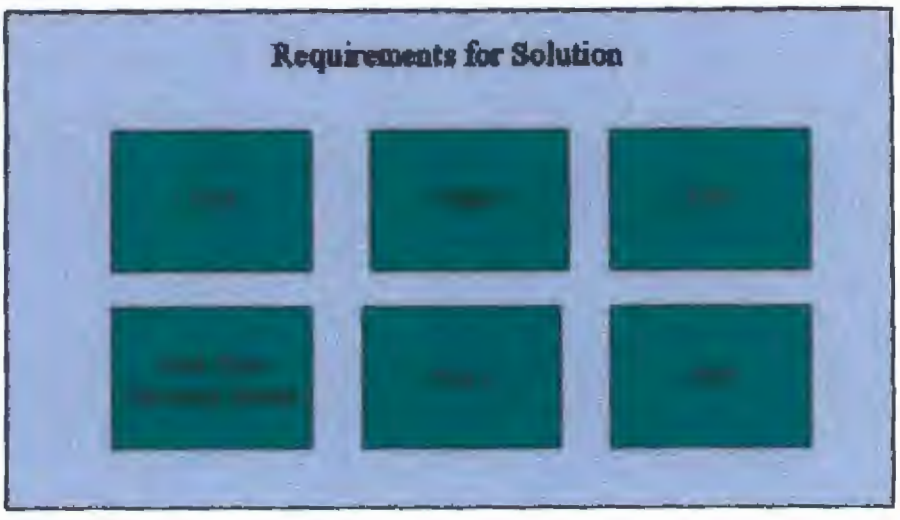


Figure 8: Requirements for Solution

This jitter is the main cause of bad time synchronization. Building on the protocol stack described, a jitter compensating SAFENET compatible time synchronization system can be developed. By implementing the software protocol stack mentioned using the components in Figure 8, and providing a custom Network Time Services software suite to handle residual system jitter, the requirements could be met. The interactions between these components and the NTS are described in Paragraph 4.5.

4.4. Selection of Hardware

The hardware used for the project formed part of the Information Management System (IMS) project for the SA Navy. Use was made of the following components, in addition to a number of Pentium PCs:

- 5 x TCUs (IMS TCU XX)
- 5 x FDDI patchcords (FDDI-P-XX)
- 1 x FDDI dual ring fibre network (FDDI)
- 7 x Multibus II development racks
- 1 x LANWatch PC, including EISA FDDI adapter
- 4 x RS232 serial terminals
- 1 x storage oscilloscope
- SUN Solaris Ultra 1 Server development host

Multibus II development racks consist of:

- 1 x IMS Network Interface card (NIC) card (CCT MBII P20 PMC host, plus PMC FDDI)
- 1 x ENET AUI Transceiver for NIC
- N x CCT MBII PMC host processor (N - depending on requirements normally 1-3)
- 1 x ENET AUI Transceiver for PMC host
- 1 x MBII development rack, including PSU, MBII backplane

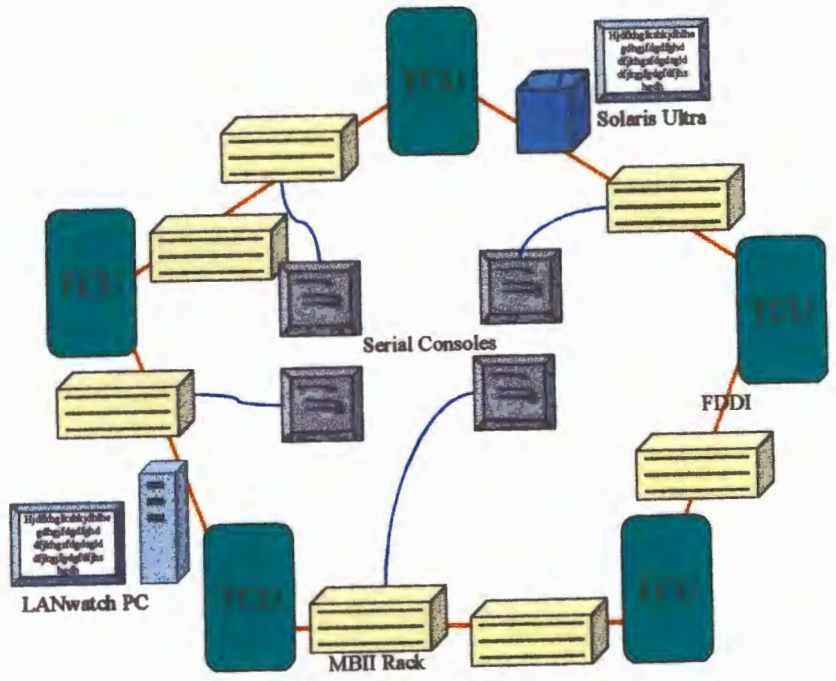


Figure 9: Development LAN

Figure 9 depicts a overview of the development LAN. For comprehensive description of the connectivity between the various components please refer to the Appendix D. The hardware used was able to provide the foundation upon which the network time synchronization could be developed. Each Multibus II (MBII) rack had an IMS NIC (as seen in Figure 10), which had a PMC FDDI daughter card linking the MBII racks. The PMC FDDI NIC was developed by C^2I^2 Systems and is aimed at the embedded platforms market. This NIC has been developed from SysKonnect's PCI FDDI NIC (see Appendix A for a detailed description). The other slots in the MBII rack could thus house the PMC host processors communicating via the MBII backplane to the NIC. The target hosts for the network time synchronization were the IMS NIC. Each IMS NIC was

equipped with both Ethernet and FDDI connectivity to one another. When in production the full FDDI network is anticipated to be at most 1km in length and to have at most 30 dual attached stations.

Figure 10: IMS FDDI NIC

4.5. Selection of Software

4.5.1. Time Protocol Decision

NTP was chosen for its superiority in terms of meeting the majority of the requirements of the project. It provided a protocol designed to be fault tolerant and ensure robust time synchronization using deterministic methods, while being the industry standard [MIL92] with the largest installed base [MIL95]. The NTP calibrates time in binary fractions of seconds and could thus represent time down to pico second accuracy. This allows for improvements in time accuracy for the future. The well documented and supported protocol has a large academic base and an ongoing development program [MIL98], [NTP98].

4.5.1.1. The NTP Time Synchronization Model

The following is a summary of the NTP Time Synchronization Model as it appears in RFC 1305 [MIL92], and describes the inner workings of NTP. For a more complete reference please refer to [MIL92]. This summary is given as a guide to the issues that directly affect this project.

An NTP client exchanges time stamps with each of the peers in its synchronization network at intervals (normally 16s – 1024s intervals). These timestamps are then used to calculate the round trip delays, clock offsets and to provide information for error estimates. The delays and offsets are processed by a clock filter algorithm to reduce jitter. From the peers a subset of those most suitable to provide accurate and valid time is selected. The process of clock selection takes place by means of two sub-algorithms. One to determine interval intersections: this is to exclude those outside of the intersection (called falsetickers). The other is based on maximum likelihood principals to improve accuracy. These offsets are then combined by means of a weighted average and

then used to drive the clock discipline algorithm, implemented as a feedback loop. The processes are depicted in Figure 11 showing the flow from the network through the clock filtering, selection and combining to the loop filter and the Variable Frequency Oscillator (VFO).

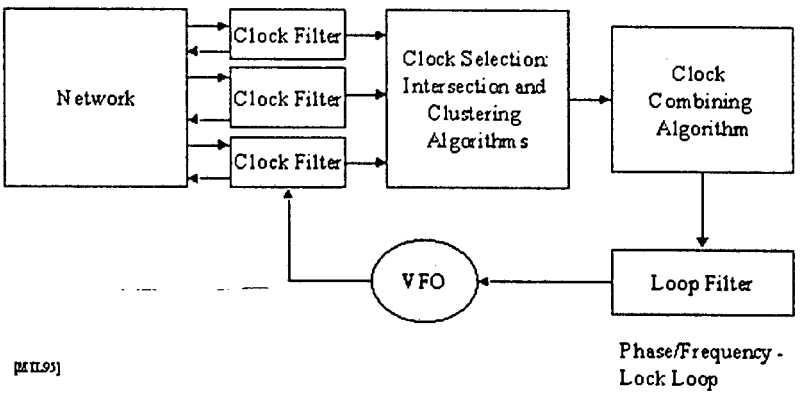


Figure 11: NTP System Model

The feedback loop processes the combined clock offset to control the VFO frequency. This VFO is a combined hardware/software programmable counter that is responsible for timing calculations and provides all timestamps. The peers are divided into strata based on the clocks they reference and how close they are to global time of Universal Coordinated Time (UTC)(sic).

4.5.1.2. Time Stamps

NTP timestamps are exchanged between peers. In each NTP message the previous three time stamps are included while the fourth is determined on arrival. Thus both peers can calculate the delay and offset (see Figure 12). Digital telephone networks use a similar symmetric, continuously sampled, time transfer

scheme [LIN80] in [MIL95]. This enables the avoidance of errors due to missing or duplicated messages.

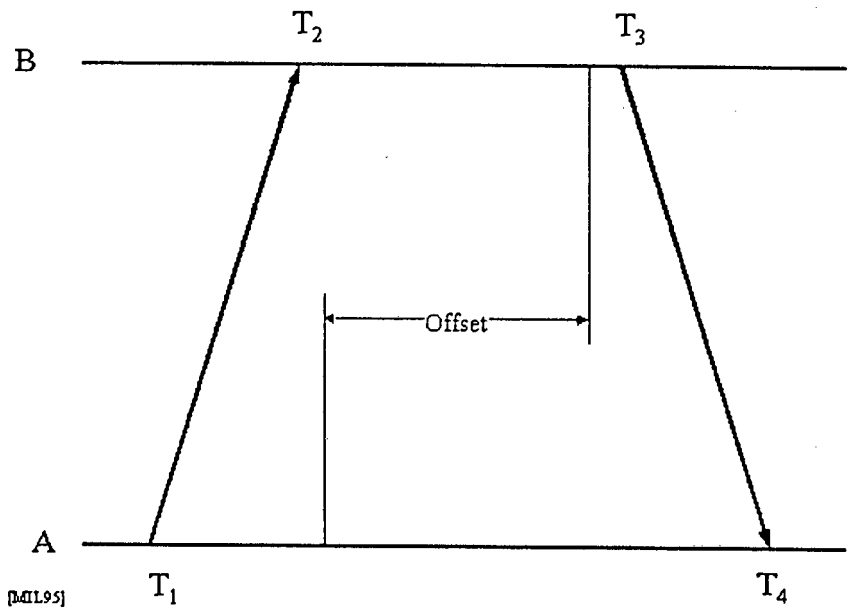


Figure 12: Measuring Delay and Offset

The above figure shows NTP timestamps numbered as they are exchanged between peers A and B. T_1, T_2 ,T_3 and T_4 are the four most recent timestamps. If clocks A and B are stable and run at the same frequency, let

$$a = T_2 - T_1 \text{ and } b = T_3 - T_4$$

If the network delay difference from A to B and from B to A (differential delay) is small then the clock offset (θ) and round trip delay (δ) of B relative to A at the time T_4 is close to

$\theta = (a + b)/2$ *and*
$\delta = a - b$

This reasoning is important – we will examine the differential delay again in the custom time services software, where the offset and delay will be further enhanced by the protocol stack. Of particular interest is the calculation of θ.

4.5.1.3. Error Estimation

Mills [MIL92a] analyses the time and frequency errors that contribute at the various levels of the synchronization subnet. The results show that the synchronization distance (λ) is the major contributor to offset error. Also of note is the dispersion error (ξ) made up of

- Maximum reading error (local and peer clocks) dependent on clock resolution and method of adjustment.
- Maximum frequency tolerance error of the local and each peer clock since last time set.
- The estimated delay contributed by variations in the network delay and latencies in the operating system on the path to the reference source(called peer dispersion)
- The estimated error contributed by the combined set of peers used to discipline the local clock (differences between the different member of the set – called select dispersion)

The error due to network delays usually dominates ξ. These delays are not normally easy to characterize as "network queues grow and shrink in chaotic

fashion and packet arrivals are often bursty" [MIL95]. However a good estimate of the maximum error contributed by all causes is

$$\lambda = \delta/2 + \xi$$

Thus the measured offset (θ) of the local clock to the reference master can be bounded to the true offset (θ_0) with a high probability to

$$\theta - \lambda \leq \theta_0 \leq \theta + \lambda$$

This is called the confidence interval. Essentially, the highest reliability is associated with the lowest stratum reference and the closest synchronization distance, while the best accuracy is to reference the lowest stratum and lowest dispersion.

4.5.1.4. Choice of NTP Implementation

Various implementations of NTP exist, and often come packaged with releases of the various flavours of UNIX and placed in PROM in routers. The implementation [NTP98] obtained from the University of Delaware by Prof. David Mills has become the standard, owing to the wide installed base and adherence to high standards of software compatibility. This makes use of the FSF [FSF98] code of ethics and uses the BSD and POSIX system calls to maintain Open Systems compatibility [UNI98].

4.5.2. Determinism and the NTP Clock Model

4.5.2.1. The Hardware Clock

The hardware clock on the motherboard is incremented at each clock tick, giving accuracy to the last tick. For example a 60Hz clock rate will be accurate to 16.67 milliseconds. Thus the NTP needs to augment the hardware clock with a software clock to provide for the added accuracy required. Most hardware clocks

now have a free running counter that can be used to interpolate between the clock ticks.

4.5.2.2. The Software Clock

The software clock is stored in 64 bits, with 32 bits each being used for the integer and fractional part of the seconds as depicted in Table 1.

64 bits	
32 bits	32 bits
Integer seconds from 0 January 1900	Fractions of a second from last second

Table 1: NTP software clock

The hardware clock is adjusted every second by a software clock correction changing the amount by which the hardware clock is incremented at the tick. The hardware clock and the hardware counter are used to represent time between ticks and are stabilized by the software clock. The synchronization of the hardware for the machines in the synchronization subnet requires accurate estimation of reading, propagation and round trip delays between the time slaves and masters. All these readings can be vastly improved by adding determinism.

4.5.2.3. Influence of Determinism in the System

The accuracy of the software clock that controls the hardware clock depends upon the accuracy of the delays with which the readings from the masters and slaves are exchanged. By reducing the reading delay inaccuracies the real offsets between the masters and slaves produce more accurate data to stabilize the hardware clocks. The determinism in the system serves to stabilize the readings and improve consistency. A simple overview of the process would be as follows:

```
FOREVER
     {
     // Possible delays:
     time_stamp = gettimestamp();          // system delay
     send_request(destination,time_stamp);
     data = receive_info ();               // round trip delay
     update_clock(data);                   // system and
                                           // propagation delay
     }
```

Non-deterministic networks, like Ethernet, produce non-quantifiable round trip delays and the return trip estimated delay is assumed to be equal for both send and receive messages, this is hardly ever true. The jitter present in reading the time from the system clocks owing to non- deterministic system latencies serves to further jeopardize the accuracy with which the software clock can be updated.

The system design used in this thesis serves to minimize both the delays and the system jitter in taking the necessary readings and in updating the clocks.

4.5.3. Network Time Services (NTS)

NTP takes care of the time management and clock coordination of the synchronization but the clock access routines still need to be fulfilled. This is normally done by the standard system calls of `clock_gettime()` or `gettimeofday()` when working directly on the IMS NIC. The nature of the architecture decrees that we provide a method to allow the MBII Application Programmer Interface (API) Layer so that host processors are able to request time services via the MBII backplane, as depicted in this set of tables adapted from [YOU96]. The NIC is connected to the other NIC's while the Application

hosts sole source of connectivity is to the NIC via the MBII backplane, as shown in Figure 13 & Figure 14.

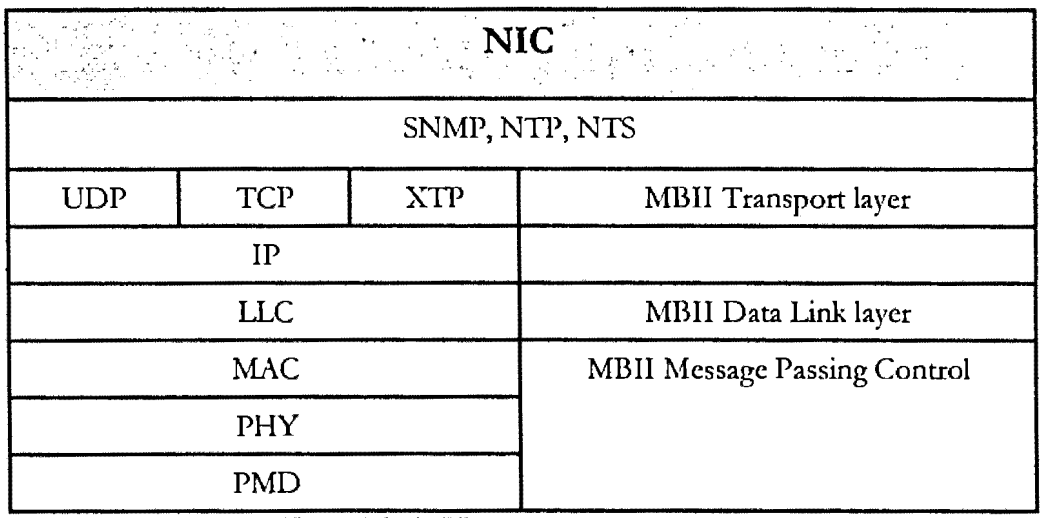


Figure 13: IMS NIC Layout

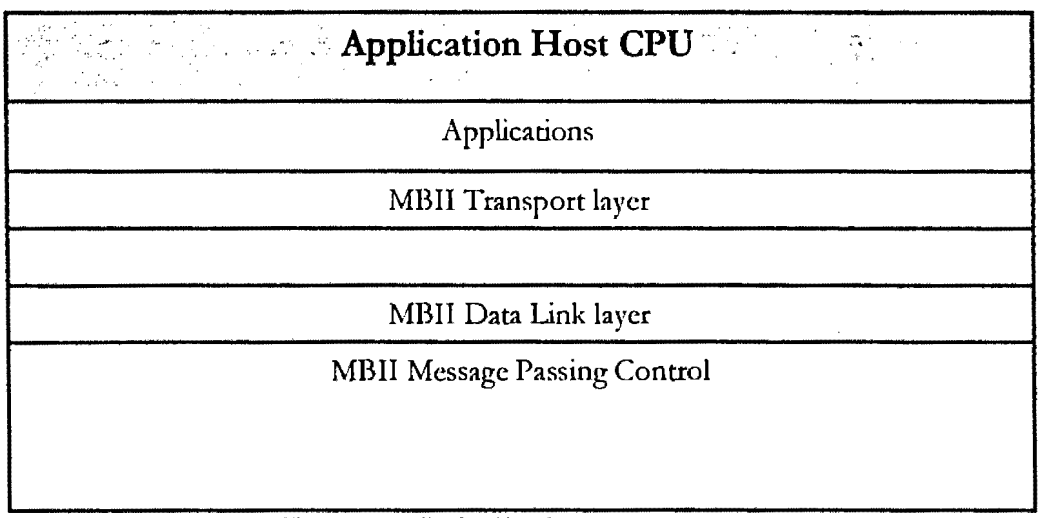


Figure 14: Application Host Layout

The opportunity thus exists to evaluate the time and to make corrections for the delays experienced within the system. A custom software suite (NTS) is thus added to the top of the protocol stack.

The NTS, unlike the NTP, is only interested in the time at an instant (current clock phase). However, the time must be that of the network/master and not the current computer. This time is defined as:

$$T(t) = T(l) + \Sigma(d) + x(t)$$

T(t) is the current master time, T(l) is the local NIC time , d is the known delays x(t) the system jitter. Σ(d) is made up of the known system delays not catered for in NTP(e.g. offset, backplane transfer, VxWorks latency).

For example: it takes 100 ± 30μs [YOU96] to obtain a time stamp. Thus by adding 100us we are accurate to 60μs. Another option is to standardize on the NIC's, as has been done in this network, ensuring that all the NICs are subject to the same delays – thus only the jitter remains, in both cases this leaves an accuracy of 60μs.

The information kept in the NTP peer structure [MIL92] is rationalized for the "trustability" (using the dispersion error (ξ) of the said offset), which is linked to the dispersion of the local clock also kept by NTP in the peer structure. The peer structure and SNMP MIB (See Paragraph 4.5.7 for a discussion on the use of SNMP) keeps all the information required to calculate our "instants time stamp", apart from the x(t) which we will seek to place a bound upon. The confidence interval:

$$\theta - \lambda \leq \theta_0 \leq \theta + \lambda$$

will be further bounded as the synchronization distance (λ) is correctable by negating asymmetry and station delays. In summary the reliability of this offset is determined by the peer and SNMP statistics and based on the node position in relation to the master. NTP is used to maintain the current time and then

when appropriate a trusted offset is added to the current time to get the NTS time. The "instants time stamp" accuracy is closely linked to the accuracy of the estimated error/offset and calculated distance , which is obtained from NTP and SNMP.

The reason this is not done in NTP is that the clock is continually trying to converge with the 0 offset possibility. The notion of keeping the clock accurate all the time would interfere with the goal of achieving a low CPU utilizing 0 offset eventually. This is summarized as a phase vs. frequency decision. Continually stepping phase would impair the VFO frequency training which is crucially important for the jitter delays. As the time stamp is manipulated after its extraction from the VFO, NTS does not interfere with the frequency compensation. And so, NTS is used to change the NTP time to "network time at an instant".

By providing a SNMP Management Information Block (MIB) in NTS we can interact with the NTP peers structure and the functioning of the NTS during run time from the Station Management Console (SMC), further adding to the time management and reconfigurability options already present in NTP.

4.5.4. Real time Operating System Decision

The use of Hard Real Time Operating System provides the secure knowledge and guarantees of code latency and true pre-emptive scheduling. The latencies are bounded and thus the throughput is guaranteed. This has the effect of making the code execution times standard even under severe load, as we have a set priority at which the code block is executed. A study of COTS Real Time Operating Systems was conducted by an investigation team, including the author, at C^2I^2 Pty (Ltd.)[CCI98]. After considering the variety of Real Time Operating Systems available, the LynxOs (from Lynx Real-Time Systems)[LYN98], iRMX,

iRMK (from Radisys)[RAD98] and Vxworks (from Wind River Systems) [WRS98] were evaluated. The basic requirements that were investigated are as follows:

- Applications must be embeddable.
- Support for different CPU architectures (x86, 68xxx, PowerPC).
- Support for Multibus II, VME and PC Platforms.
- Support for graphics applications.
- Support for networking applications.
- Industry standard development tools.

The tests that were undertaken to investigate these issues were:

- Multibus II communications between the evaluated OS & iRMX & iRMK
- Graphics performance and X11 applications.
- The University of Delaware xntp version of the NTP porting to the evaluated OS

VxWorks was chosen as the most suitable operating system to use for the distributed real time development environment. The time of day clock software for the LynxOs Operating System proved to be faulty (the ability to change the value added at each tick was non-functional). In VxWorks it was possible to alter the time of day software source and to recompile it into the kernel. Because much of the VxWorks software comes with source code that can be specialized according to the system requirements, it was chosen. For a detailed description of this report see Appendix C.

4.5.5. Transport Protocol Decision

As seen previously in 4.3 (Real Time Distributed Architecture), Young provides for a multi transport layer model. From his recommendations the target Transport layer was chosen to be XTP. However, owing to the modular design the TCP/IP version was built and tested alongside the XTP version. The choice

of XTP can be seen from the following abstract derived from the Xpress Transport Protocol 4.0. [XTP95] on [IMS98]:

"The Xpress Transport Protocol (XTP) has been designed to support a variety of applications ranging from real-time embedded systems to multimedia distributed over a wide area network. In a single protocol it provides all the classic functionality of TCP, UDP and TP4 plus many new services such as transport multicast, multicast group management, transport layer priorities, traffic descriptors for quality of service negotiation, rate and burst control, and selectable error and flow control mechanisms. XTP has the same interconnectivity as TCP/UDP/TP4 because it operates over any network layer (IP, CLNP), any datalink layer (LLC, MAC), or directly on top of the AAL of ATM. In general, XTP avoids coupling policy with mechanism; XTP offers services but the user's application defines what communications paradigm is most appropriate for its particular environment. XTP is a high performance protocol, and can sustain high throughput (92 Mbits/s over FDDI between a pair of IBM RS/6000 model 370s) and low latency (350 μs to move 100 bytes from user memory to user memory on two 50 MHz PCs connected by FDDI). Since XTP can run in parallel with all other transport protocols, and can run over whatever network layer (if any) is provided, it represents a low risk way to exploit the increased functionality required for distributed applications without sacrificing connectedness or interoperability."

XTP is described as applying particularly well to real time needs. As such, it was a natural choice to replace the TCP/IP interface on the NTP with an interface to XTP. Further, XTP was originally aimed at silicon implementation (software implementations have been built), thus opening opportunities for further decreasing jitter in future. XTP is aimed at providing fast service and multipoint communications. When combined with the variable Quality of Service available

it fits in well with the Synchronous Bandwidth Allocation (SBA) possible with the SK FDDI card. Thus the choice of XTP can be summarized in a similar way to the choice of VxWorks, namely XTP adding to the determinism of the protocol stack, by providing a way to send priority data to the SBA enable FDDI driver. This provides stability for the network and ensures timely delivery of the network packets.

4.5.6. Fiber Distributed Data Interface (FDDI)

FDDI consists of dual redundant fiber-optic token ring architecture capable of 100 Mbits/second transmission. Only one of the rings is active at any one time. The second serves as a backup. The IMS FDDI NIC specification asserts:

"Due to its efficient timed token protocol, realisable throughputs of up to 95 Mbps are possible. Owing to its high capacity, FDDI offers the capability to support both multimedia and SFT III solutions." [FDD97]

Asynchronous and synchronous data routing are available in the PMC FDDI card. The token and packets have a deterministic flow. Combined with the ability to allocate bandwidth with a Synchronous Bandwidth Allocator (SBA), the delivery of packets can be guaranteed within a time period. The SBA works by allocating a portion of the total bandwidth to each station thus ensuring the transmission of the data flagged as synchronous. The station that ensures that each station is allocated the required bandwidth is thus called the Synchronous Bandwidth Allocator [FDD97]. The SBA is used to provide NTP with guaranteed access to the network, thus helping eliminate the "network jitter" common to network communication usually. This jitter in normal Ethernet LANs is largely responsible for the majority of the random inaccuracies seen in the NTP offsets. The high network speed will also serve to significantly lower the packet turnaround delay. The envisioned IMS network has a ring of 1km, a media

propagation delay of 5.1μs, and a station delay of 0,6μs per station for 30 stations [YOU96]. Thus the maximal delay attributable to the FDDI will be

$$1 \times 5.1 + 30 \times 0.6 = 23.1\mu s$$

$$1 \times 5.1 + 2 \times 30 \times 0.6 = 41.1\mu s$$

in normal mode and wrapped mode respectively. The normal mode of operation requires the token to pass through the 30 stations, while wrapped mode describes the state where one station's FDDI card malfunction causes the loopback of the dual ring through each station twice.

Figure 15: FDDI PMC Daughter Card

The PMC FDDI card (Figure 15) was designed with the requirement of the Survivable Adaptable Fiber Optic Embedded Network (SAFENET) standard [SAF94] as a guideline and thus with its fault tolerant, deterministic and bandwidth-efficient token passing is eminently suitable for the distributed real time environment. At the time of the decision ATM was not capable of fitting in with the architecture, being unable to support STREAMS or XTP. When

considering these above facts, the choice between the Ethernet and FDDI was a simple one.

4.5.7. Simple Network Management Protocol (SNMP)

Simple Network Management Protocol (SNMP) is a standard [SNM93] that allows for the querying of the operational the state of software. By using SNMP the delay from station to station can be further narrowed down through the quantification of the delay. Depending on the size of the network involved this may not be worth while.

The major problem in the delay calculation being the calculations for offset (θ) and synchronization distance (λ). Both these calculations use a division by two factor when calculating delays. In a token ring network (like FDDI) the asymmetry of the network ensures that the delay there and back is almost never half. In the proposed network the propagation delay figure was calculated as 23.1µs in normal mode and 41.1µs in wrapped mode. This asymmetry can result in a maximum error of 11,55µs or 20,55µs respectively, should the client and server be placed immediately next to one another.

Using SNMP, a knowledge of the position in the token ring of NIC could be built and in turn using this factor, the divide by two could be substituted with the new factor. At this stage the maximum errors are too small to warrant the inclusion of this additional effort, but the inclusion of SNMP in the stack has allowed for the later addition of this capability.

The systems maintenance functions available from SNMP could be used to build SNMP functions into the timing services thus providing remote control and

routine maintenance from a console. The SNMP data is stored in SNMP Management Information Blocks.

Chapter 5

5. SOFTWARE DEVELOPMENT

5.1. Introduction

The decisions taken in Chapter 4 meant that it was necessary to integrate FDDI, SBA, XTP, NTP and develop the NTS software for a full implementation as is depicted in Figure 16 in Paragraph 5.5. The process of experimental prototyping, used to gain knowledge of the issues to be addressed for each of the software items, is described before the full implementation. This process was vital in providing insight into the specific software necessary to chart the most effective course for the implementation. The development of these various software items is discussed in this chapter and the porting work involved in the re-targeting of the software to VxWorks is described. The development of the FDDI device driver, the Data Link Provider Interface, the Xpress Transport Protocol, the Network Time Protocol and the custom Network Time Services software is described in detail. Attention was given to how an open systems programming paradigm was used and to the lessons learnt from integrating complex software stacks.

5.2. Guides for Code Porting

The following principles were developed and applied when making the needed changes to the software. The principles are in line with the guidelines and standards of the Free Software Foundation (FSF) [FSF98], a group closely linked to the Open group [UNI98] that is responsible for the Unix and POSIX [POS98] compatibility of various systems.

- Code integrity should be maintained as much as possible. Where possible changes are to be made externally to the COTS software. A set of porting code that transposes the needed COTS functions to the required functions can be supplied and linked with the COTS source. Where this is not possible the code should be clearly marked and compilable with an appropriate compiler directive.

- Code process and execution should not be altered if at all possible. Supplementary code for prerequisites should be supplied and the calls to these prerequisites should follow the point above concerning execution state and code location.

- Code written to port the COTS software should be written so that it can be generalized to other situations if needed. (Avoid writing two routines in different areas of code if one can be used)

- Use the COTS data types and provide a header file mapping those to the target operating system/compiler

- The additional code should be documented in the source to show all assumptions.

5.3. System Implementation Approach

After initial investigation the different components were used in prototype situations working as part of a known, stable environment. Once the functionality, suitability and robustness of each individual component had been confirmed, the task of deploying the components on the platform of choice commenced. Each component of the stack was redeveloped and changed as

needed (within the guidelines stated above) to operate on the target hardware. The final implementation was to be PROM'ed into the FDDI NIC.

5.4. Experimental Prototyping

Each of the software protocols was experimentally tested to verify the decisions made in the design phase. Insight gained during the prototype stage helped to ease the path for completion of full implementation.

5.4.1. NTP Prototyping

The xntp software implementation of NTP was selected as the implementation to be used. This selection was made for the reasons detailed in Chapter 4. Work on xntp is encouraged and it is guarded by a license protecting copyright but promoting enhancements[1]. This is similar to the Gnu Public License [FSF98] (called "copyleft") which accompanies the tools of the Free Software Foundation. The xntp software is maintained by Prof. D. L. Mills of University of Delaware. The aim of the prototyping is to verify NTP's suitability and to identify the problems that could be expected when implementing the software stack. NTP was thought to be a good measure of this because of the mix of system calls it uses.

5.4.1.1. Linux

The source was downloaded and after inspection it appeared to have large PC based support on the Linux [LIN98] operating system. Linux is a FSF version of Unix, which makes use of both BSD and SVR4 system calls. Linux was obtained and installed after which the then current version of NTP (xntp3.4s) was compiled. As the prototype needed a server, an additional Linux machine was

[1] Source code for xntp is freely downloadable from ftp.udel.edu/pub/ntp.

made available. The two machines were linked in an isolated Ethernet network and allowed to synchronize. The initial local clock training period is approximately 24 hours in order to stabilize clock drift.

An oscilloscope was attached to the parallel port and a program was written to send a spike to the appropriate pin each second. The machines only ran the xntp software on the operating system and the offsets where regularly recorded at 100us. The oscilloscope was used to verify and validate the xntpdc program (also part of the xntpd software). The xntpdc program allowed the querying of the xntpd daemon (main executable) and was used for obtaining statistics from the daemon. Once the xntpdc was shown to be functioning correctly the oscilloscope was dispensed with.

The Linux network was further expanded by two other machines. It was then included in the normal working network. The test network now contained a variety of x86 platforms as can be seen from Table 2 below.

Machine Name	Operating System	Processor
Ktc.ccii.co.za	Linux 1.2.13 & 1.2.3	Pentium 100
Gateway.ccii.co.za	Linux 1.2.1	486DX2-66
Laba.ccii.co.za	Linux 1.2.13 & 1.2.3	486DX2-50
Iccs1.ccii.co.za	Linux 1.2.13	486DX2-66

Table 2: Linux and NTP Test Machines

Statistics gathered overnight showed continued good results (80% of time within 250μs). However, the general working network traffic during the day revealed the real problems which were to be faced. The additional Ethernet traffic

degraded the xntpd test stations offsets to millisecond range. These offsets were unstable from polling interval to polling interval.

The initial Linux tests were successful for the following reasons:

- First installation of source ran without problems
- The Intel x86 platform (which was to be our end target) was tested to run the software correctly thus the VFO could work on the x86.
- The initial results were much better than expected in an isolated LAN with no code enhancements.

5.4.1.2. LynxOs

Whilst researching the decision to use VxWorks the real time operating system at our disposal was LynxOs[LYN98]. As the VxWorks operating system had not arrived by the time it was necessary to continue the NTP prototyping, a decision was taken to port NTP to LynxOs. LynxOs is a real time re implementation of Unix. The system call convention is based on SVR4 but suffered from some deviations at the time of testing. This phase of prototype testing allowed the author to become intimately familiar with the xntpd code as it was necessary to make a variety of changes in order to get xntpd to run on LynxOs. The machines used were chosen from the same set that had proved reliable on Linux to maintain the hardware link and relegate all the possible errors to the software domain. The setup of the machines was as shown in Table 3 below.

Machine Name	Operating System	Processor
Ktc.ccii.co.za	LynxOs 2.3	Pentium 100
Laba.ccii.co.za	LynxOs 2.3	486DX2-50
Iccs1.ccii.co.za	LynxOs 2.3	486DX2-66

Table 3: LynxOs and NTP Test Machines

Once the porting of the xntpd code to LynxOs was done it was found that although the machine would synchronize for a short time, the ability to add a differing amount at each clock tick was faulty. This routine (the `adjtime()` call) is fundamental to NTP's ability to adjust the frequency (as mentioned in Chapters 3 & 4). Owing to the proprietary nature of the LynxOs operating system the tests on LynxOs failed. LynxOs affirmed that the code was faulty but were unwilling to release the code for the operating system and unwilling to make a priority of fixing the bug themselves. This was disappointing as LynxOs was to be the first test under a hard real time operating system, with true preemptive scheduling. A major step that was taken was the upgrade of NTP from xntpd3.4s to xntpd3.4y and the integration of the porting code into this release.

5.4.1.3. SCO

SCO [SCO98] was chosen as the next NTP prototype as SCO had support for the chosen S&K PCI FDDI card (used as a template for the PMC FDDI card) which up until this stage had not been tested with NTP. SCO Open Desk Top (ODT) is also a SVR4 type system and thus the lessons learned from the LynxOs port proved valuable in porting the code to SCO. The port highlighted issues regarding simplifying the moves between operating systems and the principals listed in Guides for Code Porting above were developed. Once again the common hardware thread was maintained using the machines listed below in Table 4.

Machine Name	Operating System	Processor
Ktc.ccii.co.za	SCO ODT 5	Pentium 100
Laba.ccii.co.za	SCO ODT 5	486DX2-50
lccs1.ccii.co.za	SCO ODT 5	486DX2-66

Table 4: SCO and NTP testing Machines

SCO, like LynxOs, suffered from a broken `adjtime()` call, and the expectations of this test were not fully met. The installation and porting to another operating system added to an understanding of NTP and the interface between the operating systems used and NTP. Through this process greater knowledge was developed regarding the needs of NTP and its demands on an OS. The NTP code was updated to 3.5b and the previous ports were integrated. The main contribution of the SCO port were the following:

- NTP ran under TCP/IP on FDDI
- Network delay was radically reduced
- Dispersion errors were reduced
- Updating of NTP code simplified by porting guidelines

5.4.1.4. Summary

The initial experimental work with NTP strengthened the arguments presented in Chapter 4. The use of NTP on the various operating systems all showed the contributions that could be made by the factors in the protocol stack. At this stage NTP had been tested to work with the initial PC hardware and FDDI card, albeit not in a combined stack. The knowledge of NTP was increased and the expectations of the software were confirmed. Lessons of code portability and the myth of true open systems were exposed to show the non-conformity between major players in the Unix market. The two major streams of Unix (BSD and SVR4) were both examined and the author became familiar with the differences that affect NTP. Contact was established with the NTP research community and dialogue was opened. The frequent updates of the NTP code became known and thus the idea of modularity of layers was confirmed as a wise choice.

5.4.2. FDDI Prototyping

The source code for the Schneider and Koch [SYS98], a C^2I^2 Systems Partner company, PCI FDDI device driver was obtained. This raised the issue of communication between the layers. There are two major types of communication between protocol layers. One being BSD oriented, and the other, STREAMS oriented. The driver code obtained was STREAMS code built to work on SCO Unix. The aim of the FDDI porting exercise was to establish the portability of the code to other versions of Unix, in particular VxWorks. STREAMS provides an architecture were a message is passed from an upper protocol "down stream" to another protocol by the addition of a header on the message and vice versa on the way up the stream from the device driver. The work needed to convert the S&K source code to BSD style was deemed a waste as VxWorks had a STREAMS system that could be used, and STREAMS was seen as the de facto Unix industry standard.

5.4.2.1. SCO

The logical choice was to first confirm the workings of the source on it's native environment. The code was compiled using the native SCO C compiler and then installed into the system. The code was then compiled with the GNU [FSF98] C compiler and installed into the system to verify that the GNU compiler did not alter the byte alignment or anything else so as to render the object code non-functional. This was an important step as the GNU C compiler is the native compiler shipped with VxWorks and would form the basis for the porting of the entire stack. By providing an easy to check SCO installation, the updates of the FDDI source could be verified very quickly. The FDDI source was updated three times during development, once again reinforcing the decision in favour of a modular stack. At this point the Synchronous Bandwidth Allocator could not be tested owing to lack of functionality on the SCO platform.

5.4.2.2. LynxOs

An attempt was made to port the code to LynxOs but this was cut short when Media Access Control (MAC) layer interface was found to be missing. The SCO OS has a unique way of separating the MAC layer from the Data Link Provider Interface (DLPI), resulting in the need for a separate MAC Layer. It was felt that the time needed to port the code to LynxOs would not prove worthwhile and work was begun on the VxWorks port.

5.4.2.3. Summary

The author became familiar with SCO PCI FDDI code and the MAC driver interface issue was raised. Communication with S&K was established and the use of a STREAMS message architecture was adopted for the protocol stack.

5.4.3. XTP Prototyping

After initial investigations into the BSD message style XTP from NetXpress[NET98], the Mentat Inc.[MEN98] STREAMS based XTP source was acquired. Using a similar paradigm to that of the FDDI source, the source was compiled and tested on the Solaris OS. The VxWorks development host (a SPARC Ultra machine) was used to verify the software source when updated and to serve as base version for XTP which the port could be tested against. The XTP source was updated a similar three times during development, with two major revisions therefore being undertaken.

Because VxWorks was now available, the source was not compiled or tested on the SCO STREAMS.

5.5. Full Scale Implementation

The implementation of modular Network Time Synchronization software required the replacing of the protocols decided upon with the following COTS

software products when viewed against the Real-Time LAN profile of [YOU96], (the areas of specific interest changed during this thesis are shaded in light grey).

<table>
<tr><th>Layer No.</th><th>ISO OSI Layers</th><th colspan="2">Real-Time LAN Profile</th></tr>
<tr><td rowspan="2">9</td><td rowspan="2">Application Process*</td><td colspan="2">Software Application Tasks</td></tr>
<tr><td>Network Management Services</td><td>Timestamping Services</td></tr>
<tr><td rowspan="3">8</td><td rowspan="3">Operating System*</td><td colspan="2">POSIX 1003.1b Real-Time Extensions (Wind River Systems)</td></tr>
<tr><td colspan="2">VxWorks (Wind River Systems)
STREAMS for VxWorks (Wind River Systems)</td></tr>
<tr><td>7</td><td>Application</td><td colspan="2">Network Time Service (custom software)</td></tr>
<tr><td>6</td><td>Presentation</td><td>Network Time Protocol (xntp University of Delaware)</td><td>SNMP (Wind River Systems)</td></tr>
<tr><td>5</td><td>Session</td><td></td><td></td></tr>
<tr><td>4</td><td>Transport</td><td colspan="2">XTP for STREAMS (Mentat Inc.)</td></tr>
<tr><td>3</td><td>Network</td><td colspan="2" rowspan="2">Data Link Provider Interface for STREAMS (Wind River Systems)</td></tr>
<tr><td rowspan="2">2</td><td rowspan="2">Data Link</td></tr>
<tr><td colspan="2">STREAMS based PCI MAC FDDI Driver (Schneider & Koch DatenSysteme)</td></tr>
<tr><td>1</td><td>Physical</td><td colspan="2">ANSI FDDI PHY Protocol</td></tr>
<tr><td>0</td><td>Cable layer*</td><td colspan="2">Multimode Fiber cable Plant</td></tr>
<tr><td colspan="4">Note: Layers marked with an asterisk (*) fall outside the ISO OSI 7 layer Model</td></tr>
</table>

Figure 16: Software Stack on Real-Time LAN

The task of fully implementing the protocol stack could be divided into sections from the physical ISO OSI layer to the application and presentation layer. The standard VxWorks communications stack comes with native (BSD based) Ethernet TCP/IP software. The path to completion from the FDDI driver up as shown in Figure 16 above was as follows:

- Porting the STREAMS MAC PCI FDDI Driver to VxWorks
- Porting the VxWorks DLPI layer to communicate to the lower level STREAMS drivers
- Porting XTP for STREAMS to VxWorks
- Porting xntpd and utilities to VxWorks
- Porting xntpd and utilities to XTP from TCP/IP
- Porting the SNMP manager to VxWorks
- Design and implementation of the NTS software on the ported and integrated software
- Design and implementation of the NTS application software for testing

Although the tasks have been divided in a sequential fashion for explanatory purposes, on many occasions, the physical work took place in parallel while awaiting the responses to support that had been requested. The time taken to complete each task should be seen as incremental i.e. the time can be added

together in order to establish an idea of the total time that was taken to complete the porting exercises. The code body to be changed and ported is upwards of

200 000 lines of C code. Consequently much time was spent in understanding and deciding on the best manner in which to interact with the code.

5.5.1. VxWorks

The VxWorks [WRS98] operating system detailed in Chapter 4 was chosen as the implementation operating system. VxWorks is a hard real time embeddable kernel with Unix and POSIX and BSD like system call interfaces. VxWorks is sold with a large proportion of the OS code in order to enable the user to add additional functionality. The structure of VxWorks differs from the previous OS's as the VxWorks kernel runs all user loaded code in kernel space – in fact there is only kernel space in VxWorks, tasks are differentiated solely on priority.

This therefore requires a shift in the manner in which programs are compiled, loaded and executed. The program needs to be compiled by a cross-compiler, then downloaded and dynamically linked into the target's kernel. This can be seen in Figure 17 below. (taken from the Tornado White Paper (MCL-WHP-TO-9612) at [WRS98])

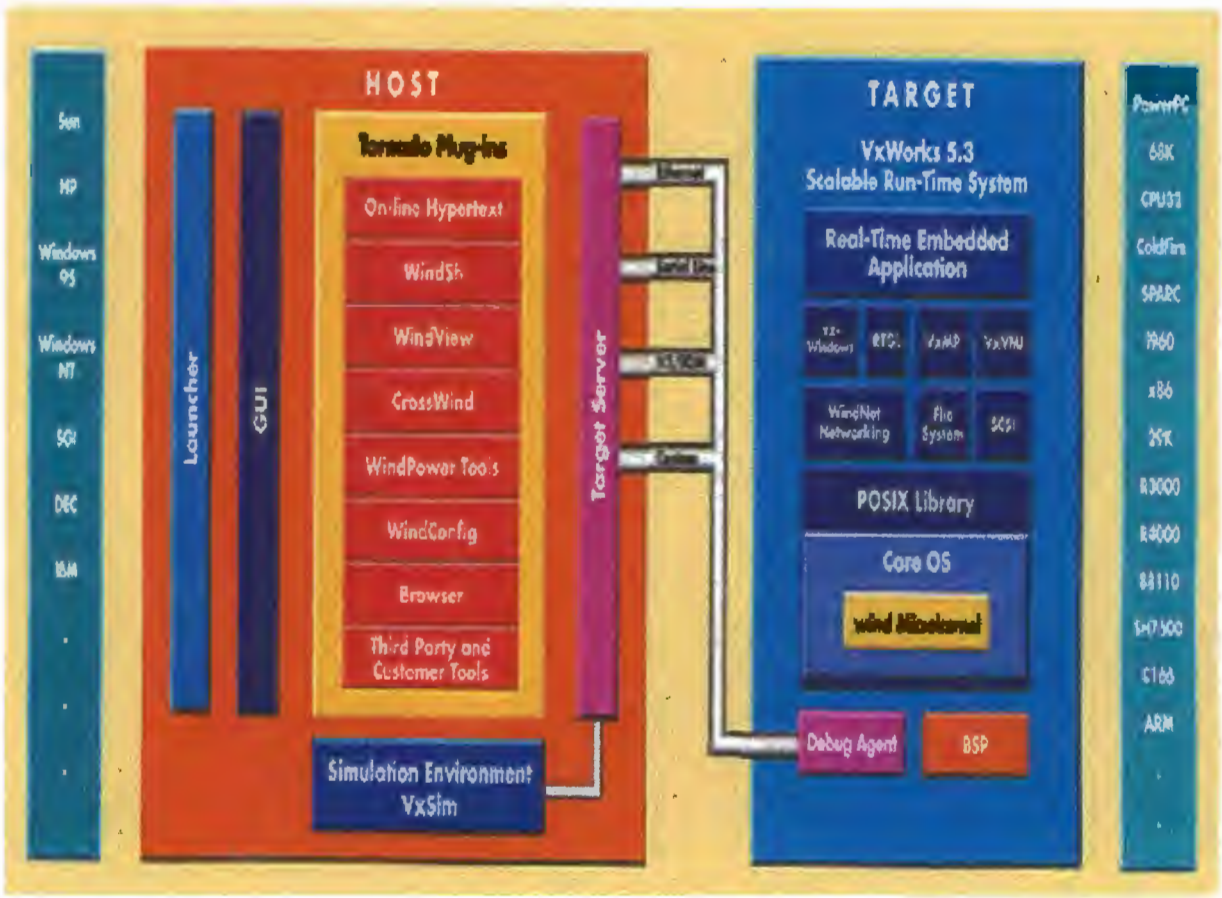


Figure 17: VxWorks Tornado Architecture

5.5.2. Testing Methodology

The software testing was broken down into unit tests, module tests, integration tests and finally, acceptance tests. Structuring the tests in such a manner ensured that should modules be updated, the path to integration remained constant. This also ensured that upgrades were checked for "stack compliance" before becoming an accepted part of the protocol stack. Each level of testing was structured in order to find problems as soon as possible and thus to avoid having to deal with minor issues in the integration stage.

The verification of each file during implementation was tested by means of individual software unit tests. These were used to test the specific blocks of code. New code was tested by this means as well as through the changes implemented into the COTS software. Unit tests ensured that code could be retested when updated and the standard set was maintained in order to ease module testing later. This was particularly important when considering that xntpd, the FDDI driver and XTP were each updated three times during the development cycle. These unit tests will further serve to integrate additional improvements at a later stage.

Once the module was nearing completion protocol tests were run to remove final errors and to ensure that the module met the prescribed requirements for the protocol concerned. The detailed Acceptance Test Requirements of the integrated NTS stack can be seen in Appendices D and E.

5.5.3. Porting PCI FDDI Driver

The porting could be divided into further stages

- Incorporating the STREAMS modules in VxWorks
- Learning the STREAMS architecture
- Writing the MAC Driver Interface (MDI) emulation code.
- Providing system call codes changes
- Moving non Interrupt Service Routine (ISR) embeddable calls
- Adding the driver to the kernel

The STREAMS module furnished with the VxWorks OS required extensive patching before it became functional. The release of STREAMS for VxWorks

was in beta testing and the IMS team was one of five teams using it. Using this as an opportunity to become acquainted with the WRS methodology the author prepared software patches, which were inserted into the VxWorks kernel and tested. This provided the opportunity to become acquainted with the STREAMS architecture and the method used to communicate via STREAMS. Once successful, the patches were forwarded to WRS and they were asked to verify that the patches would not cause any side effects.

As the SCO driver code stopped at the MDI level, and the WRS STREAMS module stopped at the DLPI layer, it was necessary to reconstruct the missing layer. The start of the reconstruction began by providing the missing MDI calls. As the SCO MDI source is proprietary the SCO header files, manual pages, the MDI driver source and the Advanced Hardware Guide [AHG95] were used and the needed calls were reconstructed. The code concerned was responsible for providing the MAC table management (for multicast purposes) and the MAC level acknowledgement and error handling.

The individual references to the MDI system calls were kept in the code and the additional functions were written in a VxWorks library that was created to link to the driver, as per the decision in the guides to porting code.

Many other SCO specific calls could be redefined to be the VxWorks equivalents via the use of C preprocessor directives. However, when this strategy proved inefficient the routine could be changed by providing the additional functionality needed inside C function wrappers. Examples of these were the PCI interface routines needed for the initialization of the MDI driver and other hardware dependent routines.

The STREAMS specific calls in VxWorks limited those permissible inside an Interrupt Service Routine (ISR). These problems were very difficult to trace as

much of the documentation for the VxWorks STREAMS is the standard Unix documentation. Consequently, the notes pointing out the ISR differences were in separate documents and found by the calls' omission from the list of permissible calls. Two examples of this kind will now be described. The `putnext()` call is needed to transfer a message from the current module to the next module. It is allowed in the ISR in most common Unixes, however the real time nature of VxWorks caused this call to be disallowed in the ISR. In the MDI source received the `putnext()` call was called directly from the ISR. This problem resulted in the need to place the received message into a local queue and an additional service routine being added to place the queued message into the next modules queue. The `allocb()` call needed to allocate memory was disallowed in the ISR for a similar reason, as a result a separate task was written to supply a ring of buffers to allocate memory. In the ISR the call to `allocb()` now called for a pointer to an unused buffer instead of actually allocating the memory. The method of loading the MDI driver into the kernel was vastly different to the SCO method and needed to be rewritten entirely.

The porting of the FDDI driver code which incorporated the Synchronous Bandwidth Allocator took four months owing to the low level of STREAMS knowledge to begin with and the problems experienced with the beta version of the VxWorks STREAMS software. This port provided much of the learning curve for STREAMS and was valuable when the later ports were managed. The code received form SysKonnect was in beat stage for SCO when the port began and the code body was updated three times during implementation. The lessons learnt from the xntpd prototypes were put to use to ensure that code was easily ported from one version to the next.

5.5.4. Porting the DLPI

The WRS DLPI library was initially designed to take messages from the upper STREAMS queues and deliver them to the native BSD style Ethernet driver. The DLPI layer needed to be extensively altered to connect to the STREAMS MDI FDDI driver below. The needs of the MDI driver were more extensive than the DLPI catered for as it was a partial subset of the DLPI 2.0 [DLP91] implementation. The porting of the DLPI could be divided into the following sub tasks:

- Module interface changes needed for downstream
- Additional calls needed by the DLPI layer
- Message header changes needed for downstream and upstream

The opening , attaching, sending and receiving sections of the DLPI needed to be altered, meaning that all areas of the code needed revisiting to update it in order to use the FDDI driver that was now available. In each case the items relating to the sending of the packet to the Ethernet driver had to be replaced with the code necessary to establish communication with the FDDI driver. The opening and attaching were changed to initialize the MDI driver and to open a Service Access Point (SAP). The send routine was changed by the addition of a Sub Network Access Protocol (SNAP) header to the incoming DLPI message and then adding functionality to flag the Synchronous or Asynchronous bit. On the receive queue the DLPI header was added and sent up to the responsible SAP.

The additional I/O control calls were developed in a similar way to the SCO MDI emulations, the DLPI Specification and the manual pages enabled the promiscuous and multicast `ioctl()` calls needed by the MDI driver to be included

As much of the knowledge acquisition had taken place during the preceding months the port went quickly and was completed within a month. The I/O control calls followed a month later. The DLPI layer is distinctive in that it is the only component that has not been updated as yet.

5.5.5. Porting XTP

Work began on the XTP code and in a manner similar to the MDI driver added a VxWorks library to link in with the XTP source. This alleviated the need for extensive changes to the upgradable code. A set of redefinition C headers was added to allow the simultaneous running of XTP and TCP/IP sockets which added greatly to the remote debugging efficiency. The code was ported to run on the native WRS DLPI within three months.

The port of the XTP to the new DLPI on FDDI was done by changing the ported DLPI layer so that the ported XTP did not need to be changed. The reason for this being the desire to contain the changes within the simpler section of code. New startup routines for the XTP were required and the author spent two weeks accomplishing the porting and running the module tests needed to ascertain the effectiveness of the port. The XTP was well documented and easy to follow owing to the employment of a similar strategy.

5.5.6. Porting NTP

NTP was the area that required the most concentration to ensure maximum benefit from the rest of the protocol suite. The NTP code was well known by this stage. As a result most of the time taken porting to VxWorks was related to real time issues and to replacing BSD or SVR4 with POSIX calls where possible (e.g. the alarm timers were changed for POSIX timers). For a partial list of the changes see the NTP selected sources in Appendix F. The initial port was undertaken as part of the investigation into real time OS's mentioned in Chapter

4. The xntpd3.4y version was ported within two months. Directly thereafter the port was upgraded to xntpd3.5b. Functionality was added to save the drift values to battery backed RAM.

The transport protocol calls in xntp were changed to use the XTP protocol as implemented by Mentat Inc. The remaining problem was the lack of an `adjtime()` call in VxWorks or in the POSIX system calls. The usrtime library from Monterey Bay Aquatic Research Institute (MBARI) written by R. Herlien was ported to provide accurate clock queries and adjustments. Although originally targeted for the Motorola 68000 it was easily ported to run on the x86 board in use. This allowed the xntpd to run accurately under XTP on VxWorks.

Subsequent to the xntpd3.5f version, the xntpd source was released with the GNU configure program which would accelerate the porting process. Changes needed to conform with the configure program were providing cross-compilation utilities and configurations for the configure program itself and changes to xntp to check for the VxWorks kernel system calls.

5.5.6.1. NTP for VxWorks included in Standard Distribution

A decision was made to submit the changes needed to get xntpd working on VxWorks to the University of Delaware. The changes where incorporated into the standard distribution, allowing other POSIX compliant OS's to benefit from the changes. This also allowed the VxWorks user community to use the xntpd port and thus provide beta test information to supplement the authors own information. In addition to this the added benefit was that the new distributions of xntpd could compile "out the box" for VxWorks. By having the author added to the Copyrights file in the distribution, mail was received indicating the extent to which the new port of xntpd was being used. Information regarding

implementations and thanks were received from the following sources (amongst others):

- Lockheed Martin [LMC98]
- Hughes Aircraft Corporation [HAC98]
- NASA Jet Propulsion Laboratories [JPL98]
- Communications Research Center (Defense Research Canada) [CRC98]
- Alcatel [ALC98]

The community of users helped with the initial testing of the port and errors were diagnosed and found much faster owing to this cooperation.

5.5.7. NTS

The NTS suite was designed using the C++ language to keep in line with the other application server software items mentioned in Figure 7. The aim was to provide a client on the MBII host with the ability to retrieve a timestamp with accompanying status information. The provision of a SNMP MIB was also included to provide the ability to both query and alter the settings in the NTS and to enhance the timestamp retrieved. The NTS was designed to be a compact add on to the system with the ability to provide additional information from NTP when needed.

The structure of NTS was simplified to allow for use on both MBII backplane and standard systems. The NTS software ran a Unix like daemon and waited on a known port for requests. Requests could be of the form get the time (or in special cases set the time). Requests to get the time were furnished with the time and status information. The ability to fix the time exported to the requesting host was

reserved for the time being and the time was passed on as read from the `gettimofday()` system call.

The SNMP MIB was maintained and when the MIB was queried it was dynamically updated to include all the peer information from the `xntp sys_peer` structure so as to allow the SNMP Manager to properly analyze the status of the local clock. Thus, internal NTS could be updated to use this peer information together with known factors to calculate the round trip delays more accurately. The efficiency of xntpd could be further enhanced after tests are studied and the results extrapolated to the particular operating scenario.

Chapter 6

6. INTEGRATION & TESTING

6.1. Introduction

Once the ported software protocols in Chapter 5 were completed the integration of the protocol suite was begun. These issues are discussed with specific reference to the interface between the layers as depicted in Figure 18.

Figure 18: Integration of Components

The modular design and individual module upgrade ability is discussed as this impacts on the ability to keep all the layers in the stack at the newest level, with minimal impact on the surrounding layers. Finally, notions of further improvements are discussed with reference to the previous design decisions.

The following was needed for the total integration of the protocol suite:

- The NTP needed to use XTP for STREAMS and the MBARI USRTIME library
- The XTP needed to use the DLPI layer
- The DLPI needed to communicate with the MDI FDDI driver
- The MDI FDDI needed to communicate with other FDDI cards

6.2. NTP

6.2.1. Integration

The specific tasks involved in integrating the NTP source onto the XTP transport layer involved replacing the TCP/IP sockets with XTP sockets. This procedure was simple once it was realized that the TCP/IP wild card delivery mechanism already built into NTP could be used, as well as the streams initialization for TCP/IP. The only other change required was the rearrangement of the socket opening to cater for the expected XTP sequence. (The Mentat XTP expected the socket options to be set only once the socket has been bound to the interface.)

The changes can be easily seen in the ntp_io.c source listing in Appendix F. Although they may appear quite extensive, they are merely form changes, and as such were not difficult to implement. The routines create_sockets() and open_sockets show the majority of changes where XTP is defined.

NTP has been upgraded numerous times since the beginning of the project, the use of the GNU configure program has made the ability to upgrade the version of NTP simple and quick. A patch file for the necessary changes for XTP and battery backed drift storage are kept which is easily installed with the GNU patch program. The process of upgrading NTP has been tested numerous times and is now a ten minute operation for the NTP and VxWorks literate.

6.2.2. Integration Test Procedures

A server synchronized to its local clock was used to check server end functionality, while a client was established and to synchronized to the server. The functionality of the xntpdc and ntpq query utilities were used to ensure the server and client. The machines were allowed to run and the network was disconnected and reconnected to check the recovery of NTP, the machines were restarted and the resynchronization time checked. The machines were then left running to check the longer term effects. The integration was a success.

6.3. USRTIME

6.3.1. Integration

Once ported the USRTIME library was compiled into the VxWorks kernel and the test code supplied with the library was used to ensure the porting process had been implemented correctly. Once this was established the configure script in NTP was changed to automatically use the USRTIME code instead of the VxWorks code.

6.4. XTP

6.4.1. Integration

The XTP was ported to use the standard VxWorks DLPI layer that communicated via the Ethernet. The integration required no changes to be made

to the XTP layer but required a change in the byte order endianess of the DLPI code and minor changes in the DLPI layer to communicate with the XTP port. The startup routines for XTP were the only changes needed to link with the new DLPI layer.

XTP was upgraded twice during the porting exercise, and once afterwards. While the process is not as simple as the NTP method, Mentat have provided patches that can be applied with the patch program. Thus new code is easily incorporated into the VxWorks compatible source code.

6.4.2. Integration Test Procedures

The Mentat source contained code to test both the virtual circuit and datagram delivery of XTP. This code was used to ensure that no problems were introduced when XTP was integrated with the new DLPI layer below.

6.5. DLPI

6.5.1. Integration

Once ported the DLPI layer required attachment to the FDDI Driver. This was done by supplying a `dlpiInit()` routine and by binding the DLPI layer to the driver on the first `dlpiOpen()` and unbinding it on the last `dlpiClose()`.

6.5.2. Integration Test Procedures

The test code supplied with the Mentat XTP source was used to test the Ethernet and FDDI DLPI layers. The Mentat source included a program called `dlpitest()`, which was used to ensure that the respective DLPI layers gave the same results in test conditions.

6.6. FDDI

6.6.1. Integration

Once the integration of the DLPI layer with the MDI FDDI Interface was complete the protocol stack was fully integrated by the connection of the FDDI cards via the cable plant.

The upgrading of the FDDI code has been done twice before as mentioned in the previous chapter. The majority of the changes needed to ensure the working of the driver have been made externally to the original source. Thus, the process of installing upgrades involves linking in the changes library to the make file and applying the needed changes via the patch program.

6.6.2. Integration Test Procedures

The `dlpitest()` routine described earlier was used to ensure that the messages flowed from one station to the next once this proved successful the XTP test routines were run between two nodes and then between many nodes in multicast testing.

6.7. NTS

6.7.1. Integration

Integration of NTS required little work owing to the fact that VxWorks was an embeddable kernel. This being the case all the variables within the kernel were visible to NTS. NTS made calls to the `gettimeofday()` and `settimeofday()` calls in the MBARI USRTIME library. References were made to the NTP peer structure and to the SNMP MIBs.

6.7.2. Integration Test Procedures

A test program was written by the author to query the time across the MBII backplane. These initial tests were used to ensure that:

- The NTS queries worked inside the integrated kernel
- The communication via the MBII backplane functioned correctly

6.7.3. Synchronization Testing Procedure

Subsequently to the above tests a program was written into the integrated kernel to wait on an interrupt on FP_INTR (Connector 2, Row c Pin 3) on the IMS NIC. Upon receiving the interrupt the kernel generated the current time of day and the NTP estimated error.

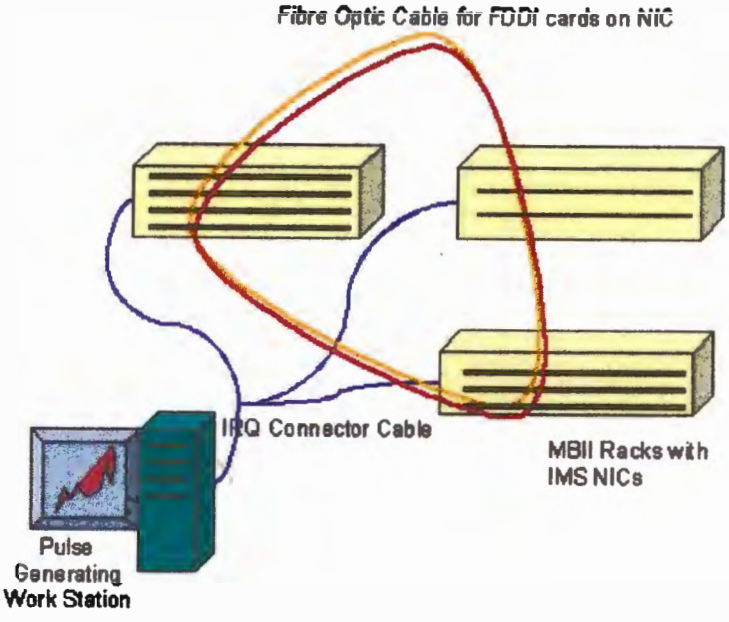


Figure 19: NTS Test Setup

A program was written to generate a spike via the parallel port pin 3 of a separate computer at a user defined time interval (eight seconds was used for these tests

unless otherwise mentioned). A cable was attached at one end to the parallel port pin 3 and at the other end to the FP_INTR of all the NICs to be tested. This process is described in detail in Appendices D & E. Log files were generated for each IMS NIC by using this method. It is these log files that are analyzed for time distribution and studied for both real and NTP estimated offsets. The analysis and findings are described in Chapter 7. The tests were conducted and the accumulated log files were then analyzed as described in Chapter 7.

Chapter 7

7. DATA ANALYSIS AND DISCUSSION

7.1. Introduction

The results of the time tests described in Chapter 6 are discussed and the performance of the synchronization is analyzed. In order to expose the nature of the synchronization offsets during the time trials the graphs of the offsets have been reproduced. The analyses of the time trials, as depicted by the graphs, are summarized into the component characteristics of reliability and performance. The performance of the NTS software is discussed and possible improvements are suggested. These suggestions are then used to draw the conclusions reached in Chapter 8.

7.2. Description of Data Collection Formats

Using the logging method described in Chapter 6, in excess of 50MB of data was collected. This was extracted to time and offset graphs and then further summarized. Selections from these graphs will be presented in this chapter and will be labeled as in Table 5 below:

Axes	Description
X axis	Time in seconds since start of test
Y axis	Absolute Binary fractions of a Second offset between machines

Table 5: Graph Label Convention

Binary fractions of seconds were used for the offsets in order to maintain the storage method of NTP and thus exclude the errors associated with rounding. Table 6 provides the necessary information to compare the two representations and includes the offsets of interest to this study.

Time Offset	Binary Fractions
1 s (second)	4 294 967 297 ($=2^{32}+1$)
1 ms (millisecond)	4 294 967.496
1 µs (micro second)	4 294.967
125 µs	536 870.912
250 µs	1 073 741.824

Table 6: Significant Offsets & Binary/Time Conversions

The duration of the tests varied between 16 and 25 hours. Six of the tests are described and the overall performance characteristics are drawn from these tests. All the tests made use of the NTP v4 (ntp4.g) except the first which was done with NTP v3 (xntp3.5.91). Unique numbers used to identify the tests were allocated at the test run time.

The graphs for the measured offsets, NTP estimated offsets and the differences are depicted. The color codes used in the graphs are represented as seen in Table 7.

Color Code	Respresents
Green	Measured Offsets (MO)
Blue	NTP Estimated Offsets (EO)
Red	Difference between MO and EO

Table 7 : Color Codes for Graphs

7.2.1. Measured Offsets

From the logged statistics gathered in Chapter 6 the time values at each interrupt were extracted. The difference between these times was taken by subtracting the slave value from the master value. This is then the offset of the slave in relation to the master. The required offset should be less than 125µs and it is against this requirement that the synchronization tests are judged.

7.2.2. NTP Estimated Offsets

The NTP keeps the estimated offsets from the master in the system peer structure. These are accessed by the interrupt service routine and stored in the log file. This estimate displays NTP's notion of the offset of the slave from the master. This value is important for determining the reliability of the offset, as we are required to report when it is outside of 125µs.

7.2.3. Discussion

The discussion of each test will also focus on the methods used to achieve better accuracy. The terms availability and reliability will be the primary measures of success. The goal is to achieve a 100 % reliable and available time synchronization service from boot time.

7.2.3.1. Availability

The *availability* of the service is the measure of how many times during the time trial a timestamp marked as within target range (within 125µs in this case) is obtained.

availability = ((# of NTP Estimates > |125us|) + (# of NTP Estimates = 0)) / # Readings

7.2.3.2. Reliability

The *reliability* of the service is determined by the statistical evaluation of the test checking the number of measured offset errors that would not be detected in real time service.

reliability = (# of Measured Offsets > |125us|) /availability

or

reliability = 100 – (undetected errors/availability)

The increase of one factor will not necessarily lead to a decrease in the other factor or vice versa. Both factors are dependent on the threshold for the target range and the degree of agreement between the NTP estimated offsets and measured offsets. Generally, the increase in reliability will show a decrease in service availability. By subtracting the NTP estimated offset from the measured offset the deviation in agreement between the two sets of data is shown.

If it is possible to increase the level of agreement, then both availability and reliability can be enhanced. The NTP Estimates are manipulated in an attempt to achieve better statistical reliability.

7.3. Time Series 22 NTP v3

This initial time series is of interest as it clearly chows the acquisition time needed to train the local clock until the two are in both phase and frequency synchronization. This is the only test in this series that makes use of the NTP v3 code. After this test was completed the NTP v4 was released and the newer code was used as it included methods to shorten the acquisition time needed.

7.3.1. Test Description

Item	Description	Value
1	Trial Length	16 hours 40 minutes
2	NTP	V3 (91)
3	Start Time	Wed 29 April 17h17
4.	# Readings	7343

Table 8 : Test Description (22)

Table 8 shows the length, start time and specific code version used for the test. Normal traffic runs during the day while no traffic runs during the night, thus the start time is significant as it shows at what stage of the day the test was run. The start time can be used as a measure of the state of the network during the test. The user can see from the offset at which stage during the test deviations occurred. A table of this nature will be shown with each time series for this purpose.

7.3.2. Measured Offsets

The acquisition time needed to train the local clock and achieve synchronization can be seen from X values 0 to 20 000 of Figure 20. Once the training period has finished the clock settles down to well within the 125µs target synchronization

time. Between the time values of 26 000 and 28 0000 the break in synchronization and re acquisition can be clearly seen. The speed at which the synchronization is acquired shows the benefit of keeping the drift values.

It is significant to note that only one reading is out of the 125 µs boundary after this acquisition process has been completed.

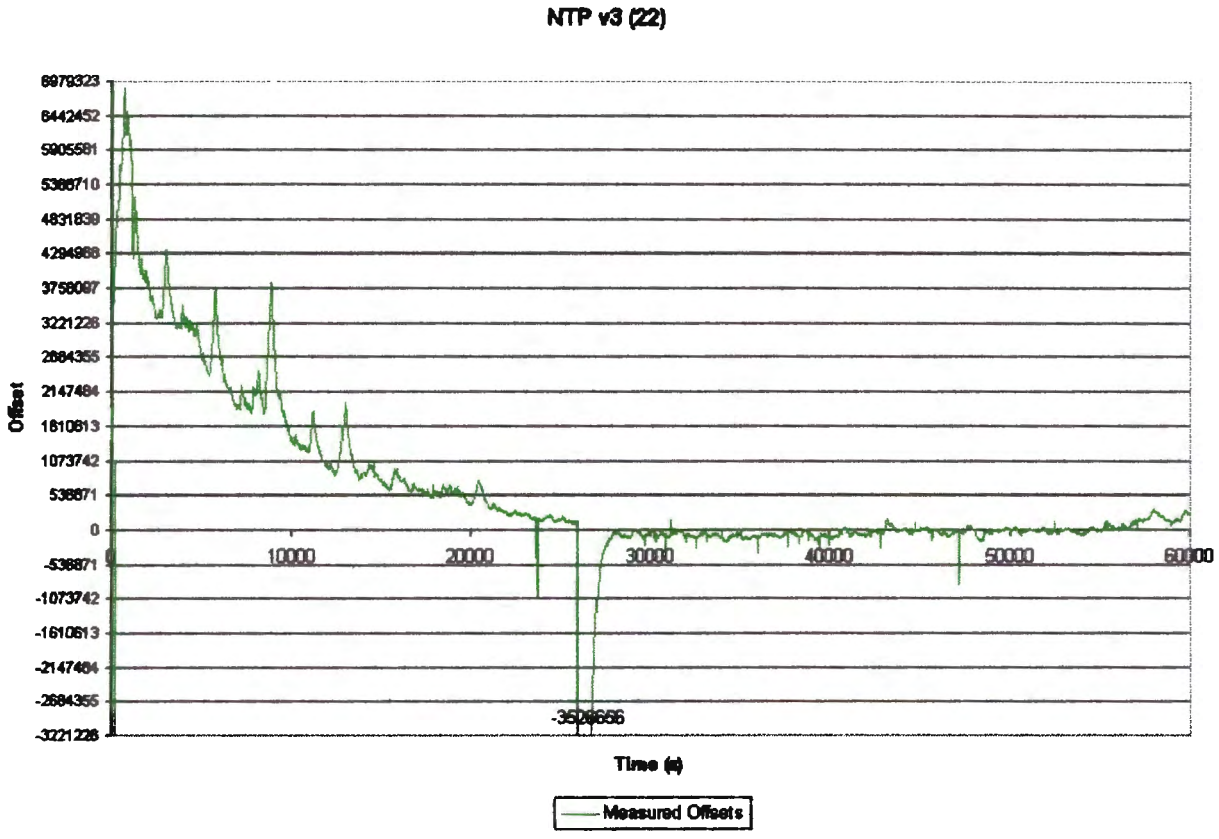


Figure 20: Measured Offsets (22)

7.3.3. NTP Estimates Offsets

The NTP `sys_peer` offsets depicted in Figure 21 show a close correlation to the real offsets measured in Figure 20. From the graph the estimates can be seen to fluctuate more widely than the measured offsets.

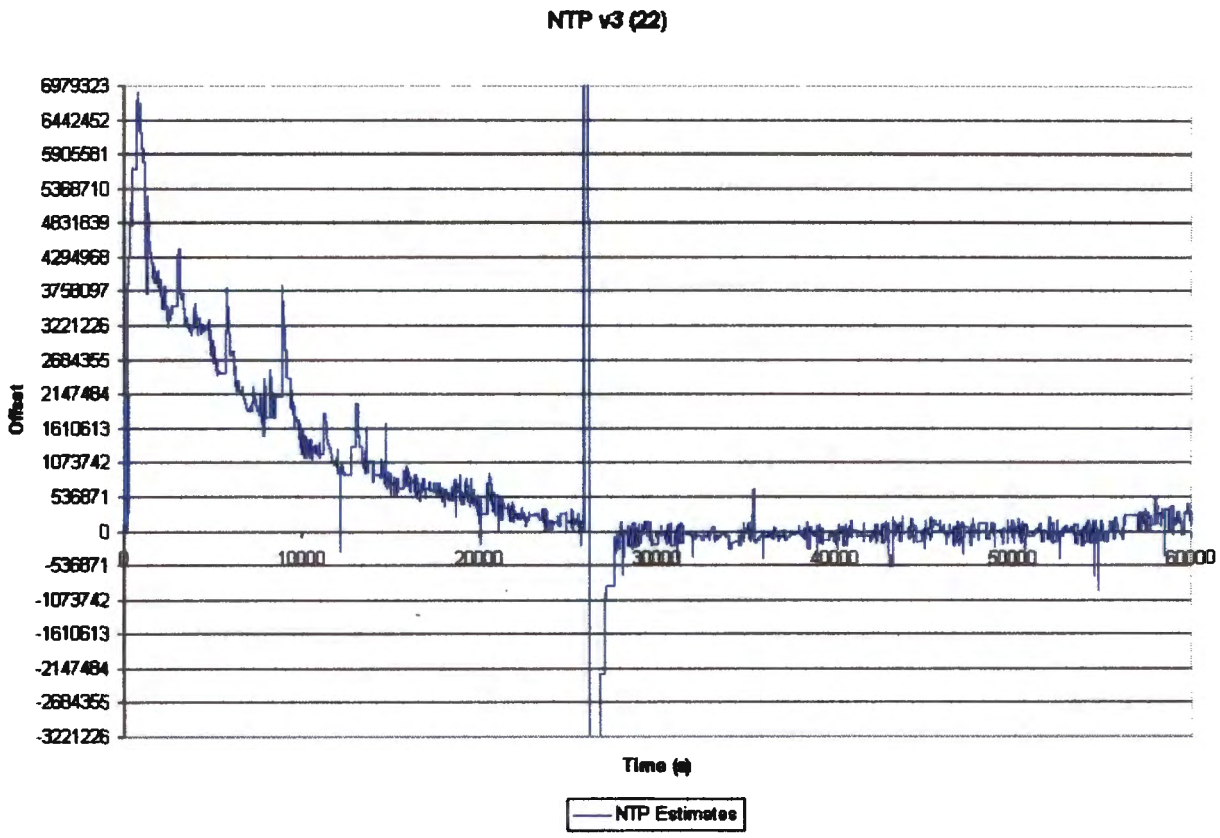


Figure 21: NTP Estimated Offsets (22)

7.3.4. Discussion

Test 22 delivered satisfactory results with the only complaint being the long acquisition time (approximately 5 hours and 30 minutes). Although this time period is well within the 24 hours period customary to NTP, however it would not be tolerable to the goals of this thesis, which seeks a method to provide

reliable time synchronization from boot up time. While it is recognized that time synchronization from boot time is not currently possible it is hoped that this can be further optimized.

The reliability of this test after the 28 000 second time interval was almost 100 % with only one timestamp error that would not be detectable in real time. The slight increase in offsets can be seen at the start of the working day (at the right hand side of Figure 20). This was noted and the daytime vs. nighttime, network traffic dependent performance was monitored in the following tests.

Item	Out of Bounds (> 125µs or 0)	Value
1.	Measured Offsets	3056
2.	NTP Estimates	3130
3.	Unsafe (not currently detectable)	101

Table 9 : Summary (22)

Table 9 shows the summary of the test giving the number of offsets read outside of the 125µs boundary. With no manipulation in the NTS layer the inherent availability of the service is 100 % with a reliability of only 58.382%.

By applying the following set of criteria, reliability can be increased at the expense of availability:

- If the `sys_peer` offset is out of range mark the time stamp invalid.
- If the `sys_peer` offset is precisely zero mark the time stamp as invalid.

Once this is applied the offsets outside of the 125µs threshold that would not be detected in the real time environment (3) are low. The availability will thus be 57.374% while reliability is increased to 97.027 %. As mentioned earlier, of these undetectable errors, 100 occurred during the acquisition period. Although 100% reliability is the goal it is far better to receive unreliable information during acquisition as this can be flagged. In extreme cases where 100% reliability is required, the entire acquisition time could be flagged and the availability will be decreased and yet yield greater reliability if needed.

From this data it can be seen that the most urgent need was to decrease the acquisition time, which was done via the introduction of the NTP v4 code.

7.4. Time Series 36 NTP v4

7.4.1. Test Description

Item	Description	Value
1	Trial Length	13 hours 53 minutes
2	NTP	V4 (G)
3	Start Time	Mon 25 May 20h28
4.	# Readings	5042

Table 10: Test Description (36)

Test 36 was the first time trial using the new NTP v4 code. This test is used to compare the acquisition times between the NTP v4 and NTP v3 code base. The readings were taken at 10 second intervals.

7.4.2. Measured Offsets

The measured offsets from this test (shown in Figure 22) depict target time acquisition within less than five minutes. Offsets greater than 125µs occur throughout the test period. These offsets could decrease the availability at these points, or worse still, could introduce unreliability - should they be undetectable. The effect of these 'erratics' will be shown in the 7.4.4.

Only three offsets greater than 250µs were shown during the test time, which may still be 'safe' depending on the synchronization level required. The normal network traffic of the LAN had no effect on the test.

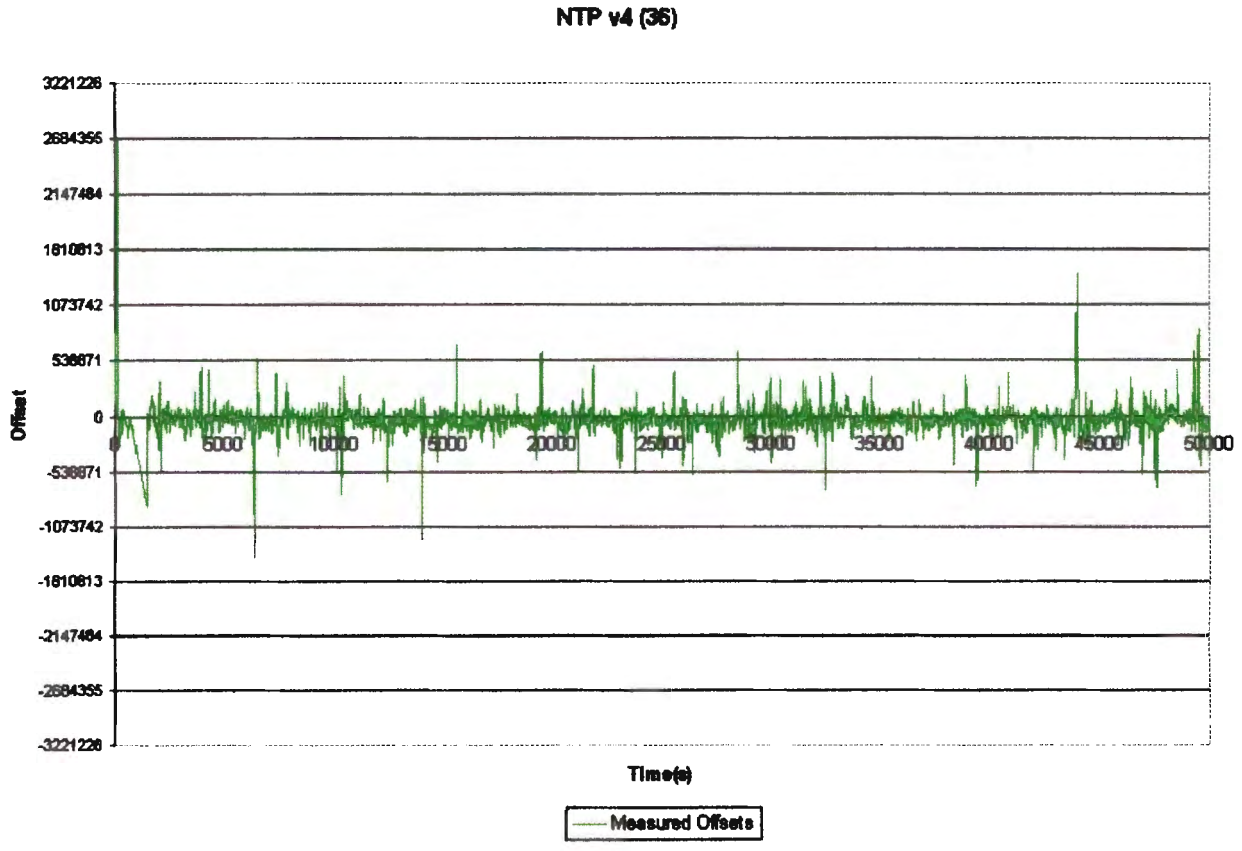


Figure 22: Measured Offsets (36)

7.4.3. NTP Estimated Offsets

As with the previous test the NTP estimates (as depicted in Figure 23) exhibited a high degree of correspondence with the measured offsets

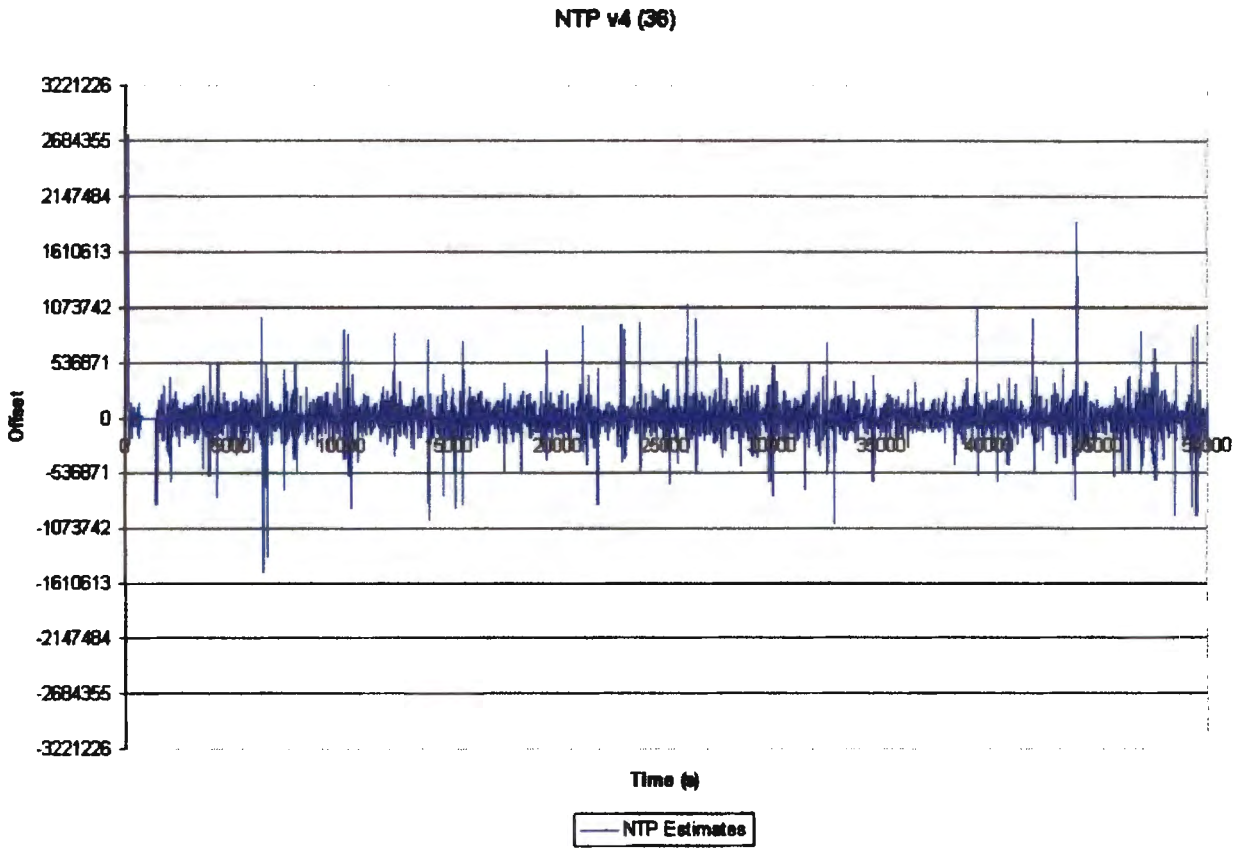


Figure 23: NTP Estimated Offsets (36)

7.4.4. Discussion

The use of NTP v4 proved successful in reducing the acquisition time, this can be ascribed to the new "burst" mode that NTP v4 uses on startup. When the process starts eight messages are sent from the client to the server in quick succession. The client then uses the data received in response to this burst of messages to calculate the synchronization prarameters.

The 'erratics' discovered during the tests raised concerns about of not being able to bound the reliability. This would result in the inability of the current implementation to be used where 100% reliability is required.

Further checks were thus needed to explore the issue of reliability. The remaining tests were run to gauge the full extent of the problem. By gathering a large body of data (greater than 100 hours) and manipulating the NTP estimates it was hoped that a programmable solution could be implemented in the NTS level code that would increase the reliability, while ascertaining the degree to which this would affect the availability.

Item	Out of Bounds (> 125μs or 0)	Value
1.	Measured Offsets	82
2.	NTP Estimates	269
3.	Unsafe (not currently detectable)	28
4.	Mean Offset	-90μs

Table 11: Summary (36)

The figures found in the summary of the previous test (see Table 9) are far less than those found in Table 11 where items(1) and (2) are concerned. The decrease can be ascribed to the shorter acquisition time needed between the two versions of the NTP code.

The major concern raised by this test is the undetectable errors(3) that occurred throughout the test period. The inherent reliability at 100 % availability was 98.373%. However, using the same criteria as in Paragraph 7.3.4, the implementation proved 99.413% reliable, availability of time stamp within target

synchronization was 94,664% during the test, which represents an improvement in availability of close to 38% solely due to the upgrade in the NTP module.

7.5. Time Series 39 NTP v4

7.5.1. Test Description

Item	Description	Value
1	Trial Length	19 hours 27 minutes
2	NTP	V4 (G)
3	Start Time	Thu 28 May 20:49
4.	# Readings	9291

Table 12: Test Description (39)

The details listed in Table 12 reflect the first in the series of tests run to gain data to be used when devising methods to statistically decrease the margin for undetectable errors.

7.5.2. Measured Offsets

The graph depicted in Figure 24 showed a new phenomenon - a specific class of 'erratic' situated in the 71 400 000 -71 600 000 range (both positive and negative) these can be seen in the 2000 second and 60 000 second interval.

This error in reading can either be ascribed to a bit flip in the clock reading or a multiplication error associated with floating point numbers. It is likely that it is the latter as this was experienced by other NTP users and discussed in the NTP Usenet Newsgroup [NTN98].

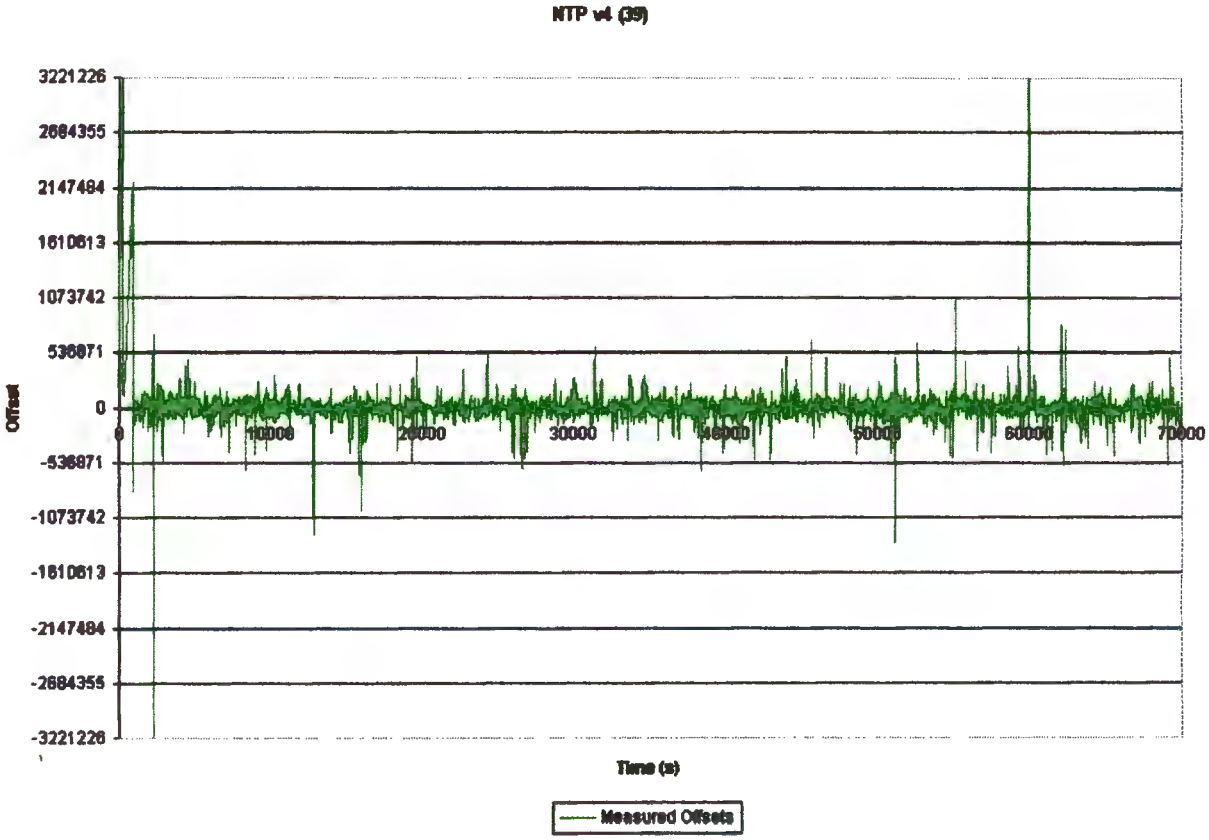


Figure 24: Measured Offsets (39)

7.5.3. NTP Estimated Offsets

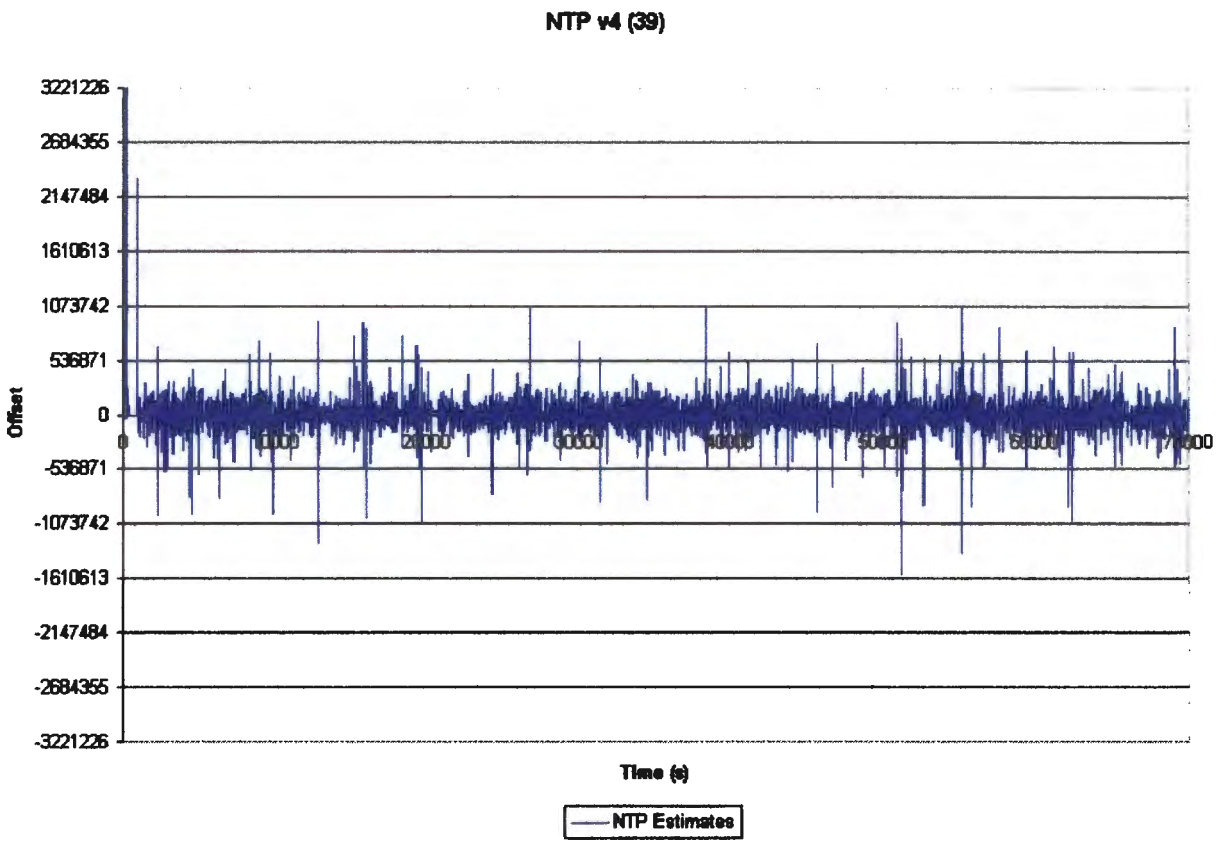


Figure 25: NTP Estimated Offsets (39)

7.5.4. Discussion

The NTP estimated offsets remained consistent between the test 36 and 39 as can be seen in Figure 25. As can be seen, the 'erratics' exhibited much the same behaviour as displayed in time series described in Paragraph 7.4. The issue of network traffic fluctuations once again can be seen to have no effect on the accuracy obtained as was shown in Paragraph 7.4.

Item	Out of Bounds (> 125μs or 0)	Value
1.	Measured Offsets	139
2.	NTP Estimates	439
3.	Unsafe (not currently detectable)	17
4.	Mean Offset	-106μs

Table 13: Summary (39)

Table 13 shows a marked decrease in the number of undetectable errors(3). This indicated the need for more data before any worthwhile extrapolations could be made. The inherent reliability at 100 % availability was 99.817%.

From Table 13 it can be seen that Test 39 provided 95.275% availability and 99.808% reliability, when using the `sys_peer` criteria, a decrease in both reliability and availability was seen.

7.6. Time Series 42 NTP v4

7.6.1. Test Description

Item	Description	Value
1	Trial Length	25 hours 00 minutes
2	NTP	V4 (G)
3	Start Time	Mon Jun 1 14:42
4.	# Readings	11943

Table 14: Test Description (42)

7.6.2. Measured Offsets

The measured offsets depicted in Figure 26 once again showed excellent general offsets punctuated by 'erratics'. Besides the anomaly mentioned earlier in Paragraph 7.5.2, only eight readings were outside of the 250µs range.

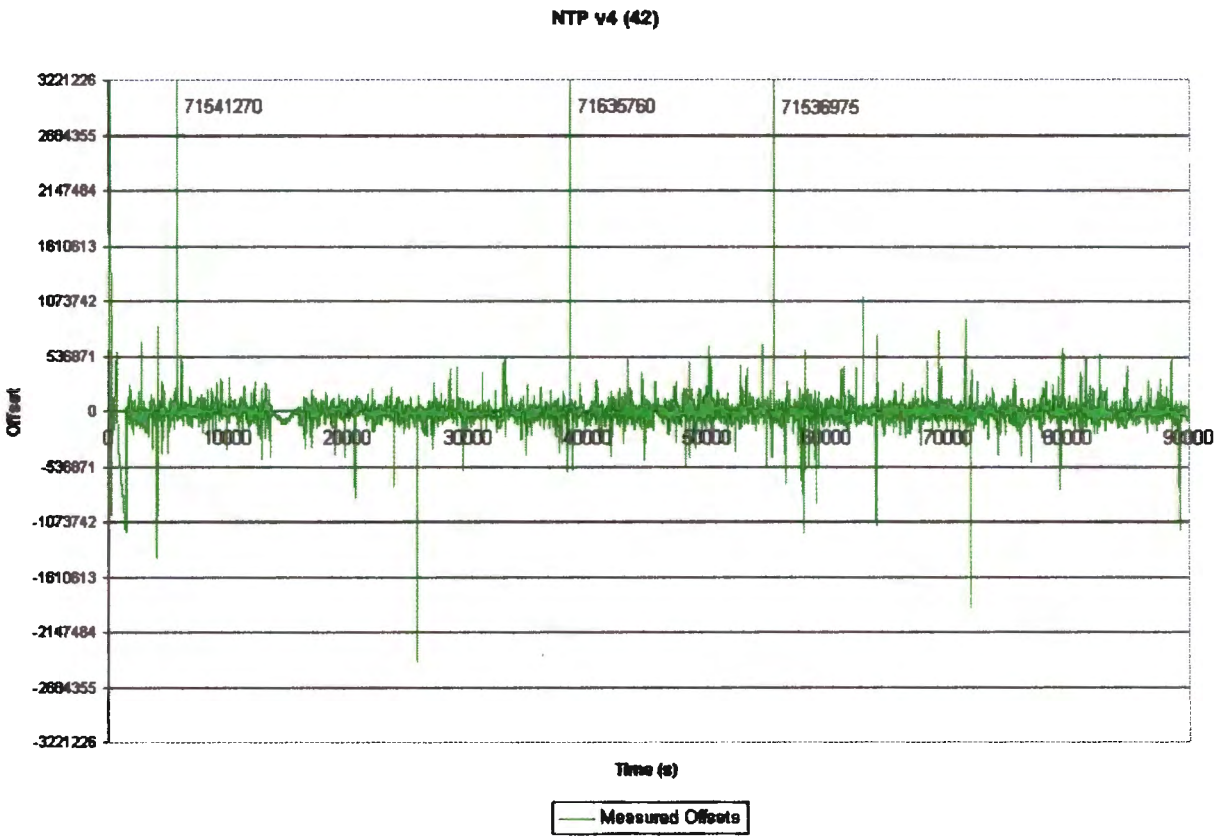


Figure 26: Measured Offsets (42)

The major region of interest is the area between 10 000 and 20 000 seconds. In Figure 27 a clear period of NTP master unavailability is seen which corresponds to a "u" shaped area of offsets in the same region in Figure 26. This area represents an ideal discontinuity between a master and slave with a properly trained client clock.

7.6.3. NTP Estimated Offsets

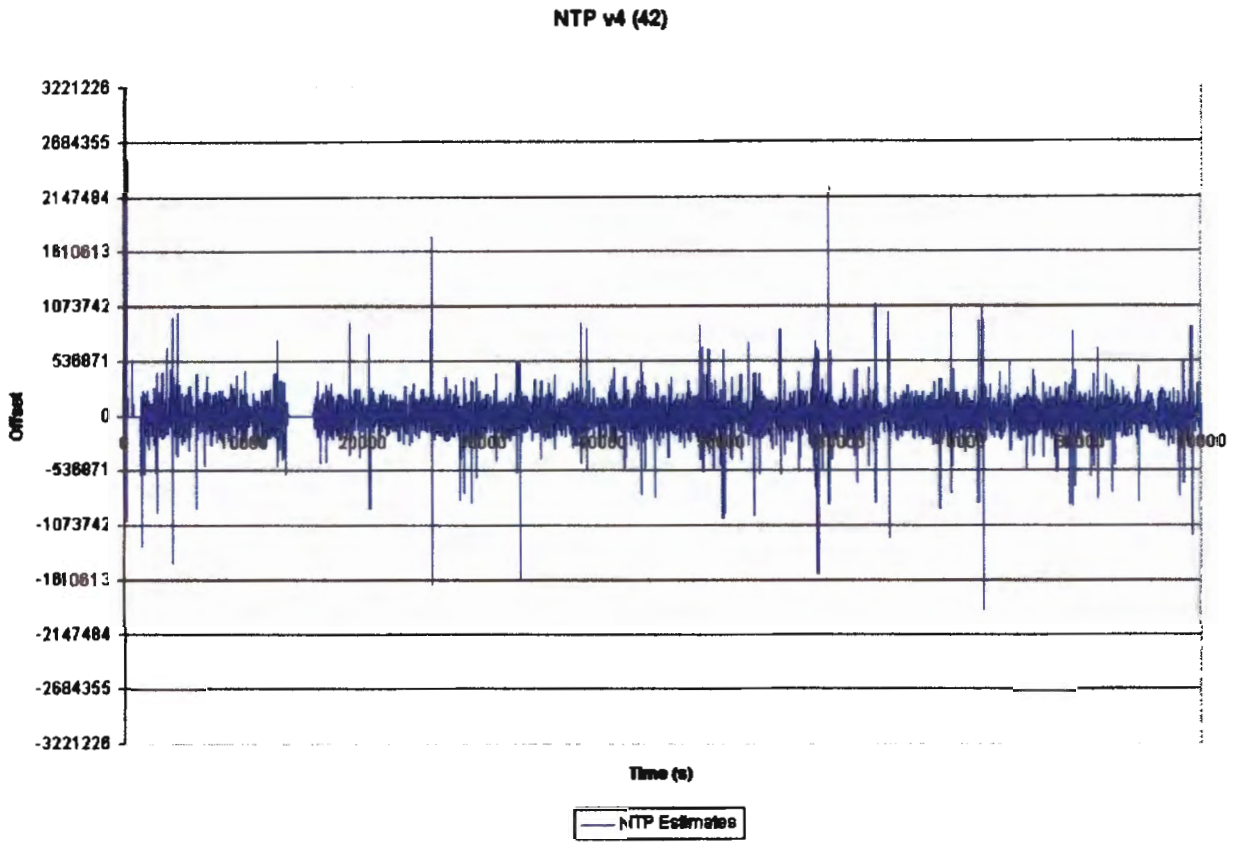


Figure 27: NTP Estimated Offsets (42)

7.6.4. Discussion

From the graphs in this paragraph a clear pattern of the impact of NTP v4 is evident. All the tests thus far using NTP v4 code have exhibited great accuracy with relatively few erratics. Should this test have reinforced the figures shown in 7.5.4, the viability of doing additional processing in the NTS layer, other than supplying additional information regarding SNMP and the sys_peers, would have been questionable.

Item	Out of Bounds (> 125µs or 0)	Value
1.	Measured Offsets	182
2.	NTP Estimates	835
3.	Unsafe (not currently detectable)	26
4.	Mean Offset	84µs

Table 15: Summary (42)

The inherent reliability of test 42 at 100% availability is 98.476%. By supplying the sys_peer criteria to the information in Table 15 reliability is 99.766% while availability drops to 93.008%.

7.7. Time Series 43 NTP v4

7.7.1. Test Description

Item	Description	Value
1	Trial Length	19 hours 27 minutes
2	NTP	V4 (G)
3	Start Time	Tue Jun 2 17h23
4.	# Readings	9007

Table 16: Test Description (43)

7.7.2. Measured Offsets

The measured offsets in Figure 28 and the NTP estimates in Figure 29 show a series of discontinuities before acquisition has taken place. This leads to a greater number of measurements outside of the range of 125µs.

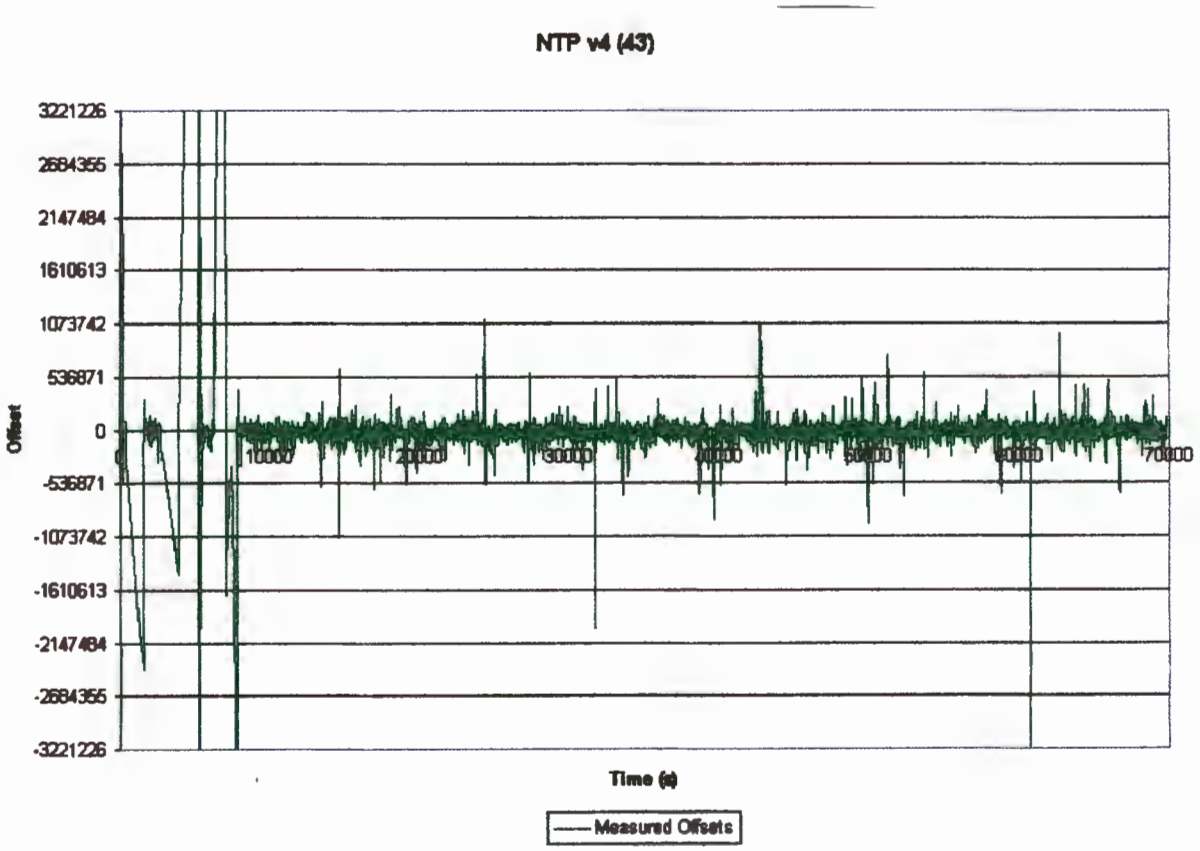


Figure 28: Measured Offsets (43)

The steady drift of the client clock from the server can be seen from 0 to 5000 seconds from the start of the test. This test took place while the NTP services were interrupted. The reason for this interruption was to test the influence of traffic instability on reliability.

7.7.3. NTP Estimated Offsets

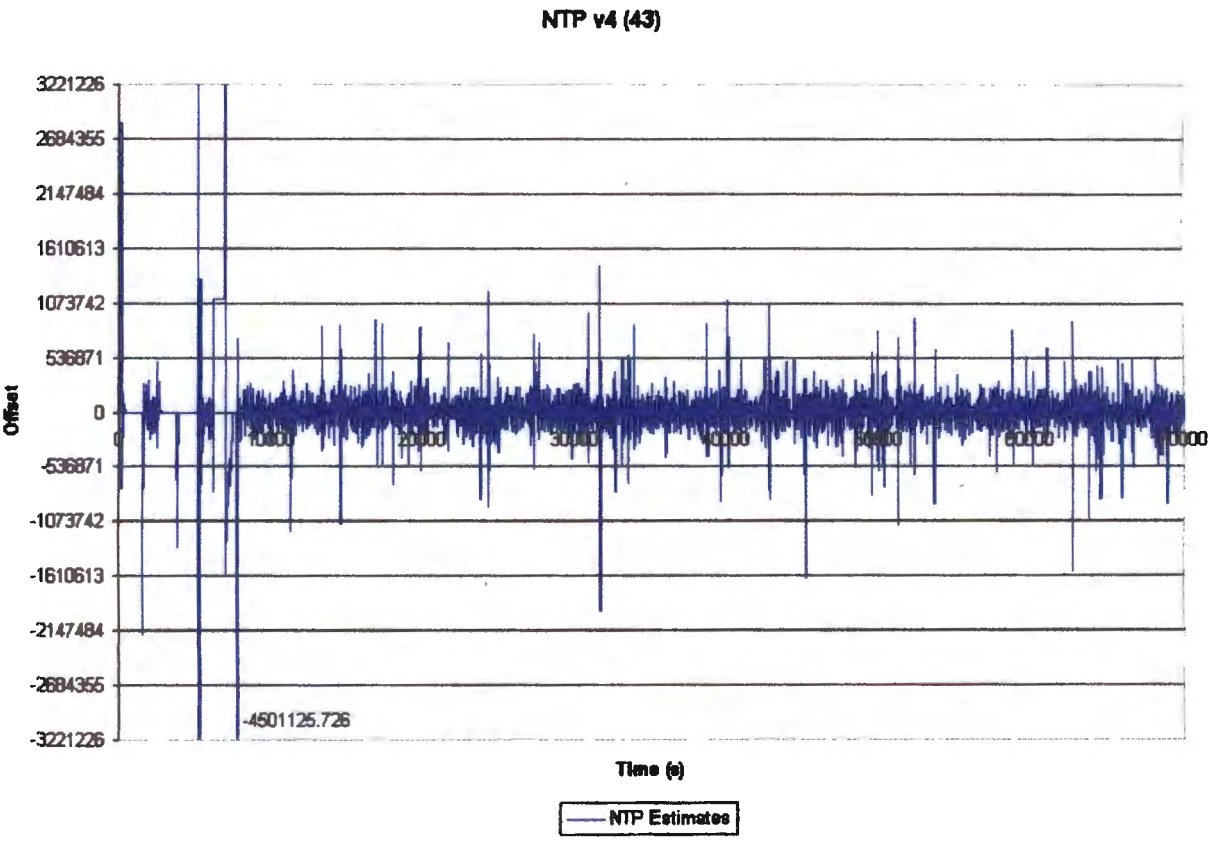


Figure 29: NTP Estimated Offsets (43)

7.7.4. Discussion

The influence of the instability at start time of this test is shown in the summary in Table 17. The out of range measured offsets(1) increased dramatically comparison with the earlier tests. As a result of this the unsafe readings(3) increased by more than two and a half times that of test 39 which operated over a longer time interval.

Item	Out of Bounds (> 125μs or 0)	Value
1.	Measured Offsets	683
2.	NTP Estimates	986
3.	Unsafe (not currently detectable)	46
4.	Mean Offset	-114μs

Table 17: Summary (43)

The inherent reliability is 92.417% thus almost one in every ten timestamps would be in error, applying the sys_peer criteria this is raised to 99.426% with 89.05% availability.

From tests 42 and 43 the results displayed in test 39 can be seen as an ideal case.

This is borne out by the results of the final test, test 44.

7.8. Time Series 44 NTP v4

This test displayed the same characteristics of the previous tests. It was run to achieve the 100 plus hours of information needed before the manipulation of the NTP Estimates could be done to check for a programmable solution in order to increase the availability and reliability of the implementation.

7.8.1. Test Description

Item	Description	Value
1	Trial Length	16 hours 40 minutes
2	NTP	V4 (G)
3	Start Time	Wed 3 Jun 13h43
4.	# Readings	9914

Table 18: Test Description (44)

7.8.2. Measured Offsets

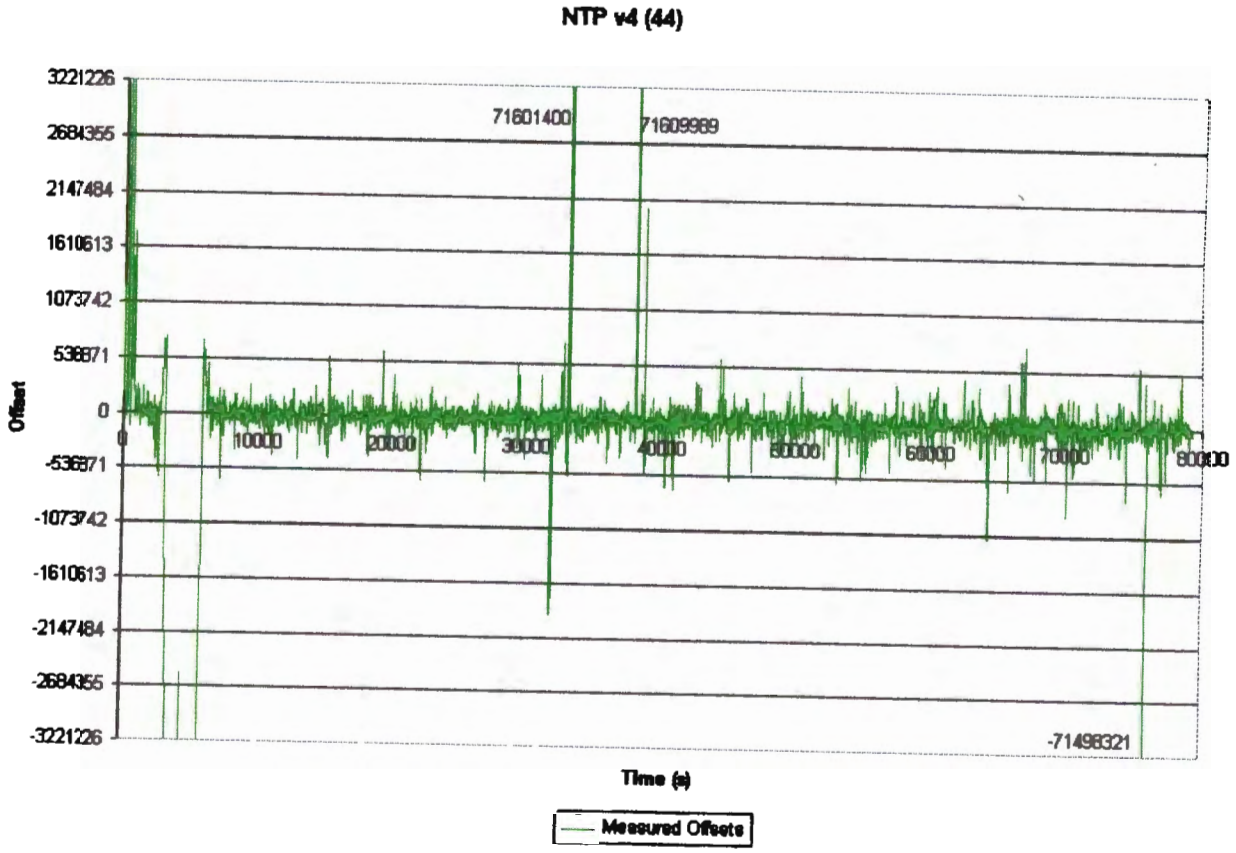


Figure 30: NTP Measured Offsets (44)

7.8.3. NTP Estimated Offsets

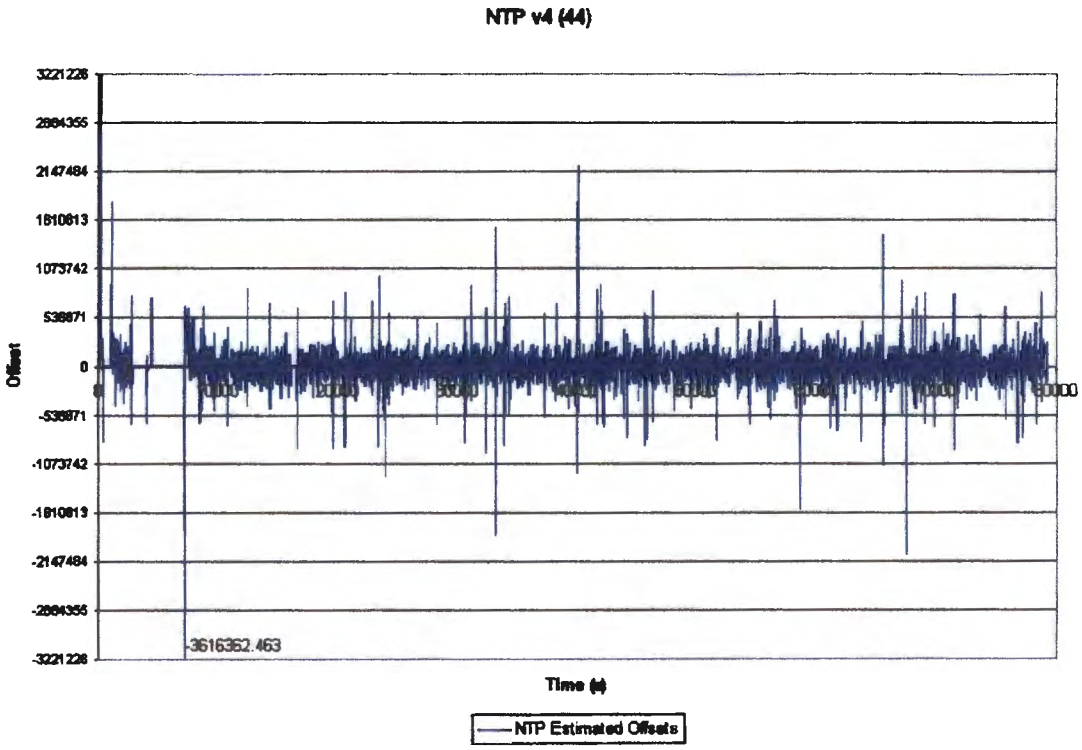


Figure 31: NTP Estimates (44)

7.8.4. Discussion

The results of this test fell within the bounds of the previous tests, being better than test 43 and worse than test 39. The results of this test are not significant on their own but serve to increase the number of readings taken and therefore to increase the statistical accuracy of the extrapolation to be made in Paragraph 7.9.

Item	Out of Bounds (> 125µs or 0)	Value
1.	Measured Offsets	478
2.	NTP Estimates	1051
3.	Unsafe (not currently detectable)	83
4.	Mean Offset	-38µs

Table 19: Summary (44)

The performance of this test was in line with the other tests as the inherent reliability was 95.178% which increased to 99.064% at 89.399% availability.

Having gathered the data needed, the overall performance can be checked and the possibility of improvement investigated.

7.9. Overall Performance

This paragraph presents a summary of the combined data and a method is proposed to increase availability with minimum impact on the reliability of the time stamps.

As has been shown in the previous tests, consistent offsets of better than 125µs have been obtained between the master and slave configurations of the nodes. Although the goal of 100% reliability and availability was not met, the average offset over all the tests, including acquisition times, is -31µs. Although this figure appears much better than the target figure, the graphs displayed indicate that this figure may be misleading, when viewed in isolation.

Item	Out of Bounds (> 125µs)	%	Value
1.	Measured Offsets	3.460	1564
2.	NTP Estimates	7.920	3580
3.	Unsafe (not currently detectable)	0.480	200

Table 20: Out of Bounds Summary

7.9.1. Inherent Performance

NTP operating over the implemented architecture yields the inherent performance. The availability is taken as 100 %, and the number of measured offsets that fell outside of the desired range give the reliability at this level. This leads to a reliability of 96.54% across the 45 197 readings taken. This figure varied from 89.050% in the worst test, to 99.817% in the best test – a range of over 10%.

As was anticipated in the design phase the inherent reliability failed to be bounded during testing. The ability of NTS to manipulate the timestamp with access to the peer structure of NTP will thus need to be used. As shown in the tests a simple set of validity criteria were used to increase the reliability while degrading the availability.

7.9.2. Manipulated Performance

A balance must be achieved between the important notions of availability of a timestamp within the target range, and the reliability of the timestamp gained. By using the NTP peer structure, the time stamp was flagged as invalid when the peer structure indicated the NTP estimated offset was out of range or unsynchronized to the master. Table 20 shows that the overall availability is

92.2% while the reliability is increased to 99.5% using the information of the NTP peer structure offsets.

These figures appear impressive, however they fall short of the 100 % availability and more importantly the 100% reliability desired. The use of the new NTP v4 contributed to a decrease in the acquisition time between the server and clients, while the acquisition is quick the range of offsets varies more greatly than in the NTP v3 instances. This can be seen when comparing Figure 20 and Figure 22.

7.9.3. Threshold Manipulation

Should it be wished to further increase the reliability, a threshold value better than the desired accuracy can be used. In Table 21 this concept is demonstrated, a threshold value of 1 is the same as the target offset desired – yielding the result described in the previous paragraph. A factor of 2 is the target offset divided by 2 and so on.

Should the NTP estimate be outside for this range the decision is taken to flag the timestamp as valid or invalid. This has the effect of using only those time stamps that originated from more conservative offset estimates.

Threshold	Unsafe	Available	Reliable
1	200	92.294%	99.559%
1.5	163	90.092%	99.632%
2	135	87.100%	99.685%
3	114	76.635%	99.700%
4	108	66.978%	99.678%

Table 21: Threshold Manipulation

It can be seen that 'thresholding' places a high cost on the availability, as depicted in Figure 32. For a 0.1% improvement in reliability a loss amounting to 5% in availability is suffered. Thresholding improves reliability by reducing the errors

undetectable in real time (as shown in column 2) by using conservative estimates when regarding a timestamp as valid.

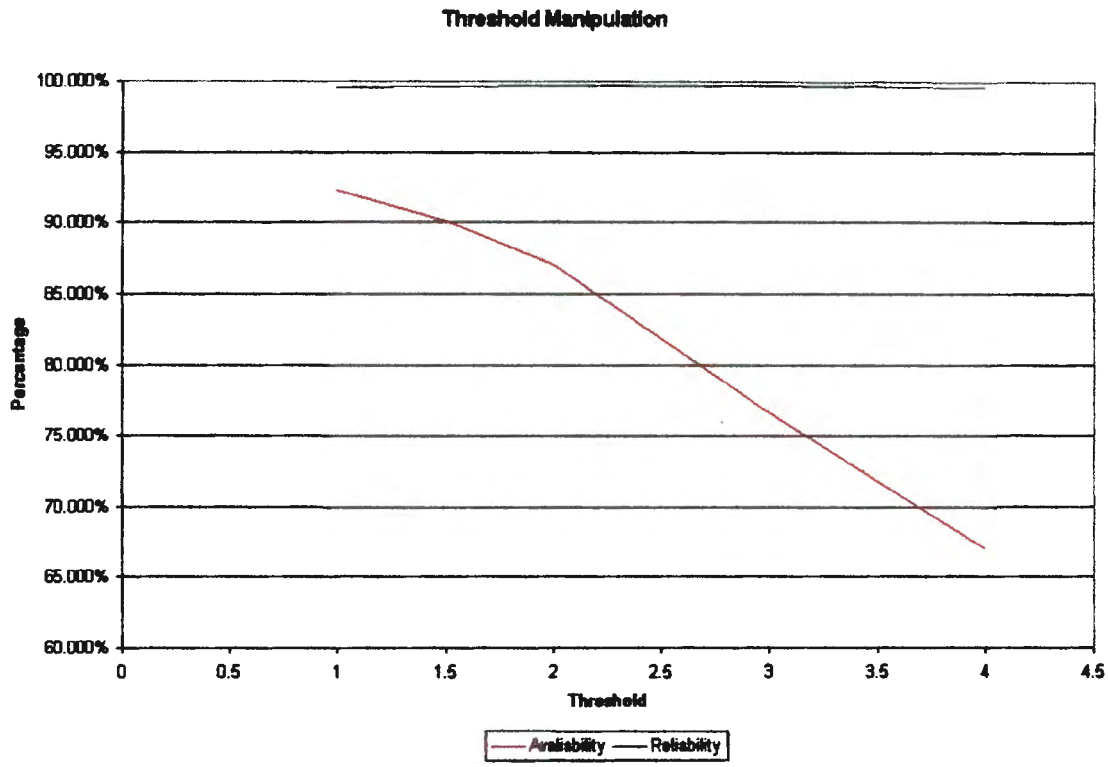


Figure 32 : Threshold Manipulation

7.9.3.1. Threshold Implementation

In the NTS software running on the IMS NIC a small section of code will need to be added. Before a timestamp is returned to the user the validation flag will be marked. The threshold value is compared to the `sys_peer->offset`, should the offset exceed the desired threshold then the timestamp is marked as unreliable.

7.9.3.2. Proposed Additions in Pseudo Code

The `ntsd` program will need the following code added at the point before it returns the time stamp to the user program.

```
// check
if (sys_peer->offset > threshold)
        status = OUT_OF_RANGE;
```

The variable threshold is a floating point number and should be set via the use of SNMP, thus this will need to be added to the NTS MIBII interface in order to tune the performance during run time. The status variable OUT_OF_RANGE will need to be defined so the user may recognize the message condition.

7.9.4. Moving Average Manipulation

Another method, seen in Table 22, which uses the moving average (MA) of the NTP estimates, proved better for gaining availability at minimal cost to the overall reliability.

MA	Unsafe	Available	Reliable
8	369	99.243%	99.154%
4	302	99.194%	99.305%
3	262	99.134%	99.394%
2	261	99.139%	99.397%
1	200	92.294%	99.559%

Table 22: Moving Average Manipulation

The first column gives the number of previous polls used when calculating the moving average. This is followed by the number of unsafe readings, which increase as more historic polls are included.

The overall performance of the implementation could be improved should it be possible to allow a drop in reliability by only 0.162%. This would allow availability to increase by 6.845% as can be seen in Figure 33.

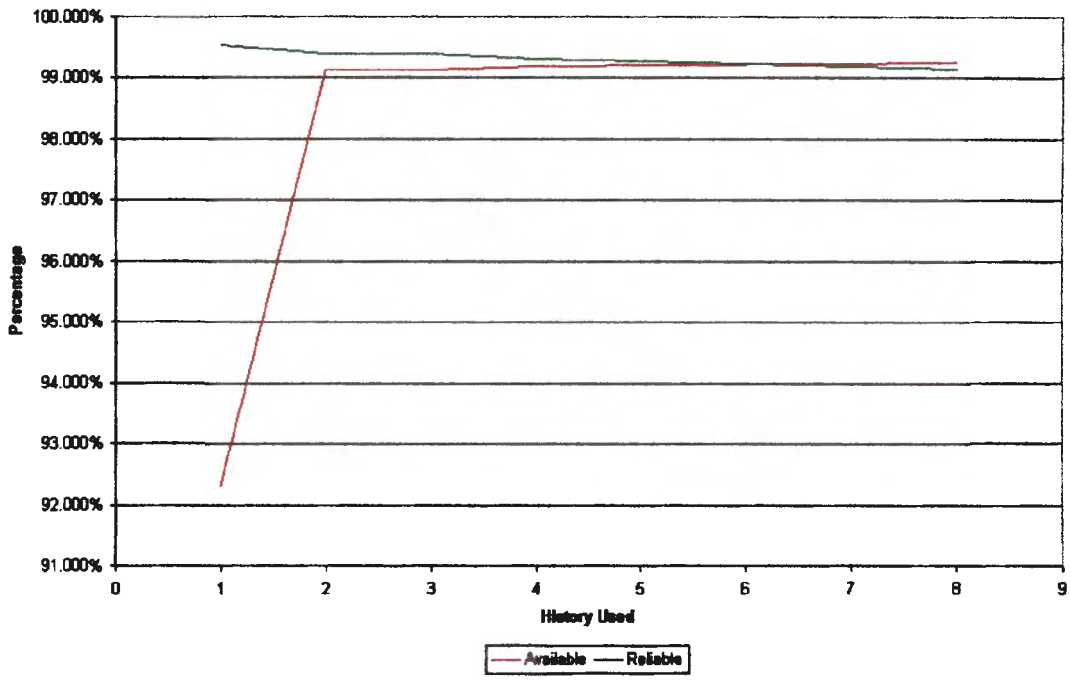


Figure 33: Moving Average Manipulation

7.9.4.1. Moving Average Implementation

As before, the NTS code on the IMS NIC will need altering. An array of values for the historical offsets will be created. This array will be used as a loop

to contain the most recent offset data needed. These timestamps will then be averaged and this average will then determine the status of the timestamp. The array will then collect the `sys_peer->offset` at a determined time interval, keeping the number of offsets required.

7.9.4.2. Proposed Additions in Pseudo Code

The following task will need to be run on the IMS NIC, to maintain the offset array:

```
// Collect data
float osarray[8];
void create_osarray(int interval, int history kept)
    {
    FOREVER {
            osarray[i++ mod history_kept] = sys_peer-> offset;
            sleep (interval);
            }
    }
```

The task will take two parameters, which, as before in Paragraph 7.9.3.2, should be reset-able from the NTS MIB for tuning during run time. The parameters are the 'interval' between gathering offset information and the number of offsets to use – the 'history_kept'. This program then runs continuously updating the history.

In the `ntsd` program the following code will be added before return of the time stamp to the user program.

```
// Generate offset to be used
offset = 0;
for (i=0, i < history_kept, i++)
       {
```

```
        offset += osarray[i];
        }
offset /= history_kept;

// check
if (offset > desired_accuracy)
         status = OUT_OF_RANGE;
```

7.9.5. Impact of the SBA

It is important to note that at this stage of tests the SBA was not activated as the communication channel problem between XTP and FDDI, using the user defined fields in XTP, had not been solved. The SBA will serve to remove the remaining jitter that is present in the system by helping to remove the remaining 'erratics' from the system.

The jitter can be ascribed to the lack of network carrier availability when NTP has 'sent' the time message. This leads to the message being delayed while waiting for its turn with the token in the FDDI ring. At present, the NTP message has to compete with other network traffic being sent from the IMS NIC. The SBA will serve to guarantee the availability of the required bandwidth to send the messages. This will provide a uniform send and receive, freeing the NTP from competition from other services on the IMS NIC.

Chapter 8

8. CONCLUSIONS

8.1. Introduction

Using a modular system integration model, network time synchronization can be brought down to a level that is applicable for distributed real time use, with reliability and availability of greater than 99% in the reported readings. Such a time synchronization system has been built to synchronize clocks on a network to within 125µs. The problem with clock synchronization can be greatly enhanced by adding determinism to the network, while maintaining component integrity of the different protocols.

The modular nature of the protocols used allows for future proofing, as any element can be replaced by its successor with minimum impact on the layer above or below. The use of accepted industry standards allows for ease of implementation and third party support. Testing by third parties to further develop and optimize these findings can be undertaken, as the components are available to industry.

Should the SBA fail to guarantee the reliability of the timestamps, use can be made of a pulse per second to synchronize the time for the fractions of the second, while the normal NTP functionality will ensure second coordination. Given the rate at which new code versions have been added to the stack, the possibility of achieving a 100% reliable solution should be attainable in the future.

The efforts taken to develop the stack so that it is, to a large degree, future proofed, will further ensure that it can utilise any progress made in it's individual components. As has already been shown in the development cycle of the software, this will also add to the increased reliability over time.

8.2. Observations

Code porting is not simple, NTP alone was over 56 000 lines of C code with much accompanying documentation. The FDDI driver was much the same in length yet had minimal accompanying documentation. Although porting is quicker than re-implementation, the time required to effectively understand both the accompanying documentation and code should not be underestimated. Adequate time should be dedicated to studying the software to be ported.

It is better to have the software source for all components intended for use, especially in areas around the critical sections. For this thesis, the critical region in the software was the clock routines that needed fine adjustment. The problems found with the LynxOs NTP prototyping (as described in Paragraph 5.4.1.2) serve to illustrate this.

8.3. Industry cooperation

All the industrial partners listed in the 'acknowledgements' section of this thesis contributed of their time and expertise. In particular, the Usenet community (particularly that of the NTP [NTN98] and VxWorks[VXN98] newsgroup members) were open-minded and extremely helpful. Numerous messages sent to and from these newsgroups, contributed to the author's understanding of the subject [CRE98] and the overall building of the system.

Later, when the NTP software additions were complete and were included in the IETG standard distribution of NTP [NTP98] the groups provided discussion on

the software and helped in refining the implementation. Messages of appreciation and requests for assistance from others wanting to use NTP have given the author a feeling of 'giving back' to the peer community that he drew upon when he was embarking on the development of the system. The author has gained a sense of satisfaction from contributing a vital part to a project that is now being used by major commercial and governmental agencies in the pursuit of technological advances in time synchronization.

8.4. Further Work Required

The SBA issues need to be resolved in order to make use of the benefits that it could provide (as mentioned in Paragraph 7.9.5). Further research needs to be undertaken in order to find ways to guarantee 100% reliability. Hard real time, time synchronization will not be able to make use of time distribution should this problem not be solved. This will mean that those application systems that can tolerate no error will have to rely on hardware solutions and expensive timing networks in the ever increasing geographic dispersion of cooperating machines.

BIBLIOGRAPHY

[AHG95] _____, *SCO OpenServer: SCO Advanced Hardware Development Kit Release 1.0*, Santa Cruz Operation, 1995

[CCI96] C²I² Systems Document No. CCII/A500/IMS/6-ICD/3, *Interface Control Document for the Information Management System Network Time Services.* Issue 1.0

[CCI96a] C²I² Systems Document No. CCII/A500/IMS/6-PIDS, *Prime Item Development Specification for the Patrol Corvette Combat Suite Information Management System.* Issue 1.3

[CCI96b] C²I² Systems Document No. CCII/A500/IMS/6-SRS/3, *Software Requirement Specification for the Information Management System Network Time Services* Issue 0.1

[CCI96c] C²I² Systems Document No. CCII/A500/IMS/6-ICD/7, *Interface Requirements Specification for the Information Management System NTS - SNMP Interface* Issue 0.1

[CCI96d] C²I² Systems Document No. CCII/A500/IMS/6-STD, *IMS Software Test Description* Issue 0.1

[CHA93] Chaudhuri, S. Gawlick, R. and Lynch,N. *Designing algorithms for distributed systems with partially synchronized clocks.* In Proceedings of the 12th Annual ACM Symposium on the Principles of Distributed Computing, pages 121--132, Ithaca, New York, USA, August 1993.

[CRI89] Cristian, F. *Probabilistic clock synchronization.* In Distributed Computing 3, Springer Verlag, 1989, 146-158.

[CRI94] Cristian F. and Fetzer C., *Probabilistic Internal Clock Synchronization*, 13th IEEE Symposium on Reliable Distributed Systems, Dana Point, CA, October 1994

[DEC89] _____, *Digital Time Service Functional Specification Version T.1.0.5.* Digital Equipment Corporation, 1989.

[DLP91] _____, *Data Link Provider Interface Revision 2.0.0*, Unix International, OSI Workgroup, 1991

[DOL97] Dolev, D.and Shavit,N. *Bounded time stamping.* SIAM Journal on Computing, 26(2):418-455, April 1997

[DON96] O'Donoghue, K.R. and Marlow, D.T., *Time Synchronization Services Aboard Surface Ships.* In Naval Engineers Journal, Nov 1996, 73-84.

[FDD97]van der Walt, S. *PMC FDDI Specification,* http://www.ccii.co.za/spec/cs_pm_fddi_spec2.htm, 1997

[GRO94] Grover, W.D. *A New Method for Clock Distribution,* IEEE Trans. Circuits & Systems, I: Fund. Theory & App., Feb. 1994, vol. 41, no. 2, pp. 149-160.

[IMS97]van der Walt, S. *Information Management System Overview,* http://www.netafrica.co.za/ccii/products/ims_specification.htm, 1997

[KOP87] Kopetz, H., and Ochsenreiter. W. *Clock synchronization in distributed real-time systems.* IEEE Trans. Computers C-36, 8 (August 1987), 933-939

[LIN80] Lindsay, W.C., and Kantak, A.V. *Network Synchronization of Random Signals,* IEEE Trans. Communications COM-28 , Aug 1980.

[MEN98a] _____, *Mentat XTP Volume 1 Programmer's Manual,* Mentat Inc. 1998

[MIL92] Mills, David L. *Network Time Protocol,* RFC 1305, Mar 1992.

[MIL94] Mills, David L. & Thyagarajan Ajit. *Network Time Protocol Version 4 Proposed Changes.* Electrical Engineering Dept. Technical report 94-10-2, Oct 1994

[MIL95] Mills, David L. *Improved Algorithms for Synchronizing Computer Network clock.* In IEEE Trans Networks, Jun 1995.

[MIL96] Mills, David L. *Time and Time Interval measurement with Application to Computer* and *Network Performance Evaluation.* Electrical Engineering Dept. Technical Memorandum, Jan 1996

[RAN91] Rangarajan S. and Tripathi, S.K. *Efficient Synchronization of Clocks in a Distributed System'.* In Proceedings of the Real-Time Systems Symposium, pages 22-31, December 1991.

[SAF94] MIL-STD-2204A, *Survivable Adaptable fiber Optic Embedded Network,* 1994

[DOL97] Dolev, D.and Shavit,N. *Bounded time stamping.* SIAM Journal on Computing, 26(2):418-455, April 1997

[DON96] O'Donoghue, K.R. and Marlow, D.T., *Time Synchronization Services Aboard Surface Ships.* In Naval Engineers Journal, Nov 1996, 73-84.

[FDD97]van der Walt, S. *PMC FDDI Specification,* http://www.ccii.co.za/spec/cs_pm_fddi_spec2.htm, 1997

[GRO94] Grover, W.D. *A New Method for Clock Distribution,* IEEE Trans. Circuits & Systems, I: Fund. Theory & App., Feb. 1994, vol. 41, no. 2, pp. 149-160.

[IMS97]van der Walt, S. *Information Management System Overview,* http://www.netafrica.co.za/ccii/products/ims_specification.htm, 1997

[KOP87] Kopetz, H., and Ochsenreiter. W. *Clock synchronization in distributed real-time systems.* IEEE Trans. Computers C-36, 8 (August 1987), 933-939

[LIN80] Lindsay, W.C., and Kantak, A.V. *Network Synchronization of Random Signals,* IEEE Trans. Communications COM-28 , Aug 1980.

[MEN98a] _____, *Mentat XTP Volume 1 Programmer's Manual,* Mentat Inc. 1998

[MIL92] Mills, David L. *Network Time Protocol,* RFC 1305, Mar 1992.

[MIL94] Mills, David L. & Thyagarajan Ajit. *Network Time Protocol Version 4 Proposed Changes.* Electrical Engineering Dept. Technical report 94-10-2, Oct 1994

[MIL95] Mills, David L. *Improved Algorithms for Synchronizing Computer Network clock.* In IEEE Trans Networks, Jun 1995.

[MIL96] Mills, David L. *Time and Time Interval measurement with Application to Computer* and *Network Performance Evaluation.* Electrical Engineering Dept. Technical Memorandum, Jan 1996

[RAN91] Rangarajan S. and Tripathi, S.K. *Efficient Synchronization of Clocks in a Distributed System'.* In Proceedings of the Real-Time Systems Symposium, pages 22-31, December 1991.

[SAF94] MIL-STD-2204A, *Survivable Adaptable fiber Optic Embedded Network,* 1994

[SMI81] Smith, T.B. *Fault Tolerant Clocking System* in Proc Fault Tolerant Computing Symp., 1981, p262-264

[SRI87] T. K. Srikanth and Sam Toueg. *Optimal Clock Synchronization.* Journal of the ACM, pages 626-645, July 1987

[STA96] Stark, G.J. *Stream Handling in Multimedia Communication Systems* Christ's College Doctor of Philosophy Thesis, University of Cambridge, April 1996

[STE90] Stevens, W.R. *Unix Network Programming*, Prentice Hall, 1990

[STR92] _____, *Programmer Guide: Streams*, Unix systems Laboratories Inc. Prentice Hall, 1992

[STR92a] _____, *Streams Modules and Drivers*, Unix systems Laboratories Inc. Prentice Hall, 1992

[SHI94] Shin, K.G. & Ramanathan, P. *Real-Time Computing: A New Discipline of Computer Science and Engineering.* In IEEE Proceedings., Jan 1994

[SNM93] Case, J. *Introduction to Version 2 of the Internet-standard Network Management Framework*, SNMP Research Inc., 1993

[SUR94] *Suri*, N. Hugue, M.M. & Walter J.W. *Synchronization Issues in Real-Time Systems.* . In IEEE Proceedings., Jan 1994

[VXW97]_____, *VxWorks Reference Manual 5.3.1*, Ed. 1, Wind River Systems Inc., 1997

[WIL90] Wilcox, D.R. *Backplane bus distributed realtime clock synchronization.* Technical Report 1400, Naval Ocean Systems Center, December 1990.

[WIL91] Wilcox, D.R. *Local area network distributed realtime clock synchronization.* Technical Report 1466, Naval Ocean Systems Center, November 1991.

[XTP95] XTP Forum, *Xpress Transport (XTP) Definition.* 1995

[YOU92] *Young*, R.M. *Real-Time Distributed System Architecture.* University of Cape Town Masters Thesis, 1992

[YOU96] Young, R.M. *Real-Time Protocol strategies for Mission Critical Distributed System Architecture.* University of Witwatersrand. Doctoral Thesis, 1996

WORLD WIDE WEB REFERENCED SITES

[ALC98] _____, Alcatel Website, http://www.alcatel.com 1998

[CCI98] _____, *CCII Systems Website* , http://www.ccii.co.za 1998

[CRC98] _____, *Communications Research Centre Website* http://www.crc.ca 1998

[CRE98] Crellin K.T. *Contributions to Peers on Usenet* http://x1.dejanews.com/profile.xp?author=Casey%20Crellin&ST=PS 1996-1998

[FSF98] *Free Software Foundation* http://www.fsf.org 1998

[HAC98] _____, *Hughes Aircraft Corporation*, http://www.hughes.com 1998

[JPL98] _____, *NASA Jet propulsion Laboratories Website* http://www.jpl.nasa.gov 1998

[LMC98] _____, *Lockheed Martin Corporation Website* , http://www.lmco.com 1998

[LYN98], *LynxOs Website* , http://www.lynx.com 1998

[LIN98], *Linux Website* , http://www.linux.org 1998

[MBA98], *Monterey Bay Aquatic Research Institute Website* , http://www.mbari.org 1998

[MEN98], *Mentat Inc Website* , http://www.mentat.com 1998

[MIL98] Mills, David L. *Prof. Mill's Website* http://www.eecis.udel.edu/~mills 1996-1998

[NTN98] *NTP Usenet Newsgroup* news:comp.protocols.time.ntp 1996-1998

[NTP98] *NTP Website* http://www.eecis.udel.edu/~ntp 1996-1998

[RAD98] *RadiSys Website* http://www.radisys.com 1996-1998

[SCO98] *SCO Website* http://www.sco.com 1996-1998

[SYS98] Schneider *and Koch Datensysteme GmbHWebsite* http://www.syskonnect.de 1998

[UNI98] *The Open Group Website* http://www.opengroup.org 1998

[VXN98] various *VxWorks Newsgroup* news:comp.os.vxworks 1996-98

[VXF98] various *VxWorks FTP Archives* ftp://ftp.ucar.edu/pub/vxworks 1996-98

[VXW98], VxWorks Operating System *Website* , http://www.wrs.com, 1998

[WRS98], *Wind River Systems Website* , http://www.wrs.com, 1998

APPENDIX A

PMS FDDI Daughter Card Short Form Specification

PMC FDDI Shortform Specification

Introduction

C²I² Systems's and SysKonnect's FDDI product portfolio includes a range of fibre-optic and copper-based FDDI network interface cards (NICs) for various PC bus architectures, as well as FDDI concentrators and switches.
C²I² Systems has developed a PMC FDDI NIC, which is ideally suited to embedded platforms. This NIC has been developed from SysKonnect's PCI FDDI NIC and as such features a wide range of compatible and qualified software drivers.

FDDI Features

FDDI meets the increasing demands of sophisticated networks :

- High data transmission rate (100 Mbps), efficiency and cost-effectiveness make FDDI the optimum solution for real-time and multimedia networks, as well as corporate backbones.
- Support of FDDI in departmental networks using copper media.
- Wide geographic range (100 km maximum circumference).
- Large number of supported nodes (up to 500 dual-attached) in the ring.
- Deterministic and bandwidth-efficient token passing access method.
- Fault-tolerant operation provides for ring reconfiguration in cases of problems such as media disruption or node failure.
- A high level of protection against tapping and malfunction when using fibre optic cabling.
- Optimum data protection and system availability by supporting SFT III server mirroring with FDDI fibre-optic MSL cards.
- Synchronous Bandwidth Allocator (SBA).

Seen from a physical point of view, an FDDI network consists of a dual-redundant fibre-optic ring. Only one of the two rings is used during normal operation. The second ring merely serves as a back-up medium. Stations in the ring are classed as Class A or Class B stations. Class A stations connect directly to the ring while Class B stations connect via a concentrator. A media interface connector is used to link FDDI stations to the transmission medium.
FDDI offers considerable bandwidth (100 Mbps) which is supported over both fibre-optic and copper media. Due to its efficient timed token protocol, realisable throughputs of up to 95 Mbps are possible. Owing to its high capacity, FDDI offers the capability to support both multimedia and SFT III solutions.

FDDI Synchronous Mode

The PCI and PMC FDDI NICs support both FDDI's asynchronous and synchronous transmission modes. Asynchronous mode is particularly suited for applications where response time behaviour and bandwidth requirements are not critical, e.g. data packet transmission in local area networks. Synchronous mode on the other had is suited for time-critical data such as networked multimedia and real-time control data. In the case of synchronous data transmission,

each station is allocated a specific proportion of the total bandwidth. Bandwidth is allocated to an end station (workstation) by a central station within the ring (e.g. fileserver) called the Synchronous Bandwidth Allocator (SBA). Allocation is done either dynamically at initiation of the specific application or statically by the network administrator.
PMC FDDI NICs allow mixed operation in both synchronous and asynchronous modes without any limitations. This is due to the fact that these NICs feature the unique AMD Supernet 3 FDDI chipset which fully supports synchronous mode.

Real-Time Support

The PMC FDDI NIC is ideally suited to distributed real-time and mission-critical applications. The board was designed with the requirement of the Survivable Adaptable Fibre Optic Embedded Network (SAFENET) standard as a guideline. Various device drivers and application software are also being developed, again with SAFENET as a guideline. Some of the most popular real-time operating systems such as VxWorks and LynxOS device drivers will be supported in addition to standard commercial network operating systems.

FDDI for Novell SFT III MSL

SysKonnect supports the Mirrored Server Link (MSL) connection of Novell's SFT III solutions with PMC FDDI NICs. The two servers to be mirrored use the MSL connection to synchronise their data. This feature offers an extremely high degree of data security and replicated fileserver operation.

Features of NetWare SFT III (MSL) using FDDI for PCI and PMC
• Server pooling: system availability can be enhanced by using a standby server in the FDDI ring
• Several server pairs may be supported on a single FDDI ring
• Existing FDDI cabling structure may be used for MSL connections
• Dual Homing
• Fibre-optic cabling protects against tapping and electromagnetic interference
• Wide geographic ranges can be covered
• Dual FDDI ring for redundant node links

FDDI for PMC

- 32-bit PCI for servers and stations
- High-performance PCI data transfer
- Two connector types available

SAS: SC or ST connectors
DAS: SC or ST connectors

- Optical Bypass Switch Control
- Up to 2 km between nodes
- Up to 500 dual-attached nodes
- Fully software configurable
- SMT Version 7.3

The PMC FDDI SAS fibre optic NIC uses an FDDI Class B fibre optic cable connection to integrate host systems with PMC Bus architecture via an FDDI concentrator into the 100 Mbps FDDI fibre optic network.
The PMC FDDI DAS NIC provides a Class A connection and integrates directly into the dual FDDI ring. NICs with ST connectors have the same optical characteristics as NICs with SC connectors.

Applications for FDDI on PCI and PMC

- Utilisation in high-performance PCI/PMC fileservers and workstations
- Digital image processing, multimedia applications, CAD, CIM, CAM, DTP
- SCADA applications
- Mobile network backbone applications
- Telephony applications
- Distributed real-time applications
- Mission-Critical applications
- SAFENET applications

Benefits

- High-capacity performance
- ANSI X3T12
- Utilisation of reliable fibre optic cabling technology
- Available as Single Attachment Station (Class B) and Dual Attachment Station (Class A) adapters
- Available with rugged ST or low-cost SC connectors
- Available in low-cost copper option (both SAS and DAS)
- Suitable for multimedia applications
- Current SMT version
- Two year warranty
- PMC FDDI provides cross-platform network support

PMC FDDI Adapter Specification

	SK-NET FDDI-ISA SK-51xx	SK-NET FDDI-EISA SK-53xx	SK-NET FDDI-MCA SK-52xx	SK-NET FDDI-PCI SK-55xx	PMC FDDI CCII/PMC/ FDDI/0
Bus interface	16-bit ISA-Bus	32-bit EISA-Bus	16/32-bit EISA-Bus	32-bit PCI-Bus	32-bit PCI-Bus electrical & PMC form factor
Network interface (fibre)	ANSI X3T9.5 and X3T12 compatible	ANSI X3T9.5 and X3T12 compatible	ANSI X3T9.5 and X3T12 compatible	ANSI X3T9.5 and X3T12 compatible	ANSI X3T9.5 and X3T12 compatible
Network Interface (Copper)	ANSI TP-PMD (MLT-3) Rev.2.1	ANSI TP-PMD (MLT 3) Rev.2.1	ANSI TP-PMD (MLT 3) Rev.2.1	ANSI TP-PMD (MLT 3) Rev.2.1	ANSI TP-PMD (MLT 3) Rev.2.1
LAN Controller	AMD Formac Plus	AMD Formac Plus	AMD Formac Plus	AMD Supernet 3	AMD Supernet 3
RAM	128 kBytes CMOS static	128 kBytes CMOS static	128 kBytes CMOS static	128 kBytes CMOS static	128 kBytes CMOS static
Flash EPROM	128 kBytes	128 kBytes	128 kBytes	128 kBytes	128 kBytes
I/O Addresses	8 switch selectable start addresses	Automatic selection of addresses	14 switch selectable start addresses	Automatic by PCI v2.1 Plug&Play assigned to the slot	Automatic by PCI v2.1 Plug&Play assigned to the slot
Interrupts	3,4,5,9,10,11, 12,15 switch selectable	5,9,10,11 software selectable	3,9,10,11 software selectable	PCI INT A	PCI INT A
DMA	3,5,6,7 switch selectable	0,5,6,7 software selectable	---	automatic depending on PCI slot	automatic depending on PCI slot
Arbitration level	---	---	8 values between 8h and Fh software selectable	---	---
Timer	3 channels @	3 channels @	3 channels @	3 channels @	3 channels @

	6,25 MHz max.	6,25 MHz max.	6,25MHz max.	6,25 MHz max.	6,25 MHz max.
Dimensions [mm]	107 x 339	127 x 275	292 x 88,3	126 x 275	150 x 74

FDDI Device Drivers

Driver	ISA	EISA	MCA	PCI	PMC
NetWare V3.1x and V4.x server	Y	Y	Y	Y	Y
Novell 4.0 HSM V1.0	Y	Y	Y	Y	Y
OS/2-NDIS LAN driver	N	Y	Y	N	N
OS/2-ODI LAN driver	Y	Y	Y	Y	Y
DOS-UPPS: Novell IPX and ODI,NDIS, PC-NFS, FTP Packet Driver, SNMP Daemon/Tools	Y	Y	Y	Y	Y
NDIS 3.0/NT-86	Y	Y	Y	Y	Y
AIX	N	N	Y	4.1	4.1
FDDI Synchronous Mode: • DOS-UPPS/ESS • OS/2-NDIS/SBA, ESS • NDIS 3.0/NT-86/SBA,ESS • UNIX (SCO, UnixWare, Interactive)/SBA, ESS • NetWare 3.1x, 4.x HSM/SBA,ESS	Y	Y	Y	Y	Y
Optional: SK-NET FDDI Unix Driver Kit (incl. SCO, UnixWare, Interactive Unix)	Y	Y	Y	Y	Y
VxWorks 5.3 (Tornado)	C	C	C	P	P
LynxOS	C	C	C	C	C
Optional: SK-NET FDDI IPX Remote Boot Kit (incl. Novell IPX Remote Boot and Documentation)	Y	Y	Y	P	P

Keys to table:
Y: Available
N: Not Available
P: In Preparation
C: Could be developed, depending on users
SBA=Synchronous Bandwidth Allocator
ESS= End Station Support

Contact Addresses
Physical:

C^2I^2 Systems (Pty) Ltd

Unit 3, 67 Rosmead Avenue

Kenilworth

Cape Town

South Africa

Postal:
C^2I^2 Systems (Pty) Ltd
P.O. Box 171
Rondebosch
Cape Town
7701
South Africa
Telephone: (+27 21) 683 5490
Facsimile: (+27 21) 683 5435
Email: info@ccii.co.za
URL: http://www.ccii.co.za/
Contact Persons:
Richard Young: rmyoung@ccii.co.za (email), (+27 82) 891 5868 (cell)
Gerhard Krüger: hgk@ccii.co.za (email), (+27 82) 892 4855 (cell)
Sean van der Walt: sean@ccii.co.za (email), (+27 82) 892 6755 (cell)

Prepared by : Sean van der Walt of C^2I^2 Systems (Pty) Ltd
Doc. Name : PMC FDDI Shortform Specification
Doc Number : CCII/G500/MARK/6 SPEC/2
Issue Date : 1997-02-17
Issue No. : 0.2
File Name : j:\admin\marktng\\specs\pmc\CS_PMC_FDDI_SPEC1.DOC
Print Date : 1997-04-22

APPENDIX B

IMS Short Form Specification

IMS SPECIFICATION

IMS Introduction

The C^2I^2 Systems Information Management System (IMS) communications architecture follows the dissemination architecture designed for distributed real time communications systems. The IMS offers latency control, message level priorities, multimedia support, high overall performance, bandwidth allocation, multicast capabilities, determinism and reliability.
The IMS is in full scale development phase for a ship board network, based on SAFENET, that will handle time -critical command and control messages, network timestamping, multimedia streams, background file transfer and data distribution from many sources to many destinations. The IMS architecture supports unicast, broadcast and reliable multicast features.
IMS is also gaining interest in mobile tactical real -time systems as well as mobile and semi -mobile air surveillance radar systems.

IMS Architecture

The C^2I^2 Systems Information Management System (IMS) communications architecture follows the dissemination architecture designed for real -time communications, refer to 'Figure 1: Dissemination Architecture' below. The IMS communications architecture supports many -to -many connections, this is best suited to distributed, time -critical information flow.

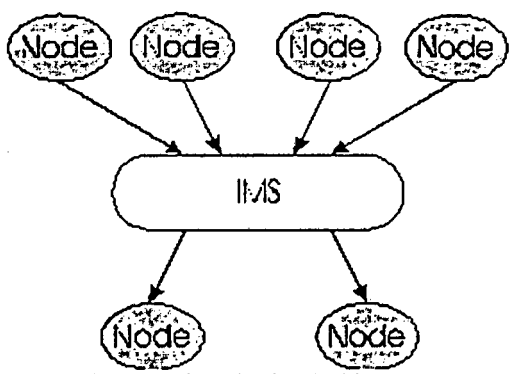


Figure 1: Dissemination Architecture

The IMS allows nodes on the network to produce and consume data in the pysical network and control of the network is distributed. IMS handles the actual data transfers between nodes on the network. Each node dynamically registers to the IMS and then becomes a producer and/or consumer. The IMS handles the

multi -casting of all data on the network, thereby allowing virtual links to be setup between the nodes on the network. This architecture is symmetric, robust to changes and failures, and very efficient.
The IMS communications architecture differs significantly from the older "point -to -point" (e.g. TCP) and "client -server" (e.g. RPC) architectures. These architectures suffer from complex connection and error recovery problems, and single points of failure.

IMS FDDI Interface

There are two types of transfer supported by FDDI, they are:

Asynchronous

Asynchronous data is produced non-repetitively by a producer and is synonymous with **event** data.

Synchronous

In Synchronous mode a producer is guaranteed a certain proportion of LAN bandwidth. Synchronous data has priority (at the FDDI level) over asynchronous data. Synchronous data is produced repetitively by a producer and is synonymous with **state** data.

IMS Protocols

Xpress Transport Protocol (XTP)

The following abstract is derived from the Xpress Transport Protocol 4.0.
The Xpress Transport Protocol (XTP) has been designed to support a variety of applications ranging from real -time embedded systems to multimedia distributed over a wide area network. In a single protocol it provides all the classic functionality of TCP, UDP and TP4 plus many new services such as transport multicast, multicast group management, transport layer priorities, traffic descriptors for quality -of -service negotiation, rate and burst control, and selectable error and flow control mechanisms. XTP has the same interconnectivity as TCP/UDP/TP4 because it operates over any network layer (IP, CLNP), any datalink layer (LLC, MAC), or directly on top of the AAL of ATM. In general, XTP avoids coupling policy with mechanism; XTP offers services but the user's application defines what communications paradigm is most appropriate for its particular environment. XTP is a high performance protocol, and can sustain high throughput (92 Mbits/s over FDDI between a pair of IBM RS/6000 model 370s) and low latency (350 µs to move 100 bytes from user memory to user memory on two 50 MHz PCs connected by FDDI). Since XTP can run in parallel with all other transport protocols, and can run over whatever network layer (if any) is provided, it represents a low -risk way to exploit

the increased functionality required for distributed applications without sacrificing connectedness or interoperability.

Network Time Protocol

The following abstract is derived from the Network Time Protocol (Version 3). NTP provides the protocol mechanisms to synchronise time in principle to precisions in the order of nanoseconds while preserving a non -ambiguous date well into the next century. The protocol includes provisions to specify the characteristics and estimate the error of the local clock and the time server to which it may be synchronised.

Network Interface Cards

The following NICs are supported by the IMS:

- CCTs CL486-DAS FDDI board
- C^2I^2 Systems PMC FDDI board
- User specified FDDI NICs

IMS Communications Design

The *Survivable Adaptable Fibre Optic Embedded Network* (SAFENET) model as defined for the US Navy, is adhered to by the IMS. SAFENET consists of the following different protocol families :

- OSI Protocol Family.
- Internet Protocol Family.

These families are further divided into protocol profiles. These profiles are :

- SAFENET LAN Profile.
- SAFENET Transport Profile.
- SAFENET Extended Profile.

The reason for the applicability of the SAFENET Model is that it is a practical, achievable implementation of the ISO OSI Model that is capable of real-time performance. It is practical because, while it redefines the 7-layer model somewhat, it leaves the major layer boundaries (i.e. the major interfaces) unchanged. It also provides the other layers required for a complete system. It is achievable because implementations (hardware and software) of all the layers exist, or are currently under full-scale engineering development.

IMS Communications Architecture

Conceptually the IMS consists of Application Interface Services (APIS), Network Time Services (NTS), File Transfer Services (FTS) and Built -In Test Services (BITS). Refer to paragraph 5.3 for a more detailed description of the IMS Application Interfaces.

The IMS Architecture consists of the shaded areas as indicated in the table below.

<table>
<tr><td>Layer No.</td><td>ISO OSI Layers</td><td colspan="6">IMS Protocol Suite</td></tr>
<tr><td>9</td><td>Application Process</td><td colspan="6">Application Jobs/Tasks/Processes/Threads</td></tr>
<tr><td>8</td><td>Operating System</td><td colspan="6">Real -Time Operating System</td></tr>
<tr><td>7</td><td>Application</td><td rowspan="3">Null</td><td colspan="2" rowspan="3">Application Interface Services (APIS)</td><td rowspan="3">Network Time Services API</td><td rowspan="3">File Transfer Services API</td><td rowspan="3">BITS API</td></tr>
<tr><td>6</td><td>Presentation</td></tr>
<tr><td>5</td><td>Session</td></tr>
<tr><td>4</td><td>Transport</td><td>TCP</td><td>ISO TP4</td><td>XTP</td><td>Network Time Protocol (NTP)</td><td>FTS</td><td rowspan="2"></td></tr>
<tr><td>3</td><td>Network</td><td>IP</td><td>CLNP</td><td>IP</td><td></td><td></td></tr>
<tr><td>2</td><td>Data Link</td><td colspan="6">IEEE Logical Link Control Class I Protocol (LLC)</td></tr>
<tr><td></td><td></td><td>SNMP</td><td colspan="2">ANSI FDDI SMT Protocol</td><td colspan="3">ANSI FDDI MAC Protocol</td></tr>
<tr><td>1</td><td>Physical</td><td colspan="6">ANSI FDDI Physical Protocol (100 Mbits^{-1})</td></tr>
<tr><td>0</td><td>Cable Plant</td><td colspan="6">62,5µ/125µ Multimode Fibre Cable Plant</td></tr>
</table>

The italics denote layers outside of the ISO OSI 7 Layers.

APIS provides the registration, produce, demand services as well as error reporting of the data flow on the network. NTS offers clock synchronisation services, user time services and time management services. FTS provides reliable, connection-oriented byte stream transport between applications executing on different host processors in the IMS. BITS allow the applications to communicate with the network management component of the IMS.

IMS Topology

The SAFENET physical topology consists of the following SAFENET-defined components:

1. Network Stations.
2. Cable Plant Connections.

3. Physical Medium Components.
4. Network Interface Cards.

Network Stations

Dual Attached Stations shall be implemented for the IMS.

Cable Plant Connections

The Cable Plant Connections for the IMS consists of a Dual Fibre Trunk and Trunk Coupling Units if required. The IMS will deviate from the SAFENET standard in the following ways :

1. Connectors may be used where required (e.g. bulkheads) in the Fibre Trunk.
2. The length between Trunk Coupling Units or NICs may be longer than 200 m if required,but not
   more than 2 000 m.
3. Connectors may be used where required (e.g. at Gland Plates).
4. Fibre Amplifiers used to guarantee optic signal integrity.

Physical Medium Components

The following Physical Medium Components are incorporated within the IMS :

1. Trunk Coupling Units (TCUs).
2. Trunk Cable.
3. Optical Interface Cable (i.e. Fibre Optics Patchchords).
4. Fibre Optic Interface Connectors (FOIC).
5. Electrical Interface Cable (Optical Bypass Switch Control).
6. Electrical Interface Connectors (Power).

Network Interface Cards

The purpose of the Network Interface Cards shall be to provide the physical interface between the Fibre Cable Plant and the nodes.
The FDDI standard intrinsically provides the physical and datalink protocols as well as Station Management (SMT). The physical protocol is supplied as the FDDI PHY protocols while the datalink protocol is supplied as the MAC (Media Access Control) protocol. At these levels, two modes of data transmission are defined, synchronous and asynchronous.

IMS Application Interfaces

APIS

Application Interface Services (APIS) allow the applications to communicate on the network without requiring any knowledge of the message sources or destinations. This allows for a degree of dataflow abstraction. The sources and destinations of messages are determined by the transport and underlying layers.
Following a cold or warm start of a sub-system processor host, a once -only APIS Initialisation command is issued to the NIC. This allows the NIC to clear its tables in respect of this host.
An application registers with the NIC utilising the APIS_OPEN() command. It shall also inform the NIC which messages it can source and which messages it requires to perform its functions with the APIS_PRODUCE() and APIS_DEMAND() commands respectively. The application does not specify the source or destination of any of these messages. The Application Interface Services will determine the destination and source addresses for the messages. To transmit a message to another application, the source application utilises the APIS_SEND_MSG() command.
An application can dynamically add and remove messages with the APIS_PRODUCE(), APIS_DEMAND(), APIS_REMOVE_PRODUCE() and APIS_REMOVE_DEMAND() commands.
When an application issues the APIS_CLOSE() command, all the messages transferred to and from it will be removed from the network.
A message on the LAN is identified by a unique message ID number. APIS interprets the Message ID as a numerical number consisting of four 16 bit fields. The hosts can allocate meaning to any field in the Message ID without effecting the operation of the APIS.
The only APIS requirement is that the Message ID uniquely identifies a message on the LAN.

NTS API

SAFENET Time Services are divided into three functional areas:

1. Clock Synchronisation Services.
2. User Time Services.
3. Time Management Services.

The clock synchronisation services provides for the co -ordination of the distributed clocks in the network.
The user time services provides for a user to obtain the current value of the clock and an indication of the accuracy and quality of the value.
The time management services provides for the definition of the managed objects used for initialisation, control and monitoring of the time services.

Each node shall provide a Network Clock to provide time values and to synchronise with all the other Network Clocks on the network. Time Format shall be as defined by SAFENET, i.e. a bit string of 64 bits representing a unsigned real number that expresses time in seconds and binary fractions of a second relative to 1st January, 1970. This representation will represent the time to a resolution of 232 ps.
The Clock Synchronisation Service synchronises all the Network Clocks on the network to the reference time, provides a service of selection of best clock, detection of failed clock, distribution of indication that a leap second is going to occur and basic reconfiguration of the time synchronisation when the clock acting as server has failed. The Clock Synchronisation Services shall be provided with the use of the Time Synchronisation Protocol.

FTS API

FTS provides reliable, connection-oriented byte stream transport between applications executing on different host processors in the IMS. Two host processors that share an FTS connection may be located either on the same backplane or in subsystems that are connected to the IMS through different NICs.
The FTS uses an addressing scheme where every user of FTS chooses a unique 64-bit name for the user's transport endpoint. The FTS transport addressing scheme is completely separate from the network addresses used by the IMS NICs. This is because the NICs use dynamically assigned network addresses, whereas it is envisioned that the communicating FTS entities will use global administered transport endpoint names.
The message interface to FTS provides a simplified OSI-like connection-oriented transport service between these transport endpoints. The data transfer over FTS connections is based on reliable byte streams, rather than sequenced message streams. After a connection has been established between two FTS transport endpoints, data transfer between the two endpoints is bi -directional.

BITS API

Built -in -Test Services (BITS) allow the applications to communicate with the network management component of the IMS. SNMP is supported for network management.

Contact Addresses

Physical:
C²I²Systems (Pty) Ltd
Unit 3, 67 Rosmead Avenue
Kenilworth
Cape Town

Postal:
C²I²Systems (Pty) Ltd
P.O. Box 171
Rondebosch
Cape Town

South Africa 7701
South Africa

Telephone: +27 21 683-5490
Facsimile: +27 21 683-5435
Email: info@ccii.co.za
URL: http://www.ccii.co.za/

Contact Persons:
Richard Young: rmyoung@ccii.co.za (email), +27-82-891-5868 (cell)
Gerhard Krüger: hgk@ccii.co.za (email), +27-82-892-4855 (cell)
Sean van der Walt: sean@ccii.co.za (email), +27-82-892-6755 (cell)

Prepared by: Sean van der Walt of C^2I^2 Systems (Pty) Ltd.
Doc. Name: IMS Specification
Doc Number: CCII/G500/MARK/6-SPEC/1
Issue Date: 1996-12-17
Issue No.: 0.1
File Name: G:\XFER\SVDW\CS_IMS_SPEC.DOC
Print Date: 1997-04-22

APPENDIX C

VxWorks Investigation Report (NTP Section)

(Excerpt from Report on CCII Investigation into Real Time Operating Systems)

Evaluation performed by Kenneth Crellin (ktc) September 1996

Note: This section assumes the reader has some familiarity with the GNU toolkit, VxWorks target/server connectivity, UNIX BSD system calls and POSIX real-time extensions.

Evaluating Tornado Tools

Windows Development Environment

Interface

By choosing an MDI interface the developers of Tornado for Windows have placed all their eggs in one basket, it is cohesive and works well if all is well, but once one component dies, or starts giving problems, the rest of the system suffers (including other programs running in windows95). A system similar to the Delphi interface would have given more flexibility. A case to demonstrate this is when you interrupt the debugger, and it tries to find out of context variables for the assembler code. This results in modal dialogues preventing you from shutting down the debugger, or simply restarting it. The only option is to kill the whole Tornado environment and individually kill the zombies (see the Reboot section below), or reboot. The separation of the tasks into their own windows or the use of non-modal dialogues would go a long way to solving this problem.

Help

Help is abysmal, the hyper text nature of the `winhelp.exe` system means that this could really be a great step up from the Unix platforms, however all you have is man pages put in the help file and most of the links on a particular page were to the same page! The search capabilities of the help are very poor.

Reboot

One of the major problems with the windows 95 development host is partly due to the windows system, or the bad programming of the applications. Once the VxWorks session has been exited there should be no tasks created by VxWorks left in memory. However the following are left lying around after a normal session with debugging and compiling. One has them still in memory (only the top five are legitimately running, the rest are zombies).

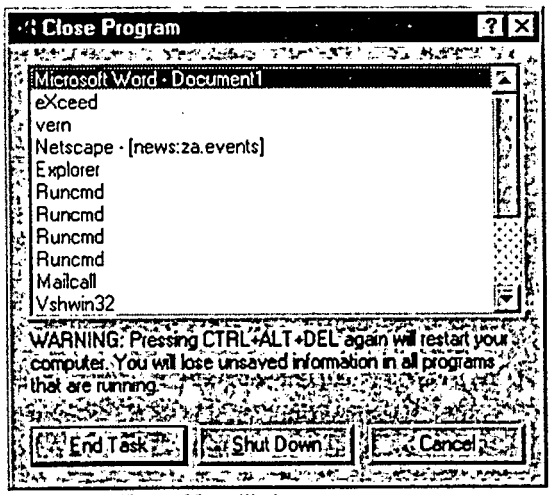


Figure: Win 95 Tasks executing ver 1

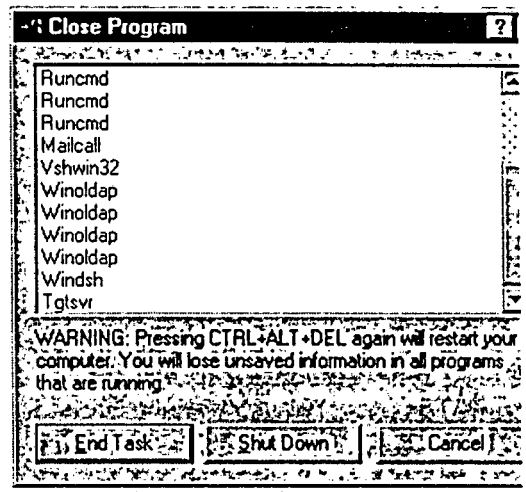


Figure: Win 95 Tasks executing ver 2

GNU Tools on Windows 95

cc386.exe (gcc compiler)

The compiler performed well and no unreasonable errors were encountered. The only problem encountered was the linking with the `ld386.exe` linker. See the example below. This was quickly corrected by calling the linker and letting it call the compiler.

```
# crt0.o is absent :: so LD  will call CC explicitly.
#$(PROGRAM): $(OBJS) $(LIB) version.o
# -del  $(PROGRAM)
# $(CC) $(COPTS) -o $@ $(OBJS) version.o $(LIB)
$(DAEMONLIBS) \
#  $(RESLIB) $(ADJLIB) $(COMPAT)


cc386 -W  -g -traditional -o ntpdate ntpdate.o
version.o .\xntpd\ntp_unixclock.o  ../lib3 .4/libntp.a
\

d:\Tornado\host\x86-win32\lib\gcc-lib\i386-wrs-
vxworks\2.6-95q2\ld.exe: cannot open crt0.o
: No such file or directory
cc386: Internal compiler error: program ld got fatal
signal 1
make.exe: *** [ntpdate] Error 0x1

Done.
```

The following code had to be substituted to fix the error of the **lacking** `crt0.o` **file.**

```
$(PROGRAM): $(OBJS) version.o $(LIB)
 -del  $(PROGRAM)
 $(LD) $(LDFLAGS) -o $@ $(OBJS) version.o $(LIB)
$(DAEMONLIBS) \
                $(RESLIB) $(ADJLIB) $(COMPAT)
```

gdbi86.exe (gdb debugger)

Supporting a remote debugger is not simple and the gdb port is quite impressive from this point of view. The information is as thorough as I have used in normal gdb, but the remote connectivity did give problems, as described in the example in the Interface section.

Owing to the traffic across the network debugging is slow. It also fails to find the program source on occasion and the breakpoint buttons operate erratically. When stepping through code it would jump to assembler rather that the next line in the same source, hopefully these problem will be rectified. When working it works well.

make.exe (make utility)

The make utility supported all the expected macro's. Changes in the make files were not needed except for the compiler, linker and other component names, and the change mentioned in the cc386.exe section.

A problem worth mentioning is the command line length of the DOS shell. The make is spawned to DOS and each individual command must have a command line length of less than 128 bytes (or 256 in 4dos shell). The problem can be demonstrated by doing a make clean in the pc486 directory and a make. It will fail with no input files because of the above problem.

Other gnu based tools used

`nm386.exe` (symbol lister for object files), `ld386.exe` (ld linker) and `ranlib386.exe` (library compiler). All these programs performed well, and no problems were experienced.

Gnu based tools not used

```
as386.exe,          cppfilt386.exe,          cpp386.exe,
objcopy386.exe,          objdump386.exe,     size386.exe,
strings386.exe, strip386.exe
```

Target Development Cycle

The major problem with the development cycle is the long time it takes from one debug session to another. The overhead of clearing the stack space, sockets and other assorted garbage left lying around after stopping a debug task is time consuming as this can only be done by rebooting.

This may be only true when porting code as most code assumes an independent memory space and once main is killed all children are already killed. Thus the clean up is done automatically, as all VxWorks tasks are linked into the kernel, this is not possible. The TCB (task control block) and stack space may be freed but the values are still there, and the program is given the same offset at which to start it picks up these variables.

Target Agent to Target Server Connectivity

Going hand in hand with the above issue is the problem that the `tgtsvr.exe` (the target server) dies during reboot and must be started again, on occasion the target server would die "spontaneously". Once the target server dies the debug session must be stopped and reinitiated as the link has been lost, thus losing stack pointer position and attachment.

Wind Foundation Classes

I made no use of these in the port of NTP.

Porting NTP

Source from vxworks archive

Originally the source code for an implementation of the NTP protocol was obtained from the VxWorks archive and it was thought the implementation and compilation of this would be trivial as the BSP should remove the hardware abstraction. In addition to this, the source supposedly ran on the 5.2 system and there have only been additions to the VxWorks system so this should also not have posed a problem. This proved not to be the case and the code had many problems. Most of the supporting code for the xntpd daemon that ran in 5.2 dated to the early 1990's and VxWorks 4.3. After much battling and coming across some very bad coding it was decided to take the standard distribution and proceed from there. The problems included blocked select calls, passing size of pointers instead of structs, lacking struct definitions.

The usrTime library from the VxWorks archive was used to emulate `gettimeofday()` and `settimeofday()` as the ansiTime library is lacking these. The construction of the ansiTime library caused problems as it was not possible to un-include it, WRS are aware of this problem and it is long standing. Undefining '#undef ANSI_TIME' in the source code did not un-include all the ansi time library calls as some of VxWorks 5.3 system calls code still had calls to the ansi time library. Similar problems exist with the SPR #: 5524, where un-defining '#undef INCLUDE_POSIX_ALL' does not remove all the POSIX calls. The calls that clashed were replaced in the usrTime library with an `ntp_` prefix, as I was loathe to remove the whole `gmtime.o` section of the `libI80486gnuvx.a`. The following changes were needed to get the usrTime library to compile:

```
#define Extern extern
#define Void void
#define Nat32 unsigned int
#define Nat16 unsigned short
#define Int32 long
#define Reg register
#define MLocal
```

Standard Distribution 3.4y

The `lib/, include/, ntpdate/, xntpd/` directories were taken. The present source is in the `Vxw_eval/Technicl/Source/Ntp/Vxntp.zip` archive on the network. The configuration of the Makefiles were changed by having the following included in the `ntp_machine.h` file:

```
#if defined(SYS_VXWORKS)
    #define RETSIGTYPE void
    #define NTP_POSIX_SOURCE
    #define _POSIX_SOURCE
```

```
    #define HAVE_NO_NICE

    /* some peculiarities that are too common for #if */
    #define fcntl ioctl /* uses only ioctl - _mostly_
compatible */
    #define getpid taskIdSelf
    #include <vxworks.h>

    #ifndef STR_SYSTEM
        #define STR_SYSTEM "VXWORKS/Tornado"
    #endif
#endif
```

Ntpdate

Minor changes were needed to the `ntpdate` which included changing include files and replacing the `fcntl()` calls with corresponding `ioctl()` calls.

The main function was replaced with an `ntpdate` function and the IP address to get the date from was coded into this function. The select call was changed to take a zero time value (it hung forever when given a NULL value). The default priority for the downloaded task was higher the netTask which handled all the UDP communications thus the application. Blocked the netTask and deprived it of processor time. After changing the priority the problem was solved. The timer() was rewritten to use the POSIX timer calls. The `ntpdate` worked well. The select system call worked as documented and passing a NULL time value will result in the application doing a blocking call, until a UDP packet arrives at one of the sockets (file descriptor).

Xntpd

Changes were made to the include files and the conventions

```
#ifdef SYS_VXWORKS
#ifndef SYS_VXWORKS
#if defined(SYS_VXWORKS)
#if !defined(SYS_VXWORKS)
```

were used throughout were changes were made except to

`#include <sys/time.h>` which was replaced with `#include <sys/times.h>` as it occurred so often. So far it has only been necessary to write a couple bits of code for missing functions. Files netdb.c, ntp_sleep.c and netdb.h were written, the code is as follows:

```
/* ntp_sleep.c by casey
 * vxworks thinks it has insomnia but
 * we have to sleep for number of seconds
 */

#ifdef SYS_VXWORKS

#define CLKRATE     sysClkRateGet()

/* I am not sure how valid the granularity is -
 * it is stolen from George Eger's port
*/
#define CLK_GRANULARITY  1 /* Granularity of system
clock in usec  */
                           /* Used to round down #
usecs/tick      */
                           /* On a VCOM-100, PIT gets
8 MHz clk,   */
                           /*  & it prescales by 32,
thus 4 usec   */
                           /* on mv167, granularity is
1usec anyway*/
                           /* To defeat rounding, set
to 1         */

#define USECS_PER_SEC    1000000L    /* Microseconds
per second    */
#define TICK
(((USECS_PER_SEC/CLKRATE)/CLK_GRANULARITY)*CLK_GRANULA
RITY)

/*
 * emulate unix sleep
 */
void sleep(int seconds)
 {
 taskDelay(seconds*TICK);
 }

/* emulate unix alarm
 * that pauses and calls SIGALRM after the seconds are
up...
* so ... taskDelay() fudged for seconds should amount
to the
* same thing.
 */
void alarm (int seconds)
 {
 sleep(seconds);
 }
```

```
#endif
 /* eof ntp_sleep.c */
```

The function `sleep()` was needed as there is a call for a delay when race clashes occur, `alarm()` was needed for a time delay before a alarm signal is sent to re-awaken the process, VxWorks does not need to send an explicit wake up.

```
/*
 * netdb.h by casey
*/

struct hostent {
 char *h_name;       /* official name of host */
 char **h_aliases; /* alias list            */
 int h_addrtype;       /* host address type     */
 int h_length;       /* length of address     */
 char **h_addr_list; /* list of addresses from name
server  */
#define h_addr h_addr_list[0] /* address, for backward
compatibility */
};

extern int h_errno;

#define TRY_AGAIN 2

struct hostent *gethostbyaddr(char * netnum, int size,
int SOCK);

/* eof netdb.h */

/*
 * netdb.c by casey
 */
#include <netdb.h>
#include <socket.h>
int h_errno;

/*
 * emulate unix gethostbyname
*/

struct hostent *gethostbyname(char *name)
 {
 struct hostent *host1;
 h_errno = 0;                /* we are always
successful!!! */
 host1 = (struct hostent *) malloc (sizeof(struct
```

```
hostent));

/* hard coded ntpvx1 should look up in etc/hosts to
get match
 * could make these a set of #defines in the netdb.h
file when
* there is more than one computer on the network
*/
 host1->h_name = "192.168.0.200";
 host1->h_addrtype = AF_INET;
 host1->h_aliases = NULL;
 host1->h_length = 2;
 host1->h_addr_list = NULL;
 return host1;
 }

/* eof netdb.c */
```

The netdb code is needed for networked calls based on names a `getservbyname()` will need to be added when it becomes necessary (when we have a VxWorks server).

Besides these additions to the code there have been numerous small changes that have been made all have been done in the systematic manner using the conventions mentioned above. The files altered at this stage are :

```
ntpv3.1\xntpd3.4\ntpdate.c
ntpv3.1\xntpd3.4\ntp_config.c
ntpv3.1\xntpd3.4\ntp_filegen.c
ntpv3.1\xntpd3.4\ntp_intres.c
ntpv3.1\xntpd3.4\ntp_io.c
ntpv3.1\xntpd3.4\ntp_sleep.c
ntpv3.1\xntpd3.4\ntp_timer.c
ntpv3.1\xntpd3.4\ntp_unixclock.c
ntpv3.1\xntpd3.4\ntpd.c
ntpv3.1\lib3.4\msyslog.c
ntpv3.1\lib3.4\netdb.c
ntpv3.1\include3.4\ntp_syslog.h
ntpv3.1\include3.4\ntp_machine.h
ntpv3.1\include3.4\netdb.h
ntpv3.1\include3.4\ntp_resource.h
```

The port is not complete yet, at this stage the main loop is running but no data is being received back from the server. The areas that could be giving problems are the following, it may not be transmitting, or receiving, or doing so on an unknown socket address.

An interesting observation: I happened to look at the Linux man pages for the following calls, in order to check their functionality as it would help in finding appropriate replacements for them, all were listed as conforming to POSIX, none are available in Tornado.

dup2(),link(), getpid(), open(), fcntl()

Conclusions and Recommendations

I am going to keep this closely tied to the porting of existing programs to VxWorks. The main differences between a normal unix/dos type operating system environment and that provided by VxWorks fall into two main categories, namely the kernel and interaction with it. This section is concluded with a list of items that may make matters easier in the long term.

Kernel

The wind kernel is primarily an embeddable real time kernel, whatever is placed into it is at the users discretion and effort. The modules are loaded into the kernel, and thus run with the same permissions. This means that the kernel carries the `main()` and that all written or ported code does not have a main program but a named main or task. This is not a major problem in itself but when one comes to porting code if this not kept in mind it can lead to problems.

Take for instance the placing of a daemon such as `xntpd` now in order to run the daemon the program creates space in memory for lists and sockets, it initializes memory dynamically and sets up the program with the intention that it will be kept running, freeing of memory is left to garbage collection later on. I can't be sure but I don't think VxWorks has any built in garbage removal. The noticeable effect, however, is upon restarting of the daemons, it will declare the sockets are still in use and the variables already assigned. This is only a problem when debugging, as in real application it will be embedded and restarting anything will be done by reboot. The main issue is when you are developing and testing you are subject to the same criteria, namely reboot after each session, this leads to a lengthy turnaround. This could be shortened by means of flash ROM for the development kernel or even HDD space for the kernel.

While debugging I suffered quite a few Kernel panics, this is something to bear in mind when evaluating the reliability and robustness of the VxWorks kernel. I was not able to ascertain the reasons, but a panic while debugging is often more serious as the memory should be especially well protected during debugging.

Interaction

The target-server configuration means the program is not built on the machine it will be executed on, thus the program must be ftp'ed to the target and execution data examined on the target or piped to the server. This does not lead to any major problems with the exception of the Crosswind (`gdbi86.exe`) debugger mentioned earlier. It is imperative that the debugger be functioning correctly and as expected for the fine tuning and bug finding on the remote target to make things easier and to save time. The other disadvantage is the lack of a built in file system for debug output, this needs to be created manually. All these things go towards making work slower.

Wishlist

Training

Training focused on the VxWorks system with special highlight given to porting and compatibility of system calls with the various flavours of Unix and Wind Kernel calls.

Help

A simple set of help files, or man pages in magnetic form that we could use to search. Would help tremendously. Perhaps we can ftp the man distribution? This would provide a man source and a source of flat test files for searching.

Compatibility libraries

We should get compatibility libraries, say sysV and BSD, if this is not possible we should build a set of libraries and calls that emulate these calls for future reference, like `sleep()`, `alarm()`, `get[host][serv]by[name][addr]()` and other calls that we may need with XTP and other ports envisaged, this can be done in the course of the porting. The major issue should be not to change the code we want to port, but where possible write the missing calls using VxWorks equivalents. This should lead to a library of commonly used calls that will eventually make porting simpler and quicker.

A different host

Among the things needed for porting big systems are standard unix tools. While gnu tools for DOS/Windows95 are available they are often subsets and not complete implementations. I have downloaded these tools from cygnus.com and will be testing them in my own time as the need arises during the course of other projects, but I believe it would be possible to port the host set up to Linux, creating a gcc that looks at the VxWorks library and headers for the compilation. I think this should be investigated as Linux has standard tools that would make porting easier and allow for an easier development environment.

APPENDIX D

IMS Software Test Description Extracts

(Excerpt from CCII/A500/IMS/6-STD)

Test Preparations

IMS Testing using the MULTIBUS II Interface

The IMS testbed consists of various hardware and software components. The hardware components are listed below for all the IMS services, this list is a superset of hardware required for testing the IMS services using the MULTIBUS II interface. Each IMS service will reference this document for hardware and software preparation.

Hardware Preparation

IMS Testbed consists of:

- 6 x TCUs (excluding 2 x spare TCUs)
- 6 x FDDI patchcords
- 1 x FDDI dual ring fibre network
- 6 x MULTIBUS II development racks
- 1 x LANWatch PC, including EISA FDDI adapter
- 6 x RS232 serial terminals
- 1 x storage oscilloscope

MULTIBUS II development racks consist of:

- 1 x IMS NIC card (CCT MBII P20 PMC host, plus PMC FDDI, plus IMS embedded application)
- N x CCT MBII PMC host processor (N depends on the number of hosts required in a particular setup)
- 1 x MBII development rack, including PSU, MBII backplane

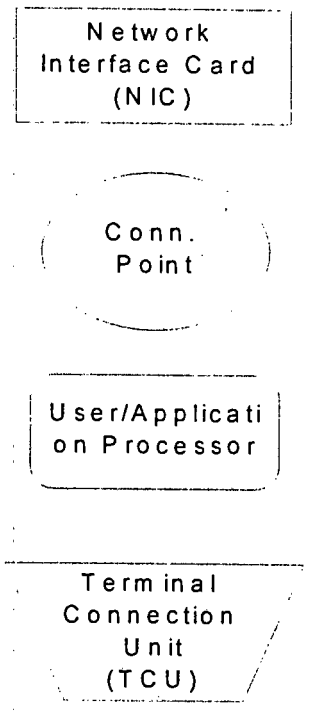
IMS Testbed Setup

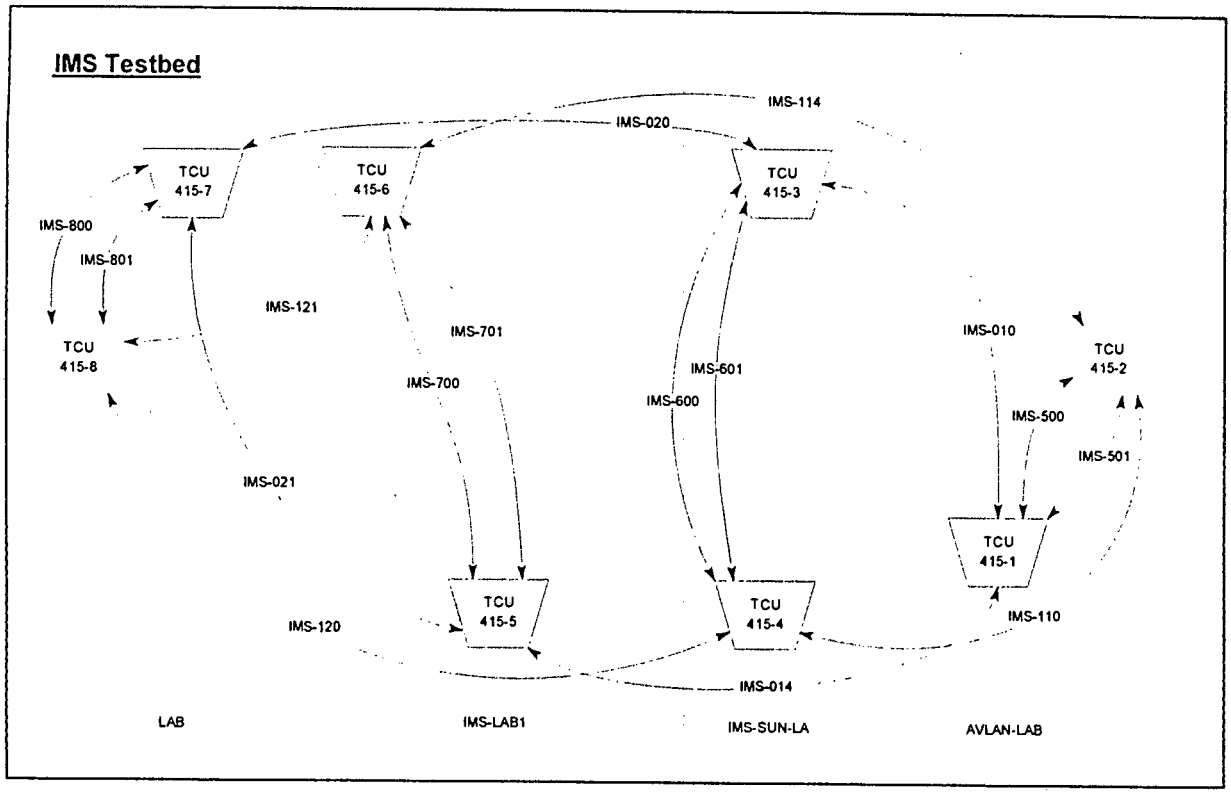
The IMS Testbed setup is displayed in the figure below. The figure displays the physical connections between each TCU.The TCUs are connected to the FDDI cable plant (main-run cable) by means of IMS-XXX FDDI cables. The TCUs will switch-in the connected NICs by means of the optical bypass switch (OBS) within the TCUs.

The redundant FDDI run cables are: IMS-121, IMS-114, IMS-110, IMS-120.

The figure does not display the connections between the TCU and the relevant IMS NICs, refer to further paragraphs for detailed information.

IMS Testbed - Legends

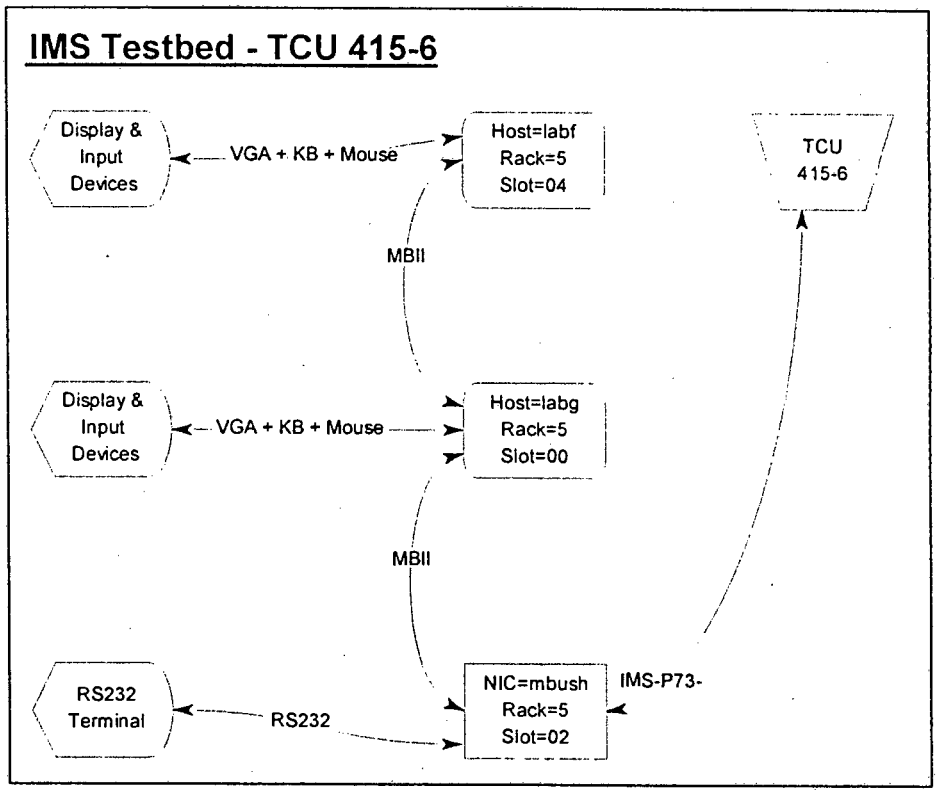




TCU 415-6 Interconnection

MULTIBUS II rack 5 consists of two MBII hosts and one IMS NIC MULTIBUS II board. These boards are connected over MBII and FDDI as displayed in the figure below. The IMS-P73-1 contains one FDDI patchcord consisting of 4 fibres and one electrical bypass control cable.

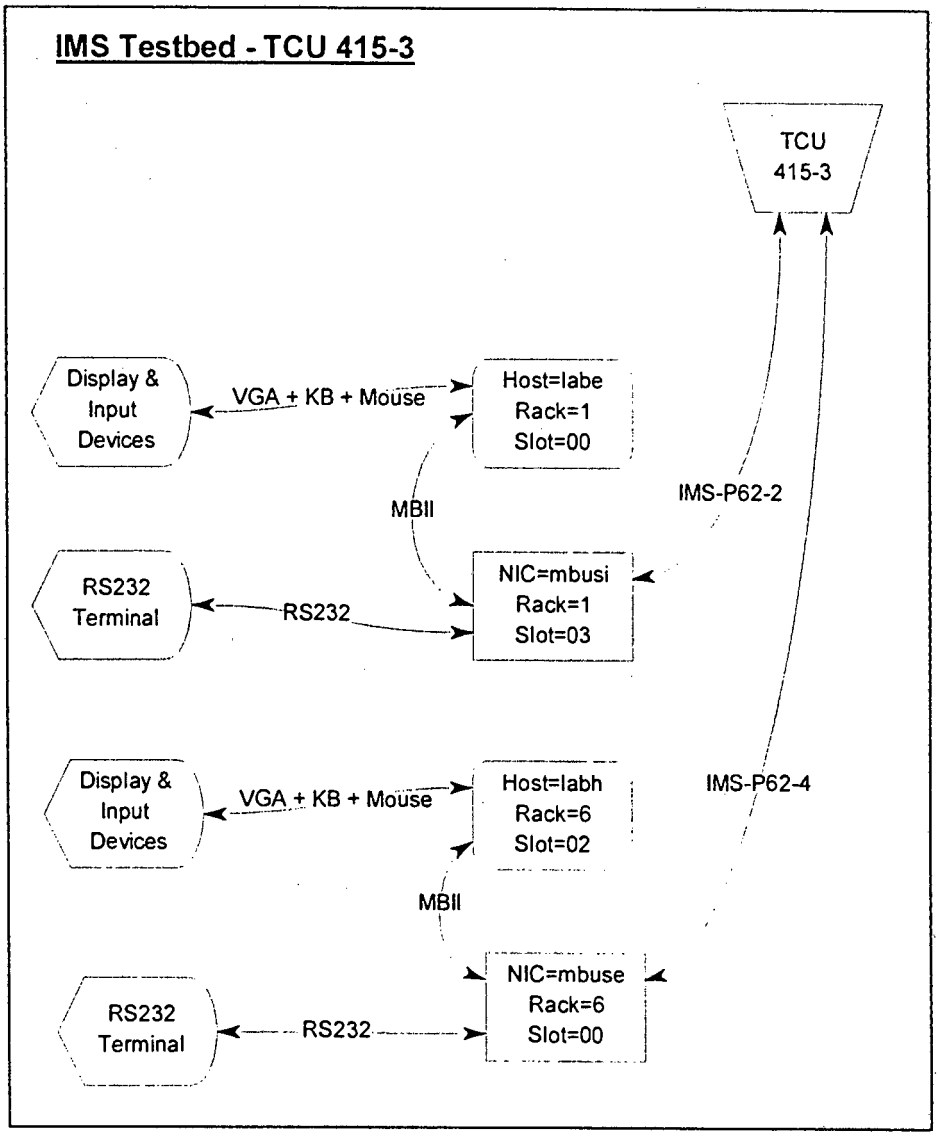
Display terminals are connected to the MBII hosts to enable the IMS Test Shell execution and interaction thereof. RS232 terminals are connected to the IMS NIC boards to view various debug outputs.


IMS Testbed - TCU 415-6
Display & Input Devices
VGA + KB + Mouse
Host=labf Rack=5 Slot=04
TCU 415-6
MBII
Display & Input Devices
VGA + KB + Mouse
Host=labg Rack=5 Slot=00
MBII
RS232 Terminal
RS232
NIC=mbush Rack=5 Slot=02
IMS-P73-

TCU 415-3 Interconnection

MULTIBUS II rack 1 and 6 are connected to TCU 415-3. The development racks consist of one IMS NIC, and one MBII Host. These boards are connected via MBII and FDDI as displayed in the figure below. IMS-P62-4 and IMS-P62-2 contains one FDDI patchcord each, consisting of 4 fibres and one electrical bypass control cable.

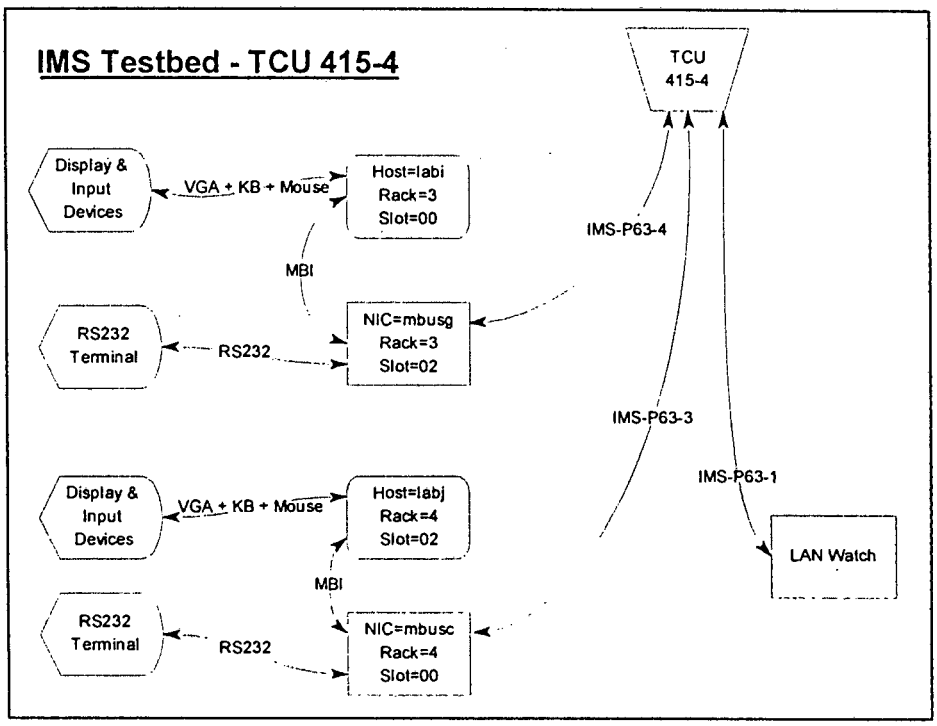
Display terminals are connected to the MBII hosts to enable the IMS Test Shell execution and interaction thereof. RS232 terminals are connected to the IMS NIC boards to view various debug outputs.



TCU 415-4 Interconnection

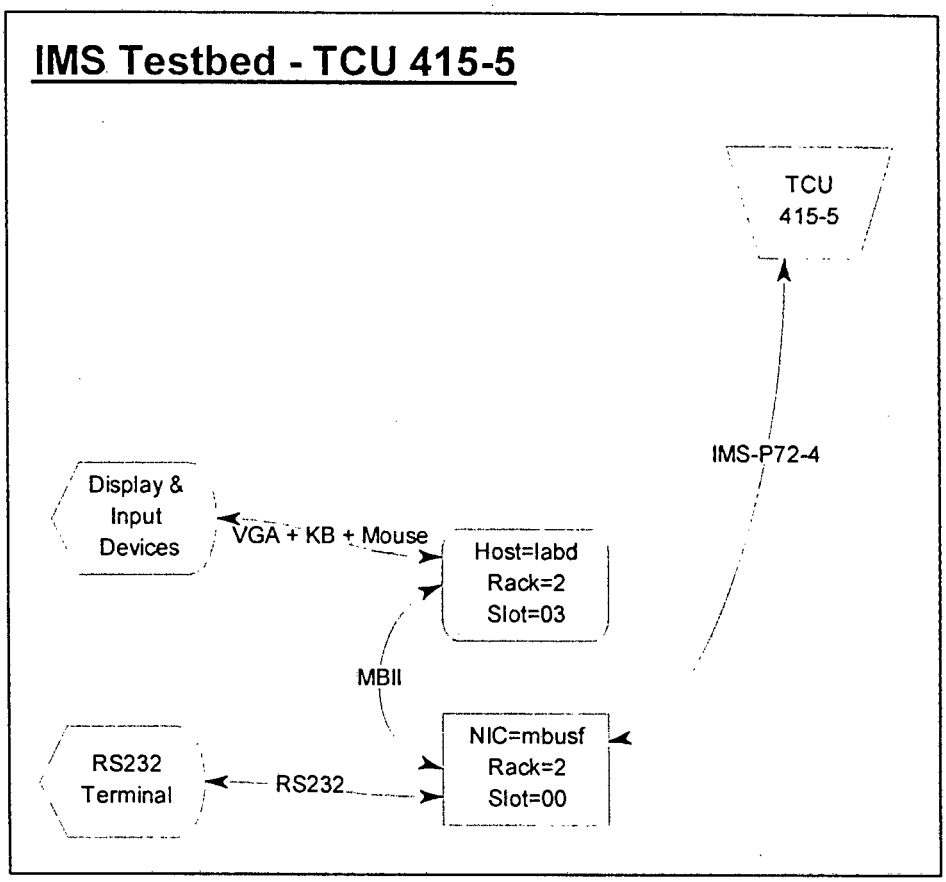
Two MULTIBUS II development racks are connected to TCU 415-4. The development racks each consist of one IMS NIC, and one MBII Host. These boards are connected via MBII and FDDI as displayed in the figure below. IMS-P63-1, IMS-P63-3 and IMS-P63-4 contains one FDDI patchcord each, consisting of 4 fibres and one electrical bypass control cable.

A LANWatch terminal is connected to the FDDI ring to capture and observe FDDI network traffic via TCU 415-4.



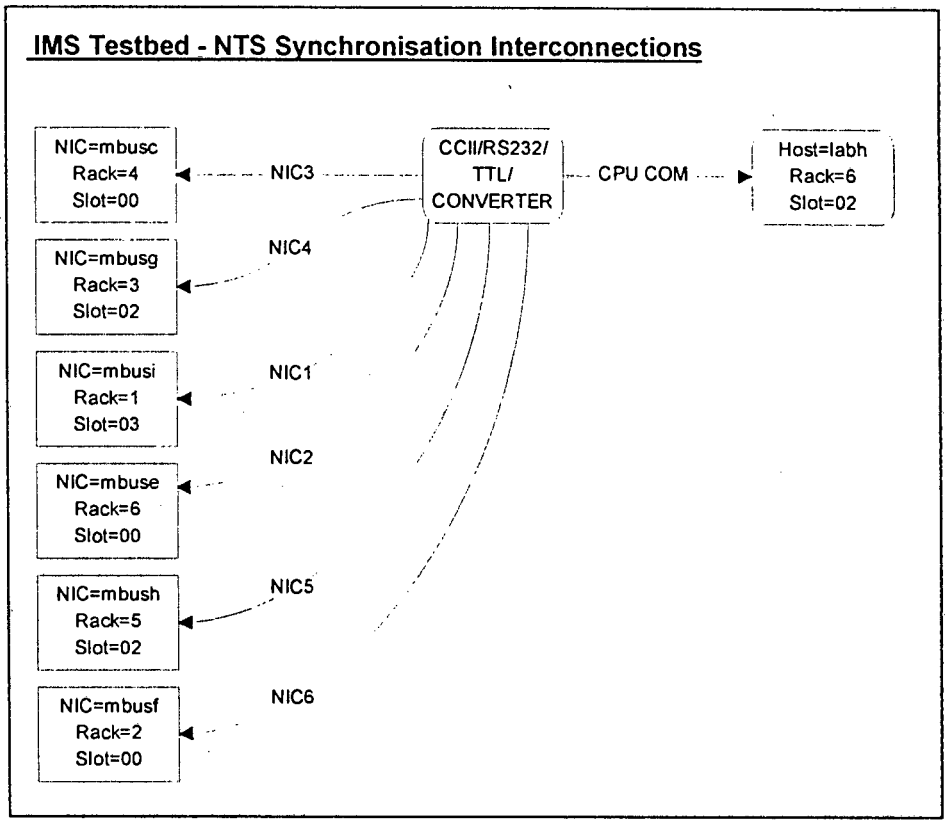
TCU 415--5 Interconnection

One MULTIBUS II development rack is connected to TCU 415-5. The development rack consists of one IMS NIC, and one MBII Host. These boards are connected via MBII and FDDI as displayed in the figure below. IMS-P72-4 contains one FDDI patchcord consisting of 4 fibres and one electrical bypass control cable.



NTS Hardware Preparation

Wire up the NTS synchronisation lines from the NTS test host to each of the NICs under test in the IMS-SUN-LAB. Connect the CCII/RS232/TTL/CONVERTER box cable labelled 'CPU COM2' to the NTS test host and the seperate cables labelled 'NIC1', 'NIC2', 'NIC3', 'NIC4', 'NIC5' and 'NIC6' to each of the IMS NICs under test.



The oscilloscope is used in NTS tests to perform timing measurements. Refer to the NTS STD for specific connection points of the oscilloscope.

APPENDIX E

NTS Software Test Description Extracts

(Excerpt from CCII/A500/IMS/6-STD/3)

NIC to NIC Synchronisation

Test the synchronisation of time between NICs.

NIC to NIC Synchronisation Test

Requirements Addressed

Test the synchronisation of time between NICs.

Prerequisite Conditions

Connect the IRQ15 control line on all NICs under test to the Interrupt 15 Trigger on the master Test Shell host. Debugging must be enabled by typing "nts_more_debug (0x3200)" at the command prompt on the NIC. This will display the NTS time at the NIC.

Test Inputs

The IRQ15 control line will be triggered to display the current NTS time on all NICs simultaneously as per **Table 1, Test 3201**.

Expected Test Results

All NTS time values displayed on the NIC Consoles are within 250 µs of each other. 250 µs corresponds to a Fraction Time value of 1 073 742.

Criteria for Evaluating Results

A successful test result is dependent on the output from the NIC Console agreeing with the Expected Test Results.

Test Procedure

a. Activate the Interrupt 15 Trigger from the Test Shell by selecting **Test 3201**. This must happen at least 60 seconds after startup to allow NTS a chance to synchronise.
b. Record the largest and smallest time values displayed on the NIC Consoles and calculate the difference.

GPS Synchronisation Test

This interface will not be implemented and thus cannot be tested.

Local NIC Isolated from LAN

Requirements Addressed

Test the behaviour of NTS when no longer attached to the LAN.

Prerequisite Conditions

Connect the IRQ15 control line on all NICs under test to the Interrupt 15 Trigger on the master Test Shell host. Debugging must be enabled by typing "nts_more_debug (0x3200)" at the command prompt on the NIC. This will display the NTS time at the NIC.

Test Inputs

The IRQ15 control line will be triggered to display the current NTS time on all NICs simultaneously, before and after removing the NIC from the LAN. Refer to **Table 1, Test 3202**.

Expected Test Results

NTS reports the quality of time as unsynchronised after being disconnected from the LAN. The NIC then returns time from its local clock, which continues to run.

Criteria for Evaluating Results

A successful test result is dependent on the output from the NIC Consoles agreeing with the Expected Test Results.

Test Procedure

a. Activate the Interrupt 15 Trigger from the Test Shell.
b. Verify that all NIC Consoles report the quality of time as 'SYNC'.
c. Unplug one of the NICs from the LAN.
d. Wait for 30 seconds and then activate the Interrupt 15 Trigger from the Test Shell.
e. Verify that the unplugged NIC's console reports the quality of time as 'UNSYNC'.
f. Record the displayed time values.
g. Wait for 10 seconds (from step (d)) and then activate the Interrupt 15 Trigger from the Test Shell.
h. Verify that the returned time values in step (d) and step (g) differ by approximately 10 seconds.

Master Clock Isolated from LAN

Requirements Addressed

Test the ability of NTS to synchronise to a secondary server in the event of the master server becoming unavailable.

Prerequisite Conditions

Connect the IRQ15 control line on all NICs under test to the Interrupt 15 Trigger on the master Test Shell host. Debugging must be enabled by typing "nts_more_debug (0x3200)" at the command prompt on the NIC. This will display the NTS time at the NIC.

Test Inputs

The master time server is removed from the LAN so that NTS synchronises to the secondary time server. The secondary time server is then removed so that NTS is no longer synchronised at all. The master time server is then reconnected so that NTS may resynchronise to it. Refer to **Table 1, Test 3203**.

Expected Test Results

NTS is synchronised to the master time server if it is connected. If not, NTS will then synchronise to the secondary server. If both time servers are disconnected, NTS will indicate that it is unsynchronised.

Criteria for Evaluating Results

A successful test result is dependent on the successful completion of the Test Procedure.

Test Procedure

a. Run the xntpdc utility on one of the NICs.
b. Type 'peers' at the NIC console and check that a '*' appears next to the master time server entry.
c. Disconnect the master time server from the LAN.
d. Wait for 60 seconds and then type 'peers' at the NIC console. Check that a '*' appears next to the secondary time server entry.
e. Activate the Interrupt 15 Trigger from the Test Shell.
f. Verify that all NIC consoles indicate that their clocks are in the 'SYNC' state (except for the master time server).
g. Disconnect the secondary time server from the LAN.
h. Wait for 60 seconds and then type 'peers' at the NIC console. Check that no '*' appears next to any of the entries.
i. Activate the Interrupt 15 Trigger from the Test Shell.
j. Verify that all NIC consoles indicate that their clocks are in the 'UNSYNC' state.
k. Reconnect the master time server to the LAN.
l. Wait for 60 seconds and then type 'peers' at the NIC console. Check that a '*' appears next to the master time server entry.
m. Activate the Interrupt 15 Trigger from the Test Shell.
n. Verify that all NIC consoles indicate that their clocks are in the 'SYNC' state.
o. Exit xntpdc by typing 'quit' at the NIC console.

Stratum Level Test

Requirements Addressed

Test the synchronisation of time between NICs when two stratum level 1 servers are present.

Prerequisite Conditions

Connect the IRQ15 control line on all NICs under test to the Interrupt 15 Trigger on the master Test Shell host. Debugging must be enabled by typing "nts_more_debug (0x3200)" at the command prompt on the NIC. This will display the NTS time at the NIC.

Test Inputs

One of the NICs is promoted to stratum level 1 (the same level as the master time server). The Interrupt 15 line is triggered as per **Table 1, Test 3204.**

Expected Test Results

NTS remains synchronised to the original master time server.

Criteria for Evaluating Results

A successful test result is dependent on the output from the NIC Consoles agreeing with the Expected Test Results.

Test Procedure

a. Run the xntpdc utility on one of the NICs.
b. Type 'peers' at the NIC console and check that the master time server has a stratum level of 1 and has a '*' next to it.
c. Type 'fudge *ip_address* stratum 1', where *ip_address* is the IP address of one of the stratum level 10 peers.
d. Type 'peers' and check that two entries show a stratum level of 1, and that the '*' is still next to the original master time server.
e. Wait for 60 seconds and then activate the Interrupt 15 Trigger from the Test

Shell.
f. Verify that all NIC consoles report that they are in the 'SYNC' state.
g. Type 'fudge *ip_address* stratum 10', where *ip_address* is the same as the one set earlier.
h. Exit xntpdc by typing 'quit' at the NIC console.

NIC Unsynchronised Test

Requirements Addressed

Test the behaviour of NTS before synchronisation is reached.

Prerequisite Conditions

Connect the IRQ15 control line on all NICs under test to the Interrupt 15 Trigger on the master Test Shell host. Debugging must be enabled by typing "nts_more_debug (0x3200)" at the command prompt on the NIC. This will display the NTS time at the NIC.

Test Inputs

The time on the local NIC will be set forward by 1 second. The IRQ15 control line will then be triggered every 30 seconds to display the times on all NICs simultaneously as per **Table 1, Test 3205.**

Expected Test Results

The time difference between the local NIC and the other NICs will decrease until the local NIC is synchronised within 250 is. While the time difference is greater than this amount, the returned quality of time value shows that the NIC is in the 'UNSYNC' or the 'CONV' state.

Criteria for Evaluating Results

A successful test result is dependent on the returned messages and the output from the NIC Consoles agreeing with the Expected Test Results.

Test Procedure

a. Use NTS_get_time to retrieve the current time value.
b. Add 1 second to the time value retrieved in the previous step and use NTS_set_time to unsynchronise the NIC by setting the time to this value.
c. Activate the Interrupt 15 Trigger from the Test Shell and immediately send a nts_get_time request.
d. Record the quality of time value in the returned message.
e. Record the largest and smallest time values displayed on the NIC Consoles and calculate the difference.
f. Repeat the last three steps every 30 seconds for the next 5 minutes.

Table 1: NIC to NIC Synchronisation Tests

Test Case		Input Parameters	Expected Results
		Interrupt 15	NIC Console
3201	NIC to NIC Synchronisation Test	Trigger	Largest and smallest time values differ by no more than 250 µs.
3202	Local NIC Isolated from LAN	Trigger Wait 30 s Trigger Wait 10 s Trigger	First time values are 'SYNC', remaining values are 'UNSYNC'. Difference between last two values is 10 s.
3203	Master Clock Isolated from LAN	Trigger (Disconnect Primary master) Wait 60 s Trigger (Disconnect Secondary master) Wait 60 s Trigger (Reconnect Primary master) Wait 60 s Trigger	First, second and fourth sets of 'quality of time' values are 'SYNC'. Third 'quality of time' value are 'UNSYNC'.
3204	Stratum Level Test	Trigger	Time values are 'SYNC'.
3205	NIC Unsynchronised Test	Trigger every 30 s for 5 minutes.	First time values are 'UNSYNC'. By end of test, time values are 'SYNC'.

APPENDIX F

Selected Source Code Listing

Owing to the large number of changes made and the extent of the systems code I have included only the configuration changes for NTP. The listings below will give the reader an overview of the scope of changes only. For a full listing of changes in `diff -r -w -u` format see the NTP distribution pacthes obtainable form [NTP98], where patches 164 and later are relevant.

ntp_machine.h

Excerpt from ntp_machine.h showing additions only

```
/*casey Tue May 27 15:45:25 SAT 1997*/#ifdef SYS_VXWORKS
/* casey's new defines */
#define NO_MAIN_ALLOWED     1
#define NO_NETDB            1
#define NO_RENAME           1

/* in vxWorks we use FIONBIO, but the others are defined for old systems, so
 * all hell breaks loose if we leave them defined we define USE_FIONBIO to
 * undefine O_NONBLOCK FNDELAY O_NDELAY where necessary.
 */
#define USE_FIONBIO         1
/* end my new defines */

#define TIMEOFDAY           0x0     /* system wide realtime clock */
#define HAVE_GETCLOCK       1       /* configure does not set this ... */
#define HAVE_NO_NICE        1       /* configure does not set this ... */
#define HAVE_RANDOM         1       /* configure does not set this ... */
#define HAVE_SRANDOM        1       /* configure does not set this ... */
#define NODETACH            1

/* vxWorks specific additions to take care of its
 * unix (non)complicance
 */

#include "vxWorks.h"
#include "ioLib.h"
#include "taskLib.h"
#include "time.h"

extern int sysClkRateGet();

/* usrtime.h
 * Bob Herlien's excellent time code find it at:
 * ftp://ftp.atd.ucar.edu/pub/vxworks/vx/usrTime.shar
 * I would recommend this instead of clock_[g|s]ettime() plus you get
 * adjtime() too ... casey
 */
/*
extern int    gettimeofday( struct timeval *tp, struct timezone *tzp );
extern int    settimeofday(struct timeval *, struct timezone *);
extern int    adjtime( struct timeval *delta, struct timeval *olddelta );
 */

/* in  machines.c */
```

```
extern void sleep (int seconds);
extern void alarm (int seconds);
/* machines.c */

/*      this is really this     */
#define getpid      taskIdSelf
#define getclock    clock_gettime
#define fcntl       ioctl
#define _getch      getchar
#define random      rand
#define srandom     srand

/* define this away for vxWorks */
#define openlog(x,y)
/* use local defines for these */
#undef min
#undef max

#endif /* SYS_VXWORKS */

#ifdef NO_NETDB
/* These structures are needed for gethostbyname() etc... */
/* structures used by netdb.h */
struct  hostent {
    char    *h_name;                /* official name of host */
    char    **h_aliases;            /* alias list */
    int h_addrtype;                 /* host address type */
    int h_length;                   /* length of address */
    char    **h_addr_list;          /* list of addresses from name server */
#define     h_addr h_addr_list[0]   /* address, for backward compatibility
*/
};

struct  servent {
    char    *s_name;                /* official service name */
    char    **s_aliases;            /* alias list */
    int s_port;                     /* port # */
    char    *s_proto;               /* protocol to use */
};
extern int h_errno;

#define TRY_AGAIN   2

struct hostent *gethostbyname(char * netnum);
struct hostent *gethostbyaddr(char * netnum, int size, int addr_type);
/* type is the protocol */
struct servent *getservbyname (char *name, char *type);
#endif  /* NO_NETDB */
```

```
#ifdef NO_MAIN_ALLOWED
/* we have no main routines so lets make a plan */
#define CALL(callname, progname, callmain) \
    extern int callmain(int,char**); \
    void callname(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10) \
        char *a0;  \
        char *a1;  \
        char *a2;  \
        char *a3;  \
        char *a4;  \
        char *a5;  \
        char *a6;  \
        char *a7;  \
        char *a8;  \
        char *a9;  \
        char *a10; \
    { \
      char *x[11]; \
      int argc; \
      char *argv[] =
{progname,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,N
ULL,NULL,NULL}; \
      int i; \
      for (i=0;i<11;i++) \
       x[i] = NULL; \
      x[0] = a0; \
      x[1] = a1; \
      x[2] = a2; \
      x[3] = a3; \
      x[4] = a4; \
      x[5] = a5; \
      x[6] = a6; \
      x[7] = a7; \
      x[8] = a8; \
      x[9] = a9; \
      x[10] = a10; \
      argc=1; \
      for (i=0; i<11;i++) \
        if (x[i]) \
        { \
          argv[argc++] = x[i];  \
        } \
     callmain(argc,argv);  \
    }
#endif /* NO_MAIN_ALLOWED */
/*casey Tue May 27 15:45:25 SAT 1997*/
```

machines.h

Excerpt from machines.c showing additions only

```
#ifdef SYS_VXWORKS
#include "taskLib.h"
#include "sysLib.h"
#include "time.h"
#include "ntp_syslog.h"

/*  some translations to the world of vxWorkings -casey */
/* first some netdb type things */
#include "ioLib.h"
#include <socket.h>
int h_errno;

struct hostent *gethostbyname(char *name)
    {
    struct hostent *host1;
    h_errno = 0;                    /* we are always successful!!! */
    host1 = (struct hostent *) malloc (sizeof(struct hostent));
    host1->h_name = name;
    host1->h_addrtype = AF_INET;
    host1->h_aliases = name;
    host1->h_length = 4;
    host1->h_addr_list[0] = (char *)hostGetByName (name);
    host1->h_addr_list[1] = NULL;
    return host1;
    }

struct hostent *gethostbyaddr(char *name, int size, int addr_type)
    {
    struct hostent *host1;
    h_errno = 0;  /* we are always successful!!! */
    host1 = (struct hostent *) malloc (sizeof(struct hostent));
    host1->h_name = name;
    host1->h_addrtype = AF_INET;
    host1->h_aliases = name;
    host1->h_length = 4;
    host1->h_addr_list = NULL;
    return host1;
    }

struct servent *getservbyname (char *name, char *type)
    {
    struct servent *serv1;
    serv1 = (struct servent *) malloc (sizeof(struct servent));
    serv1->s_name = "ntp";      /* official service name */
```

```
    serv1->s_aliases = NULL;     /* alias list */
    serv1->s_port = 123;         /* port # */
    serv1->s_proto = "udp";      /* protocol to use */
    return serv1;
    }

/* second
 * vxworks thinks it has insomnia
 * we have to sleep for number of seconds
 */

#define CLKRATE     sysClkRateGet()

/* I am not sure how valid the granularity is - it is from G. Eger's port */
#define CLK_GRANULARITY  1       /* Granularity of system clock in usec  */
                                 /* Used to round down # usecs/tick      */
                                 /* On a VCOM-100, PIT gets 8 MHz clk,   */
                                 /*  & it prescales by 32, thus 4 usec   */
                                 /* on mv167, granularity is 1usec anyway*/
                                 /* To defeat rounding, set to 1         */
#define USECS_PER_SEC       1000000L        /* Microseconds per second  */
#define TICK (((USECS_PER_SEC / CLKRATE) / CLK_GRANULARITY) * CLK_GRANULARITY)

/* emulate unix sleep
 * casey
 */
void sleep(int seconds)
    {
    taskDelay(seconds*TICK);
    }
/* emulate unix alarm
 * that pauses and calls SIGALRM after the seconds are up...
 * so ... taskDelay() fudged for seconds should amount to the same thing.
 * casey
 */
void alarm (int seconds)
    {
    sleep(seconds);
    }

#endif /* SYS_VXWORKS */
```

configure.in

Excerpt from configure.in showing additions only

```
+++ ../../patched/xntp3-5.90.1c-export/configure.in     Mon Jul
21 08:18:19 1997
@@ -10,8 +10,22 @@
+dnl  we need to check for cross compile tools for vxWorks here
+
+case "$host" in
+ $target)
+    ;;
+ *) case "$target" in
+     *-*-vxworks*)
+        CFLAGS="$CFLAGS -DSYS_VXWORKS"
+        ;;
+    esac
+    ;;
+esac
+
+dnl  we need to check for cross compile tools for vxWorks here
@@ -88,7 +102,13 @@
+
+case "$target" in
+ *-*-vxworks*)
+    ac_link="$ac_link $VX_KERNEL"
+    ;;
+esac
+
@@ -189,7 +209,24 @@
+
+#AC_C_BIGENDIAN                dnl CROSS-COMPILE?
+case "$host" in
+ $target)
+    AC_C_BIG_ENDIAN
+    ;;
+ *) case "$target" in
+     i*86-*-vxworks*)
+       # LITTLEENDIAN
+        ;;
+     *-*-vxworks*)
+        AC_DEFINE(WORDS_BIGENDIAN)
+        ;;
+    esac
+    ;;
+esac
+
+
```

```
@@ -297,9 +334,44 @@
+#AC_CHECK_SIZEOF(signed char)  dnl CROSS_COMPILE?
+case "$host" in
+ $target)
+    AC_CHECK_SIZEOF(signed char)
+    ;;
+ *) case "$target" in
+     *-*-vxworks*)
+     AC_CHECK_SIZEOF(signed char, 1)
+        ;;
+    esac
+    ;;
+esac
+
+#AC_CHECK_SIZEOF(int)          dnl CROSS_COMPILE?
+case "$host" in
+ $target)
+    AC_CHECK_SIZEOF(int)
+    ;;
+ *) case "$target" in
+     *-*-vxworks*)
+     AC_CHECK_SIZEOF(int, 4)
+        ;;
+    esac
+    ;;
+esac
+#AC_CHECK_SIZEOF(long)         dnl CROSS_COMPILE?
+case "$host" in
+ $target)
+    AC_CHECK_SIZEOF(long)
+    ;;
+ *) case "$target" in
+     *-*-vxworks*)
+     AC_CHECK_SIZEOF(long, 4)
+        ;;
+    esac
+    ;;
+esac
+
@@ -350,7 +422,7 @@
+       ac_cv_func_Xettimeofday_nargs=1,
ac_cv_func_Xettimeofday_nargs=2)
@@ -593,10 +665,13 @@
+ *)
+    case "$host" in
+     $target) ntp_warning='Which leaves us with nothing to
use!'
+esac
@@ -2173,6 +2248,17 @@
+esac
```

```
+
+case "$host" in
+ $target)
+    ;;
+ *) case "$target" in
+     *-*-vxworks*)
+        LDFLAGS="$LDFLAGS -r"
+        ;;
+    esac
+    ;;
```

ntp_io.c

Excerpt from the XTP ntp_io.c showing context and addtions for create_sockets() and open_socket() routines

```
/*
 * create_sockets - create a socket for each interface plus a default
 *                  socket for when we don't know where to send
 */
static int
create_sockets(port)
        u_int port;
{
#if defined (STREAMS_TLI) || defined (XTP)
        struct strioctl ioc;
#endif /* STREAMS_TLI */
#ifndef SYS_WINNT
        char    buf[MAXINTERFACES*sizeof(struct ifreq)];
#ifdef  XTP
    xtic_t  xtbuf;
#endif
        struct  ifconf  ifc;
        struct  ifreq   ifreq, *ifr;
#endif /* SYS_WINNT */
        int n, i, j, vs, size;
        struct sockaddr_in resmask;

#ifdef DEBUG
        if (debug)
                printf("create_sockets(%d)\n", ntohs(port));
#endif

        /*
         * create pseudo-interface with wildcard address
         */
        inter_list[0].sin.sin_family = AF_INET;
        inter_list[0].sin.sin_port = port;
        inter_list[0].sin.sin_addr.s_addr = htonl(INADDR_ANY);
        (void) strncpy(inter_list[0].name, "wildcard",
             sizeof(inter_list[0].name));
        inter_list[0].mask.sin_addr.s_addr = htonl(~0);
        inter_list[0].received = 0;
        inter_list[0].sent = 0;
        inter_list[0].notsent = 0;
        inter_list[0].flags = INT_BROADCAST;

#if !defined (SYS_WINNT)
#ifndef XTP
```

```
#ifdef USE_STREAMS_DEVICE_FOR_IF_CONFIG
        if ((vs = open("/dev/ip", O_RDONLY)) < 0) {
#else /* ! USE_STREAMS_DEVICE_FOR_IF_CONFIG */
        if ((vs = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
#endif /* USE_STREAMS_DEVICE_FOR_IF_CONFIG */
                syslog(LOG_ERR, "vs=socket(AF_INET, SOCK_DGRAM) %m");
                exit(1);
        }
#endif /* XTP */
        i = 1;

        ifc.ifc_len = sizeof(buf);

#ifdef XTP
        if ((vs = open("/dev/xtindg", O_RDONLY, 0)) < 0) {
                syslog(LOG_ERR, "vs=socket(AF_INET, SOCK_DGRAM) %m");
                exit(1);
        }

    ioc.ic_cmd = XT_IOC_ADMIN;
    ioc.ic_timout = 0;
    ioc.ic_dp =   &xtbuf;
    xtbuf.xtic_cmd = XT_IFCONFIG_GET;
    ioc.ic_len = sizeof(xtbuf);
    if(ioctl(vs, I_STR, &ioc) == ERROR  ||
        ioc.ic_len < sizeof(struct xtic_s)) {
        syslog(LOG_ERR, "get interface configuration: %m");
        exit(1);
        }
        ifc.ifc_len = sizeof(struct ifreq);
        ifc.ifc_buf = buf;
    bcopy(xtbuf.xtic_if.xtif_ifname,((struct ifreq*) buf)->ifr_name, 16);
    bcopy(xtbuf.xtic_if.xtif_ip_addr,(char *) (&((struct ifreq*)buf)-
>ifr_addr), 4);
#endif
#if defined (STREAMS_TLI)
        ioc.ic_cmd = SIOCGIFCONF;
        ioc.ic_timout = 0;
        ioc.ic_dp = (caddr_t)buf;
        ioc.ic_len = sizeof(buf);
        if(ioctl(vs, I_STR, &ioc) < 0 ||
            ioc.ic_len < sizeof(struct ifreq)) {
                syslog(LOG_ERR, "get interface configuration: %m");
                exit(1);
        }
#ifdef SIZE_RETURNED_IN_BUFFER
        ifc.ifc_len = ioc.ic_len - sizeof(int);
        ifc.ifc_buf = buf + sizeof(int);
#else /* ! SIZE_RETURNED_IN_BUFFER */
        ifc.ifc_len = ioc.ic_len;
```

```
        ifc.ifc_buf = buf;
#endif /* SIZE_RETURNED_IN_BUFFER */

#else /* ! STREAMS_TLI */
        ifc.ifc_len = sizeof(buf);
        ifc.ifc_buf = buf;
        if (ioctl(vs, SIOCGIFCONF, (char *)&ifc) < 0) {
                syslog(LOG_ERR, "get interface configuration: %m");
                exit(1);
        }
#endif /* STREAMS_TLI */

        for(n = ifc.ifc_len, ifr = ifc.ifc_req; n > 0;
            ifr = (struct ifreq *)((char *)ifr + size)) {
                size = sizeof(*ifr);

#ifdef HAVE_VARIABLE_IFR_LENGTH
                if (ifr->ifr_addr.sa_len > sizeof(ifr->ifr_addr))
                        size += ifr->ifr_addr.sa_len - sizeof(struct
sockaddr);
#endif
                n -= size;
#ifdef VMS /* VMS+UCX */
                if(((struct sockaddr *)&(ifr->ifr_addr))->sa_family !=
AF_INET)
#else
                if (ifr->ifr_addr.sa_family != AF_INET)
#endif /* VMS+UCX */
                        continue;
                ifreq = *ifr;
#ifdef STREAMS_TLI
                ioc.ic_cmd = SIOCGIFFLAGS;
                ioc.ic_timout = 0;
                ioc.ic_dp = (caddr_t)&ifreq;
                ioc.ic_len = sizeof(struct ifreq);
                if(ioctl(vs, I_STR, &ioc)) {
#else /* ! STREAMS_TLI */
                if (ioctl(vs, SIOCGIFFLAGS, (char *)&ifreq) < 0) {
#endif /* STREAMS_TLI */
                        syslog(LOG_ERR, "get interface flags: %m");
                        continue;
                }
                if ((ifreq.ifr_flags & IFF_UP) == 0)
                        continue;
                inter_list[i].flags = 0;
                if (ifreq.ifr_flags & IFF_BROADCAST)
                        inter_list[i].flags |= INT_BROADCAST;
#if !defined(SUN_3_3_STINKS)
#if defined(SYS_HPUX) && (SYS_HPUX < 8)
                if (ifreq.ifr_flags & IFF_LOCAL_LOOPBACK)
```

```
#else
                if (ifreq.ifr_flags & IFF_LOOPBACK)
#endif
                {
                        inter_list[i].flags |= INT_LOOPBACK;
                        if (loopback_interface == 0)
                                loopback_interface = &inter_list[i];
                }
#endif

#ifdef STREAMS_TLI
                ioc.ic_cmd = SIOCGIFADDR;
                ioc.ic_timout = 0;
                ioc.ic_dp = (caddr_t)&ifreq;
                ioc.ic_len = sizeof(struct ifreq);
                if(ioctl(vs, I_STR, &ioc)) {
#else /* ! STREAMS_TLI */
                if (ioctl(vs, SIOCGIFADDR, (char *)&ifreq) < 0) {
#endif /* STREAMS_TLI */
                        syslog(LOG_ERR, "get interface addr: %m");
                        continue;
                }

                (void)strncpy(inter_list[i].name, ifreq.ifr_name,
                    sizeof(inter_list[i].name));
                inter_list[i].sin = *(struct sockaddr_in *)&ifreq.ifr_addr;
                inter_list[i].sin.sin_family = AF_INET;
                inter_list[i].sin.sin_port = port;

#if defined(SUN_3_3_STINKS)
                /*
                 * Oh, barf!  I'm too disgusted to even explain this
                 */
                if (SRCADR(&inter_list[i].sin) == 0x7f000001) {
                        inter_list[i].flags |= INT_LOOPBACK;
                        if (loopback_interface == 0)
                                loopback_interface = &inter_list[i];
                }
#endif
                if (inter_list[i].flags & INT_BROADCAST) {
#ifdef STREAMS_TLI
                        ioc.ic_cmd = SIOCGIFBRDADDR;
                        ioc.ic_timout = 0;
                        ioc.ic_dp = (caddr_t)&ifreq;
                        ioc.ic_len = sizeof(struct ifreq);
                        if(ioctl(vs, I_STR, &ioc)) {
#else /* ! STREAMS_TLI */
                        if (ioctl(vs, SIOCGIFBRDADDR, (char *)&ifreq) < 0) {
#endif /* STREAMS_TLI */
                                syslog(LOG_ERR, "SIOCGIFBRDADDR fails");
```

```
                                exit(1);
                        }
#ifndef ifr_broadaddr
                        inter_list[i].bcast =
                                *(struct sockaddr_in *)&ifreq.ifr_addr;
#else
                        inter_list[i].bcast =
                                *(struct sockaddr_in *)&ifreq.ifr_broadaddr;
#endif
                        inter_list[i].bcast.sin_family = AF_INET;
                        inter_list[i].bcast.sin_port = port;
                }
#ifdef STREAMS_TLI
                ioc.ic_cmd = SIOCGIFNETMASK;
                ioc.ic_timout = 0;
                ioc.ic_dp = (caddr_t)&ifreq;
                ioc.ic_len = sizeof(struct ifreq);
                if(ioctl(vs, I_STR, &ioc)) {
#else /* ! STREAMS_TLI */
                if (ioctl(vs, SIOCGIFNETMASK, (char *)&ifreq) < 0) {
#endif /* STREAMS_TLI */
                        syslog(LOG_ERR, "SIOCGIFNETMASK fails");
                        exit(1);
                }
                inter_list[i].mask = *(struct sockaddr_in *)&ifreq.ifr_addr;

                /*
                 * look for an already existing source interface address.
If
                 * the machine has multiple point to point interfaces, then
                 * the local address may appear more than once.
                 */
                for (j=0; j < i; j++)
                        if (inter_list[j].sin.sin_addr.s_addr ==
                            inter_list[i].sin.sin_addr.s_addr) {
                                break;
                        }
                if (j == i)
                        i++;
        }
        close(vs);
#else /* SYS_WINNT */
        /* don't know how to get information about network interfaces on
Win/NT
         * one socket bound to wildcard interface is good enough for now
         */
         i = 1;
#endif /* SYS_WINNT  */
        ninterfaces = i;
```

```
        maxactivefd = 0;
FD_ZERO(&activefds);

        for (i = 0; i < ninterfaces; i++) {
                inter_list[i].fd = open_socket(&inter_list[i].sin,
                    inter_list[i].flags & INT_BROADCAST);
        }

#if defined(MCAST) && !defined(sun) && !defined(SYS_BSDI) &&
!defined(SYS_DECOSF1) && !defined(SYS_44BSD)
        /*
         * enable possible multicast reception on the broadcast socket
         */
        inter_list[0].bcast.sin_addr.s_addr = htonl(INADDR_ANY);
        inter_list[0].bcast.sin_family = AF_INET;
        inter_list[0].bcast.sin_port = port;
#endif /* MCAST */

        /*
         * Blacklist all bound interface addresses
         */
        resmask.sin_addr.s_addr = ~0L;
        for (i = 1; i < ninterfaces; i++)
                restrict(RESTRICT_FLAGS, &inter_list[i].sin, &resmask,
                    RESM_NTPONLY|RESM_INTERFACE, RES_IGNORE);

        any_interface = &inter_list[0];
#ifdef DEBUG
        if (debug > 2) {
                printf("create_sockets: ninterfaces=%d\n", ninterfaces);
                for (i = 0; i < ninterfaces; i++) {
                        printf("interface %d:  fd=%d,  bfd=%d,  name=%.8s,
flags=0x%x\n",
                                i,
                                inter_list[i].fd,
                                inter_list[i].bfd,
                                inter_list[i].name,
                                inter_list[i].flags);
                        /* Leave these as three printf calls. */
                        printf("              sin=%s",
                                inet_ntoa((inter_list[i].sin.sin_addr)));
                        if(inter_list[i].flags & INT_BROADCAST)
                                printf("  bcast=%s,",

inet_ntoa((inter_list[i].bcast.sin_addr)));
                        printf("  mask=%s\n",
                                inet_ntoa((inter_list[i].mask.sin_addr)));
                }
        }
#endif
```

```
        return ninterfaces;
}


/*
 * open_socket - open a socket, returning the file descriptor
 */
static int
open_socket(addr, flags)
        struct sockaddr_in *addr;
        int flags;
{
        int fd;
#ifndef SYS_VXWORKS
        int on = 1, off = 0;
#else
        int on = TRUE, off = FALSE;
#endif

        /* create a datagram (UDP) socket */
        if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
                syslog(LOG_ERR, "socket(AF_INET, SOCK_DGRAM, 0) failed:
%m");
                exit(1);
                /*NOTREACHED*/
        }

#ifndef XTP
        /* set SO_REUSEADDR since we will be binding the same port
           number on each interface */
        if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
                        (char *)&on, sizeof(on))!= 0) {
                syslog(LOG_ERR, "setsockopt SO_REUSEADDR on fails: %m");
        }
#endif
        /*
         * bind the local address.
         */
#ifndef SYS_VXWORKS
        if (bind(fd, (struct sockaddr *)addr, sizeof(*addr)) < 0) {
#else
        if (bind(fd, (struct sockaddr *)addr, sizeof(struct sockaddr)) ==
ERROR) {
#endif
                char buff[160];
                sprintf(buff,
        "bind() fd %d, family %d, port %d, addr %08lx, in_classd=%d flags=%d
fails: %%m",
                    fd, addr->sin_family, (int)ntohs(addr->sin_port),
                    (u_long)ntohl(addr->sin_addr.s_addr),
```

```
                    IN_CLASSD(ntohl(addr->sin_addr.s_addr)), flags);
                syslog(LOG_ERR, buff);

#ifdef XTP
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
                            (char *)&on, sizeof(on))!= 0) {
                syslog(LOG_ERR, "setsockopt IP_REUSEADDR on fails: %m");
        }
#endif

#ifndef SYS_WINNT
                close(fd);
#else
                closesocket(fd);
#endif

                /*
                 * soft fail if opening a class D address
                 */
                if (IN_CLASSD(ntohl(addr->sin_addr.s_addr)))
                        return -1;
                exit(1);
        }
#ifdef DEBUG
        if (debug)
                printf("bind() fd %d, family %d, port %d, addr %08lx,
flags=%d\n",
                        fd,
                        addr->sin_family,
                        (int)ntohs(addr->sin_port),
                        (u_long)ntohl(addr->sin_addr.s_addr),
                        flags);
#endif
        if (fd > maxactivefd)
                maxactivefd = fd;
        FD_SET(fd, &activefds);

#ifdef HAVE_SIGNALED_IO
        init_socket_sig(fd);
#else /* HAVE_SIGNALED_IO */
        /*
         * set non-blocking,
         */
#ifndef XTP
#if !defined (SYS_WINNT)
#if defined(O_NONBLOCK)
#ifndef SYS_VXWORKS
        if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0) {
                syslog(LOG_ERR, "fcntl(O_NONBLOCK) fails: %m");
                exit(1);
```

```
                /*NOTREACHED*/
        }
#else
        /* VXWORKS uses ioctl and some non-compatibilities */
        if (ioctl(fd, FIONBIO, &on) == ERROR) {
                syslog(LOG_ERR, "ioctl(O_NONBLOCK) fails: %m");
                exit(1);
        }
#endif
#else /* O_NONBLOCK */
#if defined(FNDELAY)
        if (fcntl(fd, F_SETFL, FNDELAY) < 0) {
                syslog(LOG_ERR, "fcntl(FNDELAY) fails: %m");
                exit(1);
                /*NOTREACHED*/
        }
#else /* FNDELAY */
#if defined(VMS) /* VMS+UCX */
        if (ioctl(fd,FIONBIO,&1) < 0) {
                syslog(LOG_ERR, "ioctl(FIONBIO) fails: %m");
                exit(1);
        }
#else
/*Need non blocking I/O */
#endif /* VMS+UCX */
#endif /* FNDELAY */
#endif /* O_NONBLOCK */
#else /* SYS_WINNT */
        if (ioctlsocket(fd,FIONBIO,(u_long *) &on) == SOCKET_ERROR) {
                syslog(LOG_ERR, "ioctlsocket(FIONBIO) fails: %m");
                exit(1);
        }
#endif /* SYS_WINNT */
#endif /* XTP */
#endif /* HAVE_SIGNALED_IO */

        /*
         *  Turn off the SO_REUSEADDR socket option.  It apparently
         *  causes heartburn on systems with multicast IP installed.
         *  On normal systems it only gets looked at when the address
         *  is being bound anyway..
         */
#ifndef XTP
/*      if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
            (char *)&off, sizeof(off))!=0) {
                syslog(LOG_ERR, "setsockopt SO_REUSEADDR off fails: %m");
        }
*/
#endif /* XTP */
```

```
#ifndef XTP
#ifdef SO_BROADCAST
        /* if this interface can support broadcast, set SO_BROADCAST */
        if (flags & INT_BROADCAST) {
                if (setsockopt(fd, SOL_SOCKET, SO_BROADCAST,
                    (char *)&on, sizeof(on))) {
                        syslog(LOG_ERR, "setsockopt(SO_BROADCAST): %m");
                }
        }
#endif /* SO_BROADCAST */
#endif /* XTP */

#if !defined(SYS_WINNT) && !defined(VMS)
#ifdef DEBUG
        if (debug > 1)
                printf("flags for fd %d: 0%o\n", fd,
                    fcntl(fd, F_GETFL, 0));
#endif
#endif /* SYS_WINNT || VMS */

        return fd;
}
```