Towards a Distributed Real-Time System for Future Satellite Applications

A. Rozendaal

Thesis presented in partial fulfilment
of the requirements for the degree of

Master of Science in Electronic Engineering

at the

University of Stellenbosch.

Leader: Prof. J.J. du Plessis

- December 2003 -

DECLARATION

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

ABSTRACT

The Linux operating system and shared Ethernet are alternative technologies with the potential to reduce both the development time and costs of satellites as well as the supporting infrastructure. Modular satellites, ground stations and rapid prototyping testbeds also have a common requirement for distributed real-time computation. The identified technologies were investigated to determine whether this requirement could also be met.

Various real-time extensions and modifications are currently available for the Linux operating system. A suitable open source real-time extension called Real-Time Application Interface (RTAI) was selected for the implementation of an experimental distributed real-time system. Experimental results showed that the RTAI operating system could deliver deterministic real-time performance, but only in the absence of non-real-time load.

Shared Ethernet is currently the most popular and widely used commercial networking technology. However, Ethernet wasn't developed to provide real-time performance. Several methods have been proposed in literature to modify Ethernet for real-time communications. A token passing protocol was found to be an effective and least intrusive solution. The Real-Time Token (RTToken) protocol was designed to guarantee predictable network access to communicating real-time tasks. The protocol passes a token between nodes in a predetermined order and nodes are assigned fixed token holding times. Experimental results proved that the protocol offered predictable network access with bounded jitter.

An experimental distributed real-time system was implemented, which included the extension of the RTAI operating system with the RTToken protocol, as a loadable kernel module. Real-time tasks communicated using connectionless Internet protocols. The Real-Time networking (RTnet) subsystem of RTAI supported these protocols. Under collision-free conditions consistent transmission delays with bounded jitter was measured. The integrated RTToken protocol provided guaranteed and bounded network access to communicating real-time tasks, with limit overheads. Tests exhibited errors in some of the RTAI functionality. Overall the investigated technologies showed promise in being able to meet the distributed real-time requirements of various applications, including those found in the satellite environment.

OPSOMMING

Die Linux bedryfstelsel en gedeelde Ethernet is geïdentifiseer as potensiële tegnologieë vir satelliet bedryf wat besparings in koste en vinniger ontwikkeling te weeg kan bring. Modulêr ontwerpte satelliete, grondstasies en ontwikkeling platforms het 'n gemeenskaplike behoefte vir verspreide intydse verwerking. Verskillende tegnologieë is ondersoek om te bepaal of aan diè vereiste ook voldoen kan word.

Verskeie intydse uitbreidings en modifikasies is huidiglik beskikbaar vir die Linux bedryfstelsel. Die "Real-Time Application Interface" (RTAI) bedryfstelsel is geïdentifiseer as 'n geskikte intydse uitbreiding vir die implementering van 'n eksperimentele verspreide intydse stelsel. Eksperimentele resultate het getoon dat die RTAI bedryfstelsel deterministies en intyds kan opereer, maar dan moet dit geskied in die afwesigheid van 'n nie-intydse verwerkingslas.

Gedeelde Ethernet is 'n kommersiële network tegnologie wat tans algemeen beskikbaar is. Die tegnologie is egter nie ontwerp vir intydse uitvoering nie. Verskeie metodes is in die literatuur voorgestel om Ethernet te modifiseer vir intydse kommunikasie. Hierdie ondersoek het getoon dat 'n teken-aangee protokol die mees effektiewe oplossing is en waarvan die implementering min inbreuk maak. Die "Real-Time Token" (RTToken) protokol is ontwerp om voorspelbare netwerk toegang tot kommunikerende intydse take te verseker. Die protokol stuur 'n teken tussen nodusse in 'n voorafbepaalde volgorde. Nodusse word ook vaste teken hou-tye geallokeer. Eksperimentele resultate het aangedui dat die protokol deterministiese netwerk toegang kan verseker met begrensde variasies.

'n Eksperimentele verspreide intydse stelsel is geïmplementeer. Dit het ingesluit die uitbreiding van die RTAI bedryfstelsel met die RTToken protokol; verpak as 'n laaibare bedryfstelsel module. Intydse take kan kommunikeer met verbindinglose protokolle wat deur die "Real-Time networking" (RTnet) substelsel van RTAI ondersteun word. Onder ideale toestande is konstante transmissie vertragings met begrensde variasies gemeet. Die integrasie van die RTToken protokol het botsinglose netwerk toegang aan kommunikerende take verseker, met beperkte oorhoofse koste as teenprestasie. Eksperimente het enkele foute in die funksionaliteit van RTAI uitgewys. In die algemeen het die voorgestelde tegnologieë getoon dat dit potensiaal het vir verskeie verspreide intydse toepassings in toekomstige satelliet en ook ander omgewings.

ACKNOWLEDGEMENTS

I would like to thank the following for their help, support and encouragement:

- My supervisor, Professor Jan du Plessis, for his advice during the course of the study.
- To Telkom SA Ltd. for the financial support.
- David Schleef for providing assistance in better understanding RTnet.
- Marius Theunissen, for proof-reading this thesis.
- My family and good friends for their words of encouragement and confidence in me. I sincerely appreciated the uplifting support throughout the course of the study.
- All honour to God almighty for His strength and wisdom.

LIST OF ABBREVIATIONS

ADA	Analogue-to-Digital, Digital-to-Analogue
ADC	Attitude Determination and Control
AEP	Application Environment Profile
AGP	Accelerated Graphics Port
AIST	Advanced Information System Technology
ANSI	American National Standards Institute
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
APM	Advanced Power Management
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
BEB	Binary Exponential Back-off
BIOS	Basic Input/Output System
BLAM	Binary Logarithmic Arbitration Method
BSD	Berkley Software Distribution
CA-BEB	Capture Avoidance Binary Exponential Back-off
CAN	Controller Area Network
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSMA-DCR	Carrier Sense Multiple Access with Deterministic Collision Resolution
DC	Direct Current
DIAPM	Department of Aerospace Engineering, Milan Polytechnic
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDF	Earliest Deadline First
EMI	Electro-Magnetic Interference
EOI	End of Interrupt
ESL	Electronic Systems Laboratory
FCS	Frame Check Sequence

FIFO	First-In-First-Out
FPGA	Field Programmable Gate Array
FST	Flight System Testbed
GNU	GNU's Not UNIX
GPL	General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
ICMP	Internet Control Message Protocol
IDE	Integrated Device Electronics
IDT	Interrupt Descriptor Table
IEEE	Institute for Electrical and Electronics Engineers
INET	Internet
I/O	Input/Output
IP	Internet Protocol
IPC	Inter-Process Communication
ISO	International Organisation for Standardisation
IT	Information Technology
KURT-Linux	Kansas University Real-Time Linux
LAN	Local Area Network
LCFS	Last-Come-First-Serve
LDCR	Laxity based Deterministic Collision Resolution
LEO	Low Earth Orbiting
LTSM	Latest Time to Send Message
LXRT	LinuX Real-Time
JPL	Jet Propulsion Laboratory
MAC	Media Access Control
MAT	Message Arrival Time
Mbps	Megabit per second
MD	Message Deadline
ms	millisecond
MSR	Model Specific Register
MTT	Message Transmission Time
MTU	Maximum Transmission Unit

MUP	Multi-Uni-Processor
NASA	National Aeronautics and Space Administration
NIC	Network Interface Card
OSI	Open Systems Interconnection
PASC	Portable Applications Standards Committee
PB-CSMA	Pre-emption based Carrier Sense Multiple Access
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCSMA	Predictable Carrier Sense Multiple Access
PDF	Portable Document Format
PE	Paper Empty
POSIX	Portable Operating System Interface Standard
RAM	Random Access Memory
RDMSR	Read Model Specific Register
RDTSC	Read Time Stamp Counter
RED-Linux	Real-time and Embedded Linux
RETHER	Real-time ETHERnet
ROM	Read Only Memory
RPC	Remote Procedure Call
RTAI	Real-Time Application Interface
RTHAL	Real-Time Hardware Abstraction Layer
RT-Linux	Real-Time Linux
RTnet	Real-Time Networking
RTToken	Real-Time Token
SMI	System Management Interrupt
SMP	Symmetrical Multi-Processor
SOF	Start-Of-Frame delimiter
SUNSAT	Stellenbosch UNiversity SATellite
SSTL	Surrey Satellite Technology Laboratory
TCP	Transport Control Protocol
TDMA	Time Division Multiple Access
TSC	Time Stamp Counter
TTP	Time-Triggered Protocol
UDP	User Datagram Protocol

μs	microsecond
USB	Universal Serial Bus
UTIME	Micro-Time
VT-CSMA	Virtual Time Carrier Sense Multiple Access
VTST	Virtual Time to Start Transmission
WRMSR	Write Model Specific Register

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Overview of the Satellite Environment

Satellites play a very important role in everyday life. They are used in a vast range of applications that include global positioning and navigation, remote sensing, military applications, communication and space science experiments. A short description of these applications with some examples is given in Table 1.

Table 1. Description of Various Satellite Applications.

Satellite Application	Description
Global Positioning and Navigation	A satellite based positioning and navigation system, such as the Global Positioning System (GPS), can be used on land, at sea and in the air. It is employed to for example track vehicles and to assists in the navigation of cars, ships and aeroplanes
Remote Sensing	It is the collection of environmental information using sensors hosted on Low Earth Orbiting (LEO) satellites. Usually cameras are used to collect images, but methods such as radar and infrared sensors are also used. Remote sensing has various applications in fields such as agriculture and oceanography: • **Agriculture** – To distinguish between different kinds of soil, drainage, organic content, nitrogen levels and more. The information can be used to improve yield and to reduce the wasting of valuable fertilisers and chemicals. • **Oceanography** – To determine the effect of the ocean on the environment, to analyse wave patterns, monitor marine life and more. The information is used to get a better understanding of both ocean life and behaviour.
Military Applications	Satellites are used in the air force, navy and military. Examples include the reconnaissance of enemy territory and detection of nuclear device detonations. In addition, satellites play important roles in the strategic positioning and navigation of armed forces and remote navigation of modern smart bombs.
Communication	Global telecommunication services, television transmission and the transfer of digital information are made possible with telecommunication satellites with high-speed data links.
Space Science Experiments	A variety of experiments are conducted using satellites to for example determine the effects of micro-gravity and radiation in space.

In the past, satellite projects that were undertaken became increasingly more ambitious, resulting in satellites that increased in size and functionality. Over the years the payload capacity of launch vehicles also increased and consequently the satellite sizes increased accordingly. However, in the early 1990's the space industry suffered huge financial setbacks with the failure of a number of multi-billion dollar satellite projects (e.g. the flaw in the Hubble Space Telescope's primary mirror (1990) and the loss of the Mars Observer (1993)). As a result, a paradigm shift occurred in the satellite industry (Jilla and Miller, 1995). The driving force behind the design of satellites today, is no longer the desired science or the most advanced technology, but rather the limited budget with which the (sponsored) organisations have to work.

Based on these **economical** factors, continued efforts are being made to develop more cost-effective satellites and support infrastructure, such as the operational ground station and development testbed. The Electronic Systems Laboratory (ESL) at the University of Stellenbosch started the SUNSAT (Stellenbosch UNiversity SATellite) project to conduct research and develop of affordable satellite technology. This project has included the successful development and in-orbit operation of SUNSAT-1, a remote sensing amateur micro-satellite (Milne et. al., 1999). The present study contributes to this research effort, by investigating alternative technologies in order to reduce development time and cost.

1.2 Techniques to Reduce the Cost and Development Time of Satellites

Traditionally satellites were designed using the conventional subsystem-by-subsystem design approach. Hereby, subsystems were separately designed and then integrated using an iterative process. Although this approach is proficient at accomplishing a specific mission, it can be extremely expensive and time consuming. Pre-flight satellite testing and verification are difficult, integration is complex and satellite subsystems cannot be utilised in other satellite designs without having to make significant modifications (Jilla and Miller, 1995). Consequently, the modern trend is to rapidly design satellites composed of modular subsystems, payloads and other components with standardised interfaces and utilising commercial and commercially derived hardware and software technologies (Casani and Thomas, 1995). These techniques and their associated benefits are briefly discussed.

1.2.1 Commercial Hardware Technologies

Satellites operate in the harsh environment of space where radiation exposure caused by cosmic rays[1], temperature fluctuations and other phenomena limit the operational life and performance capability of onboard electronic hardware. To compensate for these factors, highly reliable, radiation-hardened hardware technologies have been specially developed for satellite applications. These space-rated technologies, however, have several drawbacks. They are very

[1] These include *galactic cosmic rays* coming from outside the solar system, *anomalous cosmic rays* coming from interstellar space at the edge of the heliopause, and *solar energetic particles* associated with solar flares and other energetic solar events (Web reference: http://helios.gsfc.nasa.gov/cosmic.html).

expensive, have long lead times to purchase and are typically generations older and much less capable than more advanced Commercial-Off-The-Shelf (COTS) technologies.

As a result, there is a tremendous interest in the use of commercial and commercially derived hardware technologies in satellite designs. By utilising COTS technologies, more advanced capabilities can be added to satellites, component costs are reduced, development times are shortened, system power, weight and size requirements can be reduced, and component availability can be expanded. However, the two primary concerns are still the radiation and thermal conditions in space.

Currently two approaches are followed to qualify commercial hardware technologies for satellite insertion (Caldwell and Chau, 1994):

- Researchers are adapting commercial designs for use in space by modifying the commercial manufacturing processes to improve the radiation tolerance of commercial hardware. This adaptation usually results in only a minor loss of system performance. With these manufacturing techniques, radiation hardened components can be manufactured more affordably than traditional space-rated technologies. These technologies offer most of the previously mentioned benefits, but they are still relatively expensive and only a limited selection is available.
- Tests are performed on COTS hardware to determine if these components are reliable enough for the harsh environment of space (Sexton et. al., 1992). The components must have acceptable radiation susceptibility characteristics and be mechanically acceptable (pass thermal, shock, vibration and decompression tests). Significant cost reductions can result from the low individual part costs and reduced design time due to the increased capabilities and functional selections available. This is the most cost-effective approach provided that the tests can be performed inexpensively.

1.2.2 Rapid Prototyping Testbed

In response to the National Aeronautics and Space Administration's (NASA) philosophy to develop satellites "faster, better, cheaper", the Jet Propulsion Laboratory (JPL) of the California Institute of Technology established the Flight System Testbed (FST). The FST enables the development of small, high technology and low-cost satellites and their operation systems concurrently in an environment where rapid prototyping[2] of new satellite architectures and designs can be accomplished (Casani and Thomas, 1995).

The FST provides for simulated satellite development. Hereby, generic simulation models of individual satellite subsystems and payloads are developed that are customisable to meet the requirements for a specific satellite design. These models are used to create a virtual (modelled) satellite that provides valuable insight into the satellite performance prior to hardware integration. A dynamic simulator (using environmental software models) is used to provide sensor inputs and other events affecting the satellite. The FST also includes elements of the operation systems so that the entire system can be represented. Developed hardware subsystems (prototypes, engineering[3] and final flight unit) are interchangeable with their corresponding simulation models. Testing and verification comprise of hardware-in-the-loop real-time simulations. Besides development, the virtual satellite is also used for a pre-and post launch flight software and sequencing verification system (Casani and Thomas, 1995).

Having a rapid prototyping environment has several benefits that reduce costs and shorten development time:

- The environment allows concurrent development of flight and ground operation systems. These are both influenced by design changes and using this approach assists in the early identification and solving of overall (end-to-end) system problems. This approach also assists in developing and validating operation automation methods. Automation reduces operator workloads, which is not directly related to the payloads and/or mission goals (Siewert and McClure, 1995).
- It also enables the early resolution of individual satellite subsystem integration and system performance verification.
- Expandability of testbed capabilities offers support for both current and future mission requirements.
- Satellite subsystems can be re-used (i.e. the development environment facilitates technology inheritance from mission to mission).
- New technologies can be easily evaluated and integrated in current and future missions.

[2] ***Rapid prototyping*** is a design methodology that allows the designer to build a prototype, test and modify it, and through an iterative process complete the final system. A ***prototype*** is an executable version of a system that incorporates all the key elements of the final version but is incomplete in terms of functionality and robustness.

[3] The engineering unit is equivalent to the flight unit in form and function.

1.2.3 Modular and Flexible Satellite Architectures with Standardised Interfaces

Modularity refers to the ability to integrate alternate payloads, subsystems and components in the satellite architecture. It is influenced by both the system-level interfaces of the satellite subsystems[4] and the architecture (physical and logical framework) used to interconnect satellite subsystems. Modularity can be improved by using simplified system-level interfaces so that satellite subsystems can be easily connected, disconnected, replaced, swapped and/or upgraded. Choosing a suitable architecture can also reduce the inter-dependency of satellite subsystems and confine the size of the wiring harness.

Flexibility refers to the ease with which the satellite components can be integrated. The flexibility of satellite architectures is greatly improved by using standardised system interfaces and by having design continuity form current to future missions (i.e. using the same standardised interfaces for different missions). Integration is much easier if the interfaces between satellite components are both electrically and mechanically standardised i.e. conforms to formal, open standards that have been recognised by international bodies such as the International Organisation for Standardisation (ISO). Note that flexibility does not require all the satellite subsystem interconnections to be the same, just standardised.

Additional benefits associated with such architectures are:

- Using standardised interfaces that are proven in industry, reduce risk and development time. Commercial standards are well understood, proven designs are available and extensive testing and verification has been done. Development time is saved by being able to exploit a large number of knowledgeable people, rapidly prototype or produce new systems and re-use existing designs and software, such as the interface drivers.
- The time required for satellite assembly and final integration is significantly shortened.
- The architecture improves scalability. Multiple processors, peripherals, payloads and other components can be easily integrated as subsystems for a given mission. The addition of more computing power means that variation in subsystem and payload processing requirements can easily be accommodated.
- Subsystems can be duplicated for redundancy purposes.
- The architecture can utilise a tray-based structure for easier satellite assembly.

[4] Satellite subsystems, payloads and other components will be collectively referred to as "satellite subsystems" for the remainder of the thesis unless otherwise stated.

Note that satellite architectures designed and developed in the previously mentioned rapid prototyping environment are also required to be modular and flexible. As a result, most of the above mentioned benefits are mutual to those associated with having a rapid prototyping environment. Some of the standardised and space-qualified interfaces that the FST supports are RS232 and RS422 for serial point-to-point interconnections and MIL-STD 1553B (twisted pair) and 1773 (optical) bus inter-connections (Casani and Thomas, 1995).

Commercial and Local Area Network (LAN) technologies are also being used. An example is the UoSAT-12 satellite that was developed by the Surrey Satellite Technology Laboratory (SSTL) and launched in April 1999. The satellite uses Ethernet and Controller Area Network (CAN) busses for on-board data handling as illustrated in Figure 1. The Ethernet bus (IEEE Std. 802.3, 2000) connects the Earth Imaging subsystem with the primary onboard computers (both with commercial Intel 80386EX processors) and is used for high speed bulk data transfer. The CAN bus (ISO 11898, 1993) interconnects all the satellite subsystems with the primary and secondary (Intel 80186 with extensive space heritage) onboard computers and is intended for real-time telemetry and telecommand (Ward and Sweeting, 1999).

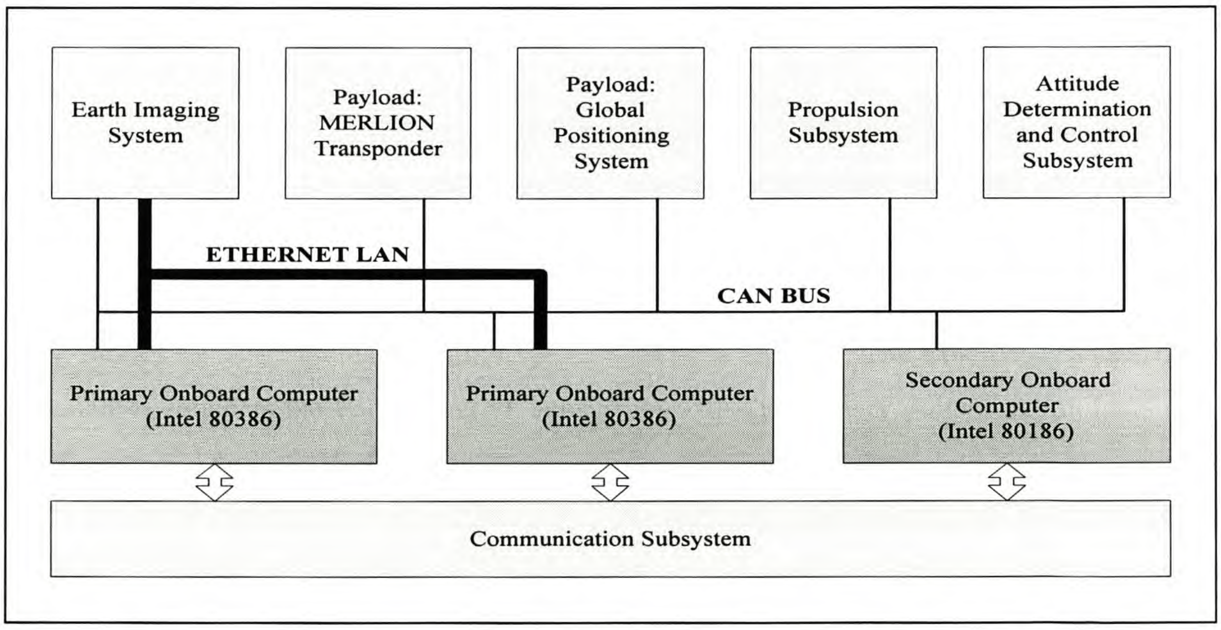
Earth Imaging System
Payload: MERLION Transponder
Payload: Global Positioning System
Propulsion Subsystem
Attitude Determination and Control Subsystem
ETHERNET LAN
CAN BUS
Primary Onboard Computer (Intel 80386)
Primary Onboard Computer (Intel 80386)
Secondary Onboard Computer (Intel 80186)
Communication Subsystem

Figure 1. Onboard Data Handling between the Subsystems of the UoSAT-12 Satellite.

1.2.4 Commercial Operating Systems

Traditional onboard satellite software consisted of tightly integrated, highly complex, customised code that was designed for a specific mission. To reduce the complexity, custom real-time operating systems running mission specific satellite applications were developed, but this approach also proved not to be ideal. In both cases, software development was very expensive, time-consuming and proven software could seldom be re-used.

To overcome these shortcomings, two approaches are currently being followed. The first is based on the design of a highly configurable and extendible satellite operating system that can be re-used for multiple missions (Batra, 1997). The initial development of the operating system is still very expensive, but the ability to re-use it will result in significant future savings in cost and development time.

The use of commercial hardware technologies onboard satellites led to the second approach, which is the adaptation of commercially available, embeddable, real-time operating systems (Stakem, 2002). This approach is the most effective by reducing costs and development time in the following ways:

- Using commercial operating systems reduce the development time that would otherwise have been required to develop an operating system. Developers can now focus resources on the development and verification of satellite applications.
- The operating system can be used in both the development and operations environment. Desktop personal computers (PC's) can be used as satellite simulators to develop and test application software for satellite subsystems.
- Having a common operating system layer for the satellite, testbed and ground operation station assists in concurrent development of flight and operation systems (i.e. easy incorporation of common software/functionality).
- Commercial operating systems are highly configurable and can be re-used in current and future missions. Established operating systems are being actively developed and maintained to continually support new processors, peripheral devices and future feature enhancements; and to maintain compliance to standards (e.g. the Portable Operating System Interface Standard (POSIX)) and other applications (e.g. Graphical User Interfaces (GUI)).
- Some operating systems support multiple user access management to the satellite (even simultaneously). This is useful for example, when satellites carry commercial payloads. It can be more cost and time efficient to allow payload providers the freedom to operate their own payloads instead of having to train satellite operation personnel to operate a specific payload (Batra, 1997).

1.3 Rationale for Using the Linux Operating Environment

The Linux operating system is a recent (released in 1991), free UNIX-like operating system that offers excellent system stability, scalability and efficiency (Pitts and Ball, 1998). It is widely used in industrial and embedded computing environments and has the potential to be used for numerous other applications. Currently, Linux is being considered to be more than a niche operating system, but not quite an operating system for the masses (Hubley and Lubrano, 2001). Linux has proven to be a very reliable, highly customisable, low-cost, desktop-based operating system that can be used for software development, networking, application serving and various other applications. In the satellite environment, Linux is currently being used as a common operating system platform for various applications related to software development and verification, mission planning and control, data processing and storage. For example, PREDICT, a Linux based, open source, multi-user satellite tracking and orbital prediction program was used in the SUNSAT-1 ground station (Magliacane, 2002). Linux is also employed to support web-based information systems to provide project status information for public interest and to facilitate collaboration between geographically dispersed researchers. The modular and highly scalable design of the operating systems also makes it ideal for applications in the embedded environment (Gillen and Kusnetzky, 2001).

In addition, the Linux operating system also has great potential to allow and/or complement the previously mentioned techniques that reduce satellite development time and cost:

- **Commercial hardware technologies** - The embeddable Linux operating system supports almost all the primary near term onboard processors that are being adapted from commercial technologies for use in space. These are derived from architectures such as the Motorola PowerPC, SPARC, MIPS, Intel and the Intel ARM families (Stakem, 2002). Provided that Linux is chosen for the flight operating system, it will promote the use of commercial and commercially derived hardware technologies.
- **Rapid prototyping testbed** – The availability of a vast range of development tools, prototyping (with and without hardware-in-the-loop simulation) and other applications, and comprehensive technical support makes Linux an ideal operating system platform for both rapid prototyping and the development of rapid prototyping testbeds. For control, data acquisition and hardware-in-the-loop simulations, Linux supports a wide range of digital and analogue sampling cards. The operating system also offers real-time support through kernel extensions and modifications. A recent addition is an application that generates real-time code from simulation models developed under MATLAB (simulation toolset), Simulink

(interactive modelling tool) and Real-Time Workshop (C code generator) (Quaranta and Mantegazza, 2001).

- **Modular satellite architecture with standardised interfaces** – Linux supports most open standards, protocols and standardised interfaces, thereby promoting modular and flexible satellite architectures. Having a LAN technology such as Ethernet or CAN onboard a satellite will significantly simplify the interfaces between subsystems and Linux can easily provide the software support for such architectures (Stakem, 2001).
- **Re-usable onboard operating systems** - The **FlightLinux** project aims to demonstrate the use of a customised, embedded distribution of the Linux operating system onboard satellites. It is a research effort funded by the NASA Advanced Information System Technology (AIST) program. The project progressed to a NASA Technology Readiness Level greater than 5, which indicates component and/or breadboard verification in a relevant environment (Stakem, 2002). The use of re-usable, open source operating systems such as Linux onboard satellites is still in its infancy, but it is believed to have the potential to revolutionise the current computing techniques used onboard satellites. An issue that still needs to be resolved is the general perception that open source is not mature versus the counter argument that such software can proof more stable than the commercial equivalent.

1.4 Problem Statement

Several systems in the satellite environment have real-time requirements that need to be guaranteed[5]:

- **Satellites** are basically robots operating in the hostile environment of space. The satellite subsystems perform various mission-critical and housekeeping functions with hard and soft real-time requirements. Examples of satellite applications/functions with hard real-time requirements include the timely execution of mission telecommand, attitude determination and control, onboard housekeeping and initiation of error handling procedures if required, onboard clock maintenance and telemetry formatting. Examples of applications/functions with soft real-time requirements include the thermal control of the physical satellite structure, data logging and memory management, which includes bulk memory scrubbing (scanning memory to find defects).
- **Ground stations** have hard real-time requirements for the accurate tracking of satellites. These requirements include accurate timekeeping and precise control of the satellite dish

[5] Most of these systems are distributed and as a result also have distributed real-time requirements.

and/or antenna array. Soft real-time functions include the timely execution of mission commands and the downloading of data.

- **Rapid prototyping testbeds** also have hard real-time requirements for mixed hardware and software testing (i.e. data acquisition, monitoring and analysis of satellite subsystems and system parameters in hard real-time) and verification (i.e. real-time hardware-in-the-loop simulations, including the ability to simulate faults).

Linux has limitations when real-time performance is required. **The Linux operating system was not designed to be a real-time operating system**. Linux is a general-purpose operating system that aims to provide acceptable average performance (i.e. all tasks are allocated a fair share of the system resources). The lack of real-time support was also a recognised shortcoming of FlightLinux that still needs to be addressed (Stakem, 2002). Fortunately several real-time extensions and modifications to the operating system are available that can be used to meet the real-time requirements. These real-time Linux operating systems were all developed to support single and symmetrical multi-processor (SMP) architectures; no real-time inter-processor communication protocols are available.

To accommodate distributed real-time requirements, the Linux operating system must also provide support for a real-time communication network. Currently, the most popular and widely used commercial networking technology is Ethernet, a local area information network specification that is also extensively supported under Linux. Its widespread use is mainly due to the proliferation of the Internet across the world and the networking of millions of PC's in both corporate and academic environments. Ethernet is also becoming a viable networking solution for embedded applications due to the increased availability of stable, mature and low-cost hardware and software, and the advantages it offers over proprietary network technologies (such as open standards and higher data transmission speeds) (Webb, 1998). However, traditional **Ethernet was not designed for real-time communication**. Modifying Ethernet for real-time communication can offer several benefits to the satellite environment:

- **Satellites** that have an Ethernet network onboard for non-real-time data handling can utilise the same network for real-time command and control communication. The integration of both real-time and non-real-time communication can provide advantages in weight, power, signal routing, reliability and overall satellite subsystem and payload integration. Ethernet can also be modified to support real-time communication and to switch between real-time and non-real-time modes of operation, trading off data throughput for network predictability.

For example, if the redundant real-time CAN network onboard UoSAT-12 should fail, the Ethernet network could serve as a backup by switching to a mode that supports real-time communication.

- The existing Ethernet local area network infrastructure can also be utilised in both the **ground station** and the **rapid prototyping testbed** for various distributed real-time applications. A single Ethernet network segment can be isolated for real-time operation or the real-time Ethernet network can be extended over multiple dedicated Ethernet networks.

1.5 Research Objectives

The research objectives for the thesis are threefold:

- The first objective is to investigate the open source, hard real-time extensions and modifications available for the Linux operating system. Both hard and soft real-time implementations exist, but hard real-time systems have the strictest design requirements and will therefore be the focus of the investigation.
- The second objective is to design a real-time communication protocol for a single segment, shared Ethernet network. Standard Ethernet cannot guarantee real-time communication and will have to be modified. Various methods have been proposed in the literature to improve the timing behaviour of Ethernet networks. A real-time Ethernet protocol will be designed based on the investigation conducted and the design constraints of the targeted architecture and satellite applications.
- The third objective is to integrate and verify the real-time communication protocol with a suitable open source, hard real-time implementation of the Linux operating system. The communication system is a critical resource of any distributed system. The addition of a hard real-time communication protocol will contribute towards the development of a Linux–based distributed hard real-time operating system that can potentially be use for various future satellite applications.

1.6 Thesis Outline

The next chapter provides an overview of real-time architectures, real-time operating systems and real-time communication. These are the main fields of research in distributed real-time systems that the thesis addresses. The basic real-time concepts and definitions used are also presented here.

The design methodology and system architecture of an experimental distributed real-time system is described in Chapter 3. The design is based on the utilisation of the Linux operating system and Ethernet networking technology.

Chapter 4 focuses on the Linux operating system. The chapter provides an overview of the operating system and discusses the limitations for real-time support. Various open source techniques to improve the real-time performance of Linux are then presented. The chapter concludes with a comparison and selection of a suitable hard real-time solution.

The Ethernet network technology and various techniques to improve the real-time communication performance are discussed in Chapter 5. A suitable technique is chosen to design and implement a real-time Ethernet communication protocol for the real-time Linux solution selected in the previous chapter.

Chapter 6 discusses the design and implementation of a software-based real-time protocol for Ethernet networks. The RTToken protocol implements a timed token bus for predictable and collision free transmission medium access control to a single-segment shared Ethernet network.

The overall implementation of the experimental distributed real-time system is described in Chapter 7. This chapter also discusses the experiments and measurements made to analyse and evaluate the performance of the distributed real-time system.

In the final chapter, conclusions are drawn and suggestions for further research are made.

2 DISTRIBUTED REAL-TIME SYSTEMS

This chapter provides an overview of distributed real-time systems with specific focus on current real-time fields of research that the thesis addresses. These are real-time architectures, real-time operating systems and real-time communication. The basic real-time concepts and definitions are presented in an informal manner to assist readers not acquainted with the area of research. Readers that are well acquainted with distributed real-time systems can refer to this chapter to determine which definition is used for each term. Concepts and definitions are presented in bold.

2.1 What is a Real-Time System?

In many applications there are strict requirements on the timing behaviour of the system. The systems that support real-time applications and ensure that the timing requirements are met are called real-time systems. More formally, a **real-time system is defined as a system whose correctness depends not only on the logical results of computation, but also on the time at which these results are produced** (Stankovic and Ramamritham, 1988). In other words, a real-time computation is wrong if the results are generated at the wrong time. To understand how the timeliness of a computation can be part of its correctness, consider the following cases:

- A navigation system computes the current position of a ship in motion. If the result is not computed within a certain time interval, the computed position is considered wrong i.e. beyond the acceptable margin of error. The permissible margin of error should be part of the system specification.
- A control system processes a sensor signal that arrives periodically. If any instance of this signal is dropped or not processed completely, for example when the processing of one instance was not completed before the arrival of the next instance, an error may occur.
- A panic signal in a nuclear reactor must be effectively responded to within a specified time interval or damage to life and/or property may result.

A general misconception of real-time systems is that they only need to be fast (Stankovic, 1988). The objective of fast computing is to minimise the average response time of a set of tasks. This does not guarantee that the timing requirements of each task will be met. A real-time system must have response times that are constraint and thus predictable. **Predictability**, not speed, is the foremost goal in real-time system design. Predictability means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made, for example, concerning failures and workloads (Stankovic and Ramamritham, 1990). Fast computing is helpful in meeting stringent timing requirements.

The complexity of real-time systems range from simple command and control of laboratory experiments (using for example simple micro-controllers) to highly sophisticated complex and distributed systems such as national air traffic control and space exploration projects. Other examples of real-time systems include process control systems, avionics tracking systems, intensive care monitoring and multimedia and communication applications. When the architecture of any of these real-time systems consists of a collection of interconnected processors, it is called a **distributed real-time system**.

2.2 Classification of Real-Time Systems

Real-time systems can be divided into two main categories. The distinction is based on the consequences that arise from not meeting the system's associated time constraints. These time constraints are normally expressed as **deadlines**, i.e. the latest instant at which a result must be produced (Kopetz, 1997). In **soft real-time systems** failure to meet deadlines leads to system performance degradation, not system failure. For example, in a video conferencing system it is desirable that frames are not missed. If it occasionally happens, the system performance will still be tolerable. Sometimes resource-inadequate solutions that will not be able to handle the rarely occurring peak-load scenarios are accepted on economic arguments.

In **hard real-time systems** failure to meet deadlines leads to system failure. Typical applications are in time critical environments such as robot-control systems and telephone switching systems. Hard real-time systems can also involve human lives, such as aeroplane flight control systems and patient monitoring systems. In these systems, if a deadline is missed, it could have catastrophic results. **Firm real-time systems** are defined to include those hard real-time systems where some low probability of missing a deadline can be tolerated (Laplante, 1997).

Hard real-time systems are designed to meet response-time constraints under all specified situations, even if they occur very rarely. The safety of commercial hard real-time systems must usually be demonstrated through a certification agency. The same design approach is followed for firm real-time systems, but a low probability exists that time constraints won't be met. Soft real-time systems do not have catastrophic failure modes and therefore a less rigorous approach to their design is often followed.

2.3 Components of Real-Time Systems

A real-time system consists of a **controlling system** and a **controlled system** (Kopetz, 1997). The controlling system consists of a human operator system and a real-time computer system. If the real-time computer system is distributed, it consists of a set of computers interconnected by a real-time communication network. The controlled system can be view as the environment with which the controlling system interacts. For example, in a satellite ground station, the dish that tracks the satellite is the controlled system and the computer and human interfaces that control the dish and monitor the tracking are the controlling system.

Two interfaces are defined for a real-time system. The **instrumentation interface** consists of sensors and actuators and interface between the real-time computer system and the controlled system. The sensors provide the controlling system with information about the current state of the environment. The actuators realise the actions computed by the controlling system. The **man-machine interface** consists of input (e.g. keyboard) and output (e.g. display) devices that interface between the human operator and the real-time computer system in the controlling system. In Figure 2, a block diagram of the components and interfaces of a real-time system is shown.

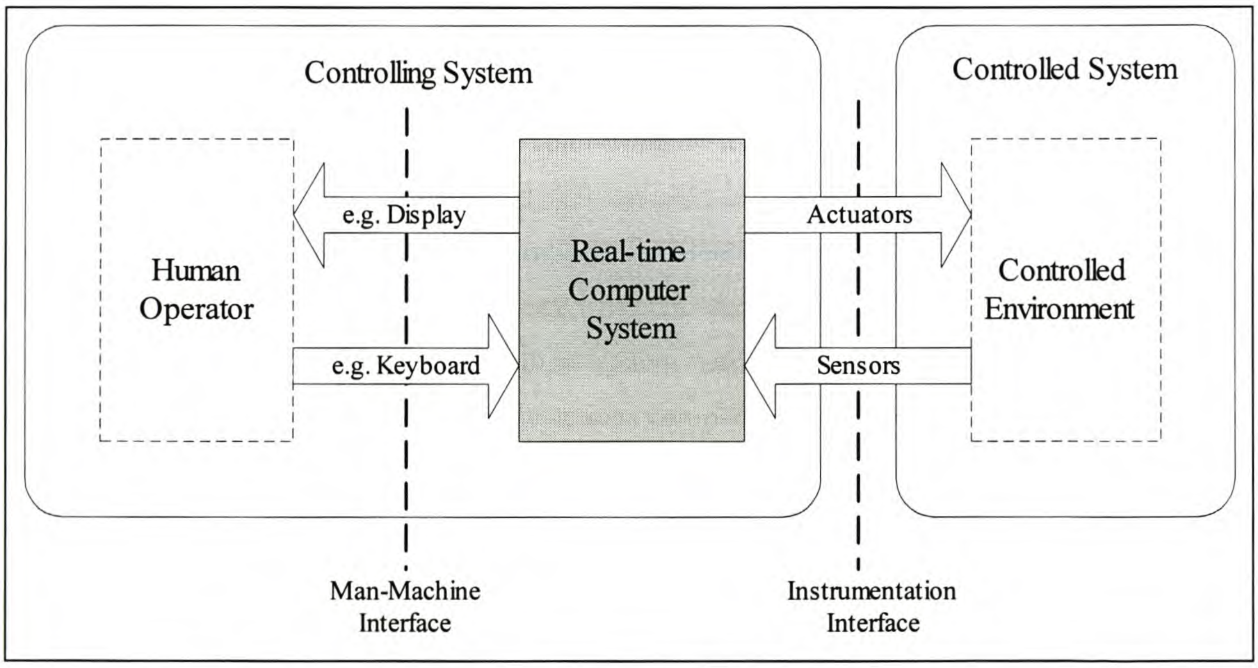

Figure 2. The Different Components of a Real-Time System.

The human operator uses the man-machine interface to perform changes to the operational configuration of the real-time computer system or for control if the system is not autonomous. In the latter case human experts make the high-level control decisions. These decisions are also based on sensor information, but it is the operator's task to determine whether a new calculation makes sense and can be applied to the controlled environment. This form of control is called **open-loop control**. The current trend however, is to replace human experts with software and thereby making real-time systems fully autonomous. The reason is to reduce operation cost and to implement sophisticated control techniques with response time requirements beyond human capabilities. Automation is possible because of the increased reliability of computers and the improvement of system models. When the operator is taken out of the control loop the form of control is called **closed-loop control**. This closed feedback loop can be seen in Figure 2, formed by the real-time computer system, the actuators and the controlled environment in the forward path, and the sensors in the feedback path (no human operator).

2.4 Motivation for Distributed Real-Time Systems

With regards to the functionality of a real-time computer system, it makes no difference whether the architecture is centralised or distributed. However, the geographically distributed nature of some operating environments (e.g. a telecommunication network) requires the use of distributed real-time computer systems. In this case, the information generated in one location is often needed in another. There are also a number of potential benefits to using distributed real-time systems:

- By executing processes in parallel on different processors, system performance and response times can be improved. However, better performance cannot always be guaranteed. This is because of the additional communication overhead involved and the complexity of task partitioning and allocation (Tsai et. al., 1996).
- System resources are more easily available. For example, a real-time system might only be able to handle the requirements of a number of real-time processes on a single processor. If additional real-time processes need to be handled, distributed processing may be a solution.
- The complexity of large systems can be controlled. This can be accomplished by partitioning the real-time system into subsystems. This reduces the effort required to understand the operation of the system.
- Effective error-containment regions can be implemented to make systems more fault-tolerant. Detected errors (that resulted from subsystem faults) can be corrected or masked before corrupting the rest of the system. In centralised architectures it is difficult to implement clean error-containment regions because most system resources (e.g. processing) are shared. Kopetz (1997) defines every subsystem in a distributed real-time system as an

error-containment region with error detection being performed at the communication interface of each subsystem.

- The reliability of a real-time system can be improved. Every subsystem in a distributed real-time system represents a unit of failure and as a result fault tolerance can also be achieved by replicating subsystems. Subsystem failures will then be masked by the replications. Again, better reliability cannot always be guaranteed because of the complexity of task partitioning and allocation (Tsai et. al., 1996).
- Expensive resources can be shared between subsystems of a distributed real-time system resulting in system cost savings. This however, makes the implementation of clean error-containment area difficult.
- Finally, a distributed real-time system is easily adaptable for use in different applications. Subsystems or nodes can easily be added or removed.

As a result, distributed real-time systems are used in a number of applications. In some cases, large computers are involved together with network systems of very high bandwidth. In other cases, several small microprocessor-based nodes are used, interconnected by simpler networking solutions. One area of application that involves both these cases is in the development and operation of satellites.

2.5 Achieving Distributed Real-Time Performance

To achieve distributed real-time performance requires an integrated solution of different aspects related to real-time systems. These aspects are categorised according to the areas of real-time research and include:

- Real-time formal methods, specification, toolset, design and analysis.
- Real-time architectures and fault tolerance.
- Real-time operating systems.
- Real-time scheduling algorithms.
- Real-time communication.
- Clock synchronisation.
- Programming languages.
- Verification, monitoring and debugging.
- Real-time databases.
- Real-time Artificial Intelligence.

These areas are actively researched and an extensive amount of literature is available. Also, various reference textbooks on real-time systems are available that provide comprehensive coverage of these areas of research (Kopetz, 1997, Laplante, 2000; Laplante, 1997; Levi and Agrawala, 1990; Tsai et. al., 1996; Wellings and Burns, 1996). For the purpose of the thesis a short discussion of distributed real-time architectures, real-time operating systems and real-time communication are provided in the following sections. The other topics fall outside the scope of the research, although some parts might overlap.

2.5.1 Distributed Real-Time Architectures

A distributed real-time computer system consists of a set of computing nodes[6] interconnected by a real-time communication network (Kopetz, 1997). Each node is a self-contained computer system with its own hardware (which includes the processor, memory and communication interface) and software (operating system and application programs). A distributed system (sometimes called a **multicomputer**) is not a multiprocessor, although nodes in a distributed system can be multiprocessors (Stankovic, 1988). **Multiprocessors** have shared memory and multicomputers have distributed memory. As a result co-operating processors in multiprocessors are tightly interconnected as a single computer. In distributed systems co-operating processors are loosely connected with a communication network and communication is based on message passing (Tsai et. al., 1996). A logical presentation is illustrated in Figure 3.

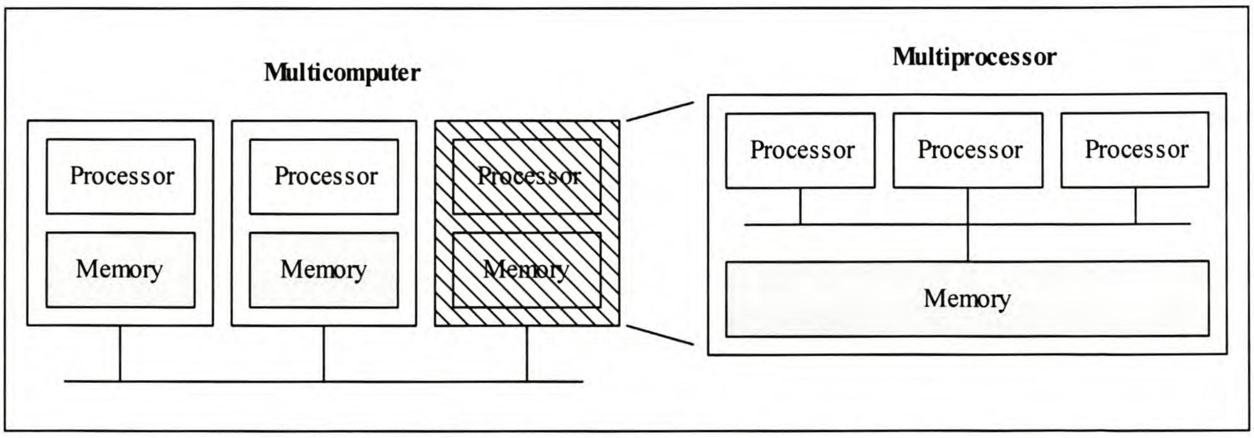

Figure 3. Distributed Computer System (Multicomputer) versus Multiprocessor.

The architecture of a distributed real-time system can be classified as **homogeneous** or **heterogeneous** (Tsai et. al., 1996). In a homogeneous architecture, all the nodes have the same

[6] Note that subsystems connected to a communication network are referred to as **nodes**.

hardware architecture and supporting software. Heterogeneous architectures on the other hand, have nodes with different hardware architectures and/or software. For example, nodes can support different general-purpose processors, microprocessors, microcontrollers, Digital Signal Processors (DSP) or even dedicated hardware implemented with Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA).

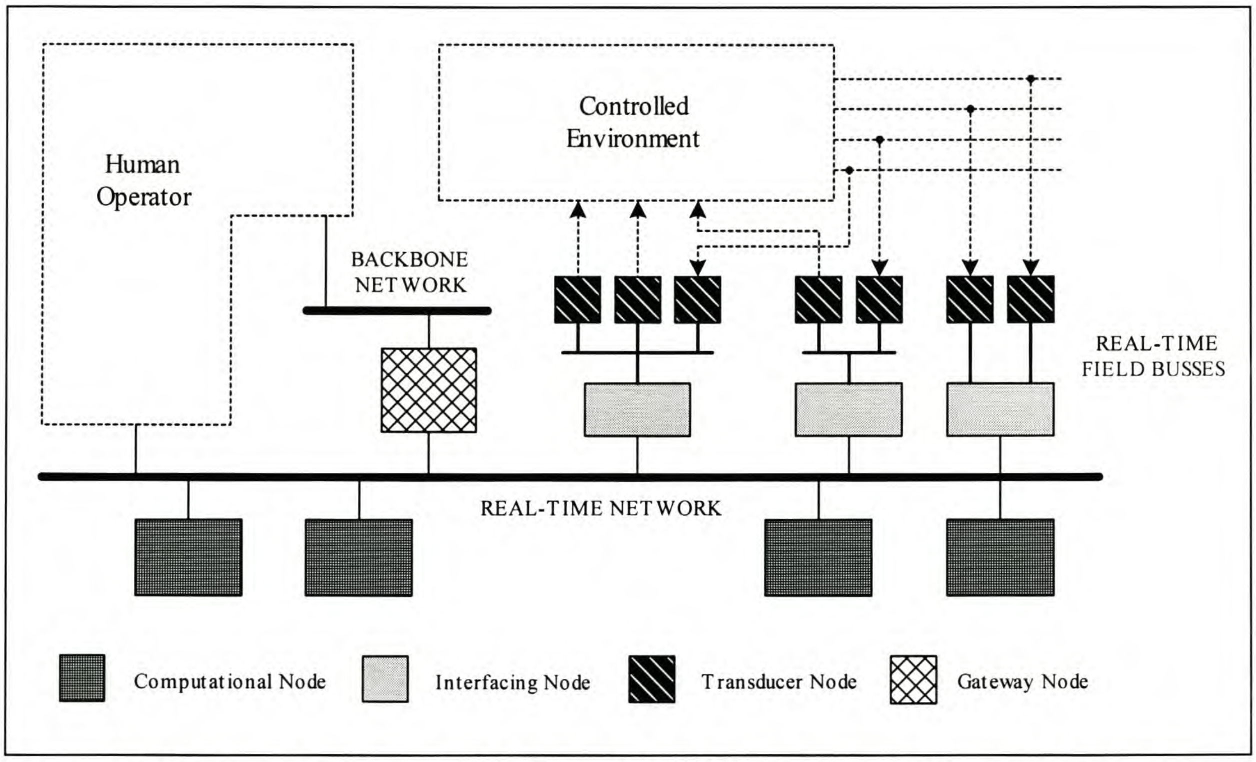

Figure 4. Example of a General Distributed Real-Time Computer System Architecture.

In most real-time applications, the distributed real-time system has three different types of communication networks that interconnect four node types (Kopetz, 1997a). These are shown in Figure 4 and briefly discussed below:

- The **computational nodes** perform some computational function in the system.
- The **interfacing nodes** have interfaces to both the real-time communication network and one or more real-time field busses.
- The **transducer nodes** represent field devices such as sensors, actuators, controllers and regulators. These nodes supported standard interfaces to the real-time field busses.
- The **gateway nodes** connect one computational cluster to another.
- The **real-time network** is the core of the distributed real-time system and allows reliable and predictable message transmission between the computational, interfacing and gateway nodes in the cluster. For better fault tolerance, the network must support both node and communication network replication. The network must also be able to provide services for

distributed clock synchronisation and node failure detection. To prevent a single point of failure, the network should be decentralised (i.e. based on distributed network control).

- The **real-time field busses** are low-cost, serial, digital bus (point-to-point and multi-point) networks that provide communication among field devices. Fault tolerance can be achieved by duplicating field devices and by using independent field busses to connect to different interfacing nodes.
- The purpose of the **backbone network** is to exchange non time-critical information (such as task schedules, logged status reports and more) between the real-time cluster and other data-processing systems.

There are a number of different alternatives to the topology and physical media of the real-time communication networks (field busses and real-time network). The network topology can be shared such as a **bus** or **ring**, or have multiple connections such as a **mesh** or **star** network. The physical medium can be copper (which is inexpensive), optical cable (which has Electro-Magnetic Interference (EMI) immunity) or wireless (for interconnecting inaccessible and moving devices). Both the physical communication media and network topology influence the cost of adding additional nodes, the ability to modify the topology, the network reliability, the complexity of communication protocols required, the data throughput rate, transmission delays and broadcast capabilities (Tsai et. al., 1996).

The general real-time distributed architecture also applies to the satellite environment. An example is the typical distributed satellite architecture. Field busses onboard the satellite connect different sensors (e.g. sun sensors) and actuators (e.g. reaction wheels) to the Attitude Determination and Control (ADC) subsystem which is both an interface and computational node of the satellite. The ADC subsystem is interconnected to other subsystems (such and the Flight Control and Data Handling subsystem) with a real-time communication network. The onboard communication subsystem is basically a gateway node that connects the satellite to one or more ground stations (other computational clusters) through wireless communication (representing by the backbone network).

2.5.2 Real-Time Operating Systems

The real-time operating system must provide predictable services to application tasks to meet the functional, fault tolerant and timing requirements of the real-time system specification. Time-sharing operating systems strive to provide an acceptable average performance and as a result the architecture of these operating systems cannot guarantee predictability. The essential real-time operating system services are summarised in Table 2 (Kopetz, 1997). Note that the **kernel** is the central part of an operating system that provides these essential services as required by other parts of the operating system and application tasks.

Table 2. Essential Real-Time Operating System Services.

Operating System Services	Description
Task Management	Tasks require resources for execution and the operating system is concerned with the provisioning thereof. In time-triggered systems, the resources can be computed off-line, based on a pre-runtime analysis of requirements. In event-triggered systems, resource requirements and availability vary at run-time and are to be assessed dynamically. The operating system can also support task pre-emptibility depending on the way tasks are scheduled.
Memory Management	In a real-time operating system, virtual memory policies (page fault and page replacement) are not used. Memory is pre-allocated to tasks, requiring knowledge of the maximum memory requirement of each task.
Time Management	Mechanisms are required for measuring the time instants at which particular events occur and the duration of time intervals between events. This problem can be adequately dealt with by a time reference of specified accuracy. In distributed real-time systems, clock synchronisation is therefore essential and can be provided by the operating system if the service is not part of the communication system.
Inter-process Communication	Inter-Process Communication (IPC) refers to a well-structured way of communication between concurrently executing tasks (Tanenbaum, 1992). Information can be exchanged directly (message passing using first-in-first-out (FIFO) queues and mailboxes) or indirectly (memory sharing). Shared memory is a common block of memory that can be accessed asynchronously by any task in the system. To avoid data inconsistencies with shared memory, mutual exclusion is enforced on critical task sections using WAIT operations on semaphore variables. Semaphores are counters allocated and released by tasks.
Error Detection	The operating system can support error detection by double execution of tasks and comparing results, monitoring interrupts and monitoring task execution times. Periodic watchdog signals (heartbeat) are typically used to detect silent node failures in distributed systems.

Real-time tasks can be categorised as being periodic, sporadic or aperiodic. **Periodic** tasks execute within regular time intervals and are characterised by a worst-case execution time, period and deadline. Periodic tasks usually have hard deadlines that coincide with the end of the current period. Sporadic and aperiodic tasks refer to a continuous series of task invocations with irregular (non-deterministic) arrival times. **Sporadic** tasks have hard deadlines and execute at arbitrary points in time, but with defined minimum inter-arrival times between two consecutive invocations. These tasks are characterised by a worst-case execution time, minimum inter-arrival time and deadline. If there are no constraints on the time of execution, the task is called **aperiodic** and is characterised by an arrival time, worst-case execution time and deadline.

Ramamritham and Stankovic (1994) divide real-time operating systems into three main categories. These are small, proprietary kernels (both commercially available and home-grown kernels), real-time extensions to commercial timesharing operating systems and research operating systems:

- **Small, Fast, Proprietary Kernels -** These kernels are typically used in small, embedded applications that require very fast and predictable execution. Small, fast, proprietary kernels can further be categorised into **home-grown kernels** and **commercial offerings**. Home-grown kernels are uniquely designed for specific applications. However, because these kernels are highly specialised, the development and maintenance costs are high. Affordable, commercial kernels are available that provide an alternative to developing and maintaining new home-grown kernels. Examples of commercial kernels include *QNX* (Hildebrand, 1992) and *VxWorks* (Wind River Systems, 1999). These kernels are very effective in small, embedded applications with well-defined environments, but due to limited kernel features it becomes increasingly difficult to guarantee predictability in larger, more complex applications. There have been efforts to overcome this limitation by increasing the scalability of these kernels. Trade-offs can now be made between systems size, performance and functionality depending on the application (Ramamritham and Stankovic, 1994). Recent versions of QNX for example, can scale from a small kernel (100 Kbytes) to fit in read only memory (ROM), to a full-featured multi-machine operating system with all the standard UNIX features (QNX, 2001).
- **Real-time Extensions to Commercial Timesharing Operating Systems** - Another way of developing real-time operating systems is by adding a real-time extension to an existing commercial operating system. Examples of these systems are *RT-UNIX* (Furht, 1991), *RT-Linux* (Barabanov, 1997), *RT-MACH* (Tokuda, 1990) and the *RTX* extension for Windows (Carpenter et. al., 1997). Compared to the proprietary kernels, these real-time operating systems are generally slower and less predictable than proprietary kernels, but have a better software development environment, have greater functionality and facilitate portability with standardised interfaces. It should be noted that the underlying timesharing nature of commercial operating systems means that various implementation, system interface and other problems need to be solved during real-time conversion and that care must be taken at run time not to use non-real-time features that can influence the overall real-time performance. To overcome these problems, the current trend is to extend new implementations of UNIX operating systems that are based on microkernels[7] (Ramamritham and Stankovic, 1994) or to add a second real-time kernel that treats the timesharing kernel as the idle task (Barabanov, 1997).

[7] A microkernel is a small kernel that provides only core operating system services. High-level operating system functionality is provided by a number of optional, co-operating services.

- **Research Operating Systems** - The development of conventional real-time operating systems is still being affected by numerous misconceptions about real-time computing (Stankovic, 1988). Proprietary kernels for example, are generally not designed to be easily scalable and most real-time extensions still emphasise speed rather that predictability (Ramamritham and Stankovic, 1994). As a result, several research projects are being conducted to overcome such misconceptions and directly address timing and fault tolerant constraints. Research operating systems include *SPRING* (Molesky et. al., 1990), *MARS* (Kopetz et. al., 1989) and *MARUTI* (Levi and Agrawala, 1990; Saksena et. al., 1995).

Operating systems are also required to provide a standardised application interface to allow applications to be interoperable and source-code portable between different operating system platforms. The POSIX standards define such an interface. The original POSIX standard was based on UNIX and produced by the Institute for Electrical and Electronics Engineers (IEEE) and standardised by American National Standards Institute (ANSI) and the ISO (Gallmeister, 1995). Currently POSIX standards are maintained and further developed by the Portable Applications Standards Committee (PASC), which is part of the IEEE (PASC, 2002). In terms of satellite applications, POSIX compliance means that application software can be re-used for different missions, software can easily be ported between the development and flight environments, and common software components can be used for the satellite, ground station and testbed.

The POSIX standards are structured as a set of optional features and compliance only requires that the features that are implemented in an operating system be specified. The implemented features of operating systems (including embedded operating systems) vary even between different versions of the same operating system (Gallmeister, 1995). To certify POSIX conformance, operating systems and hardware platforms have to be tested. The POSIX standards include specifications for testing the conformance of hardware platforms and operating systems (POSIX 2003.x), but currently test suites are only available for POSIX 1003.1a and based on the C programming language (Obenland, 2001).

The demand for real-time operating system performance resulted in the release of several real-time extensions since the original POSIX.1 standard. The basis and real-time POSIX standards are summarised in Table 3 (PASC, 2002; Obenland, 2001).

Table 3. POSIX Standards for Real-Time Application Support.

Standard	Description
POSIX 1003.1a	**Basic operating system interface and features** – Defines the interface to/and basic operating system functions. This includes control signals, the file system, file attributes, file locking, file device management, device input/output (I/O), pipes, user groups, FIFO's, system database, and support for single or multiple processes and the C programming language.
POSIX 1003.1b	**Real-time extensions** – Specifies the system interfaces to support portability of applications with real-time requirements. During the development of this standard it was called POSIX.4 and now it is officially called IEEE POSIX Std 1003.1b-1993 (Gallmeister, 1995). The standard defines memory locking, memory protection (semaphores), shared memory, message passing, prioritised scheduling, real-time signals, improved IPC and timers.
POSIX 1003.1c	**Threads** – Specifies functions to support multiple flows of control (threads) within a process. This includes condition variables, mutexes, mutex priority inheritance and priority ceiling, priority scheduling and thread attributes and control.
POSIX 1003.1d	**Additional real-time extensions** – Defines additional real-time features such as device and interrupt control, execution time monitoring of processes and threads, sporadic server scheduling and timeouts on blocking (mutex lock) functions.
POSIX 1003.1j	**Advanced real-time extensions** – Defines advanced real-time functions including barrier synchronisation, nano-sleep improvements, persistent notification for message queues, read-write locks, spin locks and typed memory.
POSIX 1003.13	**Real-time Application Environment Profile (AEP)** – Defines four different profiles (Profile 51 to 54) for real-time systems. The profiles group the functions of existing POSIX standards to cater for systems such as embedded systems that have resource limitations and where all the POSIX features may not be required. The profiles are based on whether or not the operating system supports multiple processes and a file system.
POSIX 1003.21	**Distributed real-time** – Includes functions to support distributed real-time communication such as buffer management, implementation protocols, message labels and priorities, and both synchronous and asynchronous operations.

2.5.3 Real-Time Communication

Real-time communication is the accurate and time-constrained exchange of information between nodes in a distributed real-time system. A **real-time communication system** refers to the collective technology that enables real-time communication and consists of Network Interface Cards (NIC) or communication adapters, a network, communication protocols and services. Such a communication system is a critical component of a distributed system because it directly affects the timing behaviour of the overall system. The loss of communication could result in the loss of all global system services (Kopetz, 1997).

Traditional (general-purpose) communication systems are required to either maximise the throughput or to minimise the average delay of message transmission. Real-time communication systems have different requirements as summarised by Kopetz (1997):

- The main requirement is to provide a ***predictable*** communication service i.e. a bounded (guaranteed maximum) end-to-end communication delay and minimal jitter. The communication delay is the total time duration between a sending task ready to queue a message for transmission and a receiving task being able to access the message. It is composed of four different components (Tindell et. al., 1995):

1. **Generation delay** - The time required for the sender's task to generate and queue a real-time message.
2. **Queuing delay** - The time taken for the message to gain access to the communication network.
3. **Transmission delay** - The time taken for the message to be transmitted on the communication network.
4. **Delivery delay** - The time required for the destination processor to process the message before finally delivering it to the destination task.

- Real-time communication systems are also required to be ***flexible*** to different system configurations. Configuration changes should be accommodated without requiring any software modifications and re-testing of operational parts not affected by the changes. The ISO specified the Open Systems Interconnection (OSI) framework to standardise computer communication for increased interoperability and flexibility. The OSI reference model defines seven independent layers at which different communication functions are performed (Stallings, 1997). The framework can be used when implementing real-time communication systems provided that predictable timing behaviour can be guaranteed for all the protocols and services in all of the layers.
- Distributed systems are typically built by integrating well-specified and tested subsystems. Real-time communication systems are required to be ***composable*** to ensure system properties are maintained after system integration. Composability guarantees that different subsystem combinations will work correctly without having to redesign and test any of the validated subsystems.
- Real-time communication systems are required to ***detect and correct transmission errors*** without affecting the predictability of the communication system. Senders and receivers should be promptly notified of errors that couldn't be corrected. When the loss of communication also implies the loss of control, the intended receiver should be able to detect this condition and autonomously enter a safe state. For example, when communication is interrupted or lost, a controlled valve should close to avoid overflow.

For a specific application all or only specific requirements may be applicable. It should also be noted that there is a fundamental conflict between the requirement for flexibility and error detection. Flexibility implies that the behaviour of the real-time communication system need not be fixed. Error detection on the other hand, requires a fixed behaviour so that it can be compared with the expected behaviour (Kopetz, 1997).

A **real-time message** contains real-time data and relevant control information for the proper transmission thereof. It is transmitted over the network as a data frame (a sequence of continuous bits with a defined beginning and end). In some cases, large sets of real-time data are divided into a number of smaller parts (packets)[8] that are then separately transmitted as data frames. In this case, the time constraints associated with the real-time message are based on the real-time transaction (the exchange of more than one real-time packet to deliver a real-time message).

In distributed real-time systems it is common to find the regular exchange of real-time messages between the same sender and receiver(s). A sequence of such messages constitutes a **stream**. Depending on the time interval between consecutive messages, the stream can be periodic or aperiodic. **Periodic** streams have regular time intervals (or at least with low and bounded jitter). **Aperiodic** streams have no bounded guarantee on the time interval between messages. If it is possible to guarantee a minimum inter-arrival time the stream is **sporadic**.

Real-time messages must be properly scheduled for transmission to meet the time constraints of the real-time application. The design consideration is to either minimise the number of missed deadlines or to ensure that all message deadlines are met (Malcolm and Zhao. 1995). The type of scheduling depends on the network topology such as multiple-access (e.g. shared broadcast bus or ring) or point-to-point networks (e.g. mesh network). In multiple-access networks the Media Access Control (MAC) protocol schedules messages. The protocol arbitrates access to the network and determines what messages to transmit at any given time. The access control protocol therefore directly influences the response time of the communication system to requesting nodes (Kopetz, 1997).

The rate at which real-time messages are generated for transmission can be characterised as constant or variable as illustrated in Figure 5 (Aras et. al., 1994). **Constant rate** means that fixed sized real-time messages are generated at periodic intervals. As a result the network traffic is smooth without any bursts occurring. For example, sensors in a distributed real-time control system generate real-time messages periodically. **Variable rate** refers to either fixed sized real-time messages generated at irregular intervals or variable size real-time messages generated at regular or irregular intervals. For example voice sources generate bursts of traffic followed by periods of silence.

[8] A packet is a basic unit of transmission over a network.

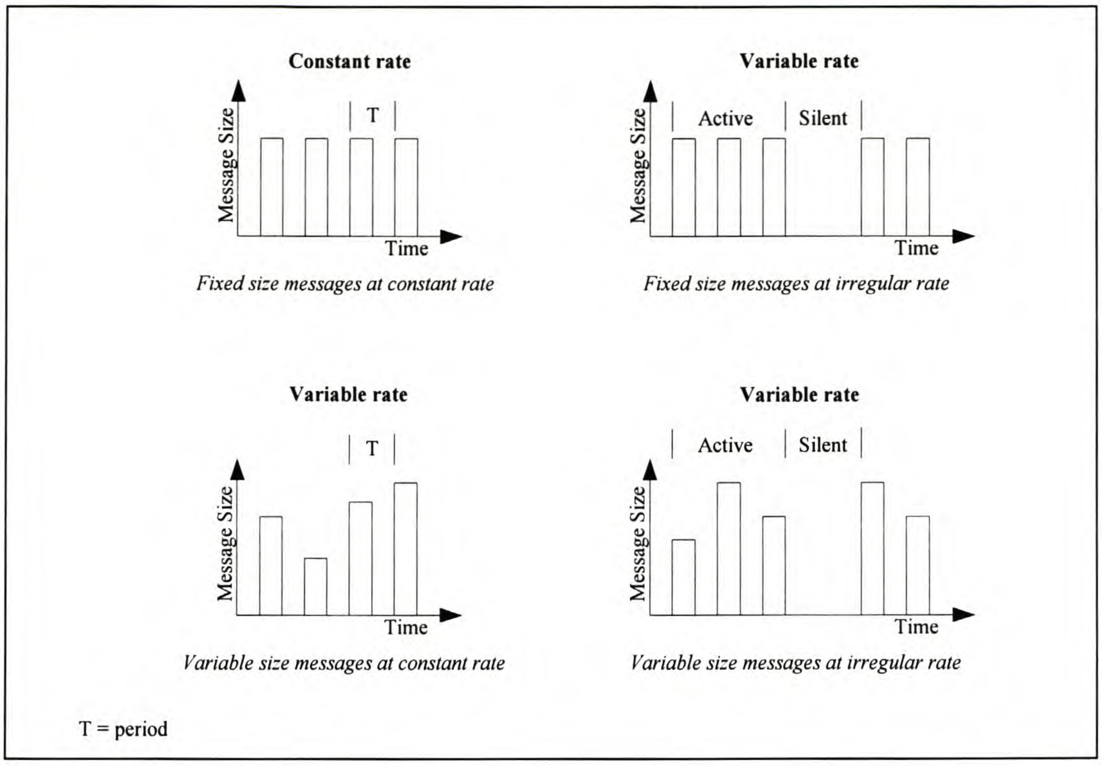

Figure 5. Illustration of Real-Time Message Generation at Constant and Variable Rates.

Two main strategies are used to transmit real-time messages (Malcolm and Zhao. 1995). The **guarantee strategy** attempts to guarantee ahead of time that the real-time messages will meet their deadlines. The guarantee may be given either during system design time or during system operation. Once a message is accepted for transmission, it is guaranteed to meet its deadline. In the **best-effort strategy**, message deadlines are generally met, but no guarantees are given. This strategy is used when there are insufficient network resources to meet all message deadlines or when the application can tolerate a certain amount of message loss (e.g. soft real-time digital video or voice). In distributed hard real-time systems, messages deadlines must be guaranteed.

Real-time communication can be **unicast** (one-to-one communication), or **multicast** (one-to-many communication), which includes **broadcast** (one-to-all communication) as a particular case. Multicasting is used in distributed applications where real-time information is simultaneously required by different nodes in the network (Kopetz, 1997). For example, satellite temperature information is required by two systems, the one is responsible for temperature control and the other for information monitoring and logging.

To conclude the discussion on real-time communication, various co-operation models are used in real-time communication systems (Thomesse, 1995). In the **master-slave** model, one node (master) controls one or mode nodes (slaves). Once the master-slave relationship is established, the direction of control is always from the master to the slaves. This model guarantees predictable real-time communication but is not very flexible since the configuration and maintenance of such a system are difficult and error prone (Kopetz, 1997). Elmenreich et. al. (2002) provides a plug-and-play solution to improve the flexibility. The **client-server** model allows each node that contains data relevant to other nodes in the network to act as the server (Thomesse, 1995). The server node responds to requests from the client nodes. A characteristic of this model is that the client starts all transactions. In a **peer-to-peer** model, all nodes have client and server capabilities and any node in the network can initiate a real-time transaction. When these models are used in real-time applications it is required to bind the response time of the server to any client request. However, these models allow for error detection because they are well suited for acknowledged data transfers. The **producer-consumer** model is also used in real-time applications. It consists of a producer node (containing data relevant to other nodes) that makes the data available to other consumer nodes in the network by transmitting it over the network (Jeffay, 1993). The consumer nodes identify and collect their relevant data. There is no explicit consumer request and the producer usually starts the transactions. This model inherently supports multicast communication.

3 DESIGN OF THE DISTRIBUTED REAL-TIME SYSTEM

In this chapter, the design of a distributed hard real-time system using the Linux operating system and Ethernet networking technology will be presented. The methodology and system architecture are described, followed by a discussion of the constraints imposed on the design.

3.1 Design Methodology

The objective of this research is to contribute to the development of a distributed real-time system with commercial and inexpensive technologies. The design focuses on two of the core components of a distributed real-time system, namely the real-time operating system and the real-time communication network (Kopetz, 1997). The real-time operating system allows predictable task execution at every system node and the real-time communication network guarantees reliable and predictable message transmission between the system nodes.

The design approach is to extend the functionality of a hard real-time operating system with a real-time communication mechanism. The methodology is also reflected in the objectives of the research and is shown in Figure 6.

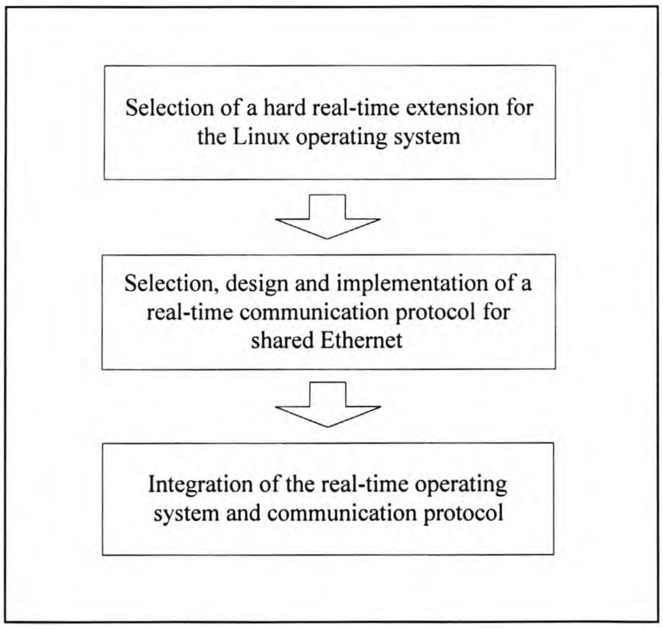
Selection of a hard real-time extension for the Linux operating system
Selection, design and implementation of a real-time communication protocol for shared Ethernet
Integration of the real-time operating system and communication protocol

Figure 6. The Structure of the Design Methodology.

Linux is a general-purpose operating system that was not intended for real-time applications. Various real-time extensions and modifications are available that have been or are still actively being developed. These are investigated and the most suitable open source, real-time implementation used for implementation of the distributed hard real-time operating system.

Standard (half-duplex) Ethernet, a popular and inexpensive LAN technology, cannot guarantee real-time communication. Various methods have been proposed in the literature to improve the timing behaviour of Ethernet for real-time applications. These are investigated to find the most suitable approach to establish hard real-time communication guarantees. A real-time communication protocols for Ethernet are designed and implemented, based on the outcome of the latter investigation and the design constraints enforced by the targeted applications.

The design is concluded with the integration and verification of the real-time communication protocol and hard real-time operating system. This is accomplished by extending an existing communication mechanism (only for localised real-time tasks) of the operating system with the real-time communication protocol. With this approach, distributed real-time tasks can communicate with minimum changes the communication interface that developers are accustomed to.

3.2 System Architecture

To meet the requirements of various real-time applications in the satellite development and operation environment requires a distributed real-time system architecture that's centralised, flexible, heterogeneous, modular and scalable. These architectural characteristics are explained:

- **Centralisation -** provides the capability of having the computational nodes of the distributed system either located in one area or remotely connectable. This allows for geographically distributed development and operation, and collaboration between remotely situated developers and researchers.
- **Flexibility -** allows the re-use of the system architecture in various applications and easy migration of technologies from rapid prototyping and validation testbed to satellites and ground systems. The onboard satellite architecture can now evolve over mission life cycles and across missions.
- **Heterogeneous architectures** – these architectures are applicable in the satellite environment where distributed testbeds, ground stations and computing environments are based on interconnected personal computers and workstations with different processing hardware and application software. Satellites also consist of subsystems and payloads that support different hardware and software technologies.

- **Modularity** - allows for the easy integration and evaluation of both new technologies and alternative implementations of subsystems. It is facilitated with standardised system-level interfaces. In the satellite testbed, both hardware components and software simulations can be integrated for application testing throughout mission life cycles.
- **Scalable architectures** – these architectures can support additional redundancy and distribution of processor load between satellite subsystems and payloads. It can also extend the processing capabilities of distributed testbeds, ground stations and satellite computing environments.

When designing a distributed hard real-time system, it is inevitable to have a particular architecture in mind that can meet the requirements of the targeted application/s. LAN architectures exhibit all of the above-mentioned characteristics required for satellite applications. The selected architecture is shown in Figure 7 and consists of a standardised, shared, **linear communication bus** to which all the subsystems (computational nodes) are connected. *Note that this architecture can be mapped to the general distributed real-time computer system architecture shown in Figure 4. The communication bus can represent the real-time network or a real-time field bus, depending on the functionality of the computational nodes.*

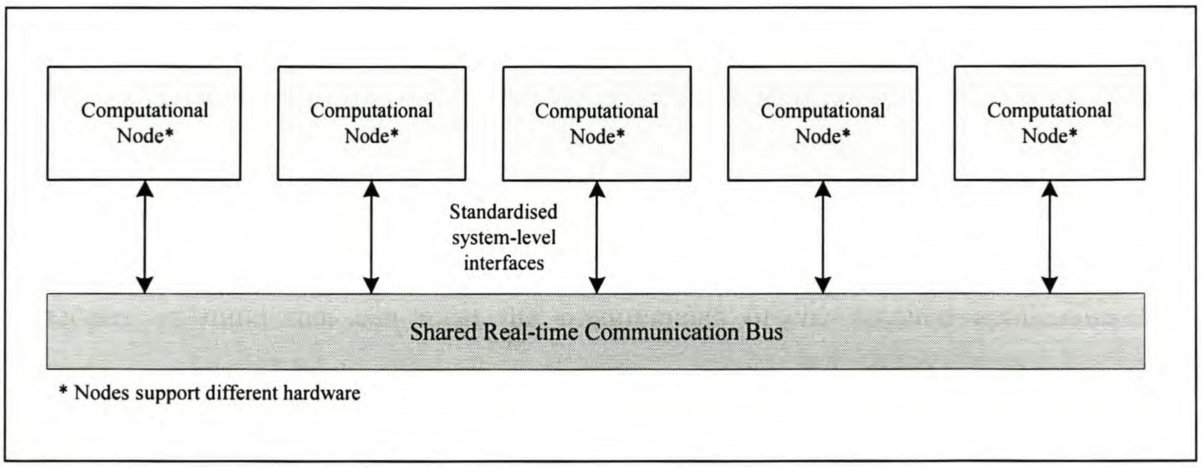
Computational Node*
Computational Node*
Computational Node*
Computational Node*
Computational Node*
Standardised system-level interfaces
Shared Real-time Communication Bus
* Nodes support different hardware

Figure 7. Reference Architecture for the Designed Distributed Real-Time System.

The computational nodes are based on commercial hardware technologies that are being adapted for near term use in space. The shared communication bus is based on half-duplex Ethernet, one of the most popular LAN technologies that is widely accepted and deployed.

3.3 Design Constraints and Assumptions

Several constraints have been imposed on the design of the distributed real-time system. This was done to narrow the scope in an otherwise exhaustive field of research:

- Both commercial and open source real-time extensions and modifications to the Linux operating system are available to improve either the soft or hard real-time performance. For the purposed of the thesis, only open source implementations that offer hard real-time performance are considered.
- Hard real-time communication over Ethernet should be accomplished without changes to the existing Ethernet hardware and network infrastructure. The real-time communication protocol should also not require specialised hardware support or network nodes to be perfectly time-synchronised. *Note that real-time communication does not imply synchronisation (co-ordination of distributed processes with respect to time), although the two are closely related* (Tsai et. al., 1996). The allocation and scheduling of distributed real-time tasks will not be considered. These are topics for further investigation and fall outside the scope of this research.
- Ethernet networks consist of multiple network segments. A segment is a specially configured subset of the larger network and boundaries are established by devices (such as routers, switches and bridges) capable of regulating the flow of network packets to and from the segment. The real-time communication protocol will be designed to provide real-time guarantees only on a single, isolated, network segment. Real-time communication over multi-segment networks will not be considered.
- Real-time messages are assumed to be single packets.

4 LINUX AS A REAL-TIME OPERATING SYSTEM

This chapter focuses on the Linux operating system. An overview of the operating system is given and the limitations for real-time performance are discussed. Various techniques to improve the real-time performance of Linux are then presented, with specific focus on open source, hard real-time extensions and modifications. Finally, a comparison is made between these hard real-time solutions.

4.1 Overview of the Linux Operating System

Linux is a modern UNIX-like operating system that was originally created by Linus Torvalds at the University of Helsinki in Finland. The operating system was made public in 1991 (Pitts and Ball, 1998). It is distributed under the "Gnu's Not UNIX" (GNU) General Public License (GPL). According to the licence agreement, the source code can be freely distributed provided that any code modifications are made publicly available in source code for at most the cost of reproduction[9] (Gillen and Kusnetzky, 2001). The availability of source code encouraged further development and today a worldwide team of enthusiasts is contributing to Linux over the Internet.

The operating system is usually distributed as part of a GNU GPL software package that includes a comprehensive selection of open source drivers, installation and configuration tools, shells and shell-command tools, software utilities, applications, documentation and more. Mainstream distributions are available from companies such as Caldera Systems, MandrakeSoft, Red Hat Software, SuSE Gmbh and TurboLinux (Hubley and Lubrano, 2001). Kernel enhancements and add-ons are also available to support embedded applications and real-time system performance requirements (Gillen and Kusnetzky, 2001).

It is evident that Linux has the potential to be employed for a broad range of applications. This currently includes desktop use with a variety of personal productivity applications such as word processing, spreadsheets, Web browsers and e-mail readers available. Linux is also used as a platform for developing and deploying Web applications which include database support, firewalls, caching, e-mail and Web hosting (e-commerce) services. In addition to this, small to large businesses are using Linux for networking solutions using available administration tools

[9] Note that the GNU GPL can be a concern for commercial developers that feel their operating system code gives their products a proprietary advantage.

and infrastructure serving applications, such as file and printer sharing, to accomplish the task. Numerous Information Technology (IT) vendors such as Hewlett-Packard, IBM, SUN Microsystems and Dell Computers realised the potential of Linux and are now offering it on their workstations and servers (Hubley and Lubrano, 2001).

Linux is also an excellent solution for embedded systems and devices that have a reasonable amount of resources[10] available and that requires networking functionality and/or user interaction (graphical interface or web-based). To use the operating system in embedded environments, various techniques are employed to reduce the size of the Linux kernel (Bird, 2000). These include reconfiguring and compressing the kernel to reduce the binary size, eliminating unnecessary files (documentation and example programs) from open source packages, finding and breaking dependencies in programs and libraries and including small and simple function libraries, and embedded versions of desktop applications. Examples of small function libraries are *BusyBox* (small GNU utility replacement available at web reference: http://www.busybox.net) and *clibc* (small version of the general C library available at web reference: http://www.uclibc.org).

4.2 Linux Compliance to POSIX Standards

The Linux operating system aims to be fully POSIX compliant (Beck et. al., 1996). The Linux 2.2 kernel, used in this research, supports all the functions specified in POSIX 1003.1a (basic operating system interface and functions) and 1003.1c (threads). The kernel also supports some functionality for real-time applications. This includes POSIX 1003.1b memory locking support and static priority scheduling functions (Kuhn, 1998). These functions improve the performance and the functionality of the Linux operating system. Note that a number of key real-time functions have not been incorporated into the operating system. As a result, real-time processes can for example be specified and prioritised over normal processes, but timing constraints such as maximum process execution time or deadline still cannot be defined (Kuhn, 1998).

[10] Linux can be used in systems with 256 Kbytes of ROM and 512 Kbytes of RAM (Random Access Memory), but the implementation will be very limited. Typically memory space in the order of 2 to 4 Mbytes of ROM, and 4 to 8 Mbytes of RAM are recommended.

Various research efforts are implementing real-time POSIX functions for the Linux operating system and experimental kernel patches are available on the Internet. These include real-time POSIX 1003.1b functions for real-time signal handling, time measurement features, synchronous and asynchronous I/O. Currently no support is available for real-time timers and message queues (Kuhn, 1998).

Support for real-time POSIX standards is useful for the development of real-time systems, but the design of the operating system still has the most significant impact on real-time performance. Real-time and general-purpose operating systems have contradictory design requirements. In real-time systems the emphasis is on timely and predictable behaviour. UNIX was originally designed as a general-purpose, time-sharing operating system (Ritchie and Thompson, 1974). Linux retained this nature and like UNIX strive to provide an acceptable average performance by balancing system throughput and response time, and ensuring fair allocation of Central Processing Unit (CPU) resources among all processes. Therefore, due to the nature of the Linux operating system and the limited real-time POSIX support, it is only suitable for certain soft (the "softest") real-time applications.

4.3 Linux Limitations for Real-Time Support

Several limitations that prevent the general-purpose, time-sharing, Linux operating system from being used as a real-time operating system are presented below (Barabanov, 1997; Yodaiken, 1999):

- **Non pre-emptive kernel** - The Linux kernel cannot be pre-empted. This means that once a process execution enters the kernel through a system call, context switches will be deferred either until the executing process blocks within the system call or exits the kernel. If a low priority process enters the kernel and a high priority process becomes ready for execution, the high priority process will have to wait until the low priority system call completes. These delays are not predictable and thus unacceptable for real-time performance.
- **Disabled interrupts -** The kernel disables interrupts to protect critical sections of code (synchronisation). As a result, the system's response to events can be delayed and external interrupts may even be lost. Note that the disabling of interrupts in Linux is not used extensively. The kernel architecture is such that particular data structures and operations are prohibited for interrupt handlers. However, the fact that interrupts are sometimes disabled compromises the system's ability to promptly respond to real-time events, making the system less deterministic.

- **Lack of high granularity timers** – Linux uses periodic clock interrupts for timekeeping and event scheduling. A hardware timer chip is programmed to interrupt the CPU at fixed rates; 10 milliseconds (ms) on Intel x86 compatibles (Hill et. al., 1998). At each interrupt the kernel updates the software clock and schedules events due for processing. The interrupt period is thus the smallest meaningful unit of time with which events can be scheduled. There is always a trade-off between the amount of time spend in handling clock interrupts and the timer resolution and scheduling granularity. This trade-off is often at odds with the requirements of real-time systems for responsiveness and predictability.
- **Fairness based scheduling** – The scheduler is the kernel function that governs the order in which processes are executed. The standard Linux scheduler maintains a linearly linked-list of all executable processes and when called, it scans the entire list before completing process selection (Card et. al., 1998). The linearity of the selection algorithm causes scheduling time to increase linearly with the number of executable processes. The undetermined process load inhibits predictable system response times. The default scheduler also ensures fair allocation of computing resources to all processes. Processes have a dynamic priority that depends on the amount of CPU time spent on a particular process. If a process uses all the computing time issued to it, its priority is lowered. For example, if the scheduler finds it desirable, a background maintenance process can run even when a high priority process requires all available computing resources. This scheduling policy does not favour real-time processing (Barabanov, 1997). The scheduler can also promote low priority tasks, based on own criteria, to avoid starvation[11] (Weinberg and Lundholm, 2001) or to minimise the hard disk head movement (Yodaiken, 1999).
- **Virtual memory with paging** – Virtual memory refers to using a portion of the hard disk to extend the RAM. Unused memory content is moved to the hard disk so that the physical memory can be used for other purposes. When the content is needed, it is moved back into memory and the virtual addresses are remapped to the physical memory addresses based on translation tables maintained by the operating system (mapping). To make translations easier, the virtual and physical memories are divided into memory pages (4 Kbytes on Intel x86 systems) (Beck et. al., 1996). Copying virtual pages from disk to main memory is called paging. For real-time systems, virtual memory introduces additional unpredictability, because the content of physical memory is not fixed and memory management (page fault and replacement) is complex and not deterministic.

[11] System starvation occurs when all the system's processes continue to run indefinitely, but fails to make any progress due to their unique interrelationships (Tanenbaum, 1992).

4.4 Real-Time Solutions for Linux

Various commercial and open source solutions are currently available to improve the real-time performance of the Linux operating system. A reference guide is maintained online by the Linux community to keep track of new products and developments (Lehrbaum, 2001; LinuxDevices, 2002). Current available solutions are summarised in Table 4.

Table 4. Various Commercial and Open Source Real-Time Linux Solutions.

Real-Time Linux	Performance	Developer	Features
Commercial Distributions			
RTLinux	Hard	FSMLabs	Commercial version of RT-Linux
Embedix Realtime	Hard	Lineo	RTAI derived with kernel enhancements
BlueCat RT	Hard	LynuxWorks	Includes RT-Linux features and kernel enhancements
Hard Hat Linux	Soft/Hard	MontaVista Software	Improved scheduler and pre-emption Includes RT-Linux for hard real-time
REDIce-Linux	Hard	REDSonic	RTAI and RED-Linux derived with kernel enhancements (Commercial RED-Linux)
Linux/RT	Hard	TimeSys	RTAI with kernel enhancements
Open Source Implementations			
ART Linux	Hard	ETL (Youichi Ishiwata)	Inspired by RT-Linux
KURT	Firm	University of Kansas	Improved scheduling resolution Timer similar to RT-Linux/RTAI
Linux/RK	Soft	Carnegie Mellon University	A resource kernel
Linux-SRT	Soft	AT&T Laboratories Cambridge	Quality of service guarantees only (No longer maintained)
QLinux	Soft	University of Massachusetts	Quality of service guarantees only
RED-Linux	Firm	University of California	Kernel blocking and scheduling improvements. Includes RT-Linux features (No longer maintained)
RTAI	Hard	DIAPM	Hard real-time application interface
RT-Linux	Hard	Institute for Mining and Technology, New Mexico	Original hard real-time operating system for Linux

Two main strategies are followed to improve the real-time performance of the Linux operating system. The first approach adds real-time capabilities by modifying the kernel. The second approach divides the kernel functionality in hard real-time and non-real-time parts. A second hard real-time kernel is added to the system that schedules the general-purpose Linux operating system for execution only when there are no real-time tasks to run. The hard real-time kernel also controls the system hardware and provides the Linux operating system with a software simulation thereof. Linux can never block interrupts or prevent itself from being pre-empted. Both these approaches have benefits and limitations that will be discussed.

Modifying the Linux operating system has the advantage that real-time tasks can run in user space. In user space, memory protection is available, making development easier and limiting kernel crashes caused by faulty real-time applications. This also means that the standard Linux services are available to real-time applications and extensive tool support is included for easy debugging.

This approach also has limitations. In most cases, substantial modifications to the Linux kernel code are required. The implementation may not meet low-latency performance requirements of some real-time applications due to long interrupt response times and context switches. Modifications that are based on the pre-emption improvement of the Linux kernel also have limitations. Due to the complexity of the kernel, the worst-case interrupt latency cannot be defined, because all the kernel code paths cannot be analysed. This limit the guarantees on interrupt latencies, and at best firm real-time performance can be achieved. Extensive pre-emptive analysis is required for new Linux kernel releases. Pre-emptibility analysis is also required for new device drivers and kernel modules to avoid adding long non-pre-emptive code paths. Due to these limitations, firm real-time performance can at best be achieved.

Adding a second hard real-time kernel has the advantage that few modifications to the Linux kernel are required. The hard real-time kernel makes standard Linux fully pre-emptable, which means that the scheduling of real-time tasks is more deterministic and can be guaranteed. With this implementation both the hard real-time kernel and real-time applications run in the same kernel space. This approach improves system performance by allowing very low interrupts latencies and fast context switches (real-time tasks have direct access to the underlying hardware).

Unfortunately this approach also has limitations. No memory protection for real-time tasks is available in kernel space. As a result, faulty real-time applications will cause the kernel to crash or become unstable. Debugging in kernel space is also difficult and limited tools are available. Application developers are required to have a proficient understanding of the kernel, device driver internals and their interactions.

4.5 Open Source Hard Real-Time Linux Implementations

The thesis focused on open source, hard real-time Linux implementations. These are freely available on the Internet at little or no cost. Most of the commercial distributions either include open source implementations, some of their features, or are influenced by them (see Table 4). There are five main open source, hard real-time extensions to Linux, namely:

- ART-Linux
- RED-Linux
- KURT-Linux
- RTAI
- RT-Linux

ART-Linux (Web reference: http://www.etl.go.jp/etl/robotics/Projects/ART-Linux/), which is similar to RT-Linux, was not considered during this research. The reason being that ART-Linux does not fall under the GPL, and reference documentation is only available in Japanese.

The remaining four open source, hard (including firm) real-time implementations were considered and will be discussed in the following sections. Note that both RED-Linux and KURT-Linux modify the Linux kernel to provide firm real-time performance. RTAI and RT-Linux add a second hard real-time kernel that schedules the full-featured Linux operating system as the lowest priority non real-time task.

4.5.1 Kernel Pre-emption Improvement with RED-Linux

RED-Linux (Real-time and Embedded Linux) is being developed at the University of California, Irvine. Pre-emption points are inserted in the kernel code to reduce the blocking time of real-time tasks. No changes are made to any of the functions and services offered by the operating system (Wang and Lin, 1999). The implementation is shown in Figure 8.

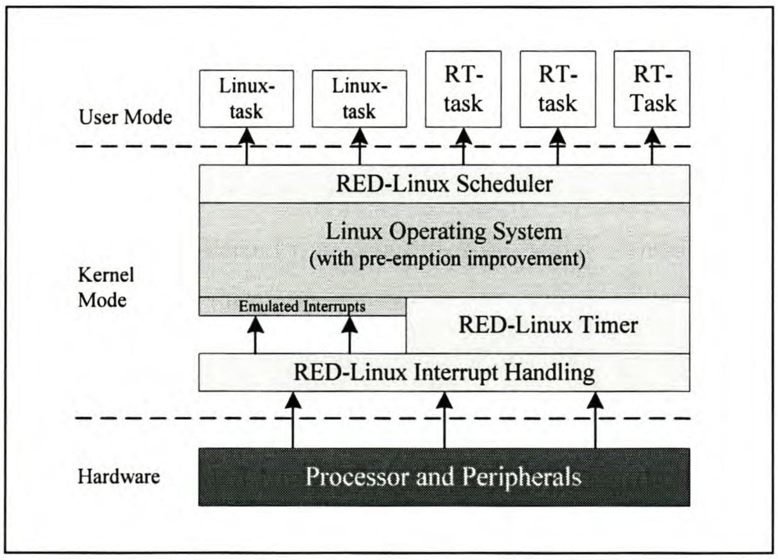

Figure 8. Implementation of RED-Linux.

Linux system calls are atomic operations with variable duration. A system call that for example reads a large block of data from disk can take several seconds to complete before releasing the processor. RED-Linux reduces the delay resolution of system calls by dividing them into smaller blocks (Wang and Lin, 1998). The minimum size of these blocks is constrained by those system calls that cannot break their execution flow (kernel thread), because other processes may change their resources. The block with the maximum execution time therefore determines the kernel scheduling resolution.

System calls are divided by forcing them to wait or "sleep" on virtual resources at predetermined instances (Wang and Lin, 1998). This approach creates deliberate backdoors (pre-emption points) for real-time tasks in the critical sections of the Linux kernel. After execution of every block, the kernel checks for pending real-time processes. If a high priority real-time task is ready for execution, a context switch to the real-time task is performed provided it is safe to do so (i.e. RED-Linux requires kernel threads to be carefully designed for pre-emption).

RED-Linux also offers a general scheduling framework that consists of two functional elements, namely the allocator and dispatcher. The *allocator* assigns the scheduling parameters to tasks according to the implemented scheduling policy. The *dispatcher* is a low-level scheduler that inspects the scheduling parameters of each task in the ready queue and dispatches the appropriate task for execution (Wang and Lin, 1999).

The scheduling framework was designed to be flexible and therefore to support priority driven scheduling, time-driven scheduling, share-driven scheduling and attribute combinations of these scheduling paradigms. Priority driven scheduling is based on the fixed or dynamic priority of real-time tasks and includes algorithms such as Rate Monotonic and Earliest-Deadline-First (Liu and Layland, 1973). Time driven scheduling uses the current time to determine the scheduling sequence of real-time tasks (e.g. the Time Triggered protocol (Kopetz, 1998)). With share-driven scheduling tasks are scheduled for execution based on the resource share assigned to each of them (e.g. Weighted Fair Queuing (Demers et. al., 1990)).

RED-Linux also ported the high-resolution timer and software interrupt mechanism from RT-Linux (Wang and Lin, 1999). The high-resolution timer provides sufficient responsiveness to events as required by many real-time applications. Software interrupt emulation is used to improve the response time of real-time tasks especially during multiple interrupt occurrences.

4.5.2 Improved Timer Resolution with KURT-Linux

KURT-Linux (Kansas University Real-Time Linux) adds real-time capabilities to the Linux operating system by improving the timer resolution and by modifying the standard scheduler to also include a scheduling policy for real-time tasks (Srinivasan et. al., 1998). The implementation is shown in Figure 9.

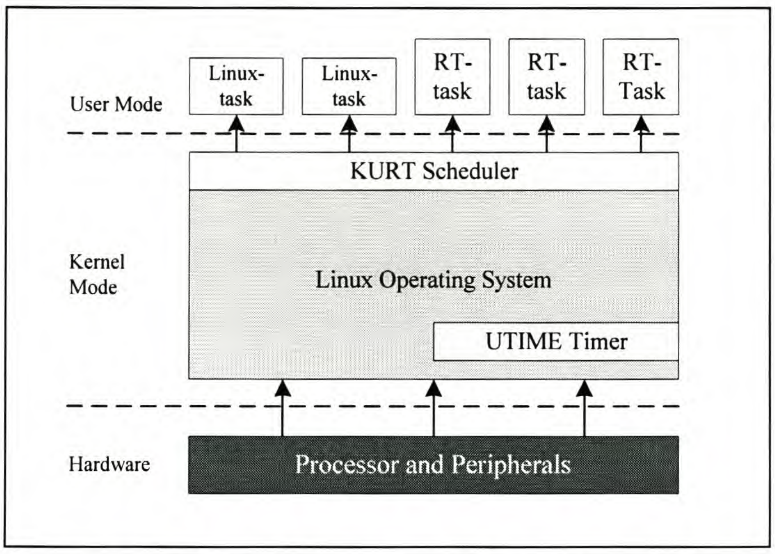

Figure 9. Implementation of KURT-Linux.

In standard Linux, the system timer is programmed to interrupt the CPU at fixed intervals. At each interrupt the kernel updates a software clock and checks for scheduled processes ready for execution[12]. The rate at which the interrupts occur determines the timing resolution of the operating system, i.e. the maximum rate at which processes can be scheduled. The timing resolution can be increased by programming the system timer to interrupt the CPU at a higher frequency, but this method also introduces significant overhead because of the additional and most often unnecessary interrupt processing.

KURT-Linux improves the timer resolution by modifying the Linux operating system with the micro-time (UTIME) extension (Hill et. al., 1998). This extension programs the system timer in one-shot mode (similar to RT-Linux) so that the processor can be interrupted with microsecond accuracy when the earliest scheduled process is ready for execution. Additional processing is only required when the timer needs to be reprogrammed with a new value after each interrupt. If

[12] The software clock is basically a counter that counts the number of interrupts (*jiffies*) since the kernel started. Each *jiffy* represents a 10 ms time interval on the Intel x86, AMD and other hardware architectures. On the Dec Alpha architectures, a jiffy represents 1 ms.

the timer is used in periodic mode, this is not needed. The UTIME extension maintains the software clock using a time stamp counter [13].

Three modes of operation are defined to control the scheduling of system resources and to minimise the degrading effect of timesharing operating system components on real-time performance. In *normal* mode, no real-time support is provided and the operating system performs like normal. In *focussed real-time* mode, only real-time processes are scheduled and allowed to execute. In *mixed real-time* mode, both real-time and non-real-time processes are allowed to execute but with real-time processes having the highest priority. In both the focussed and mixed real-time modes of operation, the KURT-Linux scheduling policy is used whereby real-time processes are scheduled explicitly i.e. using a real-time cyclic scheduler and a precompiled scheduling table (Srinivasan et. al., 1998).

Note that KURT-Linux was designed for applications with a combination of real-time and general system requirements. As a result, real-time processes are allowed to also access services that would normally only be available to non-real-time processes. Because these general system services were not optimised for real-time performance (such as allowing interrupts to be disabled), KURT-Linux is only capable to meet firm real-time performance constraints.

4.5.3 Interrupt Emulation with RT-Linux

Real-Time Linux (RT-Linux) was developed by the Institute for Mining and Technology of New Mexico (Barabanov, 1997). It is a real-time executive (small kernel and scheduler) that runs the Linux operating system as a fully pre-emptable non-real-time task. RT-Linux shares the same kernel address space, but operates separate from the Linux operating system. This means that the real-time and general-purpose operating systems can be configured and optimised independently (Lineo Inc., 2000). The implementation is shown in Figure 10.

RT-Linux inserts a software emulation layer between the Linux kernel and the interrupt control hardware. The Linux operating system is modified by replacing all interrupt control instructions

[13] Pentium processors have built-in hardware features (such as the Time stamp Counter and event counters) to monitor system performance transparently (Intel Corporation, 1997).

(assembler calls) with similar emulation instructions (Barabanov, 1997)[14]. When the Linux operating system requests to enable or disable an interrupt, the real-time kernel handles the request, thereby isolating Linux from the interrupt control hardware. With this approach, the Linux operating system can no longer influence the hardware interrupts (which previously increased the interrupt response time latencies of real-time tasks), or prevents itself from being interrupted. The real-time kernel's emulation layer also handles all hardware interrupts. Interrupts for real-time tasks are handled immediately, suspending any native Linux task if necessary. Linux operating system interrupts are queued by the emulation layer, thus insuring that no interrupt gets lost. When Linux is scheduled for execution and interrupts are enabled (the status is kept track of by the emulation layer), all the pending interrupts are serviced.

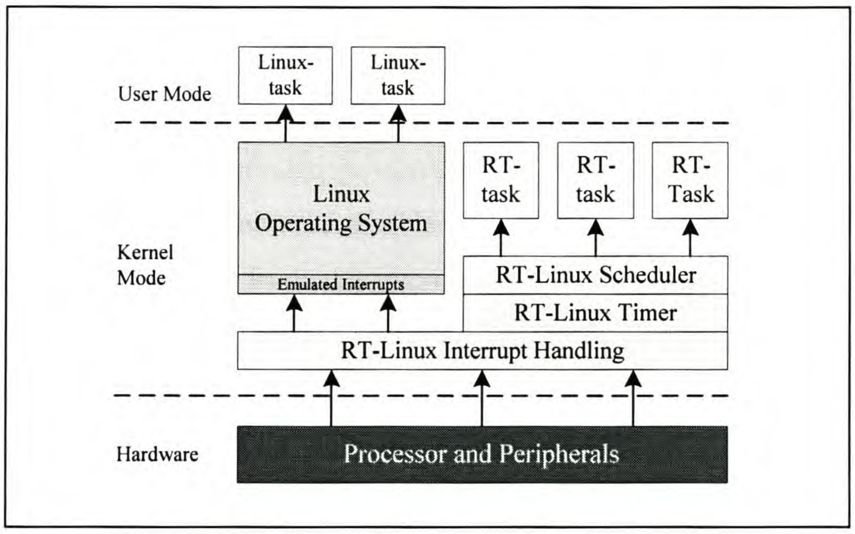

Figure 10. Implementation of RT-Linux.

In RT-Linux real-time tasks are also implemented in kernel space. These tasks cannot perform any non-real-time Linux functions, because the Linux kernel can be pre-empted at any given time. As a result, special communication mechanisms are required to allow the transfer or sharing of data between real-time tasks and/or non-real-time Linux tasks. Hereby, a real-time task can request and receive data from a task executing in user space. RT-Linux provides three IPC mechanisms for this purpose. These are FIFO pipes, mailboxes and shared memory.

[14] Instructions that disable and enable interrupts are typically called under Linux when a task switches to kernel mode (disable interrupts) and then back to user mode (enable interrupts). The instructions are cli (clear interrupt flag), sti (set interrupt flag) and iret (return from interrupt).

The default RT-Linux scheduler is a pre-emptive, fixed priority scheduler that treats the Linux operating system and associated non-real-time tasks as the lowest priority (idle) task. Linux can only operate when there are no real-time tasks and when the real-time kernel is inactive. Because of the modular nature of RT-Linux, different schedulers, scheduling policies or algorithms can be used to best suit the real-time application. They can be implemented as loadable kernel modules. Barabanov (1997) mentions two schedulers that implement the Earliest-Deadline-First (EDF) and Rate Monotonic scheduling algorithms. These algorithms are described in Liu and Layland (1973).

With real-time schedulers there is a trade-off between the rate of clock interrupts and the task release jitter. Real-time tasks usually give up the processor either voluntarily or when they are pre-empted by a higher priority task. Typically the periodic clock interrupt handler resumes these tasks. A high clock interrupt rate ensures low task release (resume) jitter, but involves high resource overheads to handle interrupts. A low clock interrupt rate, on the other hand, does not impose much overhead, but causes real-time tasks to be resumed either prematurely or too late and as a result increases the task release jitter.

As mentioned before, the Linux operating system operates the programmable hardware timer chip (Intel 8254 or equivalent) to schedule tasks with a 10 ms precision on Intel x86 compatibles. For some real-time applications the interrupt rate can be too low, resulting in real-time tasks to be resumed too late. RT-Linux solves this by operating the timer chip as a high granularity, programmable, one-shot timer. In this mode, real-time tasks can be precisely resumed (approximately 1 microsecond resolution). The system keeps global time by adding the intervals between the interrupts and simulating periodic interrupts for the Linux operating system (Barabanov, 1997).

4.5.4 Hardware Abstraction with RTAI

The Department of Aerospace Engineering, Milan Polytechnic (DIAPM), developed Real-Time Application Interface (RTAI) to support real-time applications under Linux. It consists of a Real-Time Hardware Abstraction Layer (RTHAL) which is a Hardware Abstraction Layer (HAL)[15]

[15] The HAL is a software layer that allows an operating system to interact with a hardware device at a general or abstract level, rather than at a detailed hardware level.

extended with a rich set of functions and services to support various real-time applications (Cloutier et. al, 2000). The implementation is shown in Figure 11.

RTAI has a similar architecture to RT-Linux. The Linux operating system is also treated as the lowest priority (background) task of a small real-time kernel that executes only when no real-time activity occurs. The only architectural difference between the two is the method with which real-time features are added to the Linux operating system. RT-Linux applies changes directly to the Linux kernel software and as a result static code is permanently added to the kernel (Mantegazza, 1999). This approach requires various changes to the Linux operating system software. RTAI, on the other hand, limits the level of software intrusion by adding a HAL that dynamically substitutes Linux entries with real-time equivalents using pointer redirection (substitution). A key component of the RTHAL is the Interrupt Descriptor Table (IDT), which is a data structure of all the pointers to the internal Linux data and interrupt handling functions (Lineo Inc., 2000). This approach also allows Linux to easily revert back to its standard operation by simply changing the pointers in the IDT.

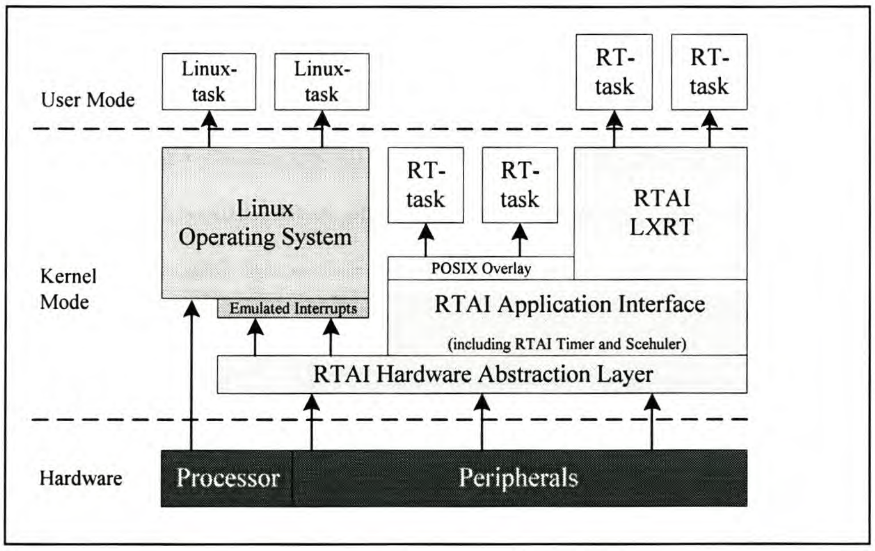

Figure 11. Implementation of RTAI.

When RTAI is activated the RTHAL generates a duplicate copy of the standard Linux IDT and interrupt handlers. The duplicate copy now becomes the valid table. All interrupt control functions are redirected to their trapped RTAI equivalents and the peripheral interrupt handling functions are directed to the interrupt dispatcher (Mourot, 1999). The RTAI interrupt dispatcher is an interrupt handler that activates the appropriate interrupt handler (Linux task when there are no active real-time tasks or RTAI scheduled real-time task). When Linux disables interrupts, they are queued by RTAI to be delivered when interrupts are re-enabled. It should be noted that processor originated interrupts (e.g. error signals such as division be zero) are still handled by the Linux kernel (Sarolahti, 2001).

RTAI offers a real-time, pre-emptive, priority based scheduler that supports either periodic or one-shot task scheduling, but not both simultaneously. Periodic scheduling is a very efficient approach (the scheduling timer is programmed once with interval time period) if the real-time application has a number of tasks with a common period. One-shot scheduling, on the other hand, is more flexible because it allows the accurate scheduling of several tasks without a common period and other random tasks. However, it requires continuous reprogramming of the scheduling timer.

In one-shot mode, the timer used by the scheduler is similar to the high-granularity timer used in RT-Linux and KURT (Lineo Inc., 2000). However, the time is measured based on the time stamp counter (or emulation if its not available) and the Intel 8254 timer chip is only used to generate the interrupts. This approach can shorten the time required to reprogram the timer chip, improving the efficiency of the one-shot mode of operation. Periodic mode is still timed by the 8254 timer chip, because of the lower overhead involved compared to the one-shot mode, even with the added improvement (DIAPM, 2002). The interrupt frequency can be adjusted to accommodate the requirements of the targeted real-time application.

Similar to RT-Linux, real-time tasks are also implemented in kernel mode (without memory protection). RTAI provides various IPC mechanisms to transfer or share data between real-time tasks and/or Linux tasks. These include real-time FIFO's, shared memory, mailboxes, semaphores and Remote Procedure Calls (RPC) (Lineo Inc., 2000).

RTAI has a symmetrical Application Programming Interface (API) that allows the use of real-time system calls within the standard user-space. With LinuX Real-Time (LXRT), real-time applications can be entirely developed in user space with memory protection, trap handling and utilising standard Linux debugging tools (Lineo Inc., 2000). After development, the application can be executed in user-space, or converted to a kernel module for execution in kernel space, which results in better system response times.

4.6 Comparison of Hard Real-Time Linux Implementations

A comparison matrix of the above-mentioned open source hard real-time Linux implementations is shown in Table 5. The four implementations were compared based on various criteria and the findings are summarised in this section. ***Note that various Web references have been used because the most recent information are updated and maintained online.***

The first five criteria listed in the comparison table relate to how the real-time limitations of Linux, as mentioned in Section 4.3, have been addressed by the different implementations. RED-Linux, RT-Linux and RTAI improve the pre-emptibility of the Linux kernel and the interrupt response time of event-triggered real-time tasks by emulating the Linux hardware interrupt control functions (Wang and Lin, 1999; Barabanov, 1997; Lineo Inc., 2000). KURT-Linux offers no mechanisms for improvement in both cases. All the real-time implementations support high-resolution timers to improve the system's predictability and responsiveness to time-triggered and scheduled events (Wang and Lin, 1999; Hill et. al., 1998; Barabanov, 1997; Lineo Inc., 2000). Flexible scheduling mechanisms for real-time tasks are also supported by all (Wang and Lin, 1999; Srinivasan et. al., 1998; Barabanov, 1997; Lineo Inc., 2000). Both RT-Linux and RTAI support system calls to control memory paging (Sarolahti, 2001).

Other criteria that were considered include the API and communication methods for real-time tasks in both the kernel and user mode of operation. Both RED-Linux and KURT-Linux only support real-time applications in user mode (Wang and Lin, 1998; Srinivasan et. al., 1998). They have custom API's with RED-Linux also offering limited POSIX support (Wang and Lin, 2000; Dinkel et. al., 2002). In both cases, no support is offered for real-time IPC between real-time tasks; only standard Linux IPC methods are available (Lineo Inc., 2000). KURT-Linux does offer limited IPC between real-time and non-real-time tasks (Srinivasan et. al., 1998).

RT-Linux and RTAI, on the other hand, offer various flexible real-time IPC methods between real-time tasks and between real-time and non-real-time tasks (Kuhn, 2001; Lineo Inc., 2000). RT-Linux offers a POSIX compliant API that supports real-time applications in kernel mode. Limited support is available for user mode real-time applications (Dankwardt, 2000). RTAI offers a feature rich, symmetrical API that is POSIX compliant (using an overlay module) and that allows the development and implementation of real-time applications in both user and kernel mode (Sarolahti, 2001).

Table 5. Comparison Matrix of Open Source, Hard Real-Time Linux Operating System Implementations.

Criteria	Firm Real-Time RED-Linux	Firm Real-Time KURT-Linux	Hard Real-Time RT-Linux	Hard Real-Time RTAI
Kernel Pre-Emption Improvement	Pre-emption points inserted in the Linux kernel	No improvement	Separate hard real-time kernel	Separate hard real-time kernel
Interrupt Response Time Improvement	Interrupt emulation similar to RT-Linux	No improvement	Interrupt emulation limits Linux from temporality disabling any hardware interrupts	Hardware abstraction layer controls interrupts (emulates Linux interrupts). Thereby, Linux cannot disable any hardware interrupts.
Timer Improvement	Includes the RT-Linux high resolution timer	UTIME high resolution timer included	Includes a high resolution timer	Includes a high resolution timer
Real-Time Task Scheduling	Precise scheduling of real-time tasks within Linux. A flexible scheduling framework is provided that can support various scheduling algorithms.	Precise scheduling of real-time tasks within Linux. Two modes for real-time scheduling are provided, one dedicated to real-time tasks.	Includes a default fully pre-emptive, priority-based real-time scheduler that supports periodic and one-shot task scheduling. Alternative scheduling modules allow EDF and rate-monotonic scheduling.	Includes a default fully pre-emptive, fixed priority-based real-time scheduler that supports periodic and one-shot task scheduling
Memory Management	Dynamic memory allocation but no real-time performance	Dynamic memory allocation, but no real-time performance	Static memory allocation before execution. Disable memory paging with mlockall system call.	Dynamic memory allocation, but no real-time performance. Disable memory paging with mlockall system call.
Inter-process Communication (IPC) between Real-Time Tasks and Data Transfer Synchronisation	No support for real-time IPC, Non-real-time Linux IPC includes: - File locks - Pipes and named pipes (e.g. FIFO) - Signals (asynchronous messages) - System V IPC: semaphores, mutexes, message queues and shared memory - Wait queues	No support for real-time IPC, only Linux IPC	- Mailboxes - Real-time FIFO's - Semaphores (POSIX) - Conditional Variables (POSIX) - Mutexes (POSIX thread mutex variables)	- Mailboxes - Message queues (POSIX) - Real-time FIFO's and named FIFO's - Remote Procedure Calls - Semaphores (POSIX) - Conditional Variables (POSIX) - Mutexes (POSIX thread mutex variables)
IPC between Real-Time and Non-Real-Time Tasks	No support for real-time IPC, only Linux IPC	- Lock-free queues - Shared memory	- Mailboxes (obsolete) - Message queues - Real-time FIFO's - Shared memory	- Mailboxes - Real-time FIFO's and named FIFO's - Shared memory

Table 5 (cont'd). Comparison Matrix of Open Source, Hard Real-Time Linux Operating System Implementations.

Criteria	Firm Real-Time RED-Linux	Firm Real-Time KURT-Linux	Hard Real-Time RT-Linux	Hard Real-Time RTAI
Application Programming Interface (API)	Custom API with support for POSIX 1003.1c scheduling threads	Custom API	API complies with POSIX 1003.13 PE51 (minimal real-time system profile) and POSIX 1003.1b. Aim is to simplify the API to improve real-time predictability.	Custom API that complies with POSIX 1003.1b by using an overlay module. Also support POSIX1003.1c (threads, mutexes and conditional variables). Aim to offer multiple features for easy development of real-time applications.
User Mode Real-Time Applications	Real-time applications are implemented in user mode	Real-time applications are implemented in user mode	Limited. Real-time functions can execute in user mode. A function is activated by an interrupt or timer (acts as a signal handler).	LXRT is a user mode API for soft real-time applications executing in user mode
Platform Support	Architectures supporting glibc 2.0, such as the Debian Linux distribution: - DEC Alpha families - Intel ARM and StrongARM platforms - Intel x86 compatible families - Motorola M680x0 - Motorola/IBM PowerPC platforms - SUN SPARC	Only Intel x86 and compatible architecture families with support for a time stamp counter	- DEC Alpha families - AMD families (K6, Athlon, Duron, etc.) - Intel ARM and StrongARM platforms - Intel x86 and compatible families - MIPS families (RM7000 and more) - Motorola/IBM PowerPC platforms	- DEC Alpha families - AMD families - Intel ARM and StrongARM platforms - Intel x86 and compatible families - MIPS families - Motorola M680x0 (no MMU) - Motorola/IBM PowerPC platforms
Embedded Implementation	Embeddable, but no information provided	Embeddable, but no information	Minirtl – a small footprint kernel	AtomicRTAI – a small footprint kernel
Development Status	Development stalled (latest version based on Linux 2.0.35, 1999) – implementation commercialised with REDIce-Linux (current status not available)	Actively developed (latest version KURT 2.4.18 v.2, March 2003)	Development stalled (latest version RT-Linux 3.1) – implementation commercialised in May 2001 (latest commercial versions RT-LinuxPro 1.2, RT-LinuxFree 3.2 , January 2003)	Actively developed (latest version RTAI 24.1.12, October 2003)
Documentation	Very limited documentation available	Limited documentation available	Extensive documentation available	Extensive documentation available
Support	No direct support available. Support available on commercial implementation, i.e. RedIce-Linux	Active support available	Active support available	Active support available

Table 5 (cont'd). Comparison Matrix of Open Source, Hard Real-Time Linux Operating System Implementations.

Criteria	Firm Real-Time RED-Linux	Firm Real-Time KURT-Linux	Hard Real-Time RT-Linux	Hard Real-Time RTAI
Licenses and Patents	GNU GPL	GNU GPL	- Open RT-Linux Patent License - allows the use of RT-Linux by software licensed under the GNU GPL or by unmodified Open RT-Linux Software in binary form - Commercial licence required if proprietary restrictions are to be imposed - U.S. Patent No. 5,995,745 – not applicable outside the USA	GNU GPL
Debugging Tools	Standard Linux debug tools available: - GNU Debugger (GDB) - Data Display Debugger (DDD) - Kernel debug stub (small program routine) for GDB (kgdb/gdbtubs)	Standard Linux debugging tools available	- Source level debugging with SMP support using GDB (DDD) - rt_printk function - Tracer (kernel and application events) - RT-Linux POSIXTrace (RTL-PT)	- Source level debugging from host linked with serial port using kgdb - rt_printk function - R2D2 – Remote real-time debugger - Linux Trace Toolkit (LTT) – Operating system monitor
Miscellaneous Development Tools	No information	No information	- Comedi – Open source driver development environment - RTiC-Lab – Real-time control implementation platform - RTLT – Matlab/Simulink based real-time graphical control system design environment	- Adaptive Domain Environment for Operating Systems (ADEOS) – a flexible environment for sharing hardware resources between multiple operating systems or multiple instances of an operating system - Carbonkernel – Real-time Operating system simulator - Comedi - Matlab/Simulink/Real-time Workshop interface – Computer aided control system design environment - Proc interface to access system info. - Xenomai – A framework for building real-time interfaces such as emulators, under GNU/Linux

The support for different processor architectures and the availability of specific embeddable (small footprint versions) real-time implementations are also criteria used in the comparison study. RED-Linux, RT-Linux and RTAI all support a wide range of processor architectures and platforms (LinuxDevices, 2002a; Web references: http://www.fsmlabs.com/developers/faq/; http://www.aero.polimi.it/~rtai). KURT-Linux however, is limited to Intel x86 and compatible architectures that support a timestamp counter (Srinivasan et. al., 1998). All the real-time implementations are embeddable with specific small footprint kernels available for RT-Linux and RTAI (Lehrbaum, 2001).

Other very important criteria that were used in the comparison study include the current development status of the real-time implementations and the availability of support in various forms that include documentation, human expertise, debugging tools and miscellaneous development tools. Both KURT-Linux and RTAI are actively being developed (Web references: http://www.ittc.ukans.edu/kurt/; http://www.aero.polimi.it/~rtai). The development on RT-Linux and RED-Linux are more focused on the commercial versions of the open source distributions (Web references: http://www.fsmlabs.com; http://www.redsonic.com).

Extensive supporting documentation is available for both RT-Linux and RTAI (Web references: http://www.fsmlabs.com/developers/man_pages; http://www.aero.polimi.it/~rtai/documentation). RED-Linux and KURT-Linux only have limited documentation available. Active support is also available for KURT-Linux, RT-Linux and RTAI (Web references: http://www.ittc.ukans.-edu/~kurt; http://www.fsmlabs.com; http://www.aero.polimi.it/~rtai). Unfortunately no direct support for RED-Linux is available; only the commercial implementation is supported (Web reference: http://www.redsonic.com).

A variety of debugging and development tools are available for both RT-Linux and RTAI. Most of the tools are customised and highly specialised because of the constraints imposed when developing real-time applications in kernel mode (Lineo, 2000; Web references: http://www.fsmlabs.com/developers/faq/; http://bernia.disca.upv.es/rtportal/apps/trace/; http://-www.rtlinx.at/applications; http://www.aero.polimi.it/~rtai/applications). Standard Linux debugging and development tools are used for RED-Linux and KURT-Linux because in both cases real-time applications are implemented in user mode.

4.7 Motivation for using RTAI

One of the research objectives was to find a hard real-time Linux operating system that is flexible enough to support various real-time applications in the satellite environment. Based on the comparison study, RTAI proved to be the most suitable open source implementation.

Both RTAI and RT-Linux can offer hard real-time performance that is very competitive with commercial operating systems such as VxWorks and QNX (Lineo Inc, 2000)[16]. RED-Linux and KURT-Linux, on the other hand, can only offer firm real-time performance.

With RTAI, the changes to the standard Linux kernel are made minimal with the inclusion of a hardware abstraction layer. The low level of intrusion improves the maintainability of the real-time code and assists in portability when new versions of the Linux operating system are released.

The real-time extensions can also be easily removed by replacing the real-time interrupt function pointers with the original Linux routines. This can be useful to verify the performance of the Linux operating system with and without real-time extensions, for example where a system is required to have interchangeable real-time and non-real-time modes of operation. It also assists in debugging situations where it is beneficial to remove certain real-time extensions. Note that the loss in performance by using function pointers instead of directly linked functions in RTAI is so minor that it can be neglected (Mantegazza, 1999)

Where RT-Linux aims to provide only the necessary real-time functions to keep the kernel small, simple and easy to maintain, RTAI provides an application-programming interface with numerous mechanisms and features to meet the requirements of various applications. This makes real-time application development easier. The developer can simply choose the best-suited method/s for a specific application. Support for POSIX standards has also been included to improve both the re-usability and portability of real-time applications.

[16] Depending on the hardware, typical performance of 4 μs context switch times, 20 μs interrupt response times, 100 kHz periodic task iteration rate (0-1,3 μs jitter) and 30 kHz one-shot task rates have been be accomplished (Lineo Inc, 2000).

RTAI also supports a memory management module that allows hard real-time tasks to dynamically allocate and release memory at run-time, thereby allowing the more efficient utilisation of system memory resources. RT-Linux can only allocate memory statically to real-time tasks before run-time.

Almost all the primary processor architectures that are being adapted from commercial technologies for use onboard satellites are supported by RTAI. Note that because RTAI is a separate hard real-time kernel added underneath the Linux operating system, it will not necessarily support the same processor architectures as Linux.

RTAI claims to be one of the most updated and maintained real-time implementations of the Linux operating system. It is still actively being developed under the GNU GPL license and continues to grow and mature through various contributions. The increased availability of proficient reference documentation, human expertise, and various kernel mode debugging and development tools reduce development times and also contribute to the increased use of RTAI in hard real-time applications. The development of open-source RT-Linux has temporarily stalled and the focus shifted to the improvement of a commercial distribution.

RTAI also has limitations that have to be considered. Firstly, the priority based, pre-emptive real-time scheduler is limited. The RTAI scheduler can only support one mode of scheduling at a time (i.e. either one-short or periodic scheduling). This limitation has been addressed in the RTAI multi-uni-processor (MUP) scheduler, but the solution is restricted to multi-processor platforms that have the Intel Advanced Programmable Interrupt Controller (APIC) functionality (Lineo Inc., 2000).[17] The RTAI scheduler also lacks flexibility. No scheduling framework (as provided by RED-Linux) is available that can support various scheduling algorithms. Another limitation is that the numerous features offered by RTAI can result in too much complexity for developers that only want a simple and deterministic hard real-time operating system of which the predictability can be relatively easily analysed (Sarolahti, 2001).

In general however, the high performance, flexible architecture, rich feature set, availability of online support and other associated benefits make RTAI the most suitable candidate for environments with various hard real-time applications and/or requirements and where benefits can be reaped from utilising a common, open source, real-time operating system.

5 ETHERNET FOR REAL-TIME COMMUNICATION

This chapter discusses the Ethernet network technology and various techniques that can be used to improve the real-time communication performance. Firstly an overview of Ethernet and the protocol that it implements are provided. Then the limitations of Ethernet for real-time communication are discussed. Finally various techniques to improve the real-time performance are categorised and discussed. A suitable technique was chosen to design and implement a real-time Ethernet communication protocol for the RTAI hard real-time operating system.

5.1 Overview of Ethernet

The term **Ethernet** refers to a family of local area network (LAN) technologies that include Ethernet, IEEE 802.3, Fast Ethernet, Gigabit Ethernet and others. The original Ethernet was a 10-Megabit per second (Mbps), baseband, LAN specification invented by the Xerox Corporation in the 1970's. It used the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol as the network access method and coaxial cable for the transmission media. In the 1980's, the Institute of Electric and Electronics Engineers (IEEE) developed the 802.3 international standard based on the original Ethernet technology. The IEEE 802.3 standard specifies the transmission media (coaxial, twisted pair, fibre optics and others), the Ethernet data frame (standardised set of bits used to transfer data; see Figure 12) and the media access control (adopted CSMA/CD from Ethernet) protocol. The standard is also backward compatible with original Ethernet. Today, all Ethernet hardware conforms to the IEEE 802.3 standard, which is embedded in the Ethernet interface hardware. *Note that the term Ethernet, as used in the thesis, will refer to both Ethernet and the IEEE 802.3 standard unless otherwise stated.*

The IEEE standard also specifies half-duplex and full-duplex modes of operation. In half-duplex mode a node may either transmit or receive data, but never both simultaneously. The media access control protocol arbitrates access to the shared network. In full-duplex mode, nodes can transmit and receive data simultaneously. Three conditions apply for full-duplex operation. The physical medium must support simultaneous transmission and reception without interference (e.g. two mediums), exactly two nodes on the LAN must be connected with a full-duplex point-to-point link and both nodes must be capable and be configured for full-duplex operation (IEEE Std 802.3, 2000).

[17] Note that the MUP scheduler can be used on multi-processor platforms with just a single mounted processor.

Note that full-duplex networks with more than two nodes are based on a star-wired **switch**, which is a network processing device that provides dedicated bandwidth though private connections (typically set up with lower layer protocol information) between two nodes on the network. The switch basically forms the central processor of a star topology network. If this network is used onboard a satellite, the switch can be a central point of failure. Instead a half-duplex **hub** (active or passive) should be used. A hub is just a common wiring point for star-topology networks. *The study will only focus on half-duplex Ethernet and the IEEE 802.3 Ethernet standard.*

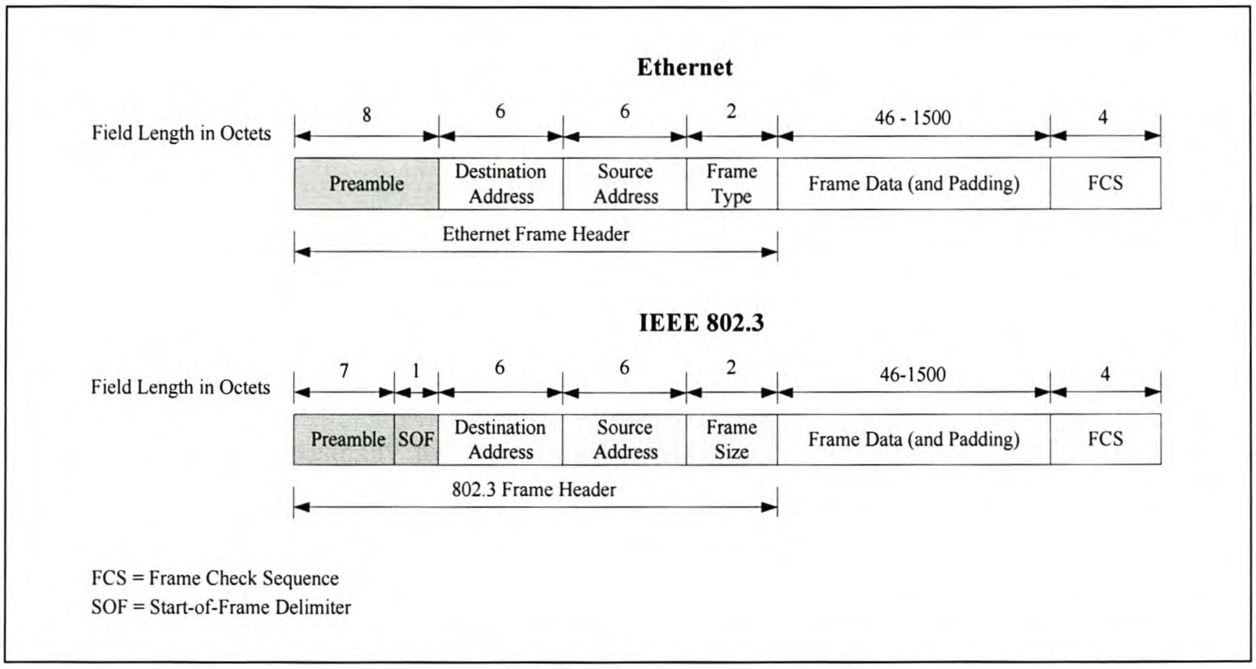

Figure 12. Structures of the Ethernet and IEEE 802.3 Frames.

5.2 IEEE 802.3 Media Access Control

To allow fair access to the shared physical transmission medium, the IEEE 802.3 standard specifies the 1-persistent, Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Media Access Control (MAC) protocol (IEEE Std 802.3, 2000). These shared networks are physically connected in a bus, tree or star-wired network topology. When one node transmits, all nodes receive the transmission. A conceptual model of the CSMA/CD network showing transmission, contention and idle periods is shown in Figure 13.

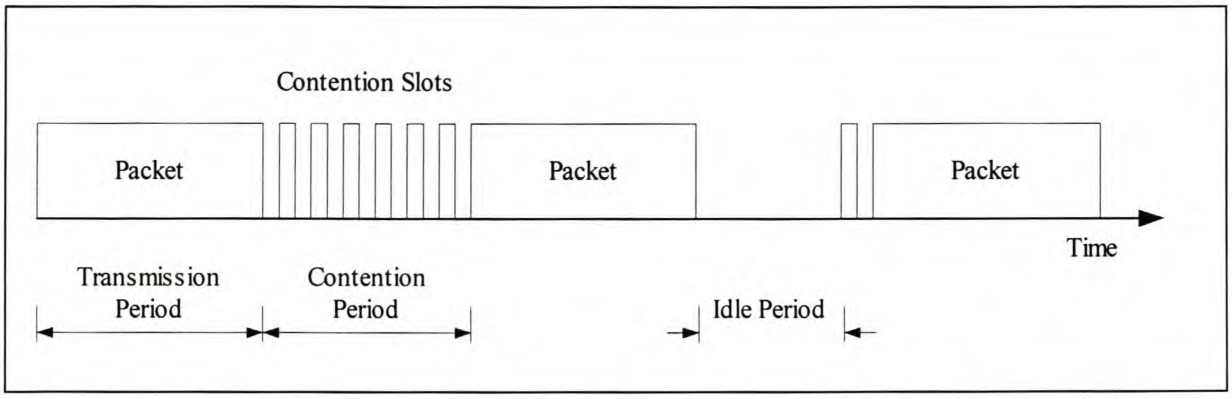

Figure 13. Conceptual Model of Carrier Sense Multiple Access with Collision Detection.

Ethernet is a shared network and all the nodes have the same ability and priority to transmit messages. When a node transmitted a message, it is allowed to access the network immediately thereafter to transmit another message (***Multiple Access***).

Network nodes are in receiving mode if they're not transmitting. A node wishing to transmit first listens to the channel to determine if another transmission is in progress (***Carrier Sense***). Once the node detects the channel to be idle, it transmits the frame immediately with probability of 1 (***1-persistent***) after a brief inter-frame delay. The delay is to provide recovery time for the MAC functions and the physical medium (IEEE Std 802.3, 2000). Since signals take a finite time to travel from one end of the network to the other, it is possible for two interfaces to sense that the network is idle and to start transmitting their frames simultaneously, resulting in a collision.

Collision Detection refers to the ability of nodes to detect collisions in the network. Collisions occur on a twisted pair star-topology network when the transceiver of the transmitting node detects activity on both the receive- and transmit pairs before it has completed transmitting. On a coaxial tree- or bus-topology network, a collision is detected when the Direct Current (DC) signal level on the cable is the same or greater than the combined signal level of two transmitters. Transmitting nodes that detect a collision will terminate transmission. According to the IEEE standard the transmission time after the collision should be long enough to ensure collision detection by all the transmitting nodes. To ensure this, the transmitting nodes transmit collision enforcement or jam sequence with unspecified content.

The truncated ***Binary Exponential Back-off (BEB)*** algorithm determines the scheduling of packet retransmissions following a network collision. The algorithm ensures that there's a low probability of simultaneous retransmissions. The retransmission delay is an integer multiple of the network slot time [18]. The number of slot times to delay before the n^{th} retransmission attempt (kept by the collision counter) is chosen as a uniformly distributed random integer r between 0 and (2^k-1), where k equals the minimum value between n and 10. After 10 failed attempts the retransmission delay is not increased any further and is frozen at 1023 (2^{10}-1) slot times. After 16 failed attempts, which is the maximum retry limit for 10Mbps and 100Mbps IEEE 802.3 LAN implementations, the transmission is aborted, the packet is discarded and an error message is generated (IEEE Std 802.3, 2000).

5.3 Ethernet Drawbacks for Real-Time Communication

Traditional Ethernet has several limitations that prevent it from being used for real-time communication. These include the following:

- Ethernet is a shared medium network. As a result, collisions can occur between messages sent from different nodes, which make it difficult to guarantee predictable message delivery delays. Collisions are difficult to control in a distributed network because it is difficult to co-ordinated network access (require the network for co-ordination) and the current status of the network is unknown. The network status is always at least as old as the propagation time of a message send between nodes.
- The Ethernet MAC protocol (CSMA/CD) uses a probabilistic approach to resolve collisions. Competing nodes have random back-off intervals when collisions occur. This causes unpredictable network access latency; the worst-case network access time is unbounded (multiple failed resolution attempts). As a result, message transmission is unpredictable. (Note that the collision resolution method does reduce the average latency of packets under light load conditions, but does not guarantee low access delay jitter.)
- The Ethernet Capture Effect is the most aggressive behaviour of the Ethernet MAC protocol and causes increased variability in network access latency. It is the behaviour of the network, when operating under high traffic load, to allow one node to hold on to the network to transmit packets consecutively, in spite of other node(s) contending for access (Ramakrishnan and Yang, 1994). Thus, if a successful node has more packets to transmit and its network interface is fast enough, it will have a progressively higher probability of re-acquiring the network. The cause for this is called the Packet Starvation Effect (Whetten et. al., 1994). The probability of accessing the network is inversely proportional to the back-off

[18] The **slot time** is defined as the unit of time for the Ethernet MAC to handle collisions. It is determined by the parameters of the IEEE 802.3 LAN implementation and for operating speeds of 10Mbps and 100Mbps, the slot time corresponds to the transmission time of 64 bytes, the minimum frame size (i.e. 51.2μs and 5.12μs respectively).

value (determined by number of collisions) and as a result new packets have a higher probability of gaining network access than older packets whose probability exponentially decreases with every failed attempt (collision). The transmission unfairness is most significant for a network with high traffic loads and a small number of active nodes.

- The network access arbitration also does not guarantee network access to any arbitrary node in the network. The duration that any node has to wait is dependent on the network traffic. Also no data throughput rate can be guaranteed for messages that consist of multiple packets.
- No distinction can be made between real-time and non-real-time data. Ethernet packets do not have priorities. This causes priority inversion when high priority packets have to wait for lower priority packets to be transmitted over the network.
- Ethernet frames have variable length that implies variable transmission times. The minimum frame size allowed is 64 bytes (octets) and the maximum 1518 bytes (IEEE Std 802.3, 2000).

5.4 Real-Time Communication Methods

Several methods have been proposed in the literature to improve the predictability of Ethernet for real-time communication. They can be classified and grouped as shown in Figure 14.

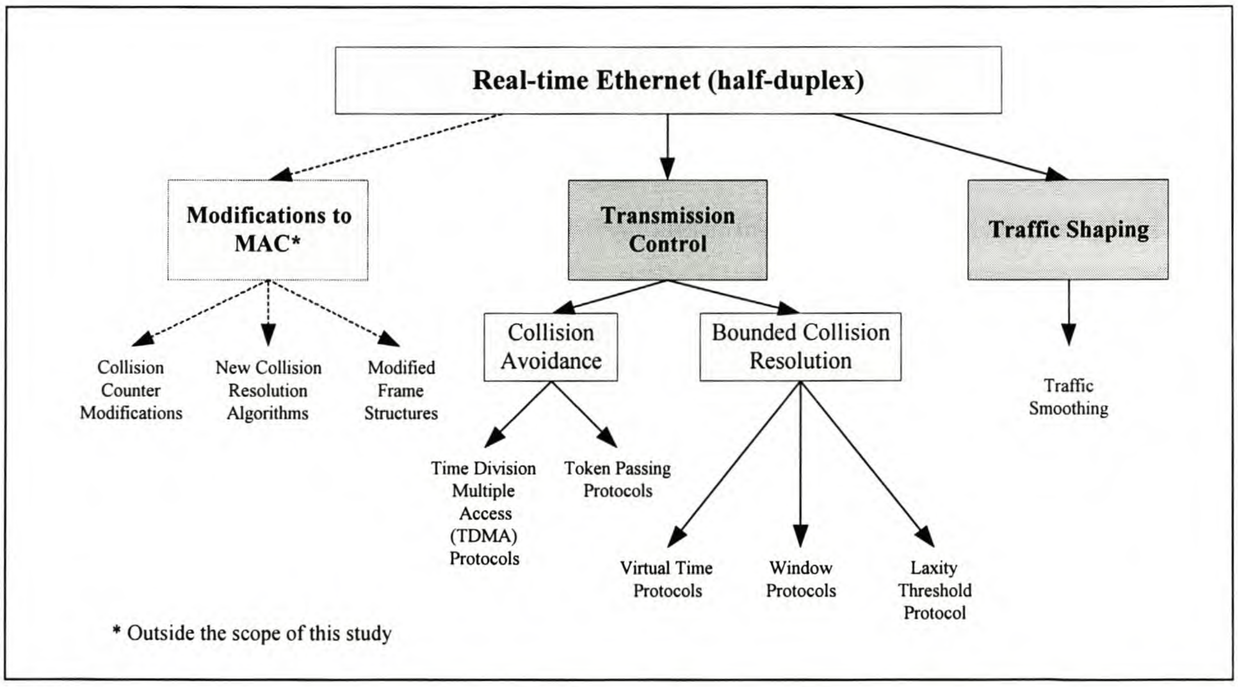
Real-time Ethernet (half-duplex)
Modifications to MAC*
Transmission Control
Traffic Shaping
Collision Counter Modifications
New Collision Resolution Algorithms
Modified Frame Structures
Collision Avoidance
Bounded Collision Resolution
Traffic Smoothing
Time Division Multiple Access (TDMA) Protocols
Token Passing Protocols
Virtual Time Protocols
Window Protocols
Laxity Threshold Protocol
* Outside the scope of this study

Figure 14. Proposed Methods for Real-Time Communication with Ethernet.

The ***modifications to the Ethernet MAC protocol*** include changes to the BEB algorithm's collision counter, replacing the collision resolution algorithms and modifications to the transmission frame structure of real-time messages. These methods are all based on changing the hardware of the Ethernet interface cards and therefore fall outside the scope of the study. For reference purposes, these are discussed in Appendix A, should programmable firmware (such as FPGA's) be considered to implement a shared real-time Ethernet communication network.

Transmission control methods can be subdivided into protocols that seek to avoid network collisions and protocols that seek to resolve collisions in bounded time. Collision avoidance protocols control (co-ordinate) access to the shared network is such a way that nodes never attempt to transmit messages simultaneously. These protocols are deterministic, but can be very inefficient e.g. when nodes have network access, but nothing to send. These inefficiencies can be overcome by protocols that offer transmission rights to a controlled number of nodes simultaneously, but with the possibility of collisions. Collisions are resolved in bounded time using probabilistic approaches i.e. repartitioning some parameter space to determine (isolate) one transmitting node.

Traffic shaping methods are based on the relationship between network utilisation and collision probability. The aim is keep the network utilisation below a given threshold to statistically bind the occurrence of collisions.

The following sections discuss the categorised transmission control and traffic shaping methods in more detail. The information will assist in the comparison study and selection of the method to be used for real-time communication over shared Ethernet.

5.4.1 Time Division Multiple Access Protocols

Time Division Multiple Access (TDMA) protocols partition global system time into predetermined time slots during which only one node can access the shared network (Kopetz, 1997). Time slots are statically assigned to network nodes at design time (static scheduling) and can have different sizes depending on the transmission requirements of individual nodes. The time slots form a sequence that is repeated periodically (called a TDMA cycle). The period of the rotation cycle is determined by the duration of the TDMA cycle. During run-time, all nodes maintain a global time reference to determine their exact time slot. No control messages are used. This requires very precise clock synchronisation between all the nodes in the network.

The ***Time-Triggered Protocol (TTP)*** is an example of a static TDMA protocol designed for real-time communication (Kopetz, 1997). There are two versions of the protocol: TTP/C (full version) and TTP/A (scaled down version). TTP/C achieves integrated and decentralised clock synchronisation by exchanging messages with temporal information. TTP/A uses a centralised, clock synchronisation approach. Hereby, a particular (master) node in the network broadcasts a synchronisation message at the start of every TDMA cycle. The TTP protocols rely on a separate controller per network (the "Bus Guardian") to terminate the operation of any node that tries to send a message outside its TDMA slot.

5.4.2 Token Passing Protocols

Token passing protocols use a token to arbitrate access to the shared communication network. The token is a special ("right to transmit") control packet that is passed between all the nodes in the network and only the recipient node is allowed to transmit packets. Other nodes are only allowed to receive packets and must wait for the token before they can transmit. This method avoids collisions on the network by allowing only one node at a time to transmit.

In the ***Timed Token Protocol*** proposed by Malcolm and Zhao (1994), network access is controlled by a token that is passed between all the nodes in round-robin bases. The protocol is based on the expected rotation time of the token, a parameter known to all network nodes at pre-runtime. The token rotation time is designed to support the response time requirements of all periodic real-time messages in the network[19]. The protocol statically assigns a part of the token rotation time to each node in the network. When a node receives the token, it can transmit periodic real-time messages for no longer than the token holding time. Messages are prioritised and queued at each node in the network. If the token arrives early (the time since the last token was received is smaller than the token rotation time), the receiving node can also transmit aperiodic real-time and non-real-time messages, but only for the duration of the time difference.

Chiueh and Venkatramani (1994) proposed the ***Real-time ETHERnet (RETHER)*** protocol for real-time communication over Ethernet. The protocol transparently switches between the traditional Ethernet CSMA/CD protocol for non-real-time message transmission and a token passing medium access control protocol when real-time and non-real-time messages need to be transmitted.

[19] The token rotation time must be large enough to allow every node in the network to transmit all periodic real-time messages, but also small enough to ensure that the token arrival time at every node is often enough so that all messages meet their deadlines.

All nodes in the network are assigned unique identifiers to resolve race conditions that could occur when multiple nodes try to initiate the switch to real-time mode. In real-time mode, the RETHER protocol distinguishes between two sets of network nodes (real-time and non-real-time). The set of real-time nodes make reservations for real-time message transmission. In the real-time mode, the token holding times for real-time nodes are determined by their bandwidth reservation and processing overheads. Non-real-time nodes have a token holding time equal to the transmission time of the maximum packet size supported by the network, also called the Maximum Transmission Unit (MTU).

The token is passed between real-time nodes based on an array of node identifiers that form part of the token. The last real-time node tags the token with a special time field before passing it to the first node in the set of non-real-time nodes. The time field contains the residual time of the token rotation period that is available for non-real-time nodes to transmit (i.e. the difference between the token rotation time and summation of the token holding times of all the real-time nodes). The first non-real-time node to receive the tagged token determines if there is enough time to transmit a non-real-time message. If there is enough time the message is transmitted, the time field is updated and the tagged token is passed to the next non-real-time node. If there is not enough time to transmit the message, the non-real-time node informs the last real-time node that it should be the first non-real-time node to receive the token in the next token rotation cycle. The token is then passed to the first node in the real-time set to start a new token rotation cycle. Thus the token rotation frequency is different for real-time and non-real-time nodes.

Real-time nodes can also be dynamically added provided that the existing transmission schedule is not affected. New admission control decisions are made locally at each node in the network and successful nodes are added to the array maintained in the token. Real-time nodes terminate transmission of real-time messages by removing them from the set of real-time node information on the token. When the last real-time node terminates transmission, it destroys the token and broadcasts a special control message that switches the network back to the traditional CSMA/CD protocol.

5.4.3 Virtual Time Protocols

In ***Virtual Time CSMA (VT-CSMA)*** protocols each node in the network maintains two clocks, a real-time clock and a virtual clock (Molle and Kleinrock, 1985). The real time clock keeps track of the *current* or *reference* time and is synchronised will all real time clocks in the network. The virtual clock keeps the *virtual* time and behaves in a manner specific to the implemented transmission policy. When the network is busy, the virtual clock is stopped. Once the network becomes idle, the virtual clock is re-initialised and starts running at a rate faster than the real time clock[20].

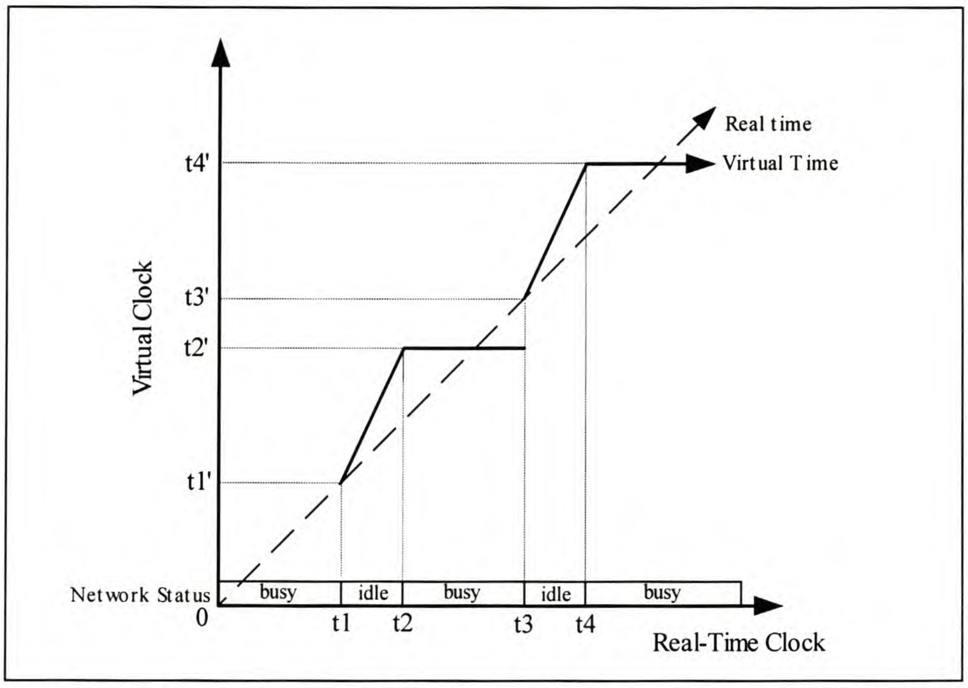

Figure 15. The Operation of the Virtual Clock in Virtual Time CSMA Protocols.

Figure 15 illustrates the general operation of the virtual clock. The virtual clock is stopped when the network is busy. When the network becomes idle the virtual clock is re-initialised to the current real time and set to run at twice the speed of the real time clock. Real-time messages ready for transmission are associated with a Virtual Time to Start Transmission (VTST) parameter. A message characteristic parameter is assigned to the VTST parameter based on the implemented transmission policy. Messages are transmitted when the network is idle and the VTST parameter is equal to the time on the virtual clock. If a collision occurs during the transmission of the message, the node either transmits the message immediately with a specified probability or modifies the VTST parameter of the message to be a random number in an interval specified by the transmission policy. The message is then re-queued for transmission. It should be noted that if the latest time to start transmission of a message has elapsed, the messages is deleted from the waiting queue i.e. the message is lost.

[20] In the VT-CSMA-A protocol the virtual clock runs at a faster rate only when is lags behind the real-time clock. When the virtual and real time clocks are equal, they run at the same rate.

Zhao and Ramamritham (1987) describe four different protocols/transmission policies (with possible message loss) that can be implemented using different message parameters in conjunction with the virtual clock. These protocols have suffixes A, T, L and D respectively and are summarised in Table 6.

Table 6. Overview of Four Different Virtual Time CSMA Protocols.

Protocol	Transmission Policy	VTST Parameter	Description
VT-CSMA-A	Minimum Arrival Time First	Message Arrival Time (MAT)[21]	When the network changes state from busy to idle, the virtual clock is not changed. When a message collision occurs, VTST is modified to be a random number in the interval [current virtual time, LTSM]. After a collision, the message thus still has a chance to meet its deadline requirement.
VT-CSMA-T	Minimum Message Length First	Message Transmission Time (MTT)[22]	When the network changes state from busy to idle, the virtual clock is re-initialised to zero. When a message collision occurs, VTST is modified to be a random number in the interval [zero, MTT]. This effectively give messages involved in a collision a higher priority than dictated by their transmission time.
VT-CSMA-L	Minimum Laxity First	Latest Time to Send Message (LTSM)[23]	When the network changes state from busy to idle, the virtual clock is re-initialised to the current real time. When a message collision occurs, VTST is modified to a random number in the interval [current real time, LTSM]. This gives collided messages still a chance to be transmitted.
VT-CSMA-D	Minimum Deadline First	Message Deadline (MD)[24]	When the network changes state from busy to idle, the virtual clock is re-initialised to the current real time. When a message collision occurs, VTST is modified to a random number in the interval [current real time, MD]. Therefore, a collided message still has a chance of being transmitted.

Simulation studies performed by Zhao and Ramamritham concluded that all the protocols are insensitive to the number of nodes in the network and thus scalable (this is partly because the network-wide load was kept the same for all simulations). It was found that the performance (message loss ratio and network utilisation) of all the protocols is sensitive to the rate at which the virtual clock runs. A high collision rate occurs under heavy network load conditions and when the rate of the virtual clock is high. Of all the protocols, the VTCSMA-D protocol proved to have the best performance in terms of message lost ratio, but it is biased towards short messages.

[21] The **Message Arrival Time** indicates the time when a message arrives at one of the nodes in the network.

[22] The **Message Transmission Time** is the total time needed to transmit a message.

[23] The **Latest time to send Message** is equal to the message deadline minus the message transmission time and the message propagation time (from one end of the network to the other).

[24] The **Message Deadline** is time by which a message must be received by its destination.

5.4.4 Window Protocols

Window protocols use a global, dynamic data structure, called a window, to schedule access to a shared network (Kurose et. al., 1983). The window defines an interval of some message parameter determined by the implemented window protocol. All the nodes maintain the current window and continuously monitor the shared network. If a node has a message with parameters in the current window and it sensed that the network was idle, it transmits the message. The transmission is successful if there was no collision on the network. In the case of a collision, all nodes on the network abort transmission. The window is repartitioned (split) and the protocol deals with these smaller windows separately and recursively. Traditional window protocols have a policy that new arriving messages are not allowed to compete for the network until all the messages in the (split) windows have been transmitted. This is to reduce the number of collisions and variation in response time for messages that are time-constrained.

Zhao et. al. (1990) proposed and simulated a window protocol suitable for real-time communication over multiple access networks such as Ethernet. The protocol implements the minimum-laxity-first transmission policy for time-constrained communication[25]. Message laxity is used for the window parameters and messages with the smallest laxity are prioritised for transmission. Zhao et. al. (1990) proved that the traditional window approach could not accurately implement the minimum-laxity-first policy when based on the latest time to send a message. To avoid the loss of any messages the lower bound of the window are set equal to the current real time. The upper bound is set to the current time plus a chosen initial window size parameter. If a collision should occur, the window size is modified instead of partitioned (as with the traditional window approach). This allows new arriving messages with small laxities to meet their deadlines without having to wait until all the previously competing messages had been transmitted. Also, messages ready for transmission can have the same laxity and will always collide for any window size. To handle laxity ties the message latest time to send parameter is modified to resolve the tie.

[25] The **laxity** of a message at given time t is the maximum amount of time the transmission of the message can be delayed without missing its deadline, given the current time t, i.e. the latest time to send the message minus the current time.

5.4.5 Laxity Threshold Protocol

The ***Pre-emption based CSMA/CD (PB-CSMA)*** protocol uses a probabilistic approach to resolve collisions on the shared network (Ulusoy, 1995). Real-time messages are classified into two priority groups (critical and non-critical), based on their laxity and a predefined threshold. The protocol prioritises critical messages in accessing the network and prevents collisions between critical and non-critical messages.

To achieve this, each node in the network continuously monitors the laxity of real-time messages ready for transmission. Non-critical messages are considered (promoted to) critical when their laxity is smaller that the predefined threshold. The threshold value is determined at design time and should be small enough to limit the amount of critical messages at a given time, but also large enough to allow messages being transmitted before their deadlines. Critical messages are transmitted when the network is sensed idle. Firstly a control message is broadcast to all the network nodes to notify the transmission of a critical message. This ensures that the critical messages will not be pre-empted. If the network is busy due to a non-critical message being transmitted, the network can be pre-empted to allow the critical message to be sent. The pre-emption is achieved by interfering with the transmission (either a short noise burst on the network or forced collision). If a collision should occur, the critical message in retransmitted immediately (but with probability of CSMA/CD protocol).

5.4.6 Traffic Smoothing

Traffic smoothing statistically bounds the medium access time (and thus the delivery of real-time messages) at every node in the shared network. Kweon et. al. (1999) proposed that a real-time message can have a network access time smaller that a predefined bound and with a certain probability if the instantaneous message arrival rate (real-time and non-real-time messages) at all the network interfaces are kept below a certain threshold. The threshold is called the network-wide input limit. A portion of the network-wide input limit, called the station-input limit, is assigned to each node in the network. To keep the message arrival rate below the required threshold, a traffic smoother was developed.

The traffic smoother is installed between the Ethernet MAC protocol layer and upper layer protocols, such as User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) over Internet Protocol (IP), at every node in the network. The traffic smoother implements two message queues to prioritise real-time over non-real-time messages at each node. It also implements a credit bucket to regulate non-real-time message transmission i.e. to reduce the number of collisions with real-time messages of other nodes in the network[26]. Changing the parameters of the credit bucket can control the burstiness of non-real-time message transmission e.g. by keeping the refresh period short and the credit bucket depth small, traffic is smoothed (made less bursty) with a fine time granularity. It is assumed that real-time messages are transmitted pseudo-periodically and at a rate much lower that the capacity of the network (i.e. already smoothed).

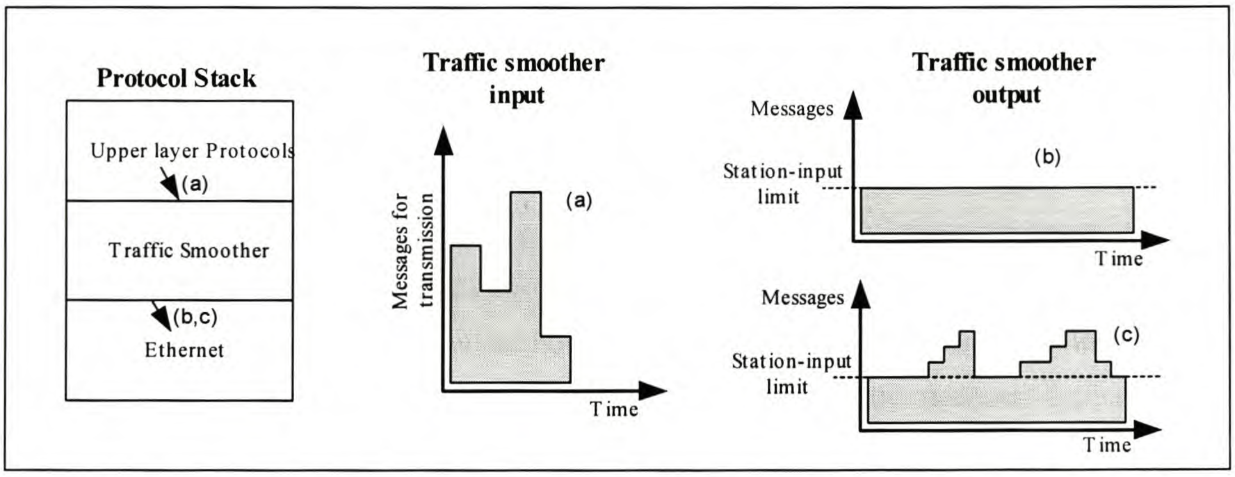

Figure 16. Traffic Smoothing of a Large Burst of Non Real-Time Traffic shown in (a) with (b) a Fixed Rate Traffic Smoother and (c) an Adaptive Rate Traffic Smoother.

The network-wide input limit is determined analytically (and based on network assumptions) and assigned equally or disproportionately among the network nodes, depending on the requirements of each node in the network. With ***fixed rate traffic smoothing*** the station-input limit for every node is fixed by the traffic smoother (Kweon et. al., 1999). However, as the number of nodes increases the station-input limit for every node must be reduced (network-wide input limit fixed). This reduces the allowed non-real-time message throughput per node and can result in unnecessary large transmission delays even when the overall network utilisation is low (i.e. only a few nodes have messages to send, but all nodes were allocated network bandwidth during a certain time period). To resolve the poor scalability of the fixed rate traffic smoother, an ***adaptive rate traffic smoother*** was purposed (Kweon et. al., 2000). This traffic smoother changes the

[26] The credit bucket is similar to the leaky-bucket regulator (Cruz, 1991) and uses credit bucket depth and refresh period as parameters. The credit bucket depth determines the amount of credits each node can store and the refresh period determines the rate at which credits are added to the bucket. Message can only be transmitted if credits are available. Typically messages are passed to the Ethernet layer and credits equal to the size of the message are removed from the credit bucket. If no credits are available, messages are buffered.

station-input limit of each node depending on the current non-real-time network load. If the network utilisation is low, nodes sending non-real-time messages can increase their station-input limit provided real-time messages are not delayed longer than experienced in the fixed traffic smoothing approach. Fixed rate and adaptive rate traffic smoothing are illustrated in Figure 16.

5.5 Comparison of Real-Time Ethernet Methods

A comparison study of the above mentioned methods have been made and are summarised in Table 7.

Table 7. Real-time Communication Methods for Shared Ethernet.

Method	Real-time guarantees	Requirement	Network Efficiency	Network access control	Supported Messages
Time Division Multiple Access (TDMA) Protocols	Hard	Accurate distributed clock synchronisation	High	Node priority	Real-time (Static)
Token passing Protocols	Hard	Control token	Medium to High	Node priority	Real-time and non-real-time (Dynamic)
Virtual Time Protocols	Hard-Firm	Accurate distributed clock synchronisation	Low	Message priority	Real-time and non-real-time (Dynamic)
Window Protocols	Hard-Firm	Accurate distributed clock synchronisation	Low	Message priority	Real-time and non-real-time (Dynamic)
Laxity Threshold Protocols	Hard-Firm	Accurate distributed clock synchronisation	Low	Message priority	Real-time and non-real-time (Dynamic)
Traffic Smoothing	Firm-Soft	Throughput threshold	Low	Message priority	Real-time and non-real-time (Dynamic)

The TDMA and Token Passing protocols provide can predictable hard real-time behaviour based on collision avoidance techniques. The Virtual Time, Window and Laxity Threshold protocols were proposed for hard real-time communication. These methods reduce network collisions by constraining the transmission of real-time messages in a fair way (collisions are part of these protocols). However, when collisions do occur, the worst-case transmission time is much higher than the average. Traffic smoothing statistically bound collisions based on the relationship between bus utilisation and collision probability. This approach provides statistical guarantees and is therefore only suitable for firm to soft real-time requirements.

Various methods require accurate distributed clock synchronisation that in not easily achieved (Levi and Agrawala, 1990). These include the TDMA, Virtual Time, Window and Laxity Threshold protocols. The performance of these methods deteriorates if the distributed clocks are not perfectly synchronised.

The TDMA protocols make the most efficient use of the available network communication bandwidth. Token passing protocols use a control message (token) to gain access to the network. The token overheads reduce network efficiency. Passing the token to nodes, which have no real-time messages to transmit, also causes unnecessary overheads. The other contention-based methods require network resources to resolve collisions. This also leads to under-utilisation of the communication network. In networks with multiple nodes the collision-free TDMA and token passing protocols can offer higher network efficiency than collision-based protocols. This is because the network efficiency decreases exponentially as the probability of collisions increases (Tanenbaum, 1989).

The Token Passing and TDMA protocols allocate access to the communication network based on node priorities. These protocols provide poor support for sporadic real-time messages. The other methods allocate network access based on the priority of real-time messages ready for transmission. TDMA protocols statically reserve communication time intervals before run-time and cannot support dynamically changes in the real-time message set supported by the system. The other protocols can accommodate both real-time and non-real-time messages and (limited) dynamic changes thereof. However, most of these protocols also assume that messages with hard real-time requirements are periodic and should be provided with guaranteed service strategies. Aperiodic messages are considered soft or non-real-time and are provided with a best effort service.

5.6 Motivation for the Token Passing Method

The real-time communication method proposed in this study for shared Ethernet, is based on a token passing protocol. A token protocol has the advantage that collision-free, hard real-time communication can be guaranteed without changes to Ethernet hardware. It is a simple medium access protocol that allows system nodes exclusive access (transmission privileges) to the network when they receive the token. This method has no prior requirements such as carrier sensing, collision resolution or accurate distributed clock synchronisation that can influence the real-time performance. Network access is fair and transmission priorities can also be supported.

Collision based protocols have not been considered due to the high variation (jitter) in network access times. This can cause priority inversion and unpredictable message transmission latencies, even with the potential for message loss. Token passing protocols are also prone to these problems but they are related to the way network access is achieved i.e. nodes have to wait for the token before real-time messages can be transmitted. However, these protocols offer more predictable message transmission latencies (smaller jitter), with an upper bound on transmission delays.

Note that in token protocols the token is also the principal disadvantage because it is the single most critical component in the operation of the network. Losing the token is a major concern because it makes the network inaccessible. Token duplication can also disrupt the protocol operation. To make the network more fault-tolerant both ring maintenance and management (token monitoring) are required.

6 DESIGN AND IMPLEMENTATION OF RTTOKEN

This chapter discusses the design and implementation of a software-based protocol that controls access to a local area multiple access Ethernet network for guaranteed hard real-time communication. The protocol is called RTToken and implements a timed token bus for predictable and collision free transmission medium access control. Performance measurements are discussed in the next chapter.

6.1 Design of the RTToken Protocol

In hard real-time communication, messages have explicit time constraints that must be met or they are considered lost. To meet these time constraints, real-time communication protocols must be deterministic, i.e. real-time messages must be properly scheduled for transmission, and the protocol must have predictable execution times and communication delays. For usability, the communication delays should also be short and with minimal jitter (Malcolm and Zhao, 1994).

The ***Real-Time Token (RTToken)*** protocol has been designed to provide guaranteed bandwidth and deterministic network access to a hard real-time node communicating over a shared Ethernet network. The protocol arbitrates access to the network and determines the transmission time of all real-time messages, with associated predictable and deterministic end-to-end communication latencies and jitter control. It is based on the timed token protocol, initially proposed by Grow (1982).

Timed token protocols are medium access control protocols that circulate a control token between nodes in the network. As the token is passed from node to node it forms a logical ring. At any time, only the node holding the token is allowed to transmit over the network and only for a limited time interval (predetermined or dynamically allocated). Steady state operation thus consists of *real-time message transmission* by the node holding the token and *token transfer*, where the token is passed to the next node. When the cycle is exhausted, the token returns to the first node and starts a new cycle. Timed token protocols are also incorporated into a number of network standards that include the IEEE 802.4 Token Bus and the Fibre Distributed Data Interface (FDDI) standards (ANSI/IEEE Std 802.4, 1990; ANSI Std X3.139, 1987).

Having a timed token protocol over Ethernet results in a **token bus** as illustrated in Figure 17. Physically the network has a linear bus topology, but logically it is organised as a token ring. The token passing mechanism avoids collisions and thereby the associated non-deterministic behaviour of the Ethernet MAC protocol (CSMA/CD) on the Ethernet network. In the context of the study, Ethernet refers to a single segment of an Ethernet network. Thus the RTToken protocol is not designed to support Ethernet networks that span across bridges, switches and/or routers.

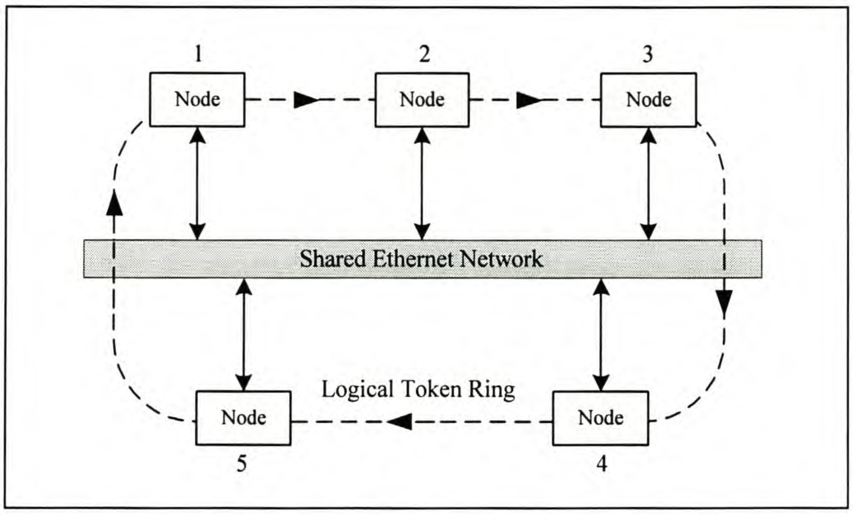

Figure 17. Logical Token Ring on the Physical Ethernet Bus with the RTToken Protocol.

6.1.1 General Design Considerations

Hard real-time communication protocols are specifically designed to satisfy the time constraints of individual real-time messages. Efficient network utilisation and protocol flexibility is of lesser concern (Kopetz, 1997). The design of the RTToken protocol is also based on these principles.

The protocol is not designed for improved network utilisation. Several real-time token protocols try to make full use of the available network bandwidth by simultaneously supporting both real-time and non-real-time communication. When no real-time messages are scheduled for transmission, the residual network bandwidth is used to transmit non-real-time messages (Chiueh and Venkatramani, 1994). The RTToken protocol however, offers no support for non-real-time messages. This reduces the chance of priority inversion whereby a hard real-time message that's ready to be transmitted, must wait for the completion of a non-real-time message transmission. Residual network bandwidth can be made available for the transmission of sporadic real-time messages, but otherwise it is unutilised (note that sporadic real-time messages are also affected by the network access control method). All nodes have to wait for the arrival of the token before any real-time messages can be transmitted.

Protocol flexibility refers to the ability of the protocol to accommodate network changes without restarting the network. The RTToken protocol is designed to be inflexible and only operates in a static network environment. Dynamic changes in the set of participating nodes are not supported. The reason is that variations in the active network nodes change the real-time message set of the overall systems, making it difficult to guarantee that the time constraints of all hard real-time messages can be met. Dynamically adding and removing nodes can also cause communication disruptions. For example, when a node wants to add itself to the network it must interrogate the status of the network and introduce itself. In both cases, this is typically done with broadcast and reply messages on the network. This process can cause a period of indeterminacy in the network, possibly leading to the loss of the token or real-time messages.

The use of commercial and unmodified Ethernet hardware also imposes a constraint on the design of the RTToken protocol. This is because the physical layer of the Ethernet protocol-stack implements the unpredictable CSMA/CD protocol in hardware. To achieve deterministic network communication, the RTToken protocol must control network access from the upper protocol layers in the Ethernet stack. It is also necessary that all nodes in the network implement the RTToken token-based network access control to ensure that there are no collisions on the network.

6.1.2 Token Handling

Token handling in timed token protocols is described with token holding times, token arrival intervals and token rotation times (Malcolm and Zhao, 1994). The relationship between these parameters is illustrated in Figure 18:

- The **token holding time** is the time duration that a node is allowed to transmit over the network after it acquired the token.
- The **token arrival interval** is the time duration between two consecutive token visits at each node in the network.
- The **token rotation time** is the time required for the token to circulate all nodes in the network. It includes the token holding times of all nodes and other overheads associated with the processing and passing of the token[27]. In most timed token protocols, the communicating nodes receive the token once during a token rotation time and in a predetermined order

[27] The token rotation time is also known as the network bandwidth allocation interval. It is periodic and therefore indicative of the overall token rotation frequency.

(Malcolm and Zhao, 1994). In this case the token rotation time is also equal to the token arrival interval for each node in the network.

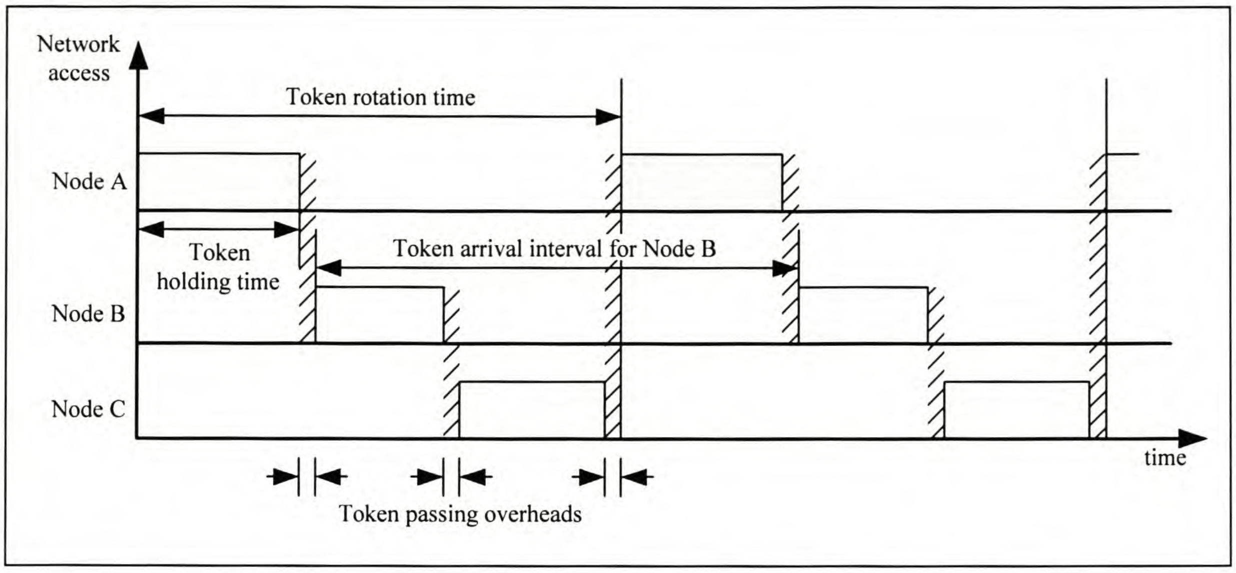

Figure 18. Relationship between the Timed Token Protocol Parameters.

The token holding time and token arrival intervals are closely related in guaranteeing hard real-time message transmission. On the one hand, nodes must be allocated adequate token holding times to complete the transmission of all real-time messages before their deadlines. On the other hand, the allocated token holding times should not increase token arrival intervals such that it causes real-time messages on other nodes to miss their deadlines.

The token holding times are determined using **network capacity allocation schemes**. These schemes are classified based on the type of information used to determine the token holding time per token arrival. An allocation scheme is considered *local* if token holding times are computed without using information of real-time messages on other nodes in the network. A *global* scheme utilises system-wide information, which includes the real-time message information of all nodes in the distributed real-time system (Agrawal et. al., 1992). The selection of a network allocation scheme is determined by the targeted application requirements. The scheme should always be able to guarantee the requirements of the given real-time message set.

The RTToken protocol is designed to statically assign fixed token holding times for every instance that a node receives the token. The token arrival times are more deterministic and nodes can now send a bounded, but variable amount of real-time messages with variable size, without influencing the relative token arrival times of other nodes and/or the token rotation time. Also, changes to the local token holding time of any node do not affect the token holding times of other nodes. If the token arrival time can be both predetermined and predictable, the token network can also be used to (periodically) time-synchronise all the distributed nodes. The disadvantages of having fixed token holding times are under-utilised network bandwidth and global priority inversion that can occur. A node holding the token can transmit real-time messages in order of priority, but with lower priorities than messages waiting for transmission on other nodes in the network. However, priority inversion is only a problem when time constraints of real-time messages cannot be satisfied.

The token holding times are empirically calculated based only on local periodic real-time message information. It depends on message delays and the number of real-time messages to be transmitted. The message delays depend mainly on the implementation of the communication system, including parameters such as message sizes, transmission times, method of transmission (fragmented packets or not), processing delays, queuing delays and others. Note that the RTToken protocol does not support fragmented messages and packet sizes are limited to the MTU of 1500 bytes, as specified for the Ethernet protocol (IEEE Std 802.3, 2000).

In general, the token arrival interval for every node is designed to be sufficiently small in order to support the real-time message transmission requirements of all the nodes in the network. The consideration of using global real-time message information depends on the application requirements. For example, if a distributed real-time application is designed with a required token rotation time, the overall sum of all token holding times should not be greater than the token rotation time.

Under less restricted conditions, the token holding times can be designed to also allow for the transmission of sporadic real-time messages. Additional transmission bandwidth can be allocated by simply increasing the token holding times of corresponding nodes in the network. The condition is that all real-time messages in the system are still guaranteed to meet their deadlines. Naturally such reservations result in the under-utilisation of network bandwidth when no real-time messages are ready for transmission. *In general token passing protocols are better suited for real-time applications with periodic message transmission requirements.*

The RTToken protocol also allows network nodes to hold the token more than once during a token rotation time. This is useful where nodes are required to periodically transmit real-time messages at rates higher than the token rotation time, as illustrated in Figure 19. This flexibility however, adds additional complexity. Nodes can have multiple token arrival intervals and/or token holding times. For example, in the figure Node A has two token holding times per token rotation time with both different durations and token arrival intervals. To guarantee that all real-time messages will meet their deadlines, separate message queues for every token holding time need to be maintained on each node. The additional token passing overheads also have to be taken into consideration when applications are designed with multiple token holding times per token rotation time. The token passing reduces the available network bandwidth for real-time message transmission and can unacceptably increase the token arrival intervals.

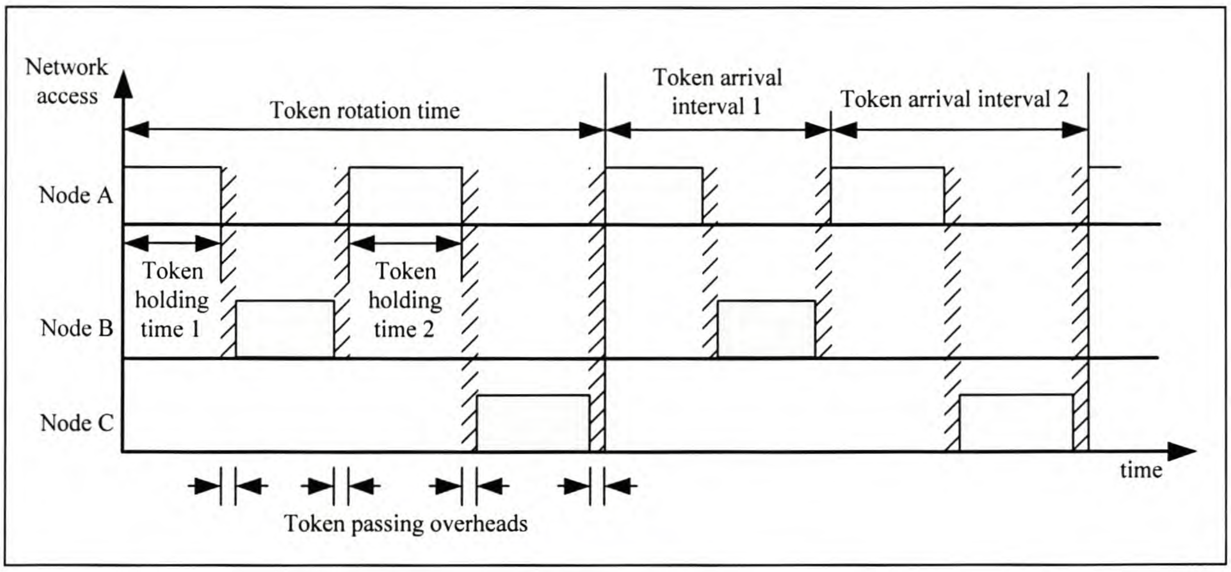

Figure 19. Variable Token Holding Times and Arrival Intervals for the RTToken Protocol.

6.1.3 Network Set-up and Termination

The real-time network is configured with all nodes supporting the RTToken protocol. Participating nodes are identified with a 6-byte Ethernet hardware address that is unique to every network interface card. These addresses are used to route the control token between the nodes. Systems with multiple network interface cards are treated as separate nodes hosted on the same hardware. Nodes are also assigned unique 4-byte IP addresses. This is because real-time messages are transmitted over the Ethernet network using connectionless UDP/IP protocols and these have different addressing schemes[28]. A mapping from the IP address to the corresponding

[28] The RTnet networking subsystem described in Section 6.2.1 supports the transmission of real-time messages over UDP/IP.

Ethernet hardware address is therefore also required. Unfortunately the dynamic address mapping method with the standardised Address Resolution Protocol (ARP) cannot be used, because of its unpredictable nature (Stevens, 1994). As a result, the RTToken protocol is designed with a deterministic address resolution scheme between IP and Ethernet addresses. Static address mappings are done at every node during network initialisation.

With RTToken, one node is designated to be the token creator and token passing initiator. The assignment is done as part of the initial network setup. Hereby token duplication is avoided and there can be no race condition between multiple nodes that want to be the initiator. Alternatively, all nodes can compete to be the token passing initiator, compensating for the possible failure of a designated token passing initiator node and as a result the network initialisation. Hereby all nodes initially support the Ethernet CSMA/CD MAC protocol. The network is switched to support the RTToken with a request to do so from any of the participating nodes. The requesting node becomes the initiator and broadcast a request to all nodes to stop further data sending and to switch to the RTToken protocol. After the nodes switch to the RTToken protocol, an acknowledgement is sent to the initiating node. The initiating node receives all acknowledgements, distributes the routing table and start circulating the token. Multiple simultaneous requests to switch to the RTToken protocol are handled by assigning static priorities to nodes. The requesting node with the highest priority gets preference over the rest. This approach is similar to that of the RETHER protocol (Chiueh and Venkatramani, 1994). *For the purpose of the study, fault tolerance for network initialisation was not researched. This is a topic for further study.*

The initiator node also generates a static routing table for token passing. The routing table is a fixed schedule to route the token from node to node through the network. It contains the token rotation sequence and token holding times of all participating nodes for one token rotation time. The routing table is broadcast to all participating nodes in the network. Alternatively, the routing table can be added to the token. However, this means that the token size now depends on the size of the routing table, resulting in variable token transmission delays for different set-ups and additional token processing overheads. The RTToken protocol is therefore designed with a fixed token size equal to the minimum size for Ethernet packets (i.e. 64 bytes). This minimises token passing overheads and transmission delay variations. The token only has an index field that relates to the distributed routing table. All the nodes read and update the field to ensure the correct token rotation sequence. Having a static routing table also contributes to the determinism of the real-time communication system, but as mentioned previously, it has the disadvantage that nodes cannot be dynamically added or removed from the network. Once the routing table is successfully distributed to all nodes, the initiator node generates the token, holds it for the node's allocated time and passes it to the next intended recipient.

The start-up sequence for the RTToken network is as follows:

1) Before run-time the token routing table and token holding times for all participating nodes are finalised. A node is selected to be token passing initiator and configured accordingly.
2) At run-time, all other participating nodes (excluding the token passing initiator) are initialised and configured to wait for the reception of the token routing table. An error report is generated if a node cannot be initialised properly. Also note that the real-time application running on each node is configured to start its real-time task set only after the token passing network is activated.
3) To start the network the token passing initiator is initialised. This node then broadcasts the token routing table to all nodes in the network and allows sufficient time for these nodes to upload and configure their localised routing tables.
4) The token passing initiator node then creates the token and holds it for the allocated token holding time before passing it on the next recipient node.
5) Once the RTToken protocol is active, all real-time tasks are signalled to start execution.

Upon termination, a node destroys the token and broadcasts an error message. All nodes in the network deactivate the RTToken protocol and no further network transmissions are allowed.

6.1.4 Admission Control

A real-time task (called the agent), with the highest priority, performs admission control on every network node. When a node receives the token, an acknowledgement message is sent to the node that last had the token and the local agent is signalled. The agent initiates the sending of real-time messages. The agent loads a timer with the allowed token holding time and suspends. When the timer expires, all further network transmissions are halted. The agent resumes by sending the token to the next node. Because the agent task can effect the execution of other real-time tasks, it is designed to cause minimal disturbance.

The successful transmission of the token indicates that a node had no pending packets in the network interface card that could cause collisions. This is important because the software has no control over data packets once they have been copied to the network interface buffer and a transmit-signal has been issued to the card.

6.1.5 Fault Detection Mechanisms

The RTToken protocol has also been designed with limited fault detection mechanisms. In token passing protocols, there is always a chance that the token can get lost or corrupted. Losing the token is a critical fault that makes the network inaccessible. This is especially true for RTToken, where the token is passed unconditionally (it is assumed that the receiving node is always available). To prevent undetected token loss, a monitoring node is introduced. The RTToken protocol uses decentralised token monitoring, whereby the last node to have the token becomes the monitor. Multiple monitoring nodes have not been considered because the additional overhead at each node in the network is not justifiable (all nodes must then monitor every token being passed).

After a node forwards the token to the next recipient, it sets a timer and waits for an acknowledgement message from the node that the token was forwarded to. The time-out value of the acknowledgement timer determines how responsive the system is to the detection of token losses. When the acknowledgement timer expires, the monitoring node checks to see if a token acknowledgement message was received and acts accordingly. Under normal conditions the recipient node will receive the token and immediately send an acknowledgement message back to the node that it received the token form.

When the node successfully received an acknowledgement message, it stops monitoring the token. If no acknowledgement message has been received before the timer expires (e.g. the next token recipient is unavailable or died), the monitoring node enters a safe error recovery mode. The RTToken protocol is designed to signal all local hard real-time tasks of a network error and to broadcast an error message to all nodes in the network. The experimental system is implemented to stop all real-time tasks and any further network transmissions. To resume, the network has to be reinitialised.

Conditions can also occur where the token gets lost even after the acknowledgement was sent to the sending node. For example, where a node goes off-line after sending the acknowledgement, but before sending the token to the next node. Such errors can be detected by either having the initiating node continuously monitor the token rotation time or by having all the nodes actively monitor their next expected token arrival times. Interference on the physical cabling can also corrupt transmissions. The RTToken protocol does not offer support for these types of errors. Only basic token loss detection was implemented. *Improving fault tolerance is a topic for further study and falls outside the scope of this research.*

6.2 RTToken Implementation

This section describes the software implementation of the RTToken protocol. The protocol is implemented as an extension to the Real-Time Network (RTnet) networking subsystem of the RTAI hard real-time operating system.

6.2.1 RTnet Networking

RTnet is an open source modification of the standard networking subsystem of the Linux operating system (LinuxDevices, 2000). It supports a Berkley Software Distribution (BSD) socket API implementation that allows real-time tasks executing in kernel mode, to send and receive real-time messages over standard IP networks. Currently, RTnet is still beta software and is thus in a pre-release testing phase (SourceForge, 2003). It is however considered stable enough for use in some real-time applications where a level of risk is acceptable. Figure 20 illustrates the protocol stack of RTnet and the correspondence in network functionality, to the Open Systems Interconnection reference framework of the ISO[29].

The RTnet networking subsystem also provides deterministic support for the UDP and IP protocols. These protocols are used to transmit real-time messages between distributed real-time tasks. Limited non-real-time support is offered for ARP and Internet Control Message Protocol (ICMP). Hereby nodes supporting RTnet can handle remote ARP requests, but cannot generate such requests dynamically. ARP requests to all participating nodes must be forced at network initialisation. RTnet can also receive and reply to ICMP echo messages (i.e. respond to pings), a useful feature when configuring a private real-time network.

[29] The BSD socket layer simplifies network communication and the porting of network applications. The INET socket layer manages the communication end points of the UDP layer or IP layer directly; represented by the *sock* data structure (Beck et. al., 1996).

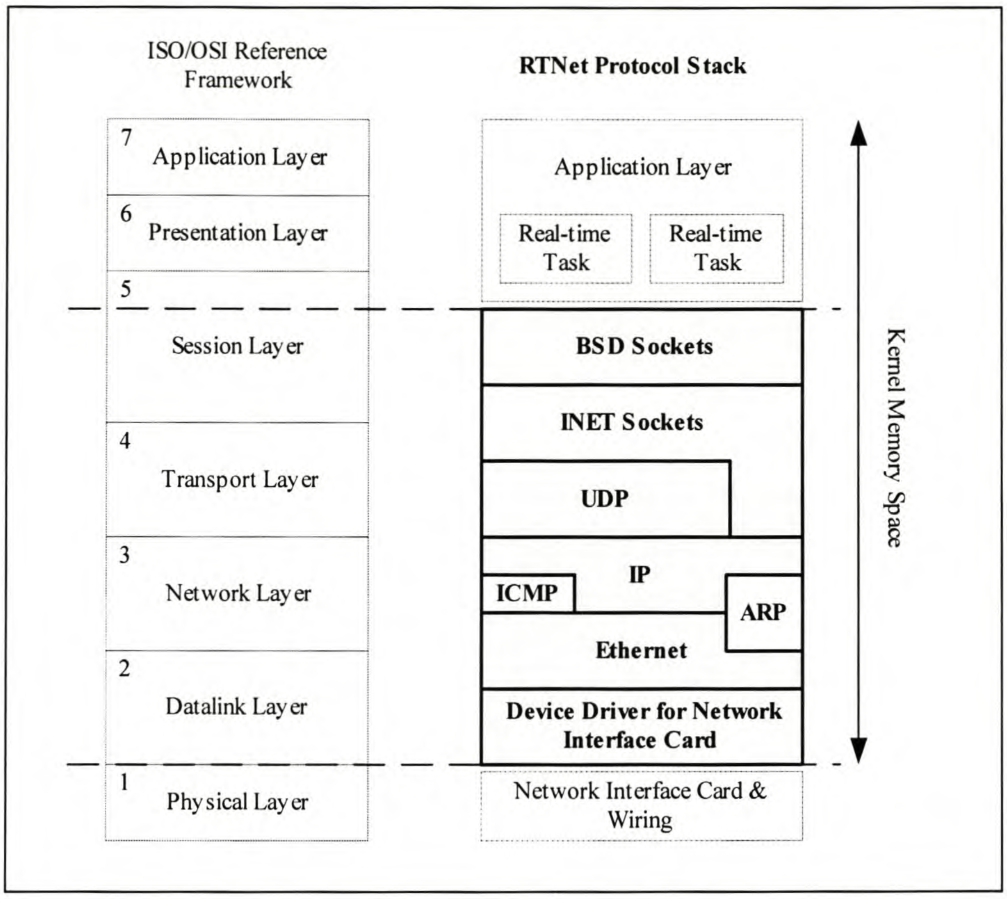

Figure 20. Protocol Stack of the RTnet Networking Subsystem.

A number of Ethernet cards are supported including DEC tulip, RTL8139 and NE2000-based network interfaces. The device drivers for these network interface cards have been modified for hard real-time communication. The method of packet transmission and reception of the modified DEC tulip device driver is briefly discussed[30]. Under light to moderate network load conditions, no buffer overflow should occur. The RTnet protocol pre-allocates buffer space for RTSKB_LEN (1000) Ethernet packets (defined in core/skbuff.c).

Packets for transmission are contained in a complex socket buffer structure called *sk_buff* (linux/skbuff.h). In this context, socket refers to a UNIX abstraction used to represent a network connection, not the API sockets described previously (Rubini and Corbet, 2001). To transmit a data packet, the *rt_dev_queue_xmit* function (rtnet/core/dev.c) is called with a pointer to the socket buffer structure. The transmission function then transmits the packet immediately through the hardware related function *tulip_start_xmit* (drivers/net/tulip_rt.c).

[30] Only DEC network interface cards have been used in the test environment, discussed in the next chapter.

Ethernet packets are received through an interrupt-driven operation. The interface card does Ethernet frame error checking before an interrupt is generated. The RTnet protocol registers the hardware interrupts of the interface card as hard real-time. When a packet is successfully received, the interrupt service routine called *tulip_interrupt* (drivers/net/tulip_rt.c) allocates and copies the packet data into the *sk_buff* socket buffer structure. The *dev* and *protocol* fields in the socket buffer structure are also assigned to identify the originating interface card and type of Ethernet packet. The interrupt service routine then calls the function *rtnetif_rx* (rtnet/core/dev.c) that signals a real-time task *rxtask* (rtnet/socket.c). The real-time task is assigned highest scheduling priority and as a result pre-empts all lower priority real-time tasks. The *rxtask* calls the *rtnet_bh* function (rtnet/core/dev.c) that initiates the appropriate protocol handler for the received packet. Because the protocol handler is called from within *rxtask*, it executes with the same priority level.

6.2.2 RTToken Access Control Protocol

The RTToken protocol extension to the RTnet protocol is illustrated in Figure 21.

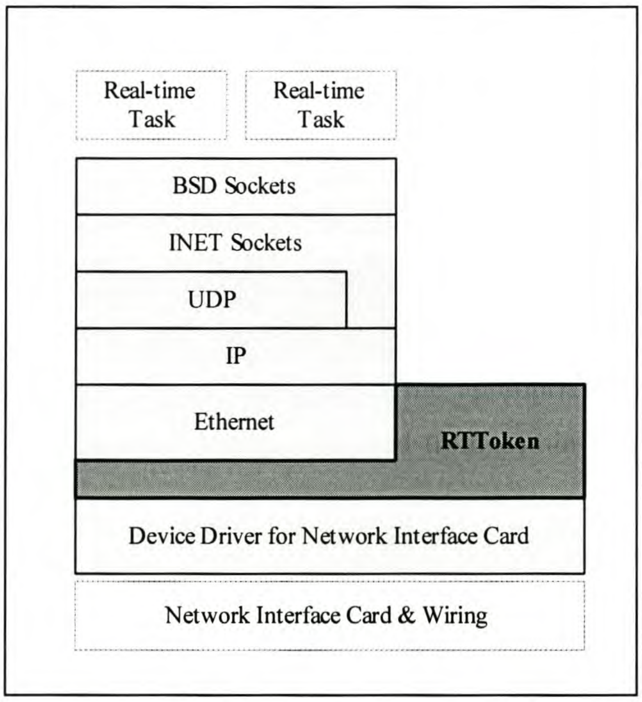
Real-time Task
Real-time Task
BSD Sockets
INET Sockets
UDP
IP
Ethernet
RTToken
Device Driver for Network Interface Card
Network Interface Card & Wiring

Figure 21. RTToken Protocol Stack.

The functionality of the RTToken protocol is implemented in the corresponding datalink and network layers of the RTnet protocol. The code is implemented in two files: *rttoken.c* and the associated *rttoken.h* header file. Other changes to the RTnet code are clearly indicated. The

RTToken software is compiled and linked to the RTnet protocol, and therefore also installed in kernel memory space as a dynamically loadable kernel module.

The RTToken protocol controls access to the shared Ethernet network. Its operation is depicted in Figure 22.

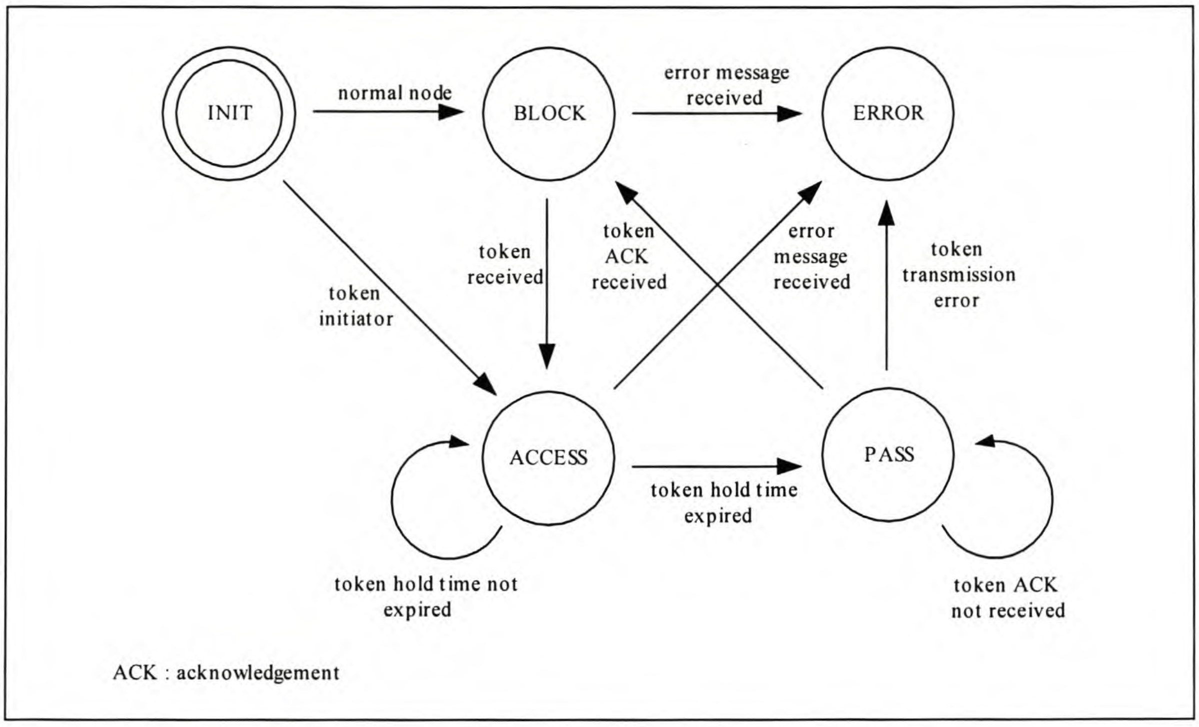

Figure 22. State Diagram of the RTToken Protocol.

In the *INIT* state, all network nodes are initialised. After initialisation, the initiator node's state changes to *ACCESS* (node is allowed to access the network - only one node can be in this state at any time), while the rest of the nodes change their state to *BLOCK* (node is *not* allowed to access the network). Since the initiator node created the token, it is allowed to be the first to access the network. If an error message is received in any of the two states, the node's state changes to *ERROR* (node performs error handling and all further network participation is ceased).

If a node in the *BLOCK* state receives the token, it sends a token receipt acknowledgement and changes state to *ACCESS*. This node is now allowed to access the network for the duration of the allocated token holding time. When the token holding time expires, the node changes state to *PASS* (the token is passed to the next recipient node which is in *BLOCK* state, and then the node waits for an acknowledgement that the token was successfully received). If the node receives no acknowledgement, an error message is broadcast and the node changes its state to *ERROR*. All the other nodes in the network will then change state from *BLOCK* or *ACCESS* to *ERROR* when the error message is received.

The following sections discuss the functions and data structures of RTToken in more detail.

6.2.3 The Token Structure

The RTToken protocol uses a specific data structure for the token and other message types. This structure has the following fields:

```
struct rttoken_hdr_t {
        unsigned char rtt_type;                 /* Token operation type */
        unsigned char rtt_txr[ETH_LEN];         /* Token sender hardware address */
        unsigned char rtt_rxr[ETH_LEN];         /* Token receiver hardware address */
        unsigned char rtt_inr[ETH_LEN];         /* Token initiator hardware address */
        unsigned char rtt_index;                /* Token routing table index */
        unsigned char rtt_index_max;            /* Number of routing table entries */
        unsigned char rtt_len;                  /* Token message length */
        unsigned short rtt_csum;                /* Token header checksum */
};
```

The token is designed to be equal to the minimum allowed Ethernet frame size, allowing 46 bytes (octets) to be allocated for the data structure. However the data structure of the token is only 24 bytes long[31]. This includes fields that have been added for information only. Additional padding by the device driver ensures that the frame is long enough for proper Ethernet operation. Figure 23 illustrates the frame format of the RTToken protocol token and the encapsulation in the minimum allowed Ethernet frame.

The *rtt_type* field defines four different message types that use the data structure. The fields in the data structure that are not applicable to a specific message type are simply set to zero. The supported message types are:

- *RTTOKEN_SEND* – Defines the token of the RTToken protocol. At any time there is only one such token message in the network.

[31] The constant *ETH_LEN* is set to 6, the size of the Ethernet hardware addresses.

- *RTTOKEN_ACK* – Defines the acknowledgement message for successful token reception.
- *RTTOKEN_SETUP* – Defines the message as containing the routing table.
- *RTTOKEN_ERROR* – Defines the message as a critical error message.

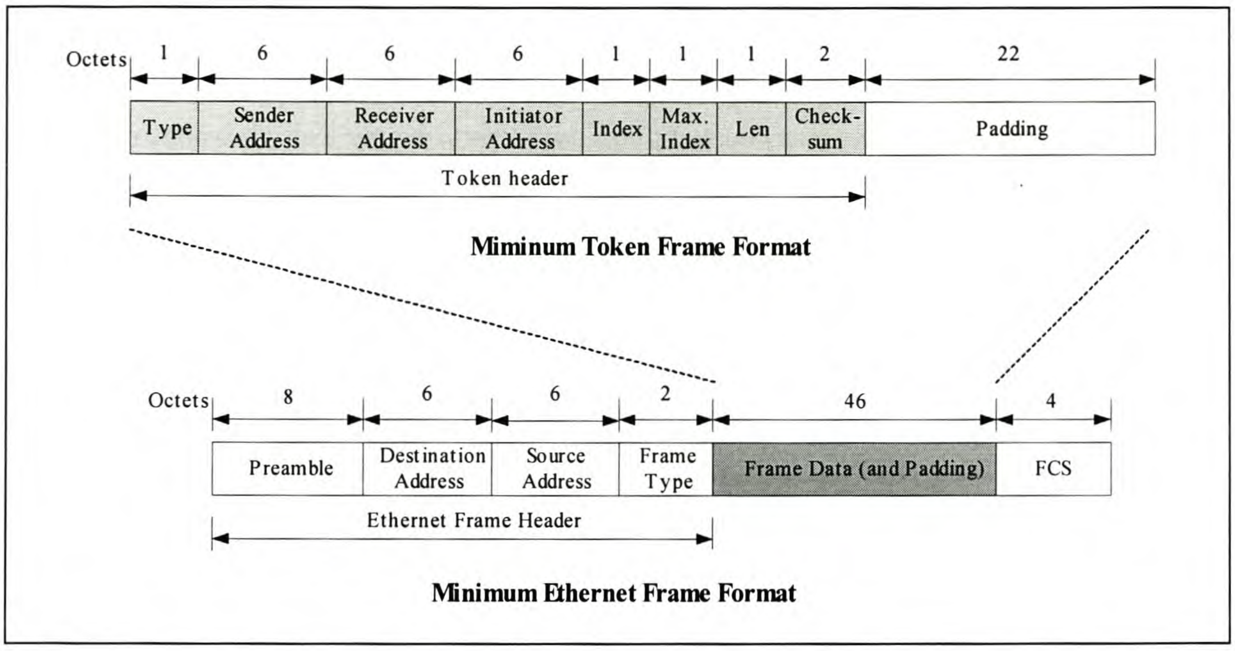

Figure 23. The Token Frame Format of the RTToken protocol.

The data structure further has three address fields. The hardware address of the last node that had the token is contained in the *rtt_txr* field. This address field is required to send the token receipt acknowledgement to the correct node. The *rtt_rxr* field holds the hardware address of the node that received the token. This field ensures that the correct node receives the token. The hardware address of the token passing initiator is contained in the *rtt_inr* field. Currently this field is added for information purposes only.

The *rtt_index* field contains the routing table index value. The value is indicative of the current token location in the token routing sequence; therefore all nodes update it. When a node receives the token, it increments the index value in order to search the routing table entries for the next token destination. If the incremented index value is greater than the maximum allowed index number, the index value is set to zero. This indicates that the token has completed its rotation cycle and must be passed back to the token initiating node (index value of zero). The maximum index number of the token routing table is contained in the *rtt_index_max* field. This field is included for information and for cross-referencing the maximum allowed index numbers.

The data structure also includes the *rtt_len* field that contains the size of a message that can be concatenated to the token structure. This field is used to transmit the routing table and indicates the length thereof. Otherwise it is set to zero. The maximum message size (routing table size) is 1486 bytes, determined by the difference between the MTU of the Ethernet frame and the data structure size (24 bytes). Note that no padding is needed if the data transmission unit is larger than the minimum requirement.

A simple error-detection method has been added to the data structure to verify the correctness of the information. The *rtt_csum* field contains a 2-byte checksum value of the data structure. Using a checksum has the advantage that computation is simple and efficient, but multiple errors cannot be detected.

The checksum computation is done with the *csum_rtthdr* function. The function is similar to that described by Braden et. al. (1988). Data in the token structure is paired to form 2-byte integers. To generate a checksum, the 2-byte sum is computed over the token header excluding the checksum field. The checksum result is the 1's complement of this sum and is placed in the *rtt_csum* field. To check a checksum, the 2-byte sum is computed over the token header including the checksum field. If the 1's complement of the summation is equal to zero, the check succeeded. The function also caters for a possible overflow during summation.

Note that the Ethernet frame also includes error detection with a 4-byte cyclic redundancy check (CRC). However, this only ensures that the data is transmitted correctly, not that the data submitted for transmission is correct. It could for example happen that the content of the data structure is erroneously changed before being submitted to the network driver for transmission.

6.2.4 Token Routing

The token's routing table has the following fields and is defined in the header file:

```
struct rttoken_rtable_t {
        unsigned char index;                    /* Table entry index */
        unsigned char node_mac[ETH_LEN];        /* Ethernet hardware  address */
        unsigned int token_holdtime;            /* Token hold time */
        struct rttoken_rtable_t *next;          /* Pointer linking the routing table entries */
};
```

The *index* field is used to pass the token between nodes in the network. The token initiator is always the first entry in the routing table, indexed as zero. Other routing table entries are automatically indexed and in incremental order. Thus, the sequence in which routing table entries are made also determines how the token will be passed between nodes in the network. The *node_mac* field holds the hardware address of the node. It is used to identify the correct recipient of the token. The *token_holdtime* field contains the time duration in nanoseconds that the node will hold the token before passing it on to the next node. The **next* field links the routing table entries.

The following token routing functions are available:

- *rttoken_rtable_init* – Initialises and links an empty array of routing table entries. A pointer variable is assigned the memory address of the first empty entry.
- *rttoken_rtable_route_add* – Adds a new entry to the token routing table. The maximum number of routing table entries is defined by the constant MAX_RTABLE, which is set to 10. The function returns an error if there's no more space available for further entries. Note that the size of each routing table entry is 16 bytes[32]. The total allowed number of routing table entries is limited to 92 by the MTU of the Ethernet packet because the RTToken protocol does not support fragmentation[33].
- *rttoken_rtable_broadcast* – All the routing table entries are broadcast with a packet structure similar to the token. The function builds the token header with the type field indicating that the packet contains the routing table, and adds the routing table as the token message. The packet is then encapsulated in an Ethernet frame for broadcast (destination hardware address in hexadecimal: ff:ff:ff:ff:ff:ff) and passed directly to the network interface device driver for transmission.
- *rtoken_rtable_lookup* – Sequentially searches the routing table for a matching index number and returns a pointer to the specific entry. Error checking is included, but errors should never occur.

6.2.5 Node Initialisation

The RTToken protocol supports two functions to initialise participating nodes:

[32] The size of each routing table entry is 1 byte larger than expected (1+6+4+4 bytes) because the compiler is allowed to pad a structure in order to achieve word or paragraph (16 bytes) alignment.

[33] The maximum allowed message field for routing information is 1486 bytes.

- *rttoken_initiator* – Initialises the token initiator node. This function updates the local node information, initialises a routing table for the node and adds itself as the first entry. In a standard application, this function is executed together with function calls to add more routing table entries (*rttoken_rtable_route_add*), to broadcast the routing table (*rttoken_rtable_broadcast*) and to start the RTToken protocol (*rttoken_start*). These additional functions were excluded from the main function call to ease implementation (variable passing), since real-time applications are implemented as separate loadable kernel modules.
- *rtttoken_node_init* – Employed to initialise the other normal nodes in the network. This function updates local node information and initialises the routing table.

6.2.6 Token Transmission and Reception

The RTToken protocol uses the *rttoken_send* function to send the token, token receipt acknowledgement, routing table and critical error messages. The function allocates a buffer for the data to be transmitted from a statically allocated pool of memory. The function then constructs a complete Ethernet (not 802.3) frame as shown in Figure 23. The type field of the Ethernet frame is assigned the constant ETH_P_RTTOKEN (0x0810). This value was chosen larger than the maximum Ethernet frame size so that the receiver can uniquely identify the frame and not mistake the type field for a frame size indication, since both Ethernet and 802.3 frame types are supported. The buffer holding the constructed Ethernet frame is then passed directly to the device driver of the appropriate Ethernet network interface card for transmission.

To receive token packets, the RTToken protocol registers the *rttoken_rcv* function as the protocol handler at network initialisation. The protocol handler is implemented as a bottom handler for a high priority real-time task (*rx_task*) and therefore executes with the same priority level (highest). The real-time interrupt service routine of the relevant network-interface card signals the latter task. As was mentioned before, RTToken protocol packets, encapsulated in Ethernet frames, are identified with the constant ETH_P_RTTOKEN. When these packets are received, the *rttoken_rcv* function gets called with a pointer to the buffer that holds the packet.

The *rttoken_rcv* function extracts the packet information required for the checksum. If the checksum is unsuccessful, a critical error is registered and handled accordingly. For the experimental system, these errors are handled with the *rttoken_stop* function[34]. Otherwise, actions are performed based on the type field (*rtt_type*) of the RTToken protocol header:

- *RTTOKEN_SEND* – When the node receives a token, all relevant token information is extracted and the buffer freed. The function then signals the admission control agent. The agent sends an acknowledgement of token receipt to the node that forwarded the token. It then allows real-time tasks to access and transmit messages over the network.
- *RTTOKEN_ACK* – The receipt of acknowledgement is noted and the associated buffer freed.
- *RTTOKEN_SETUP* – When a set-up message is received, the local routing table is re-initialised (cleared). All routing information is extracted and used to re-configure the token routing table. Finally the buffer is freed.
- *RTTOKEN_ERROR* – When the nodes receive a critical error message the RTToken protocol is deactivated and the buffer freed.

Other unknown RTToken packets should be handed over to appropriate error handling functions. These are application specific and not implemented in the experimental test system.

6.2.7 Token Acknowledgement Timer

The RTToken protocol supports the following functions for the token acknowledgement timer:

- *rttoken_acktimer_init* – Initialises the real-time task that implements the token acknowledgement timer. The timer is disabled until needed.
- *rttoken_acktimer_cleanup* – Deletes the real-time task that implements the timer.
- *rttoken_acktimer_start* – Starts the token acknowledgement timer.
- *rttoken_acktimer_do* – Implements the timer functionality. The timer is implemented by suspending a real-time task for the required duration. The timeout value is determined by the constant ACK_TIMEOUT. When the task resumes, it checks to see if the node received an acknowledgement message and acts accordingly.

6.2.8 Admission Control

The RTToken protocol performs network admission control with a real-time task (*agent)*, registered with the highest scheduling priority. The real-time agent's method is implemented in *rttoken_agent_do*. The semaphore *sync_sem* is used as the control mechanism. A **semaphore** is a protected variable invented by E.W. Dijkstra and traditionally used to synchronise access to

[34] The function broadcast a critical error message and stops the node from further network transmissions.

shared resources in a multi-processing environment (Tanenbaum, 1992). Real-time tasks are queued on the semaphore in priority order for network access. This message handling is executed by the task itself and with the appropriate priority (real-time messages inherit the priority of the associated sending task).

The following control semaphore functions are available. These, with the exception of *rttsync_sem_wait_if*, are exported and available to hard real-time tasks:

- *rttsync_sem_init* – Initialises the semaphore with a value of 0. The semaphore is therefore not available and all tasks requesting the semaphore will be blocked.
- *rttsync_sem_delete* – Deletes the semaphore. When the semaphore is destroyed, all tasks blocked and queued are allowed to execute. This function should therefore only be used when there is no tasks queued to access the network.
- *rttsync_sem_signal* – Increments and tests the value of the semaphore. If the semaphore is not positive (real-time tasks blocked), the first task in the semaphore waiting-queue is allowed to continue execution. This function does not block the caller task. Note that real-time tasks are queued based on their priority. This is accomplished by including the compile time option *SEM_PRIOD* to the RTToken software code.
- *rttsync_sem_wait_timed* – Decrements and tests the value of the semaphore. If the semaphore is not negative, the semaphore is available and the calling task is allowed to continue execution. Otherwise, the task is blocked and queued on the semaphore, but with a predefined maximum semaphore blocking duration. The calling task is allowed to continue execution when it is the first in the semaphore waiting-queue and the *rttsync_sem_signal* function is called by another task. It can also continue execution when the blocking time exceeds the maximum duration. When this happens, the semaphore value is incremented (one less task waiting on the semaphore) and the task is removed from the semaphore queue. Thus with timeout the semaphore value is unchanged.
- *rttsync_sem_wait_if* – Is similar to the previous function, but the calling task is never blocked. If the tested semaphore value is not negative, the function decrements the value. Otherwise the semaphore value remains the same.

The real-time agent is initialised with the *rttoken_agent_init* function at network start-up. The *rttoken_agent_cleanup* function is used to remove the agent. The *rttoken_agent_signal* function is used to call the agent when a valid token is received.

When the real-time agent gets signalled, it sends a token received acknowledgement to the node that sent the token. The real-time agent then does a routing table lookup to determine the allowed token holding time. A flag (*window_flag*) is set to indicate that the node has transmission privileges. The agent signals the access control semaphore with a call to the above-mentioned *rttsync_sem_signal* function before going into sleep mode for the duration of the allocated token holding time.

After the token holding time has expired, the agent resumes. First the *window_flag* is reset. No further network transmission is allowed. The *rttsync_sem_wait_if* function is called to ensure the semaphore is unavailable. The agent then looks up the next token recipient using the local routing table, sends the token and starts the associated acknowledgement timer.

To conclude the software implementation of the RTToken protocol, the use of the control semaphore in real-time tasks is briefly discussed. Example code is illustrated in Figure 24.

```
static void real_time_task_do (int t) {
    int ret;
    int ret_sem = 0;
    while(1) {
        /* Task main functionality here */
        ret_sem = rttsync_sem_wait_timed(timeout));
        if (ret_sem < 0) {
            /* Timeout : Error handling code here */
        } else {
            /* Semaphore available: Send message */
            ret = rt_sendto(sock, msg, len, 0, target_addr, sizeof(target_addr));
            rttsync_sem_signal();
    }
    rt_task_wait_period();
}
```

Figure 24. Pseudo Implementation of a Real-time Task with the Network Control Semaphore.

Real-time tasks are designed to perform their main functionality before requesting to send a message over the network. These tasks must then wait on the semaphore with the *rttsync_sem_wait_timed* function call. When the semaphore is available, the task with the highest priority gets access to the network and the semaphore becomes unavailable to others waiting in the queue. After the real-time task transmitted its message, it makes the semaphore available, provided that the node is still allowed to have access to the network.

If an error condition occurs, these real-time tasks cannot wait unconditionally for network access. Periodic real-time tasks can for example not wait longer than their period to transmit messages. The *rttsync_sem_wait_timed* function allows real-time tasks, under erroneous conditions, to remove themselves from the semaphore waiting-queue and to perform appropriated error handling procedures.

7 EXPERIMENTS AND RESULTS

This chapter discusses the experiments and measurements to analyse and evaluate the real-time performance of the proposed distributed real-time system, implemented with RTAI operating system and the RTToken communication protocol.

7.1 Evaluation Objectives

Practical experiments were performed on a prototype implementation of the proposed distributed real-time system to measure and evaluate:

- The hard real-time performance of the RTAI operating system.
- The transmission and real-time characteristics of Ethernet packets transmitted over a collision free network with the RTnet networking subsystem.
- The real-time performance and overheads of the RTToken protocol. Note that these measurements also determine the protocol efficiency i.e. the availability of network bandwidth for real-time message transmission.
- The network access latency of real-time messages as a result of the RTToken protocol.

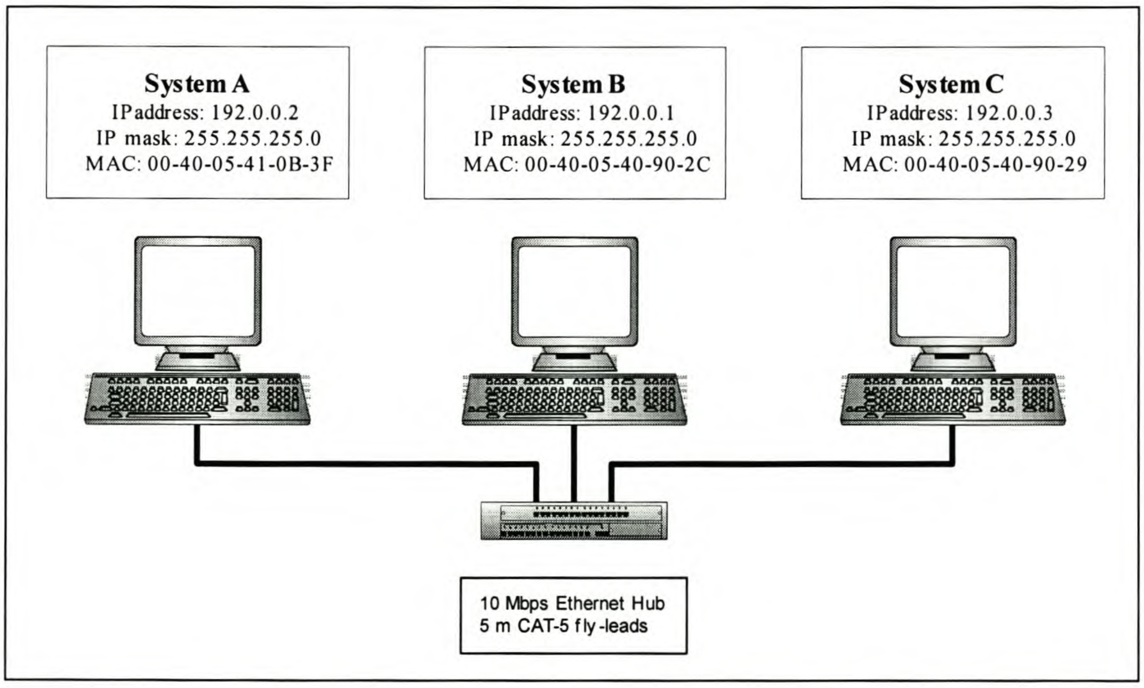

Figure 25. Experimental Prototype of the Proposed Distributed Hard Real-Time System.

7.2 Test Environment

All tests were conducted on the experimental distributed hard real-time system shown in Figure 25. The test platform consisted of three off-the-shelf desktop PC's interconnected with an isolated, single segment 10BaseT (10 Mbps, baseband transmission over twisted pair) Ethernet network, which was hub-based. The network consequently had a star topology. Note that functionally the transmission method is still the same as for a shared bus topology.

The hardware specifications for the computer nodes were:

- **System A** – 333 MHz Intel Pentium Celeron processor, ASUS motherboard (Intel 440BX two-chip set consisting of the 82443BX Host Bridge controller and 82371EB PIIX4E I/O Bridge controller)[35], 64 Mbytes of RAM and a Peripheral Component Interconnect (PCI) bus, 10/100 Mbps Fast Ethernet network card (DEC 21140 chipset).
- **System B** – 333 MHz Intel Pentium Celeron processor, ASUS motherboard (Intel 440BX two-chip set consisting of the 82443BX Host Bridge and 82371EB PIIX4E I/O Bridge), 64 Mbytes of RAM and a PCI bus, 10/100 Mbps Fast Ethernet network card (DEC 21140 chipset).
- **System C** – 466 MHz Intel Pentium Celeron Processor, Gigabyte motherboard (VIA Apollo Pro133A two-chip set consisting of the VT82C694X North Bridge controller and VT82C686A South Bridge controller), 128 Mbytes of RAM and a PCI bus, 10/100 Mbps Fast Ethernet network card (DEC 21140 chipset).

The following software was installed on all the computer nodes of the test platform:

- **REDHAT Linux release 6.2** – The software package or Linux distribution used. It included all the required configuration and development utilities, applications and documentation.
- **Linux kernel 2.2.16** – This stable, production-quality kernel (indicated by the even-numbered minor version number (x.2.x)) was used. The kernel was recompiled to pare down unnecessary default support for devices and services.
- **RTAI 1.6** – The chosen version of the open source, hard real-time Linux implementation.

[35] The Host Bridge (North Bridge) controller manages all data transfers between the CPU, main memory and the Accelerated Graphics Port (AGP). The I/O Bridge (South Bridge) controller manages the I/O-ports (serial, parallel), Universal Serial Bus (USB), keyboard controller, floppy controller and Integrated Device Electronics (IDE) controllers. For the given architecture, both Bridge controllers were interconnected with the PCI bus.

- **RTnet 0.9.1** – An experimental modification of the Linux networking subsystem. It provided a socket based networking capability to hard real-time tasks executing in kernel mode. The networking subsystem supported the Internet Protocol (IP), Internet Control Message Protocol (ICMP) and User Datagram Protocol (UDP).
- **RTToken 0.1**– The newly designed hard real-time token passing protocol.

The network deployed an active, 10 Mbps Ethernet hub with 8 connecting ports (manufacturer: OvisLink). Category 5 patch cabling was used with RJ-45 connectors. Two 5-metre fly-leads for Systems A and B, and a 10-metre fly-lead for System C were employed to from a 15-metre network collision domain (the maximum distance between any two nodes).

7.3 Performance Monitoring

To characterise the temporal behaviour of distributed real-time systems requires a monitoring system with key measurement attributes such as high accuracy, precision and timing resolution. *Accuracy* is a measure of reliability i.e. the degree of conformity of a measured value to its actual or specified value. *Precision* is a measure of repeatability i.e. the degree of mutual agreement between individual measurements of a set of measurements. The *resolution* is the smallest time interval measurable by the monitoring system.

Various processor architectures support transparent, performance monitoring features that are embedded in the hardware (Berrendorf and Ziegler, 1998)[36]. These features were also supported on the Pentium processors used in the test system. The following sections therefore discuss the performance monitoring features that were available and the approach employed to perform run-time measurements on the distributed real-time system.

7.3.1 Intel Pentium Processor Features

The Intel Pentium Celeron processors used in the test environment, support both a time-stamp counter (TSC) and two event counters. The TSC is a free running, 64-bit counter that increments every processor clock cycle and resets to zero when the processor is restarted. The event counters are programmable 40-bit counters. These can count the occurrence or duration of various processor specific events (Intel Corporation. 1997).

[36] Processor architectures include the DEC Alpha, MIPS, PowerPC, SPARC and Intel Pentium processor families.

The performance-monitoring features of the Pentium Celeron processors are associated with Model Specific Registers (MSR) as shown in Table 8. To access a feature, the relevant MSR value is loaded into the processor's ECX register followed by executing either the kernel-mode RDMSR instruction to read or WRMSR instruction to write[37]. A special user mode (non-privileged) instruction (RDTSC) is also available to read the TSC (Intel Corporation. 1997).

Table 8. Model Specific Registers supported by the Intel Pentium Celeron Processor.

MSR Value	Name	Description
10H	Time Stamp Counter	Dedicated, free running, 64-bit counter that increments every clock cycle of the processor.
11H	Control and Event Selection	Select an event (from a pre-determined list) which each Event Counter will monitor and control the operation of the event counters (monitor event occurrences or duration).
12H	Event Counter 1	Programmable 40-bit counter that counts occurrences or duration of events.
13H	Event Counter 2	Programmable 40-bit counter that counts occurrences or duration of events.

Note that besides the TSC, the other performance-monitoring features are not the same for all Pentium processors (even different models of the same processor) and are therefore not considered part of the standard processor architecture. Usage restrictions should therefore also apply to ensure software portability.

For the purpose of the study, **only the TSC was used to measure the real-time performance of the proposed distributed hard real-time system**[38]. Very precise time measurements can be achieved with limited chance of timer overflow. Theoretically a 3-nanosecond timing resolution can be achieved on a 333 MHz processor. Also, the TSC on a 466 MHz processor will take over 1000 years to overflow. The measurement precision is effected by interrupts, data cache misses when storing the beginning time stamp and processors that support out-of-order instruction execution. To mitigate these problems, disable all interrupts before reading the TSC, pre-load the cache or use registers, and use serialisation instructions (e.g. CPUID) to force the correct instruction execution sequence.

To convert the measured clock cycles on the TSC to time the clock frequency of the processors in the system has to be accurately measured. RTAI 1.6 provided a calibration utility for this

[37] A feature bit in the EDX register (reported by the CPUID instruction) indicates whether the processor supports these instructions.

[38] Entities are observed with respect to timing behaviour (not content or scheduling order).

purpose (../utils/cpu_freq_calibration/). Measurements converged after running the calibration program for approximately one hour from cold system start-up. The results are summarised in Table 9.

Table 9. Calibrated CPU Frequencies of the Test Platform.

Test platform nodes	Calibrated CPU frequency (Hertz)
System A	334090400
System B	334090400
System C	467726560

7.3.2 Time Measurement Accuracy, Precision and Resolution

Time measurements are accomplished by multiplying the clock cycles measured by the TSC with the period of the onboard system clock. For the test platform, the clock period for the different nodes is equal the inverse of the CPU frequencies as indicated in Table 9. The *accuracy* refers to how these time measurements conform to actual (atomic) time. The conformance depends on the age, quality (stability) and performance of the system clock as well as levels of noise interference (both thermal and mechanical) and voltage variations (Cypress, 1997). *For the purpose of the study, all system clocks were considered ideal with accurate clock frequencies equivalent to the tabled CPU frequencies. To compensate for variations in clock frequencies as a result of ambient temperature variations, all tests were performed after a two-hour warm-up period.*

Experiments were performed to measure the precision and resolution of the TSC on all nodes in the experimental distributed real-time system. In this context, *resolution* refers to the smallest measurable interval (number of clock cycles) between two consecutive TSC reads. For each experiment 10 000 measurements were made. All systems measured 32 clock cycles. The TSC also exhibited high *precision* with repeated consistency for all experiments. Using Table 9, the time resolution of all nodes can be calculated. The results are summarised in Table 10.

Table 10. Measured Time resolution of the Nodes in the Test Platform.

Test platform nodes	Time Resolution (ns)
System A	95.8
System B	95.8
System C	68.4

It can be concluded that accurate, precise and high-resolution measurements can be achieved with the TSC, provided the operation of underlying system clock is accurate. ***Based on these results, a maximum measurement resolution of 100 ns can be achieved to measure the real-time performance of the proposed distributed real-time systems.***

7.3.3 Software Monitoring Approach

System monitoring can be accomplished with hardware, software or a combination of both. The advantages of *software monitoring* are flexibility (simply insert monitoring code in targeted system) and the fact that no additional monitoring hardware is required. The disadvantage is that the resources of the targeted system are used; thereby possibly affecting the system's real-time performance and behaviour, i.e. change the order/sequence and timing/occurrence of events. For example, in distributed real-time systems the additional monitoring overhead on one processor can cause it to miss deadlines and/or alter the sequence of events with respect to other processors. On the other hand, *hardware monitoring* has the advantage that the monitoring system does not interfere with the targeted system. However, additional monitoring hardware is then required. The hardware is also target specific and therefore costly, and because only external interfaces to the system can be monitored, the observed level of detail is low (Tsai et. al., 1996).

```
...
/* Timestamp */
tstamp.time = rt_rdtsc();

/* Event data */
tstamp.task_id = 0;
tstamp.event = TASK_START;

/* Output data on fifo */
rtf_put(fifo0, &tstamp, sizeof(tstamp));
...
```

Figure 26. An Example of the Inserted Monitoring Software Code.

A combinational **hardware/software monitoring approach** was used in the study, whereby monitoring code was simply inserted in the targeted system's software. The functionality (trigger, record and store) of the monitoring software was separated to reduce the inserted code and thereby also minimising disturbances (perturbations) on the tested systems[39]. An example of

[39] If the monitoring software influences the real-time performance of a system, it should be permanently included in the system to be able to predict and reduce the monitoring interference by considering the monitoring overhead.

the monitoring code that was used is shown in Figure 26. The inserted monitoring code collected (read the independent hardware TSC) and transferred the run-time event information to a non-real-time application. The application then recorded the information to disk for later off-line analysis. A unidirectional buffer (FIFO) was used to asynchronously transfer data between the executing monitoring code and the recording application. The size of the buffer is important and chosen sufficiently large to avoid any buffer overflows. This approach also utilised spare non-real-time resources to reduce the effect of monitoring overheads on the real-time performance. *Note that where applicable, the inserted code should still be designed as a permanent part of the tested systems. The associated overheads must still by taken into account in order to predict and reduce monitoring interference.*

7.4 RTAI Operating System

Several experiments were conducted to measure the real-time performance of the RTAI operating system. The focus was primarily on the operating system functionality required for distributed real-time communication. Therefore, experiments included measuring the interrupt service routine latency, interrupt task latency and synchronisation/signalling methods (semaphores and suspend/resume calls). Measurements were done under different real-time and non-real-time load conditions to realistically determine the real-time characteristics of the operating system. In addition to the experiments discussed in this section, the scheduling precision of periodic real-time tasks was also measured[40]. These results were included in Appendix B, since scheduling was not formally covered in the study.

7.4.1 Load Conditions

The load conditions under which the experiments were performed are:

- **Idle systems** – Only the non-real-time background processes (of the reconfigured and scaled down Linux kernel) executed. Note that for some experiments the software monitor was also implemented on the tested system. This added additional non-real-time load to the system.
- **Real-time task load** – An artificial real-time load was generated comprising of two periodic real-time tasks with different periods (1 ms for high- and 5 ms for low-priority tasks) and priorities (1 for high- and 2 for low-priority tasks, with 0 being the highest priority allocation). Both real-time tasks performed a for-loop that executed without yielding the

[40] Scheduling co-ordinates time events such as ***when*** tasks communicate. Although scheduling is not a ***requirement*** for real-time communication it still important, especially when the receiver is not always ready to receive.

processor, i.e. giving back the control to the scheduler. The loop durations were empirically calculated to be 0.4 ms for the high- and 2 ms for the low priority task. The high priority task also frequently pre-empted the lower priority task (refer to Figure 27). Thus, for every 5 ms time interval these real-time tasks held the processor for approximately 4 ms, resulting in an approximate 80% real-time task load. Support for floating point operations was also included, adding to the context switch load.

- **Hard-disk load** – During measurements, a non-real-time hard-disk load was created on the tested systems by repetitively executing the command to list file information recursively and in long listing format (*ls –lR /*).
- **Network load** – Non-real-time network load was created by repetitively executing the flood ping command (*ping –f "IP address of tested system"*) on any system in the experimental distributed real-time system not used for testing. Ping packets were targeted at the tested system and contained 84 bytes (20 bytes for the IP header without options, 8 bytes for the Internet Control Message Protocol (ICMP) header, and 56 bytes of arbitrary data). These packets were retransmitted at the largest time interval of 10 ms or the ping return time interval.

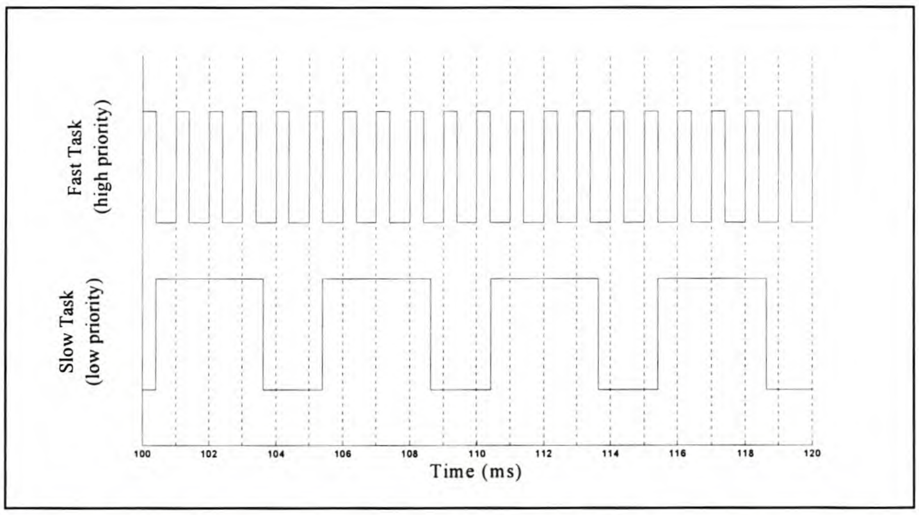

Figure 27. Real-Time Tasks used to Generate 80% Real-Time Task Load.

7.4.2 Interrupt Service Routine Latency

The **interrupt latency** or **interrupt response time** is a common performance measurement used for real-time systems. It is defined as the inherent delay between the instant that a hardware interrupt request is made and when the interrupt service routine executes (Laplante, 1997). Because an interrupt is an *asynchronous* event, the latency is dependent on the state of the system when the interrupt occurred. Both hardware and software related factors could influence the delay. This could be both deliberately (such as saving task context, determining the interrupt source and invoking the interrupt handler) and incidentally (i.e. disabled interrupts, instruction completion and prior execution of nested interrupt handlers). As a result, *interrupt latencies are variable and a measurement thereof only gives limited indication of the real-time performance and predictability*. Note that an estimate of the interrupt latency can serve as an indication of the system's responsiveness to high priority and emergency interrupts. It can also assist in the predictable characterisation of system phase lag during the evaluation of real-time control algorithms (INtime, 1998).

Experiments were conducted to measure the real-time interrupt latency of the different computer systems in the test platform. It was considered to generate interrupts with the Analogue-to-Digital, Digital-to-Analogue (ADA) digital control card, developed by SED, University of Stellenbosch (SED, 1992). However, measuring the time at which the card generated interrupts couldn't easily be determined (observed) without specialised hardware such as a store oscilloscope. This method was also tedious and very time consuming when compiling large sample sets. Instead it was decided to generate interrupts using the parallel port of a computer system. Hereby the TSC on the interrupt generating system can conveniently measure (multiple) interrupt request times. For the experiments, one system was set up to periodically send interrupt requests to the tested system and then measured the response times. The experimental set-up was similar to that used by Barabanov (1997) and is shown in Figure 28.

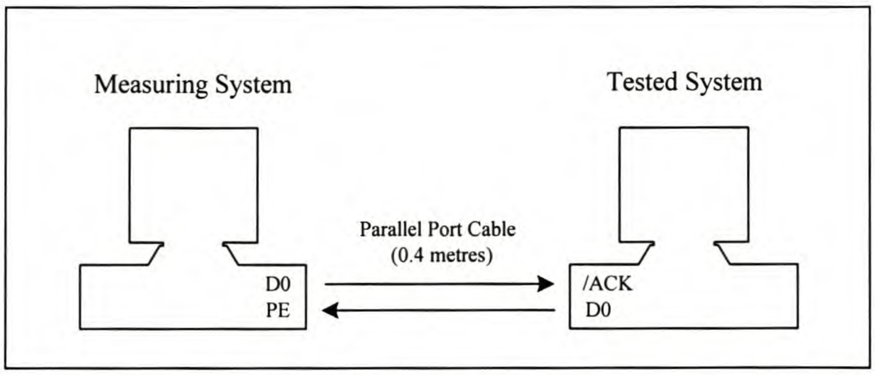

Figure 28. Experimental Set-up for Interrupt Latency Measurements.

The data output (D0) and Paper Empty (PE) input of the measuring system's parallel port were respectively connected to the acknowledge input (/ACK) and data output (D0) of the tested system's parallel port. A real-time interrupt was requested from the tested system with a positive going transition on the acknowledge input. The tested system then serviced the interrupt and responded by toggling the value on its data output (D0).

The interrupt latency was measured as the time duration between the interrupt request and the change on the PE input. As a result, measurements also included the throughput delay of the parallel port and PC's hardware. The propagation delay of the short parallel cable (0.4 metres) was considered to be negligible. Measurements were taken during steady-state operation, i.e. not from start-up. Interrupt requests were generated at 100 microseconds (μs) and 10 ms periodic time intervals. Systems A and B were used to perform the measurements. These systems had the same hardware and software configuration, allowing for consistency in the testing method of all three systems. Results were calculated based on 100 000[41] measurements per experiment for the 100 μs interrupt rate and 10 000 measurements per experiment for the 10 ms interrupt rate. These results are summarised in Table 11. The table dictates the interrupt latencies (minimum, average and maximum) and jitter. The measured interrupt latencies for System A are also shown in Figure 29 and Figure 30. Similar results were also obtained for Systems B and C and are therefore not shown.

Table 11. Measured Interrupt Latency Results for the Distributed System under Different Load Conditions.

Tested System	Testing System	Minimum Latency (μs)		Average (mean) Latency (μs)		Maximum Latency (μs)		Jitter (μs)	
Interrupt Time Interval (ms)		*0.1*	*10*	*0.1*	*10*	*0.1*	*10*	*0.1*	*10*
System A with minimum load	**System B**	**3**	**3**	**3**	**3**	**7**	**4**	**4**	**1**
System A with real-time task load	**System B**	**3**	**3**	**3**	**3**	**5**	**4**	**2**	**1**
System A with hard-disk load	System B	3	3	3	3	7	4	4	1
System A with network load	System B	3	3	3	3	**9**	4	6	1
System B with minimum load	**System A**	**3**	**3**	**3**	**3**	**7**	**4**	**4**	**1**
System B with real-time task load	**System A**	**3**	**3**	**3**	**3**	**7**	**4**	**4**	**1**
System B with hard-disk load	System A	3	3	3	3	**9**	4	6	1
System B with network load	System A	3	3	3	3	7	4	4	1
System C with minimum load	**System B**	**3**	**3**	**3**	**3**	**7**	**4**	**4**	**1**
System C with real-time task load	**System B**	**3**	**3**	**3**	**3**	**7**	**4**	**4**	**1**
System C with hard-disk load	System B	3	3	3	3	7	4	4	1
System C with network load	System B	3	3	3	3	**8**	4	5	1

[41] A very large sample set was chosen to reflect the infrequent and random occurrences of single large interrupt latencies that were observed under all load conditions.

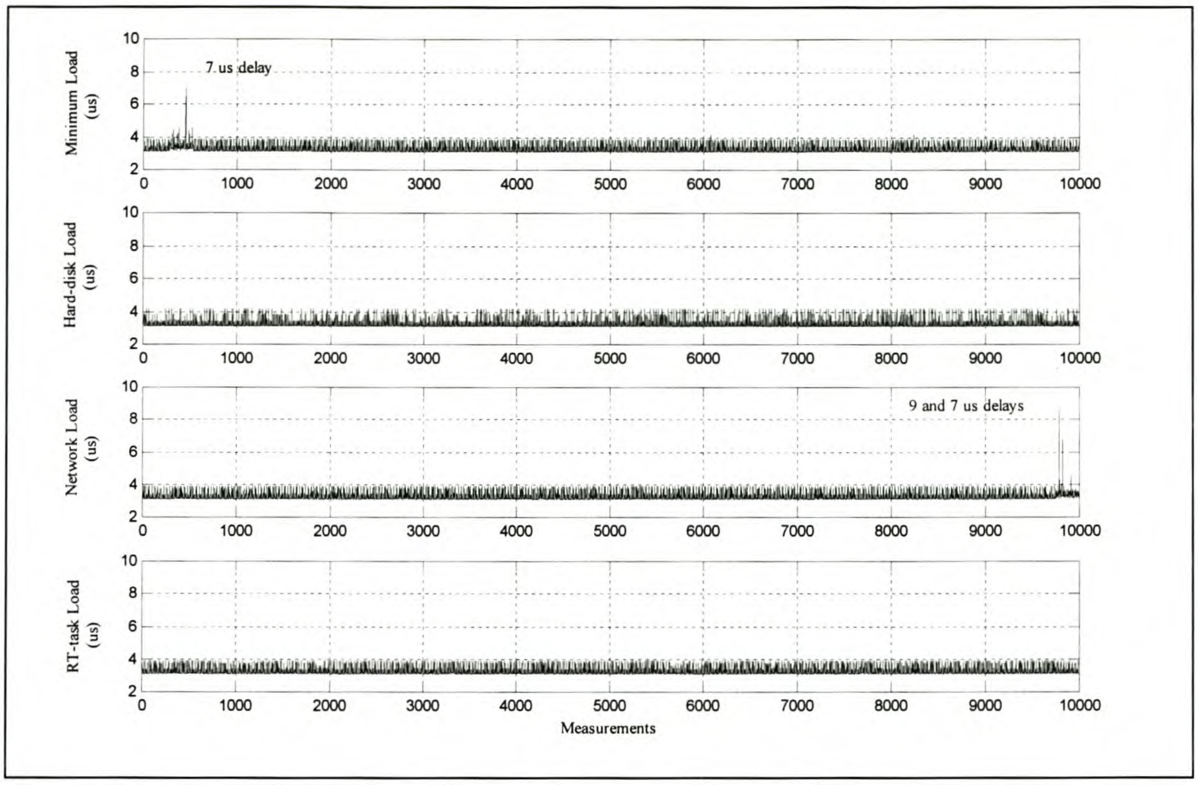

Figure 29. Measured Interrupt Latency for System A under Different Load Conditions with 100 μs Interrupt Time Intervals.

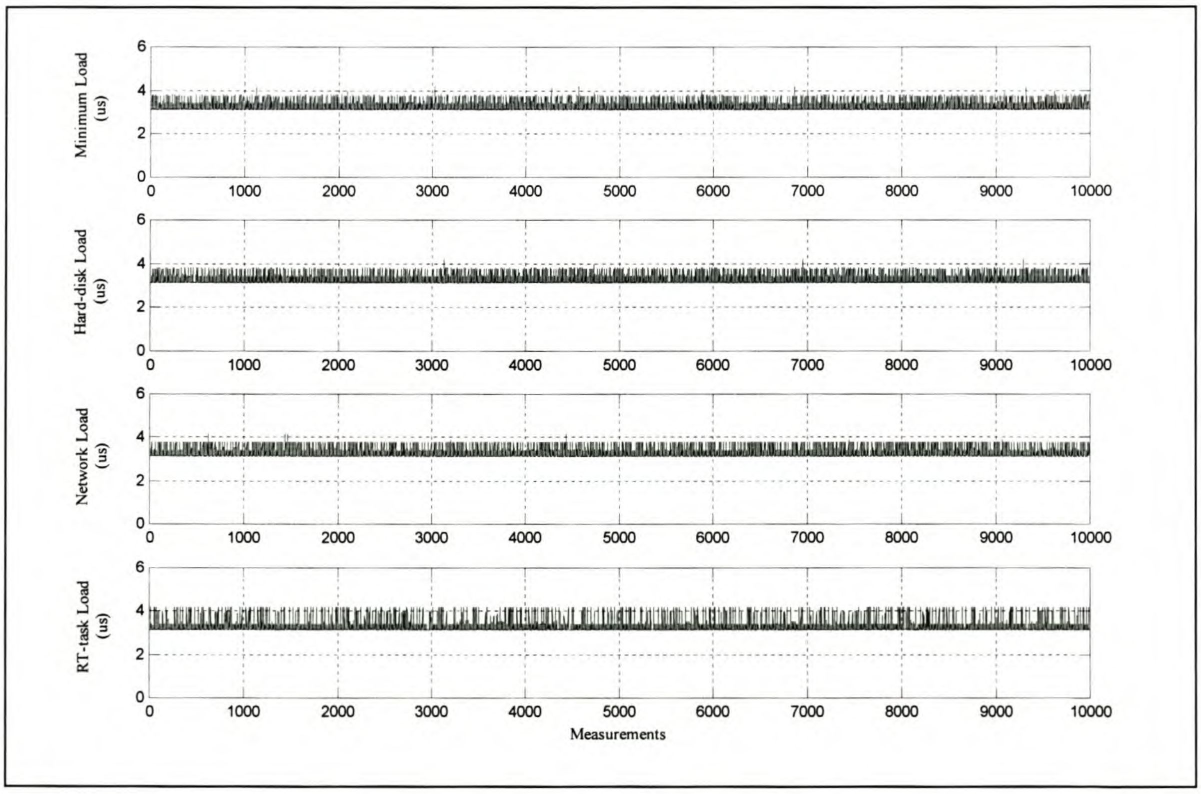

Figure 30. Measured Interrupt Latency for System A under Different Load Conditions with 10 ms Interrupt Time Intervals.

Using the parallel port, a measurement resolution not better than microseconds can be achieved. All measurements include the data throughput delay of the parallel port. This comprises of a minimum data set-up time and strobe pulse width of 1 μs (IEEE Std 1284, 2000)[42].

The **minimum interrupt latency** is primarily determined by the best performance of the underlying hardware architecture of the tested system (INtime, 1998). A minimum interrupt latency of 3 μs was measured for all systems under different loads and interrupt intervals.

Consequently, all three systems proved to offer similar hardware performance in terms of responding to interrupts. It was also observed that the faster CPU clock frequency of System C offered no significant performance improvement on the minimum interrupt latency. Thus, the interrupt response time cannot be improved by simply increasing the clock frequency of the CPU.

The **average interrupt latency** is a statistical indication of the state of the system's hardware and software when the interrupt occurred. Experiments typically measured a variation of short and longer delays. The average measured interrupt latency depends on:

- The performance of the underlying hardware architecture (that also determined the minimum interrupt latency).
- The frequency and duration of non-interruptible software or hardware events.
- The context switch time to the real-time environment if the interrupt occurred when the Linux operating system executed.

According to the results, the average interrupt latency was close to the minimum measured under all load conditions and with different interrupt intervals. It can be concluded that long interrupt delays occurred infrequently. This tendency is also seen in Figure 30. *The average interrupt latency can be used in firm real-time systems to evaluate the effect of interrupt latency on the phase lag of real-time control algorithms. However, infrequent long delays pose a problem for systems with hard real-time requirements.*

[42] Data gets latched on the rising edge of the /strobe signal after the data hold time (minimum 0.5 μs) and strobe pulse width (minimum 0.5 μs; maximum 500 μs).

The **maximum interrupt latency** was determined by either the overhead required to perform an asynchronous context switch to process the real-time interrupt, or a forced delay as a result of the present hardware and/or software state of the system. Context switch times are usually fixed in time and relatively short. Forced hardware and software delays on the other hand, have variable duration that can be long. These delays predominantly determine the worst-case interrupt latency of the system (INtime, 1998).

Commercial off-the-shelf PC hardware, the Linux operating system, device drivers and applications were not designed for real-time determinism. As a result, some of the hardware elements that can cause forced delays are (INtime, 1998a):

- **Legacy Direct Memory Access (DMA) operations** – These operations (block and demand) are non-pre-emptable and block all transfers until the transaction is completed. The use of software and device drivers that use these operations should be avoided.
- **Atomic bus cycles** - Locked bus cycles are a problem if they are used in conjunction with a peripheral that responds very slowly to a data transfer, especially for back-to-back cycles.
- **Memory error detection and correction** – Some chipsets generate a non-maskable System Management Interrupt (SMI) when a memory error has occurred. A correction algorithm is then executed to correct the error (typically provided as part of the system Basic Input/Output System (BIOS)). These undetermined interrupts can be disabled in the BIOS set-up.

Some software elements that can cause forced delays are (INtime, 1998a):

- **Disabled and masked interrupts** – This is performed by some misbehaved device drivers and the kernel in small sections of code for atomic access to registers or data structures. The use of such software should be avoided.
- **Improper use of the End of Interrupt (EOI) command** – The EOI command cannot be sent until the interrupt service routine is finished. Because some hardware cannot accommodate simultaneous access, the late issuance of EOI will delay additional interrupts being serviced. Therefore interrupt service routines should be very short to minimise delays.
- **Advanced Power Management (APM) features** – SMI interrupts are transparent, non-maskable and cannot be interrupted by normal interrupts. These features should be disabled, usually from the BIOS set-up.

The maximum interrupt latency as a result of context switches was measured under a light interrupt load (10 ms interrupt intervals). The different load conditions and the difference in CPU clock frequencies did not significantly affect the measured interrupt latencies. The effect of forced hardware/software delays was measured under heavy interrupt loads (100 μs interrupt intervals). The maximum interrupt latencies for all three systems are highlighted in Table 11. A maximum value of 9 μs was measured. Figure 29 shows the irregular occurrence of such forced delays, measured under all load conditions. Understanding and controlling the conditions that cause these delays makes it possible to optimise the system for increased real-time determinism. The maximum interrupt latency is also an important parameter when designing interrupt driven hard real-time systems. Real-time systems require bounded interrupt latency and therefore the maximum interrupt response time must be guaranteed. *For each of the tested systems, the measured maximum interrupt latencies can be taken as acceptable estimates of the worst-case interrupt response times.* Note that in general, low maximum interrupt latencies are preferred, since it increases system throughput and responsiveness. However, this should not be at the cost of predictability.

The variation in the measured interrupt latencies (**jitter**) is also indicative of the deterministic behaviour of the tested real-time systems. Jitter was measured as the difference between the minimum and maximum measured interrupt latencies. Minimal jitter of 1 μs was measured under all load conditions for 10 ms interrupt intervals. The measured jitter however, increased for 100 μs interrupt intervals. A maximum jitter of 4 μs was measured under real-time load conditions. The addition of non-real-time load made the jitter even more inconsistent, resulting in a maximum jitter of 9 μs. *Thus measurements proved that the real-time performance of the RTAI operating system deteriorated with short interrupt intervals and the addition of non-real-time load.*

7.4.3 Interrupt Task Latency

Real-time operating systems can also include an interrupt delay component when a real-time task is started by an interrupt service routine. The **interrupt task latency** is defined as the delay between the instant that the interrupt service routine signals the kernel to activate an associated real-time task (interrupt task) and when the task begins to execute (INtime, 1998). The execution of the interrupt task is a *synchronous* event i.e. it is only executed if it has the highest priority of all real-time tasks ready for execution. Therefore, it is not an additional asynchronous interrupt latency component. The relationship between the interrupt latency and interrupt task latency is shown in Figure 31.

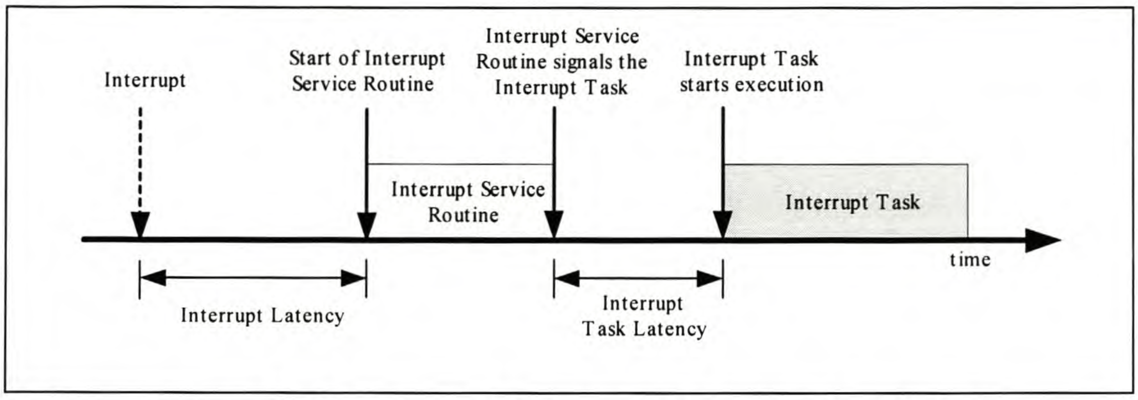
Interrupt
Start of Interrupt Service Routine
Interrupt Service Routine signals the Interrupt Task
Interrupt Task starts execution
Interrupt Service Routine
Interrupt Task
time
Interrupt Latency
Interrupt Task Latency

Figure 31. The Interrupt Latency and Interrupt Task Latency.

Associating real-time tasks with interrupt services routines is usually done out of convenience. It is not a requirement to handle interrupts. The real-time agent of the RTToken protocol is an example of a high priority real-time task signalled by an interrupt service routine.

Experiments were conducted to measure the interrupt task latency and to illustrate the dependency of the interrupt task latency on the combined real-time load of the targeted system and the priority of the real-time task signalled by the real-time interrupt service routine. To measure the latter, the target system (System A) was set up with one interrupt task and two periodic real-time tasks. The periodic real-time tasks had the same properties as those previously specified in Section 7.4.1 for the real-time load condition. Experiments were conducted with the interrupt task having the lowest and then the highest real-time task priority.

The targeted system's parallel port was configured to receive interrupt requests from another system, which generated these with 5 ms time intervals. The interrupt service routine was implemented to signal the interrupt task using a semaphore. The interrupt task was put to sleep (semaphore value incremented) as if waiting for a shared resource to be freed. When an interrupt occurred, the interrupt service routine signalled the semaphore (semaphore value decremented) as if releasing the resource and the pending interrupt task could execute.

The software monitor was set up to log specific events using timestamps. These events included the time at which the interrupt service routine signalled the interrupt task and the start and stop times of all real-time tasks. The results of these measurements were then processed and presented as state transition diagrams where:

- Positive transitions indicate the start of an event.
- Negative transition indicates the end of an event.
- The high value between transitions indicates the duration of a specific event.

The measured results for the first experiment, where the interrupt task was assigned the lowest priority, are shown in Figure 32. The figure clearly shows the variation in the interrupt task latency. This is due to the low priority interrupt task being delayed by higher priority tasks. The figure also shows how the high priority fast task pre-empts both the interrupt task and the slow task. Note that context switch events were not measured. Task pre-emption delays were only noticeable as changes in the executing time of real-time tasks. The measured event durations can be seen in the figure, with the expected real-time task execution times indicated in brackets on the vertical axis of the graph. For example, an execution time of approximately 3.2 ms was measured for the slow task, which had a calculated processing time of 2 ms. During its execution, the fast task also executed three times, adding a total execution time of 1.2 ms. Therefore, the fast task pre-empted and as a result delayed the slow task.

For the second experiment, the interrupt task was configured to have the highest priority. Figure 33 shows the interrupt task delays that were measured. It can be concluded that if the interrupt task has the highest priority of all real-time tasks ready for execution, then the interrupt task latency is determined by the method used to signal the real-time interrupt task.

Experiments were conducted to measure the interrupt task delay when the interrupt-associated task had the highest priority of all the real-time tasks. Two signalling methods were measured, namely suspend/resume function calls and semaphores. The former signalling method temporarily suspends the interrupt task with the *rt_task_suspend* function call. The task is signalled for execution when the interrupt service routine makes the *rt_task_resume* function call. Signalling with semaphores was previously discussed as part of the RTToken protocol design and implementation.

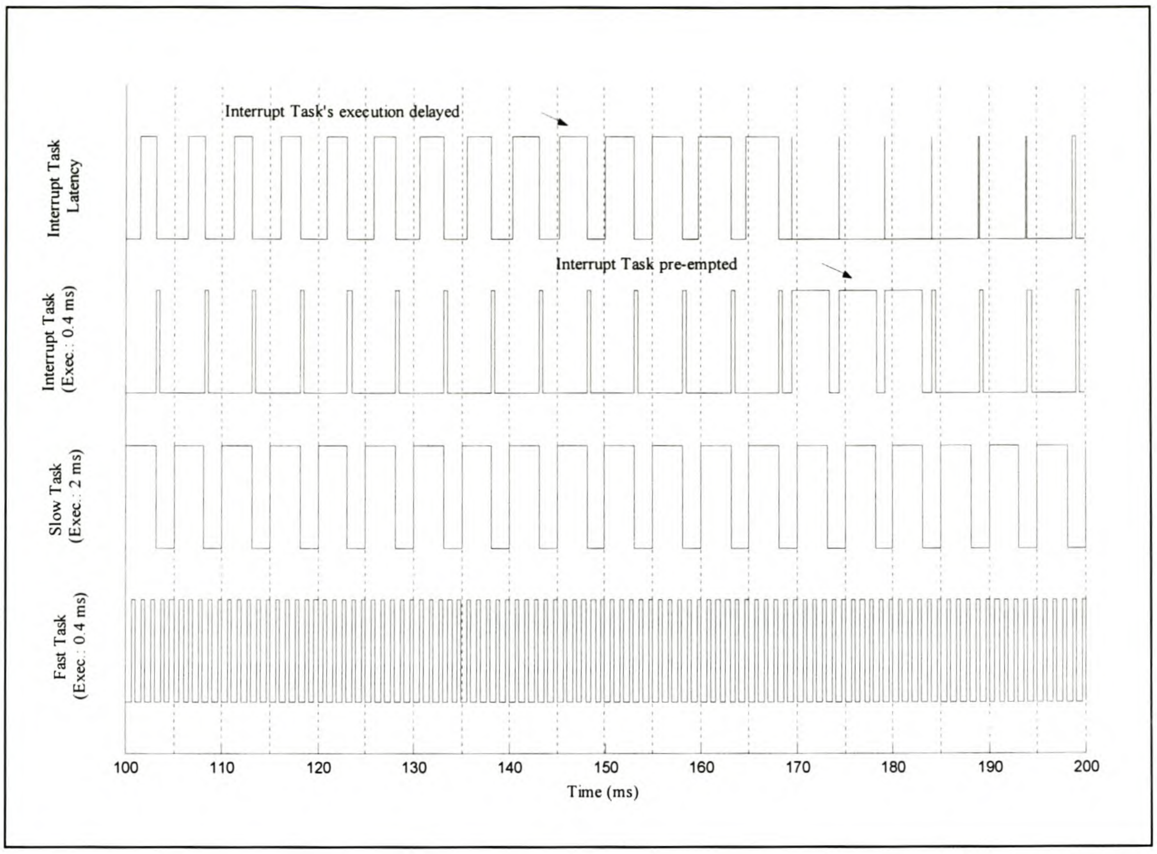

Figure 32. Interrupt Task Latency of Low Priority Interrupt Task on System A.

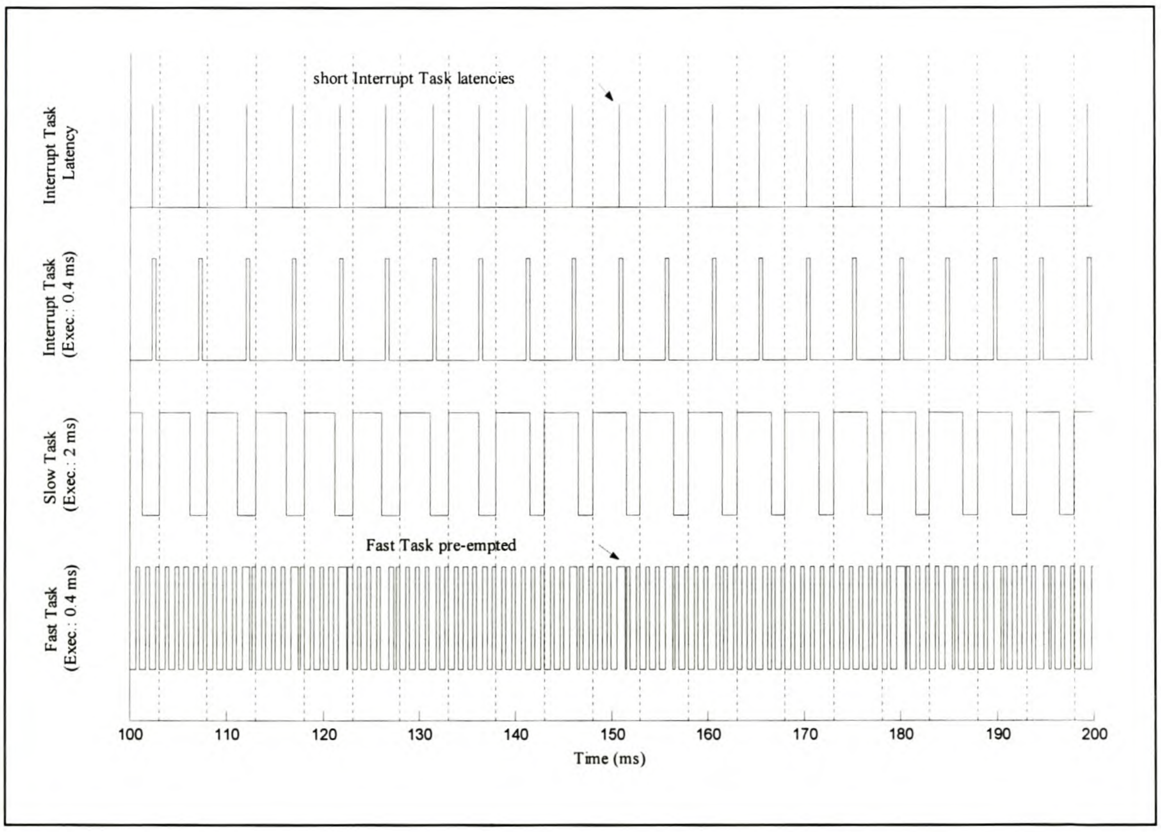

Figure 33. Interrupt Task Latency of Interrupt Task with Highest Priority on System A.

Task interrupt latency measurements were taken under different load conditions. The software monitor was implemented on all tested systems, and thus added additional non-real-time overhead. The non-real-time hard disk and network loads were also added to the tested system that already had a high real-time task load. The measured results are summarised in Table 12.

Table 12. Measured Interrupt Task Latency results for Semaphore and Suspend/Resume Signalling under Different Load Conditions.

Tested System	Minimum Latency (µs)		Average (mean) Latency (µs)		Maximum Latency (µs)		Jitter (µs)	
Signalling	*Sem*	*Sus/Res*	*Sem*	*Sus/Res*	*Sem*	*Sus/Res*	*Sem*	*Sus/Res*
System A with minimum load	**1.4**	**1.2**	**1.7**	**1.4**	**3.1**	**2.8**	**1.6**	**1.6**
System A with real-time task load	**1.2**	**1.2**	**1.5**	**1.4**	**5.2**	**2.7**	**4.0**	**1.5**
System A with real-time task and hard-disk load	1.2	1.3	2.8	5.2	9.3	13.3	8.1	12.0
System A with real-time task and network load	1.3	1.7	2.1	2.0	7.4	5.5	6.2	3.7
System B with minimum load	**1.4**	**1.2**	**1.6**	**1.4**	**3.2**	**2.8**	**1.8**	**1.6**
System B with real-time task load	**1.2**	**1.2**	**1.3**	**1.4**	**5.1**	**5.0**	**3.9**	**3.9**
System B with real-time task and hard-disk load	1.3	1.2	2.7	5.0	12.1	11.4	10.9	10.2
System B with real-time task and network load	1.6	1.7	2.4	2.1	7.3	6.5	5.7	4.8
System C with minimum load	**1.**	**0.9**	**1.3**	**1.0**	**3.2**	**4.1**	**2.2**	**3.2**
System C with real-time task load	**0.8**	**0.9**	**1.0**	**1.0**	**4.2**	**4.0**	**3.4**	**3.1**
System C with real-time task and hard-disk load	0.9	0.9	2.6	5.0	12.9	17.1	12.0	16.2
System C with real-time task and network load	1.0	1.3	1.6	1.8	9.9	10.4	8.9	9.1

Sem = Semaphore; Sus/Res = Suspend/Resume

Similar performances are measured for both the semaphore and suspend/resume signalling methods. The results for both methods showed increased jitter under high real-time task load for all tested systems. Also, the additional non-real-time load significantly degraded the real-time performance of all the systems. Measurements showed significant increases in jitter and much larger maximum interrupt task latencies. ***For real-time applications is mandatory that the non-real-time load be minimised and well characterised (for firm real-time applications) or completely removed (potentially for hard real-time applications).***

Again, the maximum interrupt task delays under predominantly real-time load were measured when irregular events occurred (forced delays as a result of the state of the system). The measurements for System B under minimum (approximately 10%) and high (approximately 90%) real-time load are shown in Figure 34. These forced delays can mainly be attributed to occurrences of priority inversion where non-real-time applications force uninterruptible hardware states optimised for throughput and acceptable average performance. An example is as Direct Memory Access (DMA) controlled data swapping between memory and hard disk. This functionality is performed on behalf of the software monitor to save logging information to hard disk.

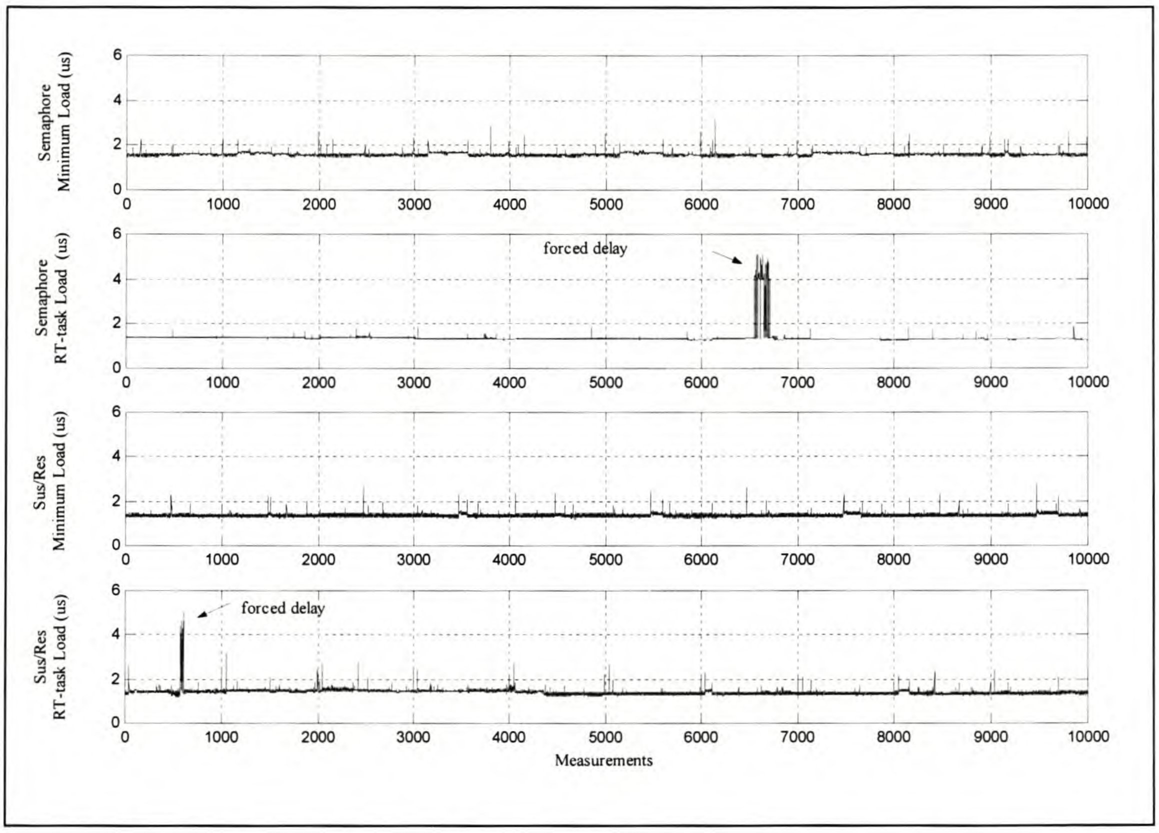

Figure 34. Measured Interrupt Task Latencies for System B under Minimum and Real-Time Task Load.

7.4.4 RTAI Performance Conclusion

The RTAI operating system delivered deterministic real-time performance for the functionality and capabilities that were tested. Overall predictable performance was measured under different real-time load conditions. The addition of non-real-time load however, degraded the real-time performance. The state transition diagrams in Figure 32 and Figure 33 also showed that the operating system fully supported prioritised task pre-emptiness. *The context switch times weren't measured and are a topic for further research.*

Experiments conducted to measure the interrupt latency proved that fast and consistent interrupt response times can be achieved with RTAI. Interrupt response times of 4 μs with 1 μs jitter were measured for 10 ms interrupt intervals. The results were consistent, even with the addition of both real-time and non-real-time loads. With interrupt intervals are shorted (100 μs), measurements showed slower maximum interrupt response times (7 μs) and increased interrupt latency variation (4 μs). Forced hardware and/or software delays were measured, even under minimum real-time load conditions. This was partially caused by the non-real-time software monitor that saved data to disk in run-time. The addition of non-real-time load increased the maximum interrupt latencies to 9 μs with 6 μs jitter. *Thus overall non-real-time load degraded the real-time performance of RTAI, especially when used with short (100 μs) real-time interrupt intervals.*

The signalling methods of RTAI also proved to be fast (1.2 μs average), but with large bounded jitter (up to 4 μs). A maximum signalling latency of 5.2 μs for the semaphore and 5 μs for the suspend/resume method were measured under real-time load conditions. The addition of non-real-time load significantly degraded the real-time performance of both signalling methods. This degradation was most prominent for high hard disk loads. Under these conditions, a maximum signalling latency of 12.9 μs for the semaphore and 17.1 μs for the suspend/resume method were measured. Again forced hardware and/or software events occurred, affecting the real-time performance of RTAI.

These preliminary results proved that the RTAI operating system to be best suited for firm real-time applications i.e. where some low probability of missing a deadline can be tolerated. The measured functionality and capability proved to deliver the best real-time performance under moderate to light real-time loads. Non-real-time load degraded the real-time performance and should either be minimised or completely removed for increased system predictability. The impact of commercial off-the-shelf hardware and software (causes of forced hardware/software delays) on the determinism of the real-time system should also be controlled.

7.5 Message Transmission with RTnet

Real-time messages are transmitted between distributed real-time tasks using the UDP/IP protocols of the RTnet networking subsystem. Real-time communication requires deterministic and predictable transmission delays and transmission delay variations (jitter). Transmission delays must be bounded (guaranteed maximum delay) and consistent (minimal jitter). Various experiments were conducted to measure these characteristics for the RTnet protocols.

As mentioned previously in Chapter 2, the communication time is composed of generation-, queuing-, transmission- and delivery delays. However, on the experimental distributed real-time system, the communication delay was measured as the time duration between when a real-time message queued for transmission at the sending node and when the message was received at the destination node. The measured delay did not include generation and delivery delays. These components were not measured, since it is generally up to the application to dictate how real-time messages will be generated and delivered (with associated delays). Since the queuing and transmission delays could not be easily distinguished due to the implementation of the Ethernet interface hardware and device driver, these delays were collectively measured and are referred to as the ***transmission delay***.

7.5.1 Transmission Delay

To measure the transmission delays (latency) of the RTnet protocols (UDP/IP over Ethernet), a node in the experimental distributed real-time system was configured to periodically send real-time messages to a receiving node. When no messages were transmitted, the network was idle. Using the parallel port, the receiving node was configured to temporarily set the parallel port output to high when it received the real-time message. After transmitting the real-time message, the sending node waited in a busy loop for the parallel port to go high. The transmission delays were measured as the time duration between the transmission of the real-time message and the change in parallel port state. These measurements were logged with a software monitor, installed on the sender.

The first experiments were performed to compare the transmission delays of differently sized packets with the theoretical transmission times achievable over a 10 Mbps Ethernet network. The size of real-time messages is confined by the maximum data field size of the Ethernet frame (1472 bytes)[43]. For each of the different Ethernet packet sizes that were transmitted, 2000 measurements were taken. These typical transmission delays that were measured are illustrated in Figure 35.

[43] Maximum Ethernet data field (1500 bytes) – IP header without option field (20 bytes) - UDP header (8 bytes).

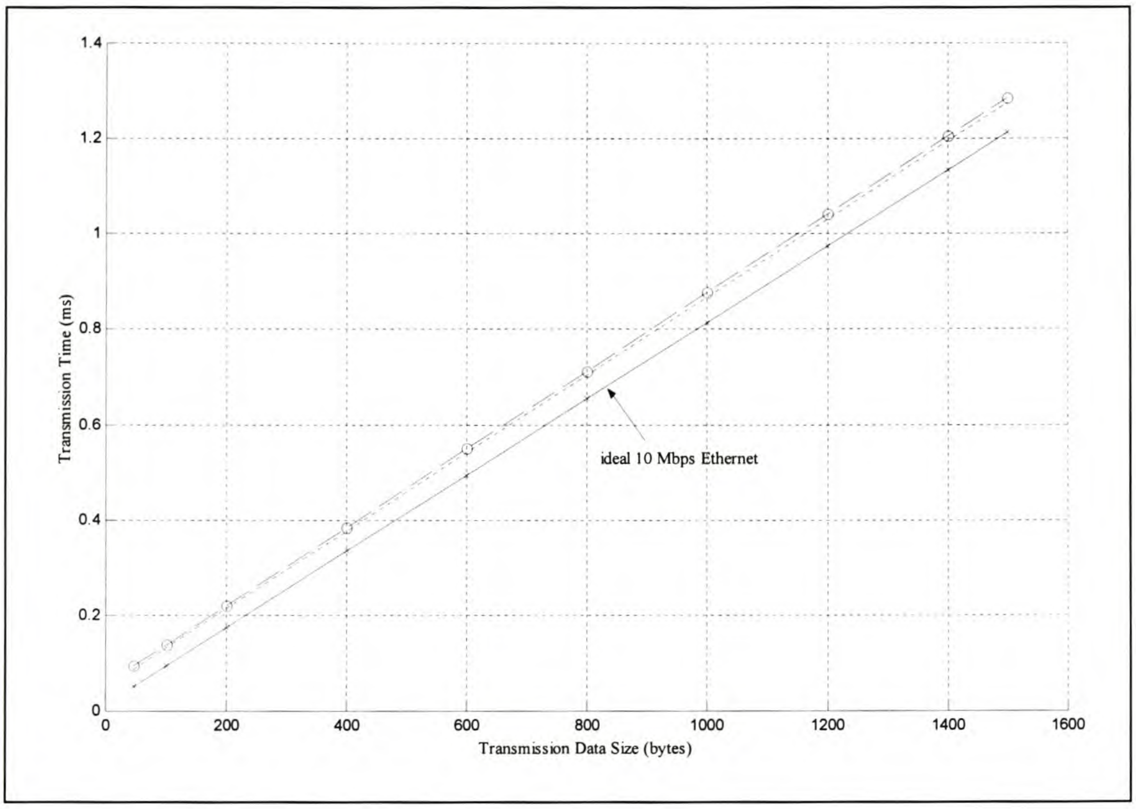

Figure 35. Measured Data Transmission Times with RTnet between Systems A and B.

Hereby real-time messages (encapsulated in Ethernet frames) were periodically transmitted (10 ms intervals) from System A to System B. The measured maximum transmission delays for the various packet sizes are encircled and interconnected with a dashed line. The average transmission delays are shown as a finely dashed line. Because similar transmission delay results are also measured between any of the other nodes, these are not shown in the figure.

Table 13 summarises the difference between the measured and ideal Ethernet transmission delays for different Ethernet packet sizes. The results indicated that the transmission delays steadily increased with larger Ethernet packets. Thus, the queuing and transmission delays of larger Ethernet packets by the RTnet protocols take longer.

Experiments with larger sample sets were performed to obtain more accurate results of the transmission delay bounds. The transmission delays were measured for the smallest and largest Ethernet packet sizes. For each of the experiments a set of 10 000 measurements were made (the first 1000 was discarded to ignore any start-up effects). Results are summarised in Table 14 and categorised according to the minimum, average and maximum transmission delays and transmission jitter. The effect of *data padding* was also measured. The minimum allowed

Ethernet data field was configured to only contain 20 bytes for the IP protocol header, 8 bytes for the UDP header and 18 bytes of padding.

Table 13. Transmission Delay Difference between Measured and Ideal Ethernet networks.

Data Transmission	Transmission Delay Difference between Measured and Ideal Ethernet Networks (μs)									
Ethernet Data Field Size (Bytes)	***46***	***100***	***200***	***400***	***600***	***800***	***1000***	***1200***	***1400***	***1500***
Transmitting from A to B	43	45	47	49	55	57	62	68	72	71
Transmitting from A to C	54	56	56	60	63	72	70	73	74	78
Transmitting from B to A	42	44	45	53	55	63	62	64	68	78
Transmitting from B to C	56	58	56	57	62	65	69	70	74	76
Transmitting from C to A	54	60	53	56	60	66	68	69	84	75
Transmitting from C to B	55	60	55	58	68	64	69	73	91	91

The experimental results showed that the transmission delays between Systems A and B (bi-directional) were approximately 10 μs less for small Ethernet packets and 8 μs less for large Ethernet packets than the delays measured between other nodes in the system. The results of three experiments that were performed to measure the transmission delays of minimum sized Ethernet packets are shown in Figure 36. The different delays can be attributed to the underlying hardware of the nodes, such as the PCI and memory bus systems. It is not as a result of the different Ethernet fly-lead lengths, since the maximum propagation delay of an Ethernet link segment cannot exceed 1 μs (IEEE Std 802.3, 2000). The experiments also showed that the effect of data padding was minimal.

Table 14. Measured Data Transmission Latencies and Jitter.

Data Transmission	Minimum Latency (ms)		Average Latency (ms)		Maximum Latency (ms)		Jitter (ms)	
Transmitting from A to:	***B***	***C***	***B***	***C***	***B***	***C***	***B***	***C***
Minimum data (46 bytes)	0.086	0.095	0.087	0.099	0.095	0.113	0.009	0.018
Minimum data (18 bytes padding)	0.086	0.096	0.088	0.099	0.095	0.114	0.009	0.018
Maximum data (1500 bytes)	1.274	1.281	1.276	1.285	1.286	1.301	0.012	0.019
Transmitting from B to:	***C***	***A***	***C***	***A***	***C***	***A***	***C***	***A***
Minimum data (46 bytes)	0.095	0.086	0.098	0.088	0.111	0.094	0.016	0.008
Minimum data with padding	0.095	0.085	0.098	0.087	0.112	0.097	0.017	0.012
Maximum data (1500 bytes)	1.279	1.273	1.283	1.275	1.295	1.287	0.017	0.013
Transmitting from C to:	***A***	***B***	***A***	***B***	***A***	***B***	***A***	***B***
Minimum data (46 bytes)	0.093	0.095	0.096	0.097	0.115	0.114	0.021	0.020
Minimum data with padding	0.093	0.095	0.095	0.098	0.116	0.115	0.023	0.020
Maximum data (1500 bytes)	1.280	1.284	1.284	1.287	1.303	1.305	0.023	0.021

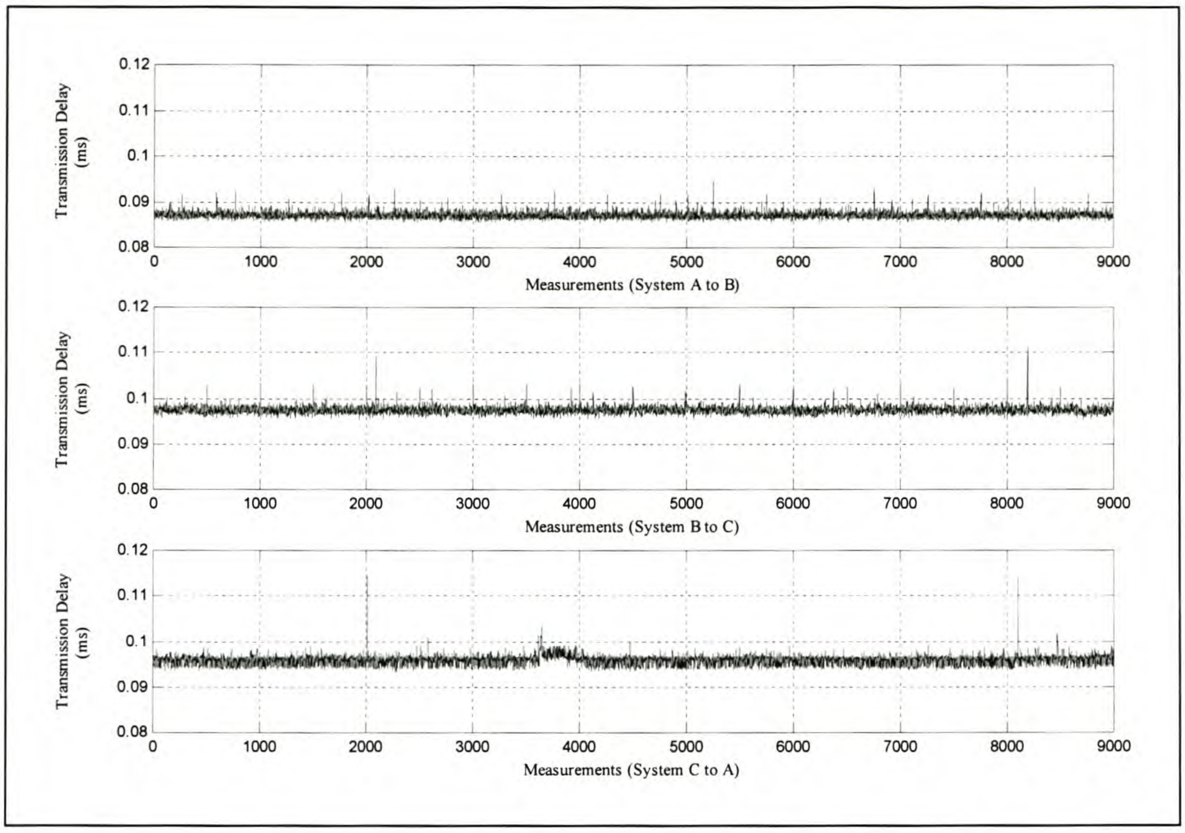

Figure 36. Measured Transmission Delays for Minimum Sized Ethernet Packets between all Systems in the Experimental Distributed Real-Time System.

7.5.2 Transmission Jitter

Transmission delay variation (jitter) is another way of measuring the determinism of a real-time communication system. It's usually calculated as the maximum difference between a desired transmission time and the individually measured transmission times. *However, for the purpose of the study it was defined as the difference between the maximum and minimum transmission times.* The results for the measured transmission jitter are also summarised in Table 14.

Reasonably high transmission jitter was measured between the nodes in the experimental distributed real-time system. The maximum jitter for both minimum and maximum sized Ethernet packets was measured at 23 μs. The minimum measured jitter was 9 μs for the smallest and 12 μs for the largest Ethernet packets. Bi-directional transmissions between Systems A and B measured lower jitter than transmissions to and from System C. These results indicated that the hardware of System C was less deterministic for real-time communication. Note that all systems had the same network interface cards, but System C had a different hardware architecture in comparison to Systems A and B. Overall however, the experiments proved that the RTnet networking subsystem offered consistent and bounded transmission jitter.

7.5.3 RTnet Performance Conclusion

Consistent transmission delays with bounded jitter were measured between all nodes in the experimental distributed real-time system. Transmission delays depend on the RTnet networking subsystem, the underlying hardware and the size of transmitted Ethernet packets.

The measured transmission delays increased proportionally for larger Ethernet packets. The rate of increase was slightly higher rate than that for ideal Ethernet networks. Figures similar to Figure 35 can therefore be used calculate the true rate at which the transmission delay increases (gradient of measured transmission delays). Such figures can also be used to estimate the transmission delay for a specifically sized Ethernet packet sent between nodes in a distributed real-time system.

To conclude, predictable real-time communication can be achieved with the RTnet networking subsystem. The applicability however, depends on whether the empirically calculated transmission delay and jitter meet the requirements of the targeted real-time application.

7.6 Real-Time Token Protocol

Hard real-time communication systems have no tolerance for message loss. The RTToken protocol also addressed this requirement. By controlling network access to a shared Ethernet network, both deterministic and collision-free real-time message transmission can be achieved.
Several experiments were conducted to measure the performance and protocol efficiency of the RTToken protocol. The first experiments measured the accuracy of the token holding time for all the nodes in the experimental distributed real-time system. Experiments were also conducted to measure the token arrival interval and token rotation time. The difference between the measured token rotation time and the total token holding time of all nodes, gives an indication of the RTToken protocol overheads for a specific configuration. This however, excludes the overheads involved in receiving token acknowledgement messages. To determine the responsiveness of the fault tolerance mechanism of the RTToken protocol, the token acknowledgement response times were also measured under steady state operation.

7.6.1 Token Holding Time

The RTToken protocol allocates fixed token holding times to all communicating nodes in the distributed real-time system. The token holding time determines the time duration that a node is allowed to transmit over the network after it acquired the token. The cumulative token holding times of all nodes also determine the token rotation time. Therefore, the accuracy of the token holding times of all the nodes in the network are crucial for deterministic network access control. Variations change both the token arrival interval and network access time, potentially causing real-time messages to miss their transmission deadlines.

To measure the accuracy of the token holding time for all the nodes in the distributed real-time system, the RTToken protocol was configured to broadcast a routing table, allocating 1 ms, 10 ms and 100 ms token holding times to all nodes in the network. The software monitors were installed on all nodes to log the token holding times for off-line processing. Each experiment consisted of 10 000 measurements with the first 1 000 discarded. The results are summarised in Table 15.

Table 15. Measured Token Holding Times for All Nodes in the Tested Systems.

Token Holding Times	Minimum (ms)	Average (ms)	Maximum (ms)	Jitter (ms)
System A:				
Token holding time of 1 ms	0.997	1.002	1.003	0.003
Token holding time of 10 ms	9.996	10.002	10.004	0.004
Token holding time of 100 ms	99.996	100.001	100.004	0.004
System B:				
Token holding time of 1 ms	0.996	1.002	1.006	0.006
Token holding time of 10 ms	9.995	10.002	10.004	0.005
Token holding time of 100 ms	99.996	100.001	100.004	0.004
System C:				
Token holding time of 1 ms	0.998	1.003	1.006	0.006
Token holding time of 10 ms	9.999	10.002	10.007	0.007
Token holding time of 100 ms	100.001	100.002	100.004	0.004

The experimental results proved that the token holding time of the RTToken protocol was very accurate and consistent. A maximum jitter of 7 μs was measured. Thus, the performance of the RTToken protocol in terms of accurate token holding times can be considered predictable, provided the jitter is bounded. The measured token holding time of 10 ms for all the nodes in the experimental distributed real-time system is shown in Figure 37.

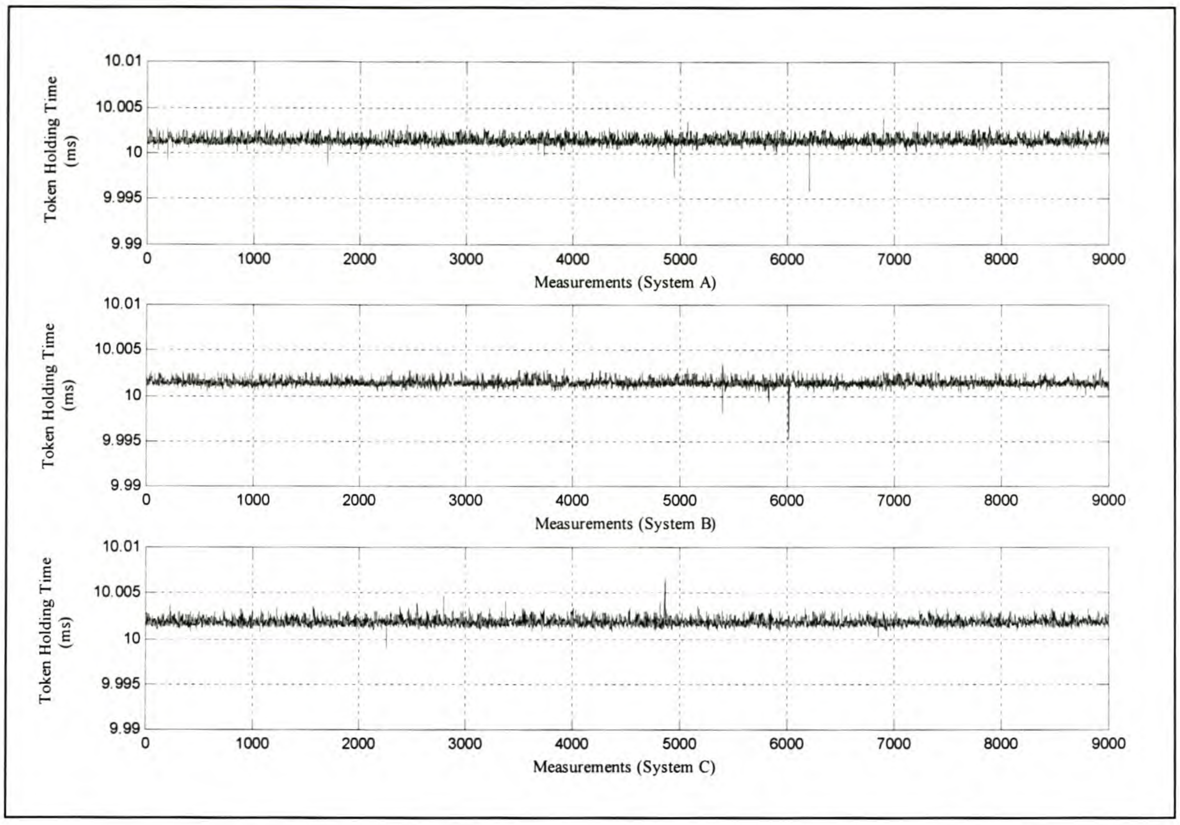

Figure 37. Measured Token Holding Times of the RTToken Protocol.

7.6.2 Token Arrival Interval

The token arrival interval is an important parameter in real-time token passing communication protocols. The time at which real-time messages are ready for transmission, must be co-ordinated with the availability of network resources, otherwise these messages will miss their transmission deadlines and be considered lost. Predictable token arrival times allow real-time tasks to schedule their transmission times accordingly. Token protocols with deterministic token arrival intervals can also be used to periodically time-synchronise nodes in a distributed real-time system. Hereby, all nodes will know the expected arrival time of the token.

Experiments were conducted to empirically determine the token arrival times at all nodes in the experimental distributed real-time system. The token arrival interval is calculated as the difference between consecutive token visits. Similar to previous experiments, nodes were configured with 1 ms, 10 ms and 100 ms token holding times. The RTToken protocol was configured to pass the token between all the nodes. For each experiment, 10 000 measurements were made with the first 1 000 discarded to measure steady state operation. Software monitors were installed on all the nodes to log measurements. Figure 38 illustrates the measured token arrival intervals on Systems A, B and C. All nodes were configured with 10 ms token holding times. The overall results obtained are summarised in Table 16.

Table 16. Measured Token Arrival Intervals for All Nodes in the Tested Systems.

Token Arrival Interval	Minimum (ms)	Average (ms)	Maximum (ms)	Jitter (ms)
Measured at System A:				
Token holding time of 1 ms	3.**253**	3.**261**	3.**309**	0.056
Token holding time of 10 ms	30.**257**	30.**262**	30.**297**	0.040
Token holding time of 100 ms	300.**258**	300.**264**	300.**284**	0.026
Measured at System B:				
Token holding time of 1 ms	3.**252**	3.**261**	3.**311**	0.058
Token holding time of 10 ms	30.**255**	30.**262**	30.**322**	0.067
Token holding time of 100 ms	300.**251**	300.**259**	300.**316**	0.066
Measured at System C:				
Token holding time of 1 ms	3.**252**	3.**261**	3.**310**	0.058
Token holding time of 10 ms	30.**254**	30.**263**	30.**312**	0.058
Token holding time of 100 ms	300.**257**	300.**264**	300.**310**	0.054

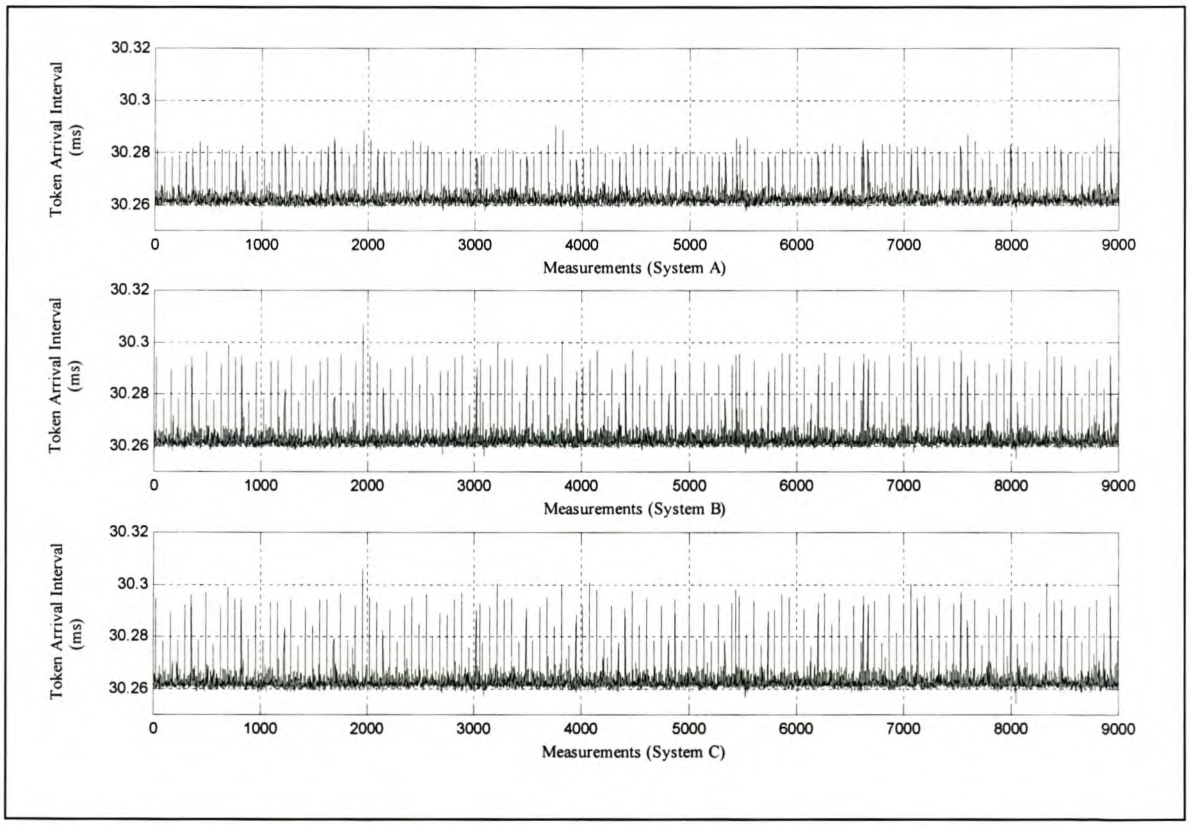

Figure 38. Measured Token Arrival Intervals of the RTToken protocol at All Nodes in the Tested System.

For all nodes in the network, similar token arrival intervals were measured for each experiment. Maximum token arrival intervals of 3.311 ms, 30.322 ms and 300.316 ms were respectively measured. The minimum token arrival intervals were respectively 3.252 ms, 30.254 ms and 300.251 ms. The error bound on the token arrival interval (jitter) was calculated as the difference between the maximum and minimum measurements for each experiment. Hereby, a maximum jitter of 67 μs was measured on System B for the experiment with 10 ms token holding times. In general the measured jitter was consistent for all experiments, averaging 54 μs. It can be concluded that the token arrival interval was deterministic, but with large bounded error margins.

7.6.3 Token Rotation Time

In the previous experiments, each node received the token once during a token rotation cycle. The token arrival intervals are therefore also indicative of the token rotation time. In general, the token rotation time is a global parameter measured at all nodes in the network.

Because all nodes in the distributed real-time system measured the same token arrival time (token rotation time), these measurements were expected to be consistent. The token arrival interval results are reworked in Table 17 to evaluate the accuracy of the token rotation times measured at all nodes.

Table 17. Measured Token Rotation Times for All Nodes in the Tested Systems.

Token Rotation Times	System A (ms)	System B (ms)	System C (ms)	Variation (ms)
Token holding time of 1 ms				
Minimum token rotation time	3.253	3.252	3.252	0.001
Average token rotation time	3.261	3.261	3.261	0.000
Maximum token rotation time	3.309	3.311	3.310	0.001
Token holding time of 10 ms				
Minimum token rotation time	30.257	30.255	30.254	0.003
Average token rotation time	30.262	30.262	30.363	0.001
Maximum token rotation time	30.297	30.322	30.312	0.025
Token holding time of 100 ms				
Minimum token rotation time	300.258	300.251	300.257	0.007
Average token rotation time	300.264	300.259	300.264	0.005
Maximum token rotation time	300.284	300.316	300.310	0.032

Minimum variations in the token rotation times were measured between 1 and 7 μs. The averaged token arrival time variations were between 0 and 5 μs. These variations were limited. However, larger variations were observed between the maximum measured token rotation time values. The maximum measurements varied between 2 and 32 μs. It was also observed that these measured variations in token rotation times increased with larger token holding times assigned to the nodes in the network (worst variations were measured for experiments with large token holding times).

It can be concluded that the token rotation time can be accurately determined at all nodes in the network, but with some error bound. *The observation that the error bounds increased with larger token rotation times is a topic for further investigation and is not covered in this study.*

7.6.4 RTToken Protocol Overheads

The token rotation time includes the token holding times of all the nodes and other overheads associated with the processing and passing of the token. Previous results proved the token holding times of the RTToken protocol to be very accurate. Therefore, the measured token arrival times (token rotation times) can be used to calculate the processing and token passing overheads of the RTToken protocol for the distributed real-time system configuration. The protocol overheads were thus approximated as the difference between the measured token arrival intervals and the expected token holding times of all nodes for one token cycle. Assuming ideal token holding times of 1 ms, 10 ms and 100 ms respectively, the overheads are highlighted in bold in Table 16. Note that the token overheads include the sending of the token acknowledgement message, but not the reception thereof. A minimum overhead of 252 μs and a maximum of 322 μs were measured for the RTToken protocol in the test configuration.

Experiments were performed to measure the token transmission and processing times of the RTToken protocol. The processing time was measured as two components. The first component was the time difference between the reception of the token and the start of the token holding time. This measurement included the processing time required to do a checksum on the received token. The second processing time was measured from after the token holding time until the agent task finished execution. *Note that the processing time included the creation and passing of the token to the device driver of the interface card.* Thus there is some overlap between the token transmission times and the processing times of the RTToken protocol.

The experimental set-up was the same as described in the previous experiments. The processing overheads of the RTToken protocol were measured with software monitors installed on all nodes in the network. The token transmission times were measured with the same technique used to measure the transmission times of the RTnet protocol. After the transmission of the token, the sending node waits in a busy loop for the parallel port to go high (set by the receiving node). The transmission delays were measured as the time duration between the transmission of the real-time message and the change in parallel port state. Experiments consist of 5 000 measurements with the first 100 discarded to reflect steady state operation. The results for the above-mentioned configuration are summarised in Table 18.

Table 18. Measured RTToken Protocol Overheads at All Nodes in the Tested Systems.

Token Holding Times	Minimum (ms)	Average (ms)	Maximum (ms)	Jitter (ms)
System A:				
Processing time after receiving a token	0.006	0.006	0.015	0.009
Processing time after token holding time	0.008	0.008	0.010	0.002
Token transmission time to System B	0.080	0.082	0.086	0.006
System B:				
Processing time after receiving a token	0.006	0.006	0.012	0.006
Processing time after token holding time	0.008	0.008	0.010	0.002
Token transmission time to System C	0.082	0.084	0.089	0.007
System C:				
Processing time after receiving a token	0.004	0.004	0.011	0.006
Processing time after token holding time	0.008	0.008	0.011	0.003
Token transmission time to System A	0.081	0.083	0.087	0.006
RTToken protocol overheads (ms)*	***0.283***	***0.289***	***0.331***	***0.047***

* *Estimation because the token creation and handling are included in the token transmission and processing times.*

Similar token transmission times were measured for all nodes in the network. A maximum token transmission time of 89 μs was measured with up to 7 μs of jitter. Thus the token transmission times of the RTToken protocol measured an error bound of 9%. The measured protocol overheads were also similar to the previous approximations made. For the tested configuration, maximum time duration of 322 μs was measured. This included three token transmissions. Thus on average, the protocol processing took about 14 μs on each node. Note however, that large error margins were measured for the token processing times; especially the processing after a node received the token. This can be attributed to variations in the routing table look-up times.

7.6.5 Token Acknowledgement Response Time

The token acknowledgement response time is the time duration between sending a token to the next recipient, and receiving an acknowledgement that the token was received. To measure this, the token was passed between all nodes, configured with 10 ms token holding times. Software monitors on all nodes logged both the time that the token was sent and the time that the acknowledgement was received (another 10 000 measurements). The token acknowledgement response time was calculated off-line as the difference between the latter two measurements (again the first 1 000 calculations were discarded). The results are summarised in Table 19, with some measurements illustrated in Figure 39.

Table 19. Measured Token Acknowledgement Response Time for All Tested Nodes.

Token Acknowledgements	Minimum (ms)	Average (ms)	Maximum (ms)	Jitter (ms)
Measured at System A:				
Received from System B	0.161	0.162	0.169	0.009
Received from System C	0.162	0.163	0.170	0.008
Measured at System B:				
Received from System C	0.161	0.163	**0.178**	0.016
Received from System A	0.161	0.163	0.171	0.009
Measured at System C:				
Received from System A	0.162	0.164	0.172	0.011
Received from System B	0.162	0.164	0.171	0.009

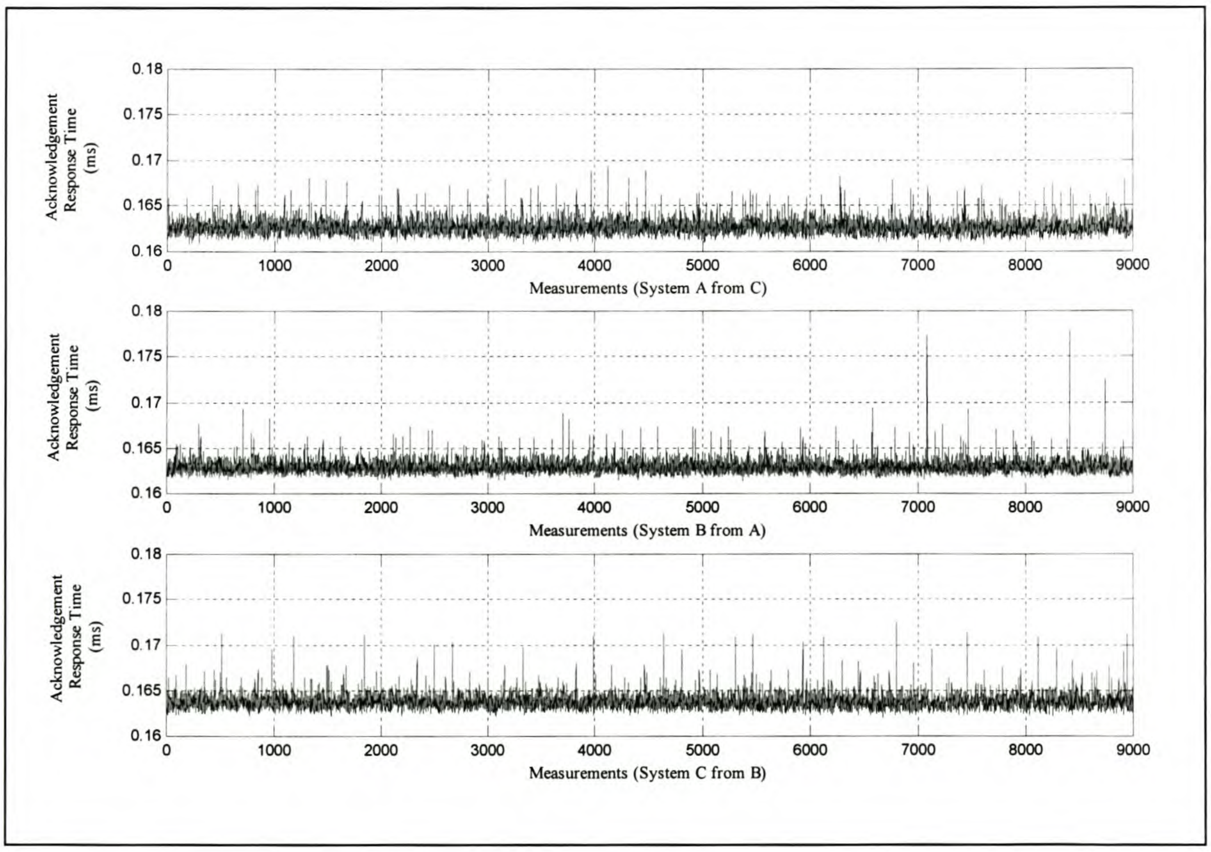

Figure 39. Measured Token Acknowledgement Response Times between All Systems.

Experiments measured a worst-case acknowledgement response time of 178 µs. In general the responsiveness of all the tested systems were similar. Note that under normal operation, the token recipient would respond immediately when a token was received. Therefore, the measured results are also an indication of the fastest possible acknowledgement response times. *These results, when used to program the acknowledgement time, determine the maximum responsiveness of the RTToken protocol in detecting token losses.*

7.6.6 RTToken Performance Conclusion

The RTToken protocol proved to offer deterministic token holding times and predictable token arrival intervals to all nodes in the network. All nodes in the experimental distributed real-time system could also accurately measure the token rotation time. Furthermore, the processing overheads of the protocol were also minimised and measured consistent for the configuration that it was tested with[44]. The RTToken protocol also measured reasonable fault-detection responsiveness in relation to the token acknowledgement response times. Thus, overall the protocol proved to offer deterministic performance to facilitate real-time communication over shared Ethernet networks.

7.7 RTToken Network Access Latency

The RTToken protocol delayed network access and therefore the transmission of real-time messages. Real-time messages ready for transmission have to wait for the arrival of the network access control token before they can be transmitted. If these messages miss their transmission deadlines they are considered lost. An experiment was therefore conducted to measure the network access latency as a result of the RTToken protocol. The experiment also illustrated that distributed real-time applications with RTToken can be designed such that real-time message transmission deadlines are always met.

The experimental distributed real-time system was configured with 20 ms token holding time for all nodes in the network. Therefore, both the token arrival interval and token rotation time were approximately 60.3 ms for all nodes (an estimated protocol overhead of 0.3 ms according to Section 7.6.4). Each node implemented a periodic real-time task that executed with 120 ms intervals. These tasks transmitted real-time messages of 72 bytes over the Ethernet, using the

[44] Note that increased variations can be expected for larger configurations, requiring more time for routing table handling.

RTnet UDP over IP protocols (thus 100 bytes in the Ethernet frame's data field). Real-time messages were transmitted as follows:

- The real-time task on System A transmitted to a receiving task on System C.
- The real-time task on System C transmitted to a receiving task on System B.
- The real-time task on System B transmitted to a receiving task on System A.

The real-time tasks are also configured with 50 ms transmission deadlines. Note that for the specified configuration, network transmission deadlines longer than 40.3 ms (difference between the token rotation time and the node's token holding time) are always met, provided the token arrival intervals are deterministic. For the experiment, software monitors are installed on all nodes and configured to log 3000 measurements. The network access latencies measured for Systems A, B and C are shown in Figure 40.

For the test configuration, a maximum network access latency of 36.222 ms (System C) was measured. Therefore, no real-time tasks missed their transmission deadlines. It can be concluded that the RTToken protocol is capable of guaranteeing bounded network access latencies, as required for real-time communication.

In relation to the measured network access latencies, a shortcoming of the semaphore functionality of the RTAI operating system was observed. The RTToken protocol used a semaphore to control access to the shared Ethernet network. Hereby real-time tasks were queued on the semaphore in priority order for network access. The *sem_wait_timed()* function was used to implement transmission deadlines. However, it was observed that this function influenced the scheduling of periodic real-time tasks. Depending on how long a real-time task was queued on the semaphore, it changed the scheduling period of the real-time task. The changes in the scheduling period of the real-time tasks in the previous experiment are shown in Figure 41. The measurements are related to the network access latency measurements shown in Figure 40.

It was observed that the scheduling period of tasks that temporarily queued on the semaphore, varied between 120 ms (expected) and 170 ms. Further tests proved that the maximum scheduling jitter (50 ms) equalled the configured transmission deadlines. *This erroneous behaviour of the timed semaphore function of the RTAI operating system is a topic for further investigation.*

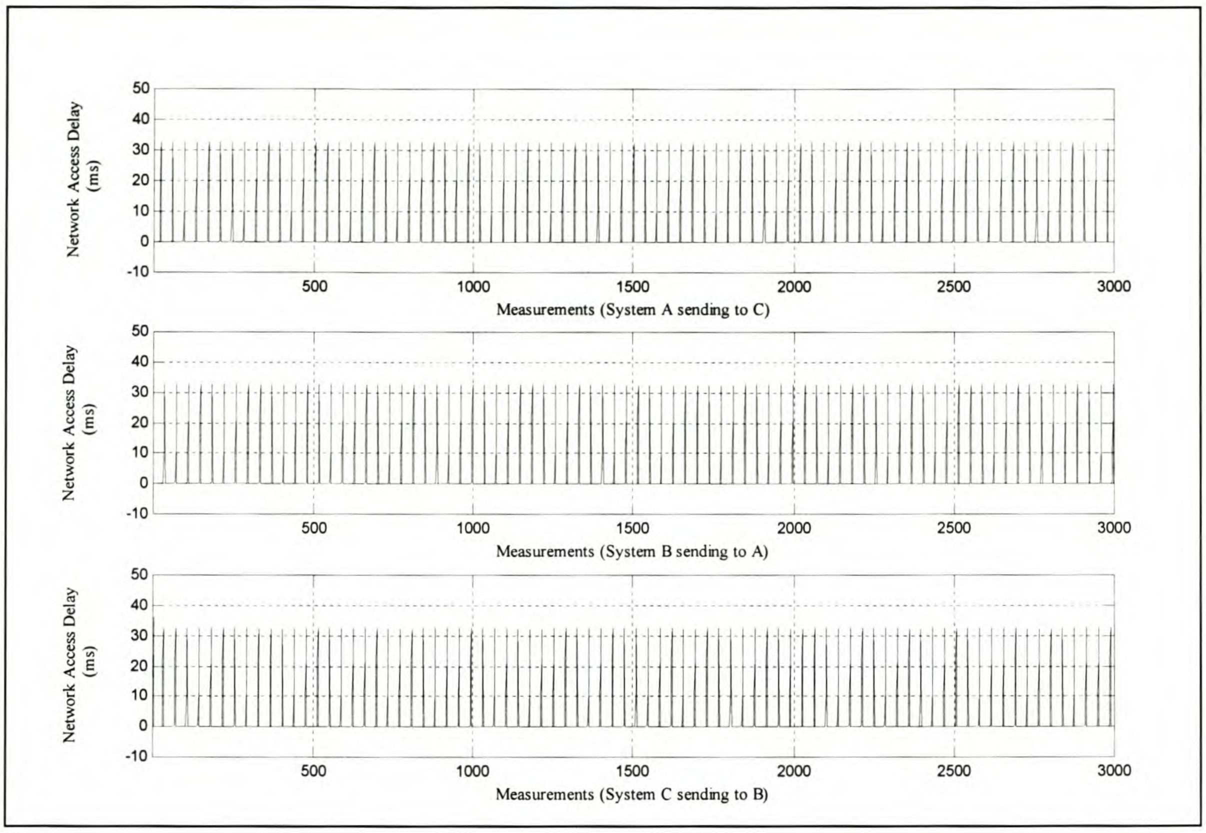

Figure 40. Measured Network Access Latency for the Experimental System, Configured with 20 ms Token Holding Times and 50 ms Transmission Deadlines.

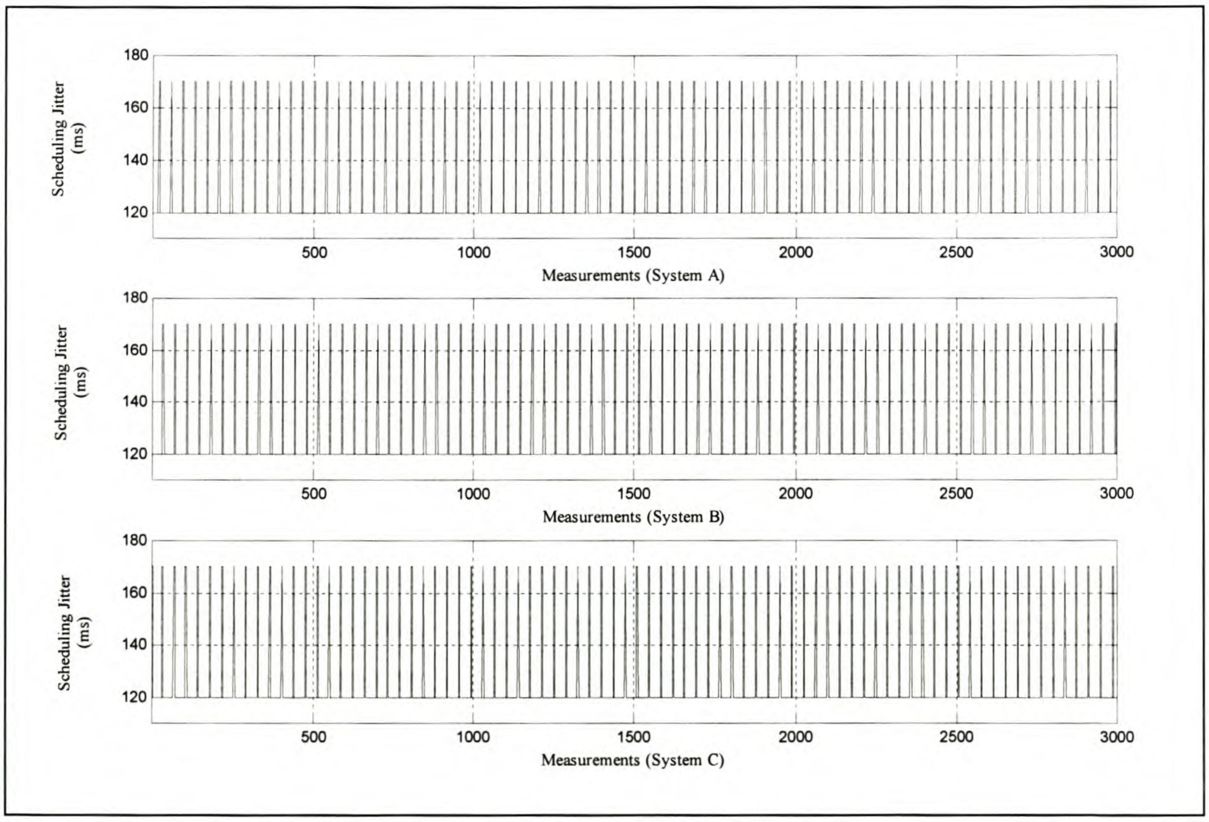

Figure 41. Erroneous Scheduling Jitter of Periodic Tasks (120 ms) Queued on a Timed Semaphore.

8 CONCLUSIONS

This study investigated hardware and software technologies with the potential to reduce the development time and cost of satellites as well as their supporting infrastructure. These technologies also addressed a common requirement of various systems in the satellite environment. The requirement for distributed hard real-time computing was addressed with research conducted on hard-real-time operating- and communication systems.

Various real-time extensions and modifications are available for the Linux operating system. A literature survey proved sufficient for the identification of an open source, hard real-time operating system suitable as the basis for further development. Other real-time extensions were either very limited in functionality, lacked true hard real-time performance, or open source developments stalled due to commercialisation. In contrast, RTAI offered more functionality, is still actively being developed and extensive documentation is available.

Quantitative results showed that the RTAI operating system delivered deterministic real-time performance in the absence of non-real-time load. Consistent and fast interrupt response times of 4 μs (1 μs jitter) were measured for 10 ms interrupt intervals. Non-real-time loads however, significantly degraded real-time performance. The signalling methods of RTAI also proved to be fast (1.2 μs average), but with large bounded jitter (approximately 4 μs), degrading with non-real-time load. The use of commercial off-the-shelf PC hardware also introduced irregular hardware events that affected the real-time performance of RTAI. The performance degradation was most prominent under high hard disk and network load conditions. To compensate, sources of interference were minimised, but experiments still showed the random occurrence of such events. Based on the results of this study, the RTAI operating system can at best be used for firm real-time applications, i.e. where some low probability of missing a deadline can be tolerated. Ideally non-real-time load should be removed, real-time load should be moderate and commercial hardware should be characterised for real-time applications.

The second objective was to design a real-time communication protocol for single segment shared Ethernet networks. Various methods have been proposed in the literature to improve the timing behaviour of Ethernet networks. A comparative study concluded that a token passing protocol is the simplest and most effective method to guarantee hard real-time communication, without changes to the Ethernet hardware. Token protocols arbitrate access to the shared network with a circulating token, thereby controlling the transmission time of all messages. The RTToken protocol was designed to assign fixed token holding times to all nodes in the network (distributed real-time system), trading off efficient bandwidth utilisation for real-time determinism. The protocol was implemented as a modular, loadable kernel module for easy re-usability.

In the various experiments conducted, the RTToken protocol proved to deliver predictable performance. For example, the protocol measured a maximum token holding time variation of 7 μs in a test configuration where all nodes were allocated 10 ms token holding times. A worst-case variation in the token arrival intervals of 67 μs was measured for all nodes. Individual nodes were able to determine the token rotation time with a maximum error bound of 32 μs. The RTToken protocol overheads were also minimised, with an average processing time of 14 μs per node and a maximum token transmission time of 89 μs (with up to 7 μs of jitter) between nodes. It can be concluded that the overall performance of the RTToken protocol is deterministic, but with some amount of error (jitter). Thus, when the RTToken protocol is used in real-time applications, both the jitter and protocol overheads need to be considered.

In token passing protocols, losing the token is a critical fault that can make the network inaccessible. As a result, the handling of token losses is characteristic of these protocols. In this study, the RTToken protocol was only designed with limited fault tolerance. The protocol monitored the transmission of the token between nodes. Hereby, the receiving node sent an acknowledgement to the sending node. As a result, conditions can however still occur where token loss is undetectable. In addition, no error handling was included for network buffer overflows.

The third objective was to integrate the RTAI operating system and the RTToken protocol. RTAI supports the RTnet networking subsystem that allows real-time tasks to communicate over Ethernet. The supported UDP/IP protocols of RTnet were used to communicate between distributed real-time tasks. These protocols proved to deliver consistent transmission delays (between 115 μs and 1305 μs, depending on the message size) with bounded jitter (23 μs maximum), measured on an otherwise idle (i.e. collision free) network and under minimum system load conditions. The RTToken protocol was configured to control network access (transmission times) with a RTAI semaphore on each node. Tasks ready to transmit real-time messages were queued on the semaphore and in priority order, until the token arrived. Thus, the real-time messages also inherit the priorities of the sending tasks. When the token arrives, the highest priority task is signalled to start transmission. A test implementation proved that the RTToken protocol could provide predictable and bounded network access to real-time tasks. However, it was observed that the timed semaphore function of the RTAI operating system affected the scheduling precision of periodic real-time tasks. These tasks are delayed depending on the duration queued on the semaphore. The maximum delay was equal to the time-out value assigned to the semaphore function.

In conclusion, the thesis proved that the proposed commercial hardware and software technologies could address the requirement for distributed real-time computing. With the Linux operating system being one of the fastest growing operating systems and continued efforts being made to establish Ethernet as a networking standard in the industrial environment, these technologies cannot be ignored. Although in it's infancy, efforts are already made to apply these technologies in satellite applications. It is proposed that once matured, these technologies will be a viable solution for a variety of satellite applications. This potential justifies that an assertive effort be made to further research and develop these technologies. It is recommended that future work be conducted in a group effort to retain the expertise.

8.1 Thesis Contributions

This thesis has presented the motivation, background theory, technique and results obtained towards the development of a distributed real-time system for future satellite applications. The contributions of the thesis stem from the defined research objectives. These are:

- The comparison and evaluation of currently available open source hard real-time extensions and modifications for the Linux operating system.
- The design, implementation and verification of a hard real-time communication protocol for shared Ethernet networks.
- The extension of a hard real-time Linux operating system (RTAI) with a real-time communication protocol for shared Ethernet networks. This improved the potential of Linux to be used in various distributed real-time applications in the satellite and other environments.

8.2 Future Research

Based on the technologies used and developed in this study, future research should focus on:

- The offered functionality of the RTAI real-time operating system needs to be verified and corrected. For example, tests showed that the timed semaphore function degraded the schedulability of periodic real-time tasks. Various functions were also implemented with no proper return values.

- Both development and debugging tools should be investigated. Development tools are available that can, for example, generate real-time source code from simulation models developed under MATLAB, Simulink and Real-Time Workshop. Debugging tools such as the Linux Trace Toolkit can graphically display a system's behaviour for a specific period of time.
- The scalability of the RTAI operating system needs to be investigated. Various techniques can be used to reduce the footprint of the RTAI operating system. These, together with the portability to the various supported embedded processors need to be evaluated. Portability can be also be improved by extending the compliance to POSIX standards.
- Although the thesis only focused on open source real-time extensions to the Linux operating system, various commercial modifications and extensions are also available. These also need to be researched.
- The fault tolerance of the RTToken protocol needs further improvement to cater for all conditions where the token could be lost. The protocol can also be extended to support multi-segment shared Ethernet networks.
- Should programmable firmware (such as FPGA's) be considered to implement a real-time, shared Ethernet network, various methods are proposed and should be investigated.
- Switched Ethernet is becoming a very popular technique to improve the performance of shared Ethernet. However, by simply adding a switch to an Ethernet network, it is not enough to guarantee real-time communication. The switch introduces additional and undeterministic transmission latencies for uncontrolled network loads. For real-time communication, appropriate admission control policies must be added. This is a topic for further research.
- It is recommended that the distributed real-time system be further extended with task scheduling algorithms (EDF, Rate Monotonic or others) and distributed time synchronisation (GPS, Network Time Protocol (NTP), tightly synchronised clock cards and others). Using design methodologies (task graphs and end-to-end timing constraints), the system should be further evaluated in distributed real-time applications.

9 REFERENCES

Agrawal, G., Chen, B., Zhao, W., Davari, S. 1992. *Guaranteeing Synchronous Message Deadlines with the Timed Token Medium Access Control Protocol*. International Conference on Distributed Computing Systems. Pp. 468-475.

ANSI/IEEE Std 802.4. 1990. *Information processing systems – Local area networks – Part 4: Token-passing bus access method and physical layer specifications* (Revision of ANSI/IEEE Std. 802.4-1985). Adopted by ISO/IEC and redesignated as ISO/IEC 8802-4:1990.

ANSI Std X3.139. 1987. *Fiber Distributed Data Interface (FDDI), Token Ring Media Access Control* .

Aras, C.M., Kurose, J.F., Reeves, D.S., Schulzrinne, H. 1994. *Real-time Communication in Packet-Switched Networks*. Proceedings of the IEEE. Vol. 82(1), pp.122-139.

Barabanov, M. 1997. A Linux-based Real-Time Operating System. Master of Science Thesis, New Mexico Institute of Mining and Technology.

Batra, R.K. 1997. The Design of a Highly Configurable Reusable Operating System for Testbed Satellites. 11th AIAA/USU Conference on Small Satellites, Logan, UT.

Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D. 1996. *Linux Kernel Internals*. Addison Wesley Longman Limited.

Berrendorf, R. Ziegler, H. 1998. *PCL – The Performance Counter Library: A common interface to access hardware performance counters on microprocessors, Version 1.2*. Central Institute for Applied Mathematics, Research Centre Juelich GMBH, Germany. Web reference: http://www.fz-juelich.de/zam/PCL/

Bird, T. 2000. *Right-sizing Linux for Embedded Devices*. Embedded Linux Expo and Conference (ELEC 2000) Presentation. June 2000. Web reference: http://www.linuxdevices.com/

Braden, R.T., Borman, D.A., Partridge, C. 1988. *Computing the Internet Checksum*. RFC 1071.

Caldwell, D.W., Chau, S.N. 1994. *Spacecraft Information Systems: Principles and Practice*. JPL Document (no number). 25 April 1994. Web reference: http://fstsrv.jpl.nasa.gov/-library/papers/avionix/

Card, R., Dumas, É., Mével, F. 1998. *The Linux Kernel Book*. John Wiley & Sons Ltd.

Carpenter, B., Roman, M., Vasilatos, N., Zimmerman, M. 1997. *The RTX Real-Time Subsystem for Windows NT*. Proceedings of the USENIX Windows NT Workshop, Seattle, WA, pp. 33-37, August 1997.

Casani, K.E., Thomas, N.W. 1994. *The JPL Flight System Testbed*. Proceedings of the Spring '94 SPIE Conference, Orlando, FA.

Chiueh, T. Venkatramani, C. 1994. *Supporting Real-Time Traffic on Ethernet*. Proceedings of the IEEE Real-Time Systems Symposium (RTSS'94).

Cloutier, P. Mantegazza, P. Papacharalambous, S. Soanes, I. Hughes, S. Yaghmour, K. 2000. *DIAPM-RTAI Position Paper*. 21st IEEE Real-Time Systems Symposium (RTSS'2000) – Real-time Operating Systems Workshop. Orlando.

Cruz, R.L. 1991. *A calculus for network delay, part I: network elements in isolation*. IEEE Transactions on Information Theory.

Cypress, 1997. Clock Terminology. Application notes. October 1994 – Revised 9 July 1997. Web reference: http://www.cypress.com

Dankwardt, K. 2000. *Comparing real-time Linux alternatives*. 11 October 2000. Web reference: http://www.linuxdevices.com/articles/AT4503827066.html

Demers, A. Keshav, S. Shenker, S. 1990. *Analysis and Simulation of a Fair Queueing Algorithm*. Journal of Internetworking Research and Experience. pp.3-26, October 1990.

DIAPM. 2002. *DIAPM – RTAI Beginners Guide*. Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano - Real Time Application Interface. Web reference: http://www.aero.polimi.it/~rtai/

Dinkel, W., Niehaus, D., Frisbie, M., Woltersdorf, J. 2002. *KURT-Linux User Manual*. University of Kansas. Draft version. 29 March 2002. Web reference: http://www.ittc.ukans.edu/kurt/

Elmenreich, W., Haidinger, W., Peti, P., Schneider, L. 2002. *New Node Integration for Master-Slave Fieldbus Networks*. 20th IASTED International Conference on Applied Informatics, Innsbruck, Austria.

Furht, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., Roberts, M. 1991. *Real-time UNIX systems: design and application guide*. Kluwer Academic Publishers.

Gallmeister, B.O. 1995. *POSIX.4 - Programming for the Real World*. O'Reilly & Associates Inc.

Gillen, A., Kusnetzky, D. 2001. *Embedded Operating Environments: How does Linux Compare?* An IDC Whitepaper. Web reference: http://www.idc.com

Grow, R.M. 1982. *A timed token protocol for local area networks*. Proceedings of Electro'82. Token Access Protocols, May 1982.

Hildebrand, D. 1992. *An Architectural Overview of QNX*. Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp. 113-126, April 1992.

Hill, R., Srinivasan, B., Pather, S., Niehaus, D. 1998. *Temporal Resolution and Real-Time Extensions to Linux*. Information and Telecommunication Technology Centre, Department of Electrical Engineering and Computer Sciences, University of Kansas. Technical Report. Web reference: http://www.ittc.ukans.edu/kurt/

Hubley, M. Lubrano, C. 2001. *Linux Operating System Distributions: Perspective*. Technology Overview, 21 August 2001. Gartner Research.

IEEE Std 1284. 2000. *IEEE Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers (IEEE 1284-2000).*

IEEE Std 802.3. 2000. *IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements - Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*. 2000 Edition (Incorporating IEEE Std 802.3, 1998 Edition, IEEE Std 802.3ac-1998, IEEE Std 802.3ab-1999 and IEEE Std 802.3ad-2000). Adopted by ISO/IEC and redesignated as ISO/IEC 8802-3:2000(E).

Intel Corporation. 1997. *Model Specific Registers and Functions*. Pentium Processor Family Developer's Manual, Chapter 16. Web reference: http://developer.intel.com

INtime, 1998. *INtime Interrupt Latency Report: Measured Interrupt Response Times*. Technical Paper, November 1998. Web reference: http://www.radisys.com

INtime, 1998a. *Determinism and the PC Architecture. Applying PC Hardware to Real-time Applications.* Technical paper. June 1998. Web reference: http://www.radisys.com

ISO 11898. 1993. *International Organisation for Standardisation (ISO) International Standard 11898: Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communication.*

Jeffay, K. 1993. *The Real-time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*. Proceedings of the ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, USA. pp. 796-804. February 1993.

Jilla, C.D., Miller, D.W. 1995. *Satellite Design: Past, Present and Future*. International Journal of Small Satellite Engineering, Issue No. 1.

Kopetz, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

Kopetz, H. 1997a. *Component-based design of large distributed real-time systems*. 14th IFAC Workshop on Distributed Computer Control Systems (DCCS'97), pp. 171-177.

Kopetz, H. 1998. *The Time-Triggered Model of Computation*. Proceedings of the IEEE Real-Time Systems Symposium. Madrid, Spain. pp. 168-177. December 1998.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., Zainlinger, R. 1989. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, Vol. 9(10), pp. 25-40.

Kuhn, M. 1998. *A Vision for Linux 2.2: Posix.1b Compatibility and Real-Time Support*. Web reference: http://www.cl.cam.ac.uk/~mgk25/linux-posix.1b.txt

Kuhn, B. 2001. *An overview of real-time Linux*. Real-time and Embedded Systems forum. Presentation 9 April 2001. Web reference: http://www.opengroup.org/rtforum/

Kurose, J.F. Schwartz, M. Yemini, T. 1983. *Controlling window protocols for time-constrained communication in a multiple access environment*. Proceedings of the 8th IEEE International Data Communication Symposium.

Kweon, S. Shin, K.G. Workman, G. 2000. *Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing*. Proceedings of the IEEE Real-Time Technology and Applications Symposium.

Kweon, S. Shin, K.G. Zheng, Q. 1999. *Statistical Real-Time Communication over Ethernet for Manufacturing Automation Systems*. Proceedings of the IEEE Real-Time Technology and Application Symposium.

Laplante, P.A. 1997. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. Second Edition. IEEE Press.

Laplante, P.A. 2000. *A Practical Approach to Real-Time Systems: Selected Readings*. IEEE, Inc.

Lehrbaum, R. 2001. *A Survey of Embedded Linux Packages*. The Linux Journal. Issue 82, pp. 38-40. 1 February 2001.

Le Lann, G. Rivierre, N. 1993. *Real-time communication over broadcast networks: The CSMA-DCR and the DOD-CSMA-CD protocols*. Technical Report. Institute National De Recherche En Informatique Et En Automatique, France.

Levi, S.T., Agrawala, A.K. 1990. *Real-Time System Design*. McGraw-Hill Publishing Company.

Lineo Inc. 2000. *DIAPM RTAI Programming Guide 1.0*. September 2000. Web reference: http://www.aero.polimi.it/~rtai/

LinuxDevices. 2000. *Lineo announces GPL real-time networking for Linux: RTnet*. 27 July 2000. Web reference: http://www.linuxdevices.com/sponsors/SP8515396280-NS4023517008.html (Developer: David Schleef)

LinuxDevices. 2002. *The Real-time Linux Software Quick Reference Guide*. October 2002. Web reference: http://www.linuxdevices.com/articles/AT8073314981.html

LinuxDevices. 2002a. *RED-Linux: for real-time and Embedded systems*. Web reference: http://www.linuxdevices.com/articles/LK7432493143.html

Liu, C.L., Layland, J.W. 1973. *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*. Journal of the ACM, Vol. 20(1), pp. 44-61.

Magliacane, J.A. 2002. PREDICT: a multi-user satellite tracking and orbital prediction program. Web reference: http://www.qsl.net/kd2bd/predict.html

Malcolm, N. Zhao, W. 1994. *Use of the Timed Token Protocol for Real-Time Communications*. IEEE Computer. Vol. 27(1), pp 35-41.

Malcolm, N. Zhao, W. 1995. *Hard Real-Time Communication in Multiple-Access Networks*. Journal of Real-Time Systems. Vol. 9(1), pp. 35-78.

Mantegazza, P. 1999. *Dissecting DIAPM RTHAL-RTAI*. Web reference: http://www.aero.polimi.it/~rtai/

Milne, G.W., Mostert, S., Schoonwinkel, A., Du Plessis, J.J., Steyn, W.H., Van der Westhuizen, K., Van der Merwe, D.A., Grobler, H., Koekemoer, J.A., Steenkamp, N. 1999. *SUNSAT - Launch and First Six Months' Orbital Performance*. 13th AIAA/USU Conference on Small Satellites.

Molesky, L., Ramamritham, K., Shen, C., Stankovic, J., Zlokapa, G. 1990. *Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel*. Extended abstract, IEEE Workshop on Real-Time Operating Systems and Software, May 1990.

Molle, M.L. 1994. *A New Binary Logarithmic Arbitration Method for Ethernet*. Technical Report CSRI-298, University of Toronto, Toronto, CA.

Molle, M.L., Kleinrock, L. 1985. *Virtual time MSMA: Why two clocks are better than one*. IEEE Transactions on Communications. Vol. COM-33(9).

Mourot, P. 1999. *RTAI Internals Presentation*. Web reference: http://www.aero.polimi.it/

Norden, S. Balaji, S. Manimaran, G. Siva Ram Murthy, C. 1999. Deterministic protocols for real-time communication in multiple access networks. Computer Communications. Vol 22(2), pp. 128-136.

Obenland, K.M. 2001. *The Use of POSIX in Real-time Systems, Assessing its Effectiveness and Performance*. Embedded Systems Programming Vol. 14(2).

PASC. 2002. *Status Report: PASC Status (including POSIX)*. Portable Applications Standards Committee of the IEEE Computer Society. Last updated on 14 January 2002. Available at: http://www.pasc.org/standing/sd11.html

Pitts, D., Ball, B. 1998. *Red Hat^{TM} Linux Unleashed, Third Edition*. SAMS Publishing.

Quaranta, G. Mantegazza, P. 2001. *Using MATLAB-Simulink Real-Time Workshop (RTW) to Build Real Time Control Applications in User Space with RTAI-LXRT*. Real-time Linux Workshop. Milano, 2001.

QNX Software Systems Ltd. 2001. *QNX Version 6.1.0: System Architecture*. Web reference: http://www.qnx.com

Ramakrishnan, K.K. Yang, H. 1994. *The Ethernet capture effect: analysis and solution*. Proceedings of the 19th Conference on Local Computer Networks.

Ramamritham, K., Stankovic, J.A. 1994. *Scheduling Algorithms and Operating Systems Support for Real-Time Systems*. Proceedings of the IEEE, Vol. 82(1), pp. 55-67.

Ritchie, D.W., Thompson, K. 1974. *The UNIX time-sharing system*. Communications of the Association for Computing Machinery. Vol. 17(7), pp. 265-375.

Rubini ,A. Corbet, J. 2001. Linux Device Drivers. Second Edition. O'Reilly & Associates Inc.

Saksena, M. Da Silva, J., Agrawala, A.K. 1995. *Design and Implementation of Maruti-II*. In Advances in Real-Time Systems, Son, S.H. (Ed.), Prentice Hall.

Sarolahti, P. 2001. *Real-Time Application Interface*. Research seminar on Real-Time Linux and Java, Spring 2001. Department of Computer Science, University of Helsinki. Web reference: http://www.aero.polimi.it/~rtai/

SED. 1992. *Analogue-to-Digital, Digital-to-Analogue and Digital I/O Card for the PC*. Technical Manual: Technical Description and Operation.

Sexton, F.W., Fleetwood, F.W., Aldridge, C.C., Garrett, G., Pelletier, J.C., Gaona, J.I. Jr. 1992. *Qualifying Commercial ICs for Space Total-Dose Environments*. IEEE Transactions on Nuclear Science, Vol. 39(6), pp.1869-1875.

Siewert, S., McClure, L.H. 1995. *A System Architecture to Advance Small Satellite Mission Operations Autonomy*. 9th AIAA/USU Conference on Small Satellites.

SourceForge. 2003. *Project: RTnet - Real-Time Networking for RTAI*. Web reference: http://www.sourceforge.net/projects/rtnet

Srinivasan, B., Pather, S., Hill, R., Ansari, F., Niehaus, D. 1998. *A firm real-time system implementation using commercial off-the-shelf hardware and free software*. Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'98).

Stakem, P.H. 2001. *FlightLinux Project: Onboard LAN. Technical Report*. NASA Advanced Information System Technology (AIST) research effort NRA-99-OES-08. Initial Release: 31 May 2001.

Stakem, P.H. 2002. *FlightLinux: a Viable Option for Spacecraft Embedded Computers*. Earth Science Technology Conference (ESTC 2002). Pasadena, CA. June 2002.

Stallings, W. 1997. *Data and Computer Communications*. Prentice Hall International, Inc. Fifth Edition.

Stankovic, J.A. 1988. *Misconceptions about real-time computing: A serious problem for next generation systems*. IEEE Computer Society Press.

Stankovic, J.A. 1992. *Real-Time Computing*. BYTE magazine invited paper, pp. 155-160, August 1992.

Stankovic, J.A., Ramamritham, K. 1988. *Tutorial: Hard Real-Time Systems*. IEEE Computer Society Press.

Stankovic, J.A., Ramamritham, K. 1990. *What is Predictability for Real-Time Systems*. Real-Time Systems Journal, Vol. 2, pp. 247-254.

Stevens, W.R. 1994. *TCP/IP Illustrated, Volume 1. The Protocols.* Addison-Wesley Professional Computing Series.

Tanenbaum, A.S. 1989. *Computer Networks*. Prentice-Hall International, Inc.

Tanenbaum, A.S. 1992. *Modern Operating Systems*. Prentice-Hall International, Inc.

Thomesse, J-P., Mammeri, Z., Vega, L. 1995. Time in Distributed Computing Systems: Co-operation and Communication Models. 5th IEEE Workshop on Future Trends of Distributed Systems, Cheku Island, Korea. IEEE Computer Society Press, August 1995, pp. 41-49.

Tindell, K., Burns, A. and Wellings, A.J. 1995. *Analysis of Hard Real-Time Communications*. Real-Time Systems Vol. 9(2), pp. 147-171. Kluwer Academic Publishers. September 1995.

Tokuda, H., Nakajima, T., Rao, P. 1990. *Real-Time Mach: Towards a Predictable Real-Time System*. Proceedings of USENIX Mach Workshop, October 1990.

Tsai, J.J.P., Bi, Y., Yang, S.J.H., Smith, R.A.W. 1996. *Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis*. A Wiley-Interscience Publication.

Ulusoy, O. 1995. A Network Access Protocol for Hard Real-Time Communication Systems, Computer Communications, Vol. 18(12), pp. 943-948.

Wang, Y.C., Lin, K.J. 1998. *Enhancing the real-time capability of the Linux kernel*. Proceedings of the 5th Conference of Real-Time Computing Systems and Applications (RTCSA'98).

Wang, Y.C., Lin, K.J. 1999. *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*. Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99).

Wang, Y.C., Lin, K.J. 2000. *A short tutorial on Redice-Linux installation and programming*. Web reference: http://www.redsonic.com

Ward, J., Sweeting, M. 1999. *First In-Orbit Results from the UoSAT-12 Minisatellite*. 13th AIAA/USU Conference on Small Satellites.

Webb, W. 1998. *Ethernet invades embedded space*. EDN magazine design feature. 11 September 1998. pp. 71-80. Web reference: http://www.ednmag.com

Weinberg, B., Lundholm, C. 2001. *Embedded Linux: Ready for Real-Time*. Whitepaper. MontaVista Software. Web reference: http://www.mvista.com

Wellings, A., Burns, A. 1996. *Real-Time Systems and Their Programming Languages*. Second Edition. International Computer Science Series. Addison-Wesley Publishing Company.

Whetten, B. Steinberg, S. Ferrari, D. 1994. *The packet starvation effect in CSMA/CD LAN's and a solution*. Proceedings of the IEEE Local Computer Networks. pp. 206-217.

Wind River Systems. 1999. *VxWorks Programmer's Guide 5.4, Edition 1, Part #: DOC-12629-ZD-01*. Web reference: http://www.windriver.com. 6 May 1999.

Yavatkar, R. Pai, P. Finkel, R. 1992. *A Reservation-based CSMA protocol for Integrated Manufacturing Networks*. Technical Report 216-92. Department of Computer Science, University of Kentucky.

Yodaiken, V. 1999. *The RT-Linux Manifesto*. Proceedings of the 5th Linux Expo, Raleigh, North Carolina, USA. May 1999.

Zhao, W. Ramamritham, K. 1987. *Virtual Time CSMA Protocols for Hard Real-Time Communication*. IEEE Transactions on Software Engineering. Vol. 13(8), pp. 938-952.

Zhao, W. Stankovic, J.A. Ramamritham, K. 1990. *A Window Protocol for Transmission of Time-Constrained Messages*. Proceedings of IEEE Transactions on Computers. Vol. 39(9), pp. 1186-1203.

A REAL-TIME ETHERNET HARDWARE MODIFICATIONS

A.1 Collision Counter Modifications

Molle (1994) proposed the ***Binary Logarithmic Arbitration Method (BLAM)*** algorithm as an improvement to the Binary Exponential Back-off (BEB) algorithm currently being used in the IEEE 802.3 CSMA/CD Ethernet protocol. The BLAM algorithm solves the inefficiencies of the BEB algorithm, in particular the large and unpredictable transmission delays under moderate to high network load conditions and the Ethernet Capture Effect (unfairness characteristics of BEB). The BLAM algorithm was reviewed for standardisation by the IEEE 802.3w (Enhanced MAC Algorithms) working group.

The BLAM algorithm modifies the collision counter of the back-off algorithm. The collision counter determines the back-off period of a node when a collision has occurred. The traditional BEB algorithm increments the collision counter asymmetrically, which effectively resembles a Last-Come-First-Serve (LCFS) queue where packets that incurred multiple collisions will experience increasingly large delays for retransmission. The BLAM algorithm on the other hand, increments the collision counter symmetrically. When a collision occurs, the counter is incremented by one. After a successful transmission the collision counter is reset to zero. If the network idles for two or more consecutive time periods since the last collision counter increment, it is decremented by one. These modifications promote transmission fairness and reduce the network access delay variance of the BEB algorithm.

The BLAM algorithm also includes a network-holding timer. Hereby, a node can continuously transmit packets until the timer expires. Other nodes will only start transmission after the timer expires or when the network is sensed idle. By limiting the continued transmission duration of a node, the Ethernet Capture Effect is avoided.

The ***Capture Avoidance Binary Exponential Back-off (CA-BEB)*** protocol was designed to solve the Ethernet Capture Effect for networks with a small number of nodes (Ramakrishnan and Yang, 1994). The protocol modifies the collision counter during the early stages (first two collisions) of collision resolution. When a collision occurs during the transmission of the second packet of a non-interrupted consecutive transmit, the node backs off for two time periods. If the other colliding packet is new (no previous collisions), it will be transmitted next (back-off of zero or one period). After the two time periods, the packet is ready for retransmission. If a second collision should occur, the back-off value is set to zero (immediate retransmission). The standard BEB algorithm applies for subsequent collision resolution.

A.2 New Collision Resolution Algorithms

The ***CSMA with Deterministic Collision Resolution (CSMA-DCR)*** protocol was designed to resolve network collisions in a deterministic way (Le Lann and Rivierre, 1993). The protocol replaced the traditional BEB algorithm with a binary search tree algorithm. Hereby, the network nodes are assigned an index number (key) and then statically mapped to the vertices of a binary search tree[45]. During the collision resolution mode, transmission rights are given to network nodes based on the position of every node in the binary search tree. The tree thus represents a hierarchy of node priorities. For example, Figure 42 shows five nodes indexed and mapped (the mapping is left to the designer) to the vertices of an arbitrary collision resolution binary search tree. Using pre-order transversal[46], the order in which the nodes will be allowed to transmit after a collision is Node 1, Node 2, Node 3, Node 4 and Node 5.

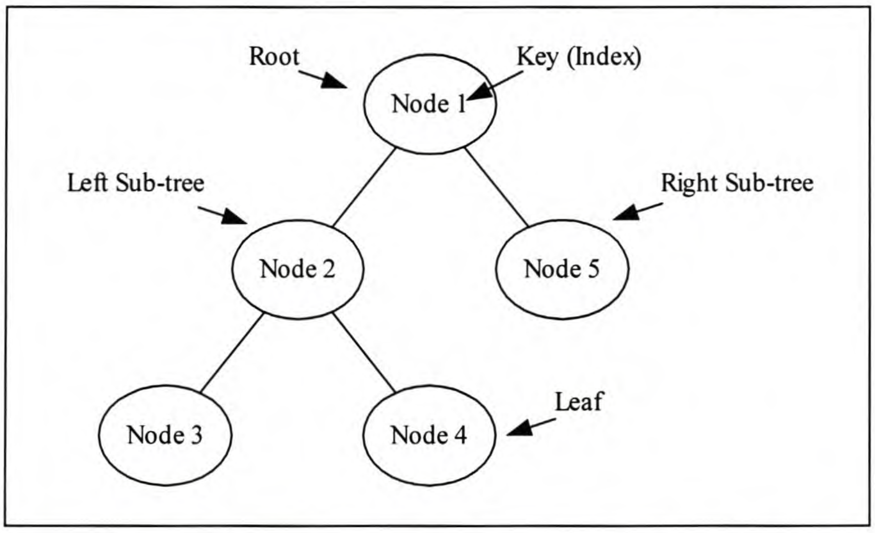

Figure 42. Mapping of Network Nodes into a Binary Search Tree.

[45] In a Binary Search Tree, the top node is called the **root**. Each node has below it a **left sub-tree** and/or a **right sub-tree**. A node with no sub-trees is called a **leaf**. The network node **keys** are stored individually in each binary search tree node.

[46] With pre-order traversal the contents of a tree is sorted in pre-order. The root node is first, followed by the left sub-tree and then the right sub-tree (repeated at different tree depths).

Norden et. al. (1999) proposed the ***Laxity based Deterministic Collision Resolution (LDCR)*** protocol to prevent possible priority inversion that can occur as a result of the static transmission order that the latter protocol enforces. Each node in the network maintains a sorted queue of real-time messages. Messages are prioritised based on laxity. The protocol follows the same ordered collision resolution mode as before. When a node is allowed to transmit, it transmits a notify message to inform all nodes of the message requirements of the sending node. If the laxity of the first message queued at the sending node, is less than a fixed threshold, it is transmitted. If the laxity is negative, the message is dropped and if it is greater than the threshold, the message is deferred and per order transversal continues. After the collision resolution mode, the protocol enters the least laxity first mode if some deferred messages still need to be transmitted. Previously deferred messages are either transmitted in least laxity first order among nodes or dropped (negative laxity). The collision resolution and least laxity first modes define a period in the LDCR (compared to the collision resolution mode in CSMA-DCR). Thus, a new period begins after all deferred messages have either been transmitted or dropped.

A.3 Ethernet Frame Modification with Off-line Scheduling Strategy

The ***Predictable-CSMA (PCSMA)*** protocol extends the Ethernet CSMA/CD protocol with a static off-line scheduling strategy to support both real-time and non-real-time communication (Yavatkar et. al., 1992). The extension was specifically designed for periodic real-time messages. Non-real-time packets are still handled by the standard Ethernet CSMA/CD protocol.

Network arbitration between nodes transmitting periodic real-time messages are based on a bandwidth reservation policy. These nodes reserve bandwidth (transmission slots) before run-time. Because the bandwidth reservation of nodes sending real-time messages with different time periods will conflict, the protocol includes a conflict resolution algorithm that can shift the scheduled transmission of one of the conflicting nodes so that the delivery times of both nodes are still met.

To ensure that non-real-time packets cannot interfere with the transmission of real-time packets, the PCSMA protocol uses a special real-time frame structure with a large pre-emption field. The different frame structures of Ethernet (802.3) and PCSMA are shown in Figure 43. When a non-real-time packet collides with a real-time packet, the non-real-time node will detect the collision, stop transmitting and back off according to the Ethernet CSMA/CD protocol. The real-time node will persist with transmission. The pre-emption field is designed to be long enough that the effect of the collision will not interfere with the real-time content of the packet. The resultant disadvantage is significant overheads in real-time messages.

If a node is scheduled to transmit a real-time packet, but finds the network busy because a non-real-time packet is being transmitted, the node must wait for the network to become idle. Because the node missed it's scheduled transmission time and more than one node might be ready to transmit a real-time packet, all the nodes execute an earliest deadline first rescheduling procedure to compute new transmission times in order to avoid network conflicts.

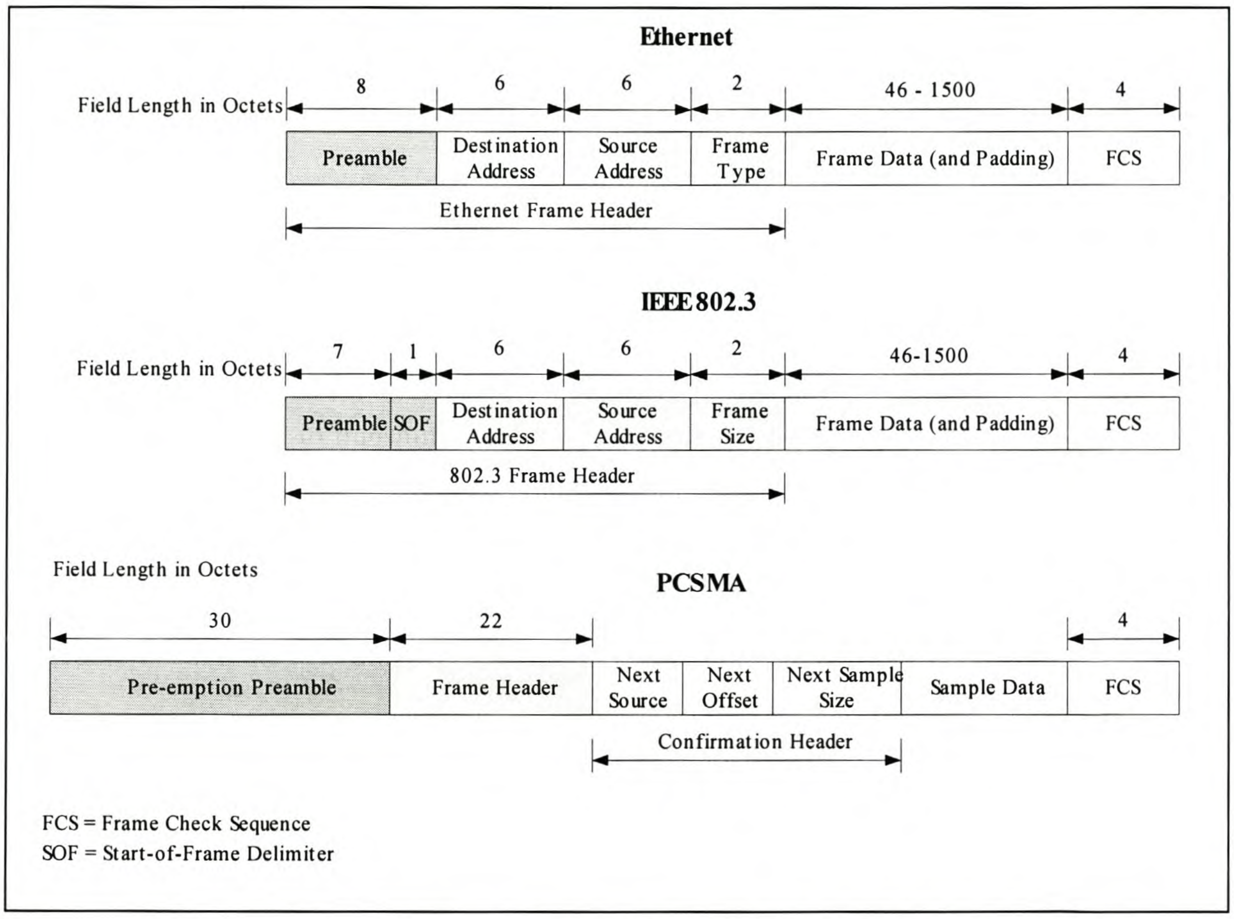

Figure 43. Comparative Structures of the Ethernet, IEEE 802.3 and PCSMA Frames.

B RTAI TASK SCHEDULING PRECISION

B.1 Periodic Task Scheduling

Experiments were conducted under different real-time and non-real-time load conditions to measure the scheduling jitter and drift of a periodic real-time task running under the RTAI operating system and assigned the highest priority. The task was scheduled to execute periodically with 1 ms intervals. All the systems in the experimental testbed were measured and the results are summarised in Table 20.

Table 20. Measured Scheduling Drift and Jitter under Different Load Conditions.

Tested System	Minimum Interval (μs)	Average Interval (μs)	Maximum Interval (μs)	Jitter (μs)	Schedule Drift after 5 min. (ms)
System A with minimum load	**996**	**1000**	**1004**	**4 (0.4%)**	**-0.36**
System A with hard-disk load	990	1000	1012	12 (1.2%)	-0.35
System A with network load	993	1000	1007	7 (0.7%)	-0.36
System A with real-time task load	**994**	**1000**	**1007**	**7 (0.7%)**	**-0.36**
System B with minimum load	**996**	**1000**	**1004**	**4 (0.4%)**	**-0.36**
System B with hard-disk load	989	1000	1011	11 (1.1%)	-0.36
System B with network load	993	1000	1008	8 (0.8%)	-0.36
System B with real-time task load	**993**	**1000**	**1007**	**7 (0.7%)**	**-0.36**
System C with minimum load	**991**	**1000**	**1011**	**11 (1.1%)**	**2.21**
System C with hard-disk load	984	1000	1016	16 (1.6%)	2.20
System C with network load	985	1000	1018	18 (1.8%)	2.20
System C with real-time task load	**991**	**1000**	**1011**	**11 (1.1%)**	**2.20**

In general the scheduler of the RTAI operating system proved to offer very acceptable scheduling performance with a maximum measured jitter under 11 μs for real-time task load conditions. The worst scheduling jitter for all systems was observed under non-real-time hard-disk load conditions (up to 18 μs was measured). The jitter measurements for System A and C are shown in Figure 44 and Figure 46. Note that jitter was calculated as the maximum difference between the expected scheduling time (1 ms) and the scheduling time that was measured.

The scheduling drift for all systems was also calculated. Both System A and B showed a negative drift of 0.36 ms after 5 minutes. System C measured a positive drift of 2.2 ms after 5 minutes. The measured scheduling drifts for System A and C are shown in Figure 45 and Figure 47. The scheduling drift indicated the inaccuracies involved in using the PC hardware clock. Thus hard real-time performance depends on hardware and software predictability. ***Precision hardware timing circuitry is recommended.***

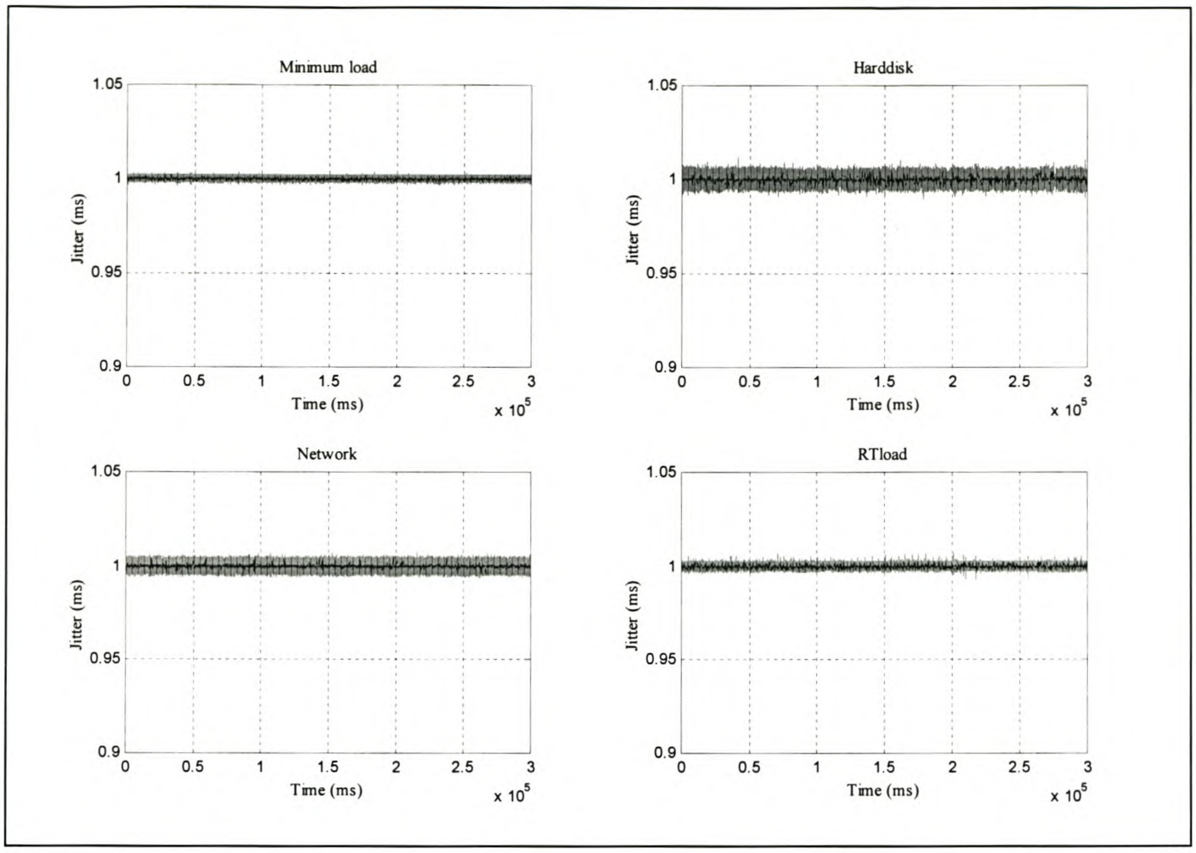

Figure 44. System A Scheduling Jitter Measured under Different Load Conditions.

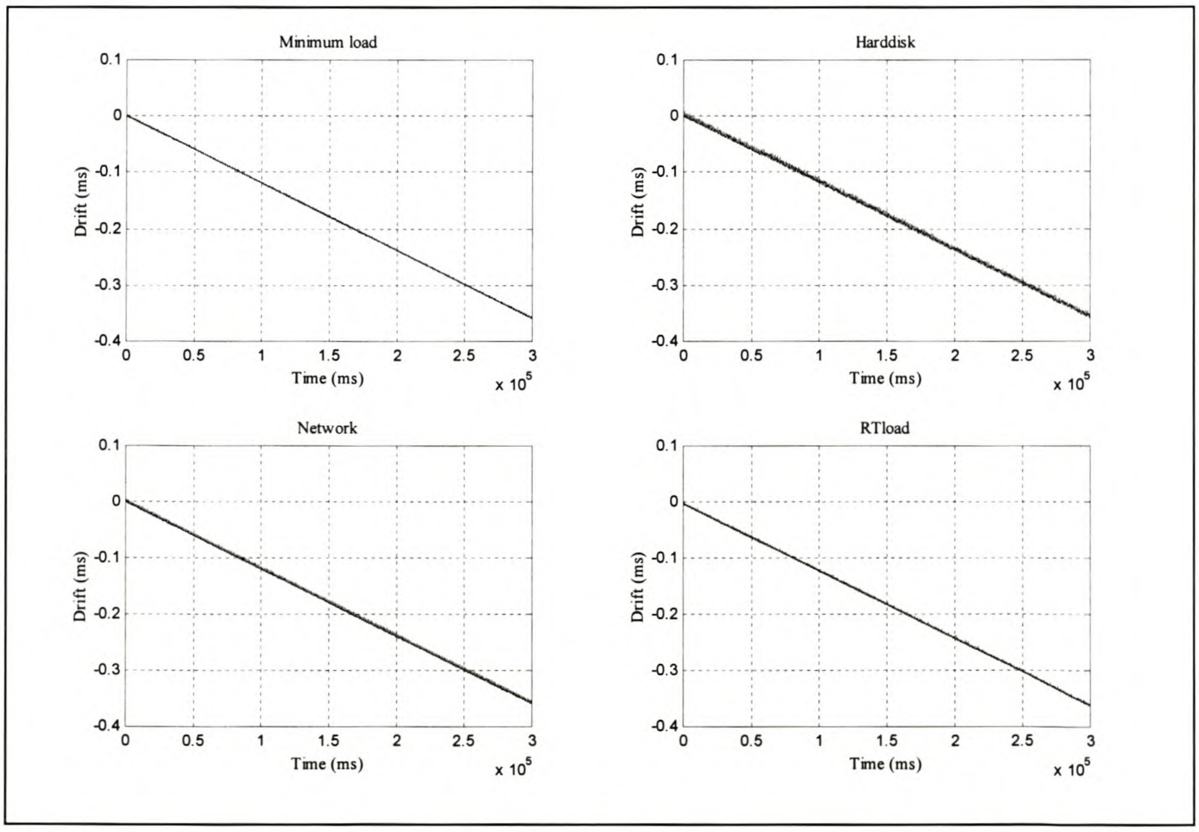

Figure 45. System A Scheduling Drift Measured under Different Load Conditions.

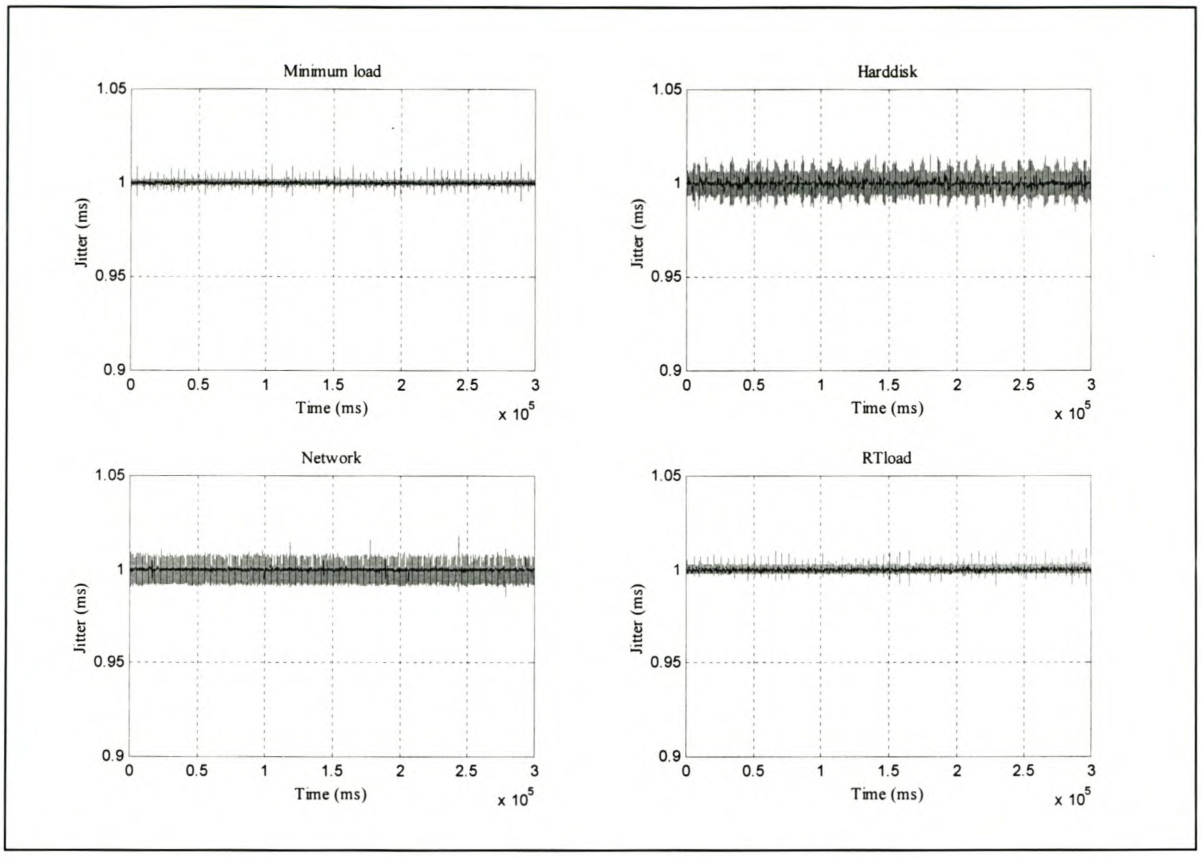

Figure 46. System C Scheduling Jitter Measured under Different Load Conditions.

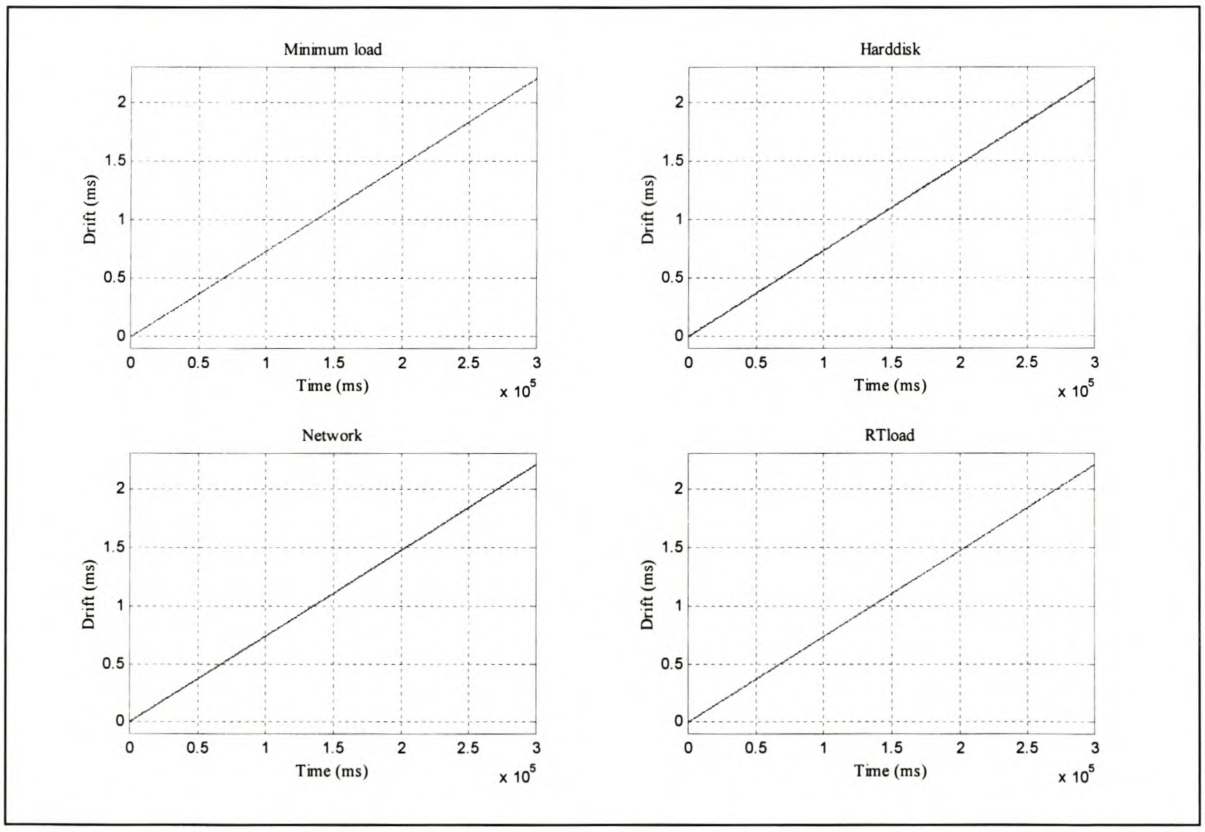

Figure 47. System C Scheduling Drift Measured under Different Load Conditions.

C COMPACT DISC CONTENT

The attached compact disk contains the thesis in both Microsoft Word 2000 and Portable Document Format (PDF) formats. Additional content is stored in the following directories:

C.1 installs

This directory contains installation files for:

- Linux 2.2.16 Operating System.
- RTAI 1.6 hard real-time extension.
- RTnet 0.9.1 networking subsystem.

C.2 mlabfigs

This directory contains:

- .fig files for all the figures in the thesis generated with Matlab.
- Matlab programs to generate plots of the measured results.

C.3 rttoken

This directory contains all the program files of the RTToken protocol.

C.4 tests

This directory contains all the test programs and results of the experiments that were conducted on all three the test systems in the experimental distributed real-time system. These are divided into three directories based on the software system that was tested:

C.4.1 rtai

Four subdirectories are included under this directory:

- **cal_load** – Contains test programs and measurements to empirically calculate the processing times of the real-time load tasks.
- **measure_irq** – Contains test programs and measurements of the interrupt latencies.
- **measure_irq_task** – Contains test programs and measurements of the interrupt task latencies, including the RTAI operating system semaphore and suspend/resume signalling methods.
- **task_sched** – Contains test programs and measurements of the scheduling precision of periodic real-time tasks with RTAI.

C.4.2 rtnet

One subdirectory (**measure_msg**) contains the test programs and measurements of the message transmission latencies and jitter of RTnet.

C.4.3 rttoken

Five subdirectories are included under this directory:

- **rtt_ack** – Contains test programs and measurements of the token acknowledgement response times.
- **rtt_holdtime** – Contains test programs and measurements of the token holding time of the RTToken protocol.
- **rtt_ohead** – Contains test programs and measurements of the RTToken protocol overheads.
- **rtt_rotation** – Contains test programs and measurements of the token rotation and token arrival intervals.
- **rtt_send** – Contains test programs and measurements of the network access delay as a result of the RTToken protocol.