Task Oriented Fault-Tolerant Distributed Computing for Use on Board Spacecraft

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Muhammad Fayyaz
Department of Engineering
University of Leicester

2015

Abstract

Current and future space missions demand highly reliable, High Performance Embedded Computing (HPEC). The review of the literature has shown that no single solution could meet both issues efficiently at present addressing HPEC as well as reliability. Furthermore, there is no suitable method of assessing performance for such a scheme.

In this thesis a novel cooperative task-oriented fault-tolerant distributed computing (FTDC) architecture is proposed, which caters for high performance and reliability in systems on board spacecraft. In a nut shell, the architecture comprises two types of nodes, a computing node and an input-output node, interfaced together through a high-speed network with bus topology. To detect faults in the nodes, a fault management scheme specifically designed to support the cooperative task-oriented distributed computing concept is proposed and employed, which is referred to as Adaptive Middleware for Fault-Tolerance (AMFT). AMFT is implemented as a separate hardware block and operates in parallel with the processing unit within the computing node. A set of metrics is designed and mathematical models of availability and reliability are developed, which are used to evaluate the proposed distributed computing architecture and fault management scheme.

As a new development, extending the current state of the art, the proposed fault-tolerant distributed architecture has been subjected to a rigorous assessment through hardware implementation. Implementation approaches at two levels were adopted to provide a proof of concept: a board level and a Multiprocessor System-on-Chip (MPSoC) level. Both distributed computing system implementations were evaluated for functional validity and performance.

To examine the FTDC architecture performance under a realistic space related distributed computing scenario a case-study application, representing a satellite Attitude and Orbit Control System (AOCS), was developed. The AOCS application was selected because it features a time critical task execution, in which system failure and reconfiguration time must be kept minimal. Based on the case-study application, it was demonstrated that the FTDC architecture is capable of fully meeting the desired requirements by timely migrating tasks to functional nodes and keeping rollback of task states minimal, which proves the advantages of the adopted cooperative distributed approach for use on board spacecraft.

Dedication

To My Late Mother, Father, and Father-in-Law.

May Allah rest their soul in peace.

Acknowledgements

"In the name of Allah, the most generous, most benevolent."

First of all, I would like to thank Almighty Allah, who gave me the power to achieve my goals. Then, I am very grateful to my supervisor Prof. Tanya Vladimirova for inculcating professionalism towards the achievement of my goals and for being a source of motivation and inspiration throughout the research. Her guidance and encouragement were instrumental in crafting and ensuring success in this research program. Without her advice and support, breaking dead ends would have been fatal.

I am also thankful to my second supervisors Prof. Mike Warrington and Dr. Alistair McEwan for their guidance and sharing of wisdom to ensure my work makes an impact. I am very grateful to Prof. Micheal Pont for his valuable feedback and encouragement during the APG exam. I am very thankful to the staff of the Engineering Department; especially Michelle Pryce for her support to sort out University related academic matters.

The research work would not be possible without the generous funding and support provided by the University of Leicester, UK, and the European Commission as part of the FP7 Project "ReVuS: Reducing the Vulnerability of Space Systems". I am very thankful to both organizations. Special thanks go to Jean-Michel Caujolle from Airbus Defence & Space, France for his guidance.

There are numerous students and academic staff that have supported me in technical and non-technical ways. These include, in no particular order, Paul William, Tom Robotham, Ian, Irfan, Muhammad Zia, Saad A. Malik, Jing, Xiaojun Zhai, Julie Clayton, Farah, and Ioannis. I am very thankful to all of you.

Last, but not the least; I am very grateful to my wife, mother-in-law, and colleagues for their support and encouragement throughout the Ph.D. research.

Declaration of Authorship

I hereby declare that this submission is my own work, and that is the result of work done during the period of registration. To the best of my knowledge, it contains no previously published material written by another person. None of this work has been submitted for another degree at the University of Leicester or any other University.

Contents

List of Figures

List of Tables

List of Abbreviations

A

ASIC	Application Specific Integrated Circuit
AMFT	Adaptive Middleware for Fault Tolerance
AOCS	Attitude and Orbit Control System
AWSN	Aerospace Wireless Sensor Network

B

BC	Bus Controller

C

CAN	Controller Area Network
CCM	Core Computing Module
CMP	Chip Multiprocessor

D

DCN	Distributed Computing Node
DES	Distributed Embedded System

E

ECC	Error Correcting Code
EGSE	Electrical Ground Support Equipment
EMC	Electromagnetic Compatibility
EMI	Electromagnetic Interference
ESA	European Space Agency

F

FD	Fault Detection
FDIR	Fault Detection, Isolation, and Recovery
FEC	Forward Error Correction
FM	Fault Management
FPGA	Field Programmable Gate Array
FT	Fault-Tolerant
FTDC	Fault-Tolerant Distributed Computing
FTD-OBC	Fault-Tolerant Distributed On-board Computer
FTDS	Fault-Tolerant Distributed System

H

HBM	HeartBeat Message
HPEC	High Performance Embedded Computing

J

JPL	Jet Propulsion Laboratory

L

LEO	Low Earth Orbit

M

MIPS	Million Instruction per Second
MPSoC	Multiprocessor System-on-chip

N

NASA	National Aeronautics and Space Administration
NMP	NASA’s New Millennium

O

OBC	On-board Computer
OBDH	On-board Data Handling
OMG	Object Management Group
OS	Operating System

P

PPIF	Point-to-point Interface
PRHB	Periodic Reception History Broadcast
PU	Processing Unit

R

REE	Remote Exploration and Experimentation
RTOS	Real-Time Operating System

S

SAR	Synthetic Aperture RADAR

SBST	Software-based Self-Test
SEEs	Single Event Effects
SEU	Single Event Upset
SoC	System-on-a-chip
SRAM	Static Random Access Memory
STAR	Self-Testing and Repairing
SUM	State Update Message
SNS	Supervisory-based Network Surveillance

T

TDMA	Time Division Multiple Access
TLM	Task List Message
TMR	Triple Modular Redundant
TTCAN	Time-Triggered Controller Area Network
TTEthernet	Time-Triggered Ethernet

U

UART	Universal Asynchronous Receiver Transmitter

W

WDT	Watchdog Timer
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network

X

XPP	eXtreme Processing Platform

List of Symbols

α_e	=	Attitude Angle Vector
ω_e	=	Attitude Angular Velocity Vector
t_{cs}	=	Duration of TDMA Slot
e	=	Exponential function
λ	=	Failure rate
s_0, s_1	=	States
t_D	=	Fault Detection Time
t_{FM}	=	Fault Message Sending Time
t_{TX}	=	Fault Message Transmission Time
t_{TM}	=	Fault Message Reception and Scheduling Time
$\frac{dP_{s0}(t)}{dt}$	=	First order derivative for State-0
$\frac{dP_{s1}(t)}{dt}$	=	First order derivative for State-1
$\forall$	=	For all
s	=	Laplace operator

$P_{s0(s)}$	=	Laplace Transform for State-0
$P_{s1(s)}$	=	Laplace Transform for State-1
$B^d{}_t$	=	Magnetic Field at time instance t
$B^d{}_{t-1}$	=	Magnetic Field at time instance t-1
B^{dot}	=	Magnetic Field derivative
m_b	=	Magnetometer measurement in body frame
m_i	=	Magnetic Field model value in inertial frame
Mbps	=	Megabits per second
$\mu sec.$	=	Microseconds
ms	=	Milliseconds
$[M_x, M_y, M_z]$	=	Moments
n	=	Number of Nodes
p	=	Number of Processors
k	=	Number of Replicas
$n_{rollback}$	=	Number of State Rollback
q	=	Number of subsystems
v	=	Number of Tasks
R_i	=	Node Reliability
t_R	=	Node recovery time
t_F	=	Node failure time
#	=	Number of Tokens

θ	=	Pitch Angle
t_{FD_period}	=	Period of Fault Detection Task
t_s	=	Primary to redundant Switching time
b	=	Probability of Success
$t_{FD\_processing}$	=	Processing Time for Fault Detection Task
$\prod$	=	Product
t_{Reconf}	=	Reconfiguration Time
m	=	Redundant unit per subsystem/ Number of migration
$R_{cent\_sys}(t)$	=	Reliability of centralized system
$R_{tmr\_sys}(t)$	=	Reliability of Triple Modular Redundant system
$R_{dist\_sys}(t)$	=	Reliability of distributed system
μ	=	Repair rate
Φ	=	Roll angle
R^{bi}	=	Rotation matrix from inertial to body frame
Δ_{SD}	=	State Data
$R_{sys}(t)$	=	System reliability
$P_{s0(0)}$	=	State-0 initial value
$P_{s1(0)}$	=	State-1 initial value
$P_{s0}(t)$	=	State-0 probability in time
$P_{s1(t)}$	=	State-1 probability in time

x	=	State Matrix
s_b	=	Sun sensor measurement in body frame
s_i	=	Sun sensor model value in inertial frame
T	=	Task Period
t	=	Time
u	=	Torque Vector
$t_1 b$	=	Triad Vector Component 1 of sensor-1 in body frame
$t_2 b$	=	Triad Vector Component 2 in body frame
$t_3 b$	=	Triad Vector Component 3 of $t_1 b \; x \; t_2 b$ body frame
$t_1 i$	=	Triad Vector Component 1 of Model-1 in inertial frame
$t_2 i$	=	Triad Vector Component 2 in inertial frame
$t_3 i$	=	Triad Vector Component 3 of $t_1 i \; x \; t_2 i$ of in inertial frame
R^{bt}	=	Triad to body frame rotation matrix
R^{it}	=	Triad to inertial frame rotation matrix
t_{sch}	=	Time required to schedule the Migrated Tasks
Δt	=	Time interval of failure
ψ	=	Yaw Angle

Chapter 1

Introduction

Distributed embedded systems are ubiquitous and have deeply penetrated into our society [1]. Major sectors targeted by distributed systems are telecommunications, automotive, avionics, industrial automation, robotics, consumer electronics, medical, and aerospace. The overall value of the embedded sector market worldwide is about 1600 billion € per year [2], a large part of which is attributable to distributed systems. The widespread deployment of distributed embedded systems is due to them being able to support important system properties, such as high reliability, scalability, high performance [3]. A single processor failure in a distributed embedded system can be avoided by distributing the system computing workload among multiple processors [4] [5]. For critical applications, such as spacecraft, control of nuclear reactors, etc., using a distributed system could increase the system reliability significantly.

In general, a distributed computing system is any computing system that involves multiple processors, remotely located from each other, where each processor plays a particular role in the execution of a computation or control problem. This type of distributed computing is referred to as *physically distributed* computing. An advanced form of distributed computing, which is referred to as *cooperative distributed* computing, involves collaboration among processors, in which an individual processor solves a part of a larger problem. Cooperative distributed computing is achieved via a distributed computing system that comprises a set of *p* processors connected via a network. A computing problem is then divided into *v* tasks, and each processor is

assigned a subset of *v* tasks [6, 7]. Cooperation among processors can make the distributed computing approach more efficient and reliable in comparison with just physically distributed computing units.

In this thesis a novel cooperative task-oriented fault-tolerant distributed computing (FTDC) architecture is proposed, which utilizes the advantages of cooperative distributed computing. Therefore, in the rest of the thesis the term "distributed computing" refers to "cooperative distributed computing". An example of a legacy centralized on-board computer (OBC) [8] on board the Sentinel-2 spacecraft is depicted in Figure 1.1. However, in high precision spacecraft control, a significant amount of data processing is required, which a centralised OBC is not able to support [9]. In addition, the physical redundancy scheme used in current spacecraft on-board computers is limited to a single processor failure, and its performance is constrained by the operating frequency that cannot be increased beyond a certain limit [10]. To address these and other requirements of modern spacecraft, the use of distributed computing is essential. The aim of this thesis is to propose a viable distributed OBC design, capable of achieving a significantly higher computing performance and reliability, to replace the traditional centralised OBC design, exemplified in Figure 1.1.

The intra-spacecraft application of distributed computing, outlined above, could be extended to serve the purpose of supporting emerging modern distributed space systems. For example, it can be employed in a constellation of spacecraft for applications requiring inter-spacecraft links, such as synthetic aperture radar (SAR), high precision spacecraft control, real-time optical imaging etc.. To illustrate this concept a SAR distributed space system is shown in Figure 1.2. Synthetic Aperture Radar is a well-known remote sensing technique that captures images of target objects on Earth using the motion of antenna to control image resolution. The SAR that is usually mounted on an aircraft or spacecraft, is referred to as monolithic SAR. The monolithic SAR is limited in terms of resolution. Contrary to the monolithic SAR, on-going research on distributed SAR proved that it is capable of producing a large synthetic aperture [11, 12]. To create such a large synthetic antenna aperture, a spacecraft cluster is used as shown in Figure 1.2. One of the satellites acts as a master, while the others act as slaves. The master spacecraft is equipped with a transmitting antenna for the transmission of the SAR signal, while the slave spacecraft are used to

receive and process the signal. Each spacecraft has its own processor for the execution of tasks. Each processor is assigned tasks accompanied with a chunk of data for processing. Cooperative distributed computing among these spacecraft is necessary, as the spacecraft cluster is required to work collaboratively to achieve an outcome. A distributed SAR imaging system, based on two satellites, where each satellite can acquire, divide and distribute data chunks to the other satellite for onward transmission to the ground station, is described in [13, 14]. Distributed computing in such a system will allow improving revisit time, image resolution and targeting viewing.

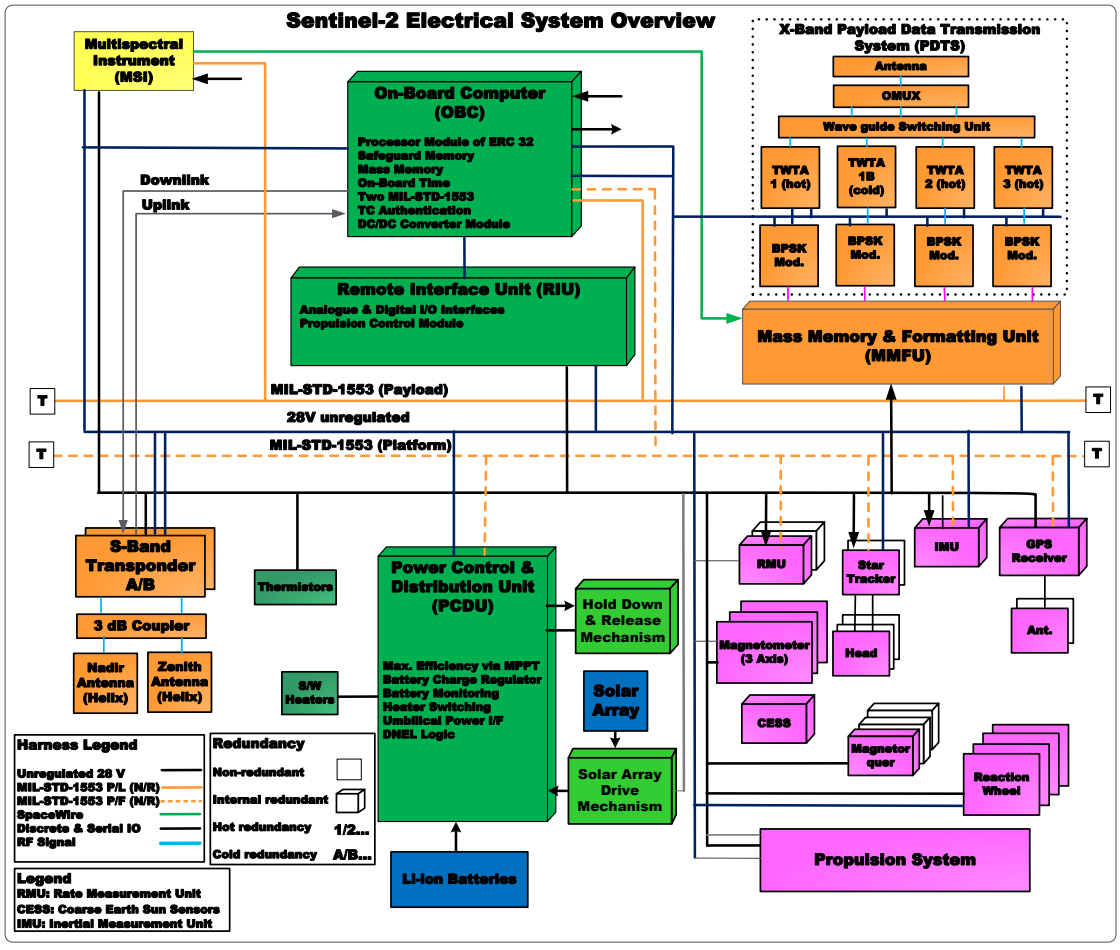


Figure 1.1: Block Diagram of the Sentinel-2 Centralized Computing System [8].

Other emerging space applications in which distributed computing is essential include space weather monitoring, fractionated spacecraft and distributed imaging. These applications are inherently distributed and must employ distributed computing among the spacecraft to achieve the overall mission objective. At present there are a

few technological challenges, particularly in the design of wireless inter-satellite communication links that need to be overcome in order to implement distributed computing among spacecraft.

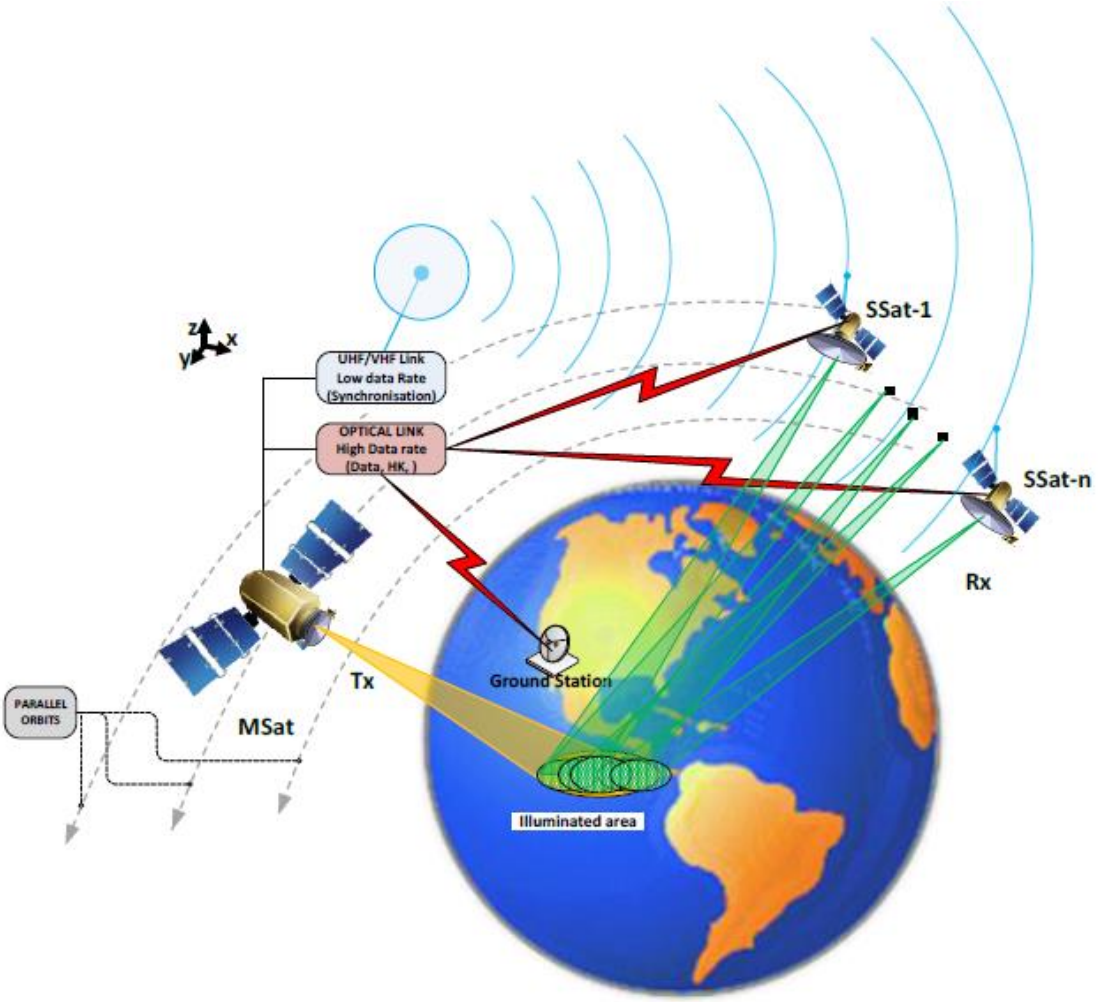


Figure 1.2: Distributed Synthetic Aperture Radar [12].

This thesis addresses space applications, however many other applications may directly benefit from the distributed computing approach. A number of terrestrial applications, such as autonomous cars [15] and distributed robots can also profit from this approach in comparison to a tightly coupled design [16]. Autonomous cars, for example, rely on a real-time sensor data processing and interpretation of complex control and navigation algorithms and in addition the underlying computing platform must support fault tolerance. The traditional standalone dual redundant embedded computing units (ECUs) are not sufficient to support such fault-tolerant data-driven

driverless operations. Therefore, a distributed computing system that provides enhanced performance and reliability is essential.

An on-board embedded distributed computing system has to operate under the influence of severe environmental conditions, which could cause a failure of a processor or a network. A main reason of a failure in space systems is radiation [17, 18]. Radiation can damage electronic components via total ionizing dose (TID) and single event effects (SEEs). TID is a slow phenomenon and can be overcome by suitable metallic shielding. SEEs, which are caused by high-energy particles, can instigate bit flipping that can lead to a temporary failure of a processor. Furthermore, physical damage by space debris, particularly in low earth orbit (LEO), is another cause of systems failures [19-21].

The efficiency of a computing application, running on a distributed computing platform is severally degraded in the presence of failures. A distributed computing system comprised of p processors can tolerate up to p-1 failures, and then the overall system performance would be equal to a uniprocessor system. Similarly, a 'p' processors system can achieve a p-fold increase in computational efficiency (speed-up) [6]. Thus both computational efficiency and fault-tolerance, cannot be achieved simultaneously. Thus, the challenge is to develop an efficient fault-tolerant technique that can achieve fault-tolerance by graceful degradation in computational efficiency. Distributed computing can support such a fault-tolerant technique, as it inherently consists of multiple processors that can either be utilized for high computational efficiency or fault-tolerance purpose. Failure related issues, which are intrinsic to distributed computing systems, are as follows:

- A crash of a processor can lead to a loss of all tasks that belong to it.
- A malicious failure of a processor can disguise itself, "giving a wrong impression" to the other processors that it operates in normal mode.
- After a processor restart, following a failure, it needs to be made aware of the state of the overall computation progress.

- To minimise the loss of efficiency, processors should not only perform their assigned tasks but they must be able to detect the occurrence of a failure of a processor in coordination with the other processors.

In this thesis, a novel approach to fault-tolerant distributed computing is presented. It provides high computational efficiency during normal operation and supports graceful degradation in case of failures. This approach is particularly designed for space applications but, in general, it can be applied to any distributed embedded computing application.

1.1 Motivation

The motivation behind this research is threefold:

- Demands for high-performance on-board processing in a single spacecraft mission, is continuously increasing and could not be met with by a standalone dual redundant processor [22]. By distributing a computational problem to multiple processors, these demands can easily be met.
- Traditional redundancy-based approaches could not be utilized in design of a fault-tolerant distributed computing system because in such approaches the peer redundant processor achieves 2:1 redundancy only. Distributed computing systems are inherently redundant, comprising multiple processors that can be intelligently used to achieve higher reliability.
- The same fault-tolerant distributed computing approach can be directly or indirectly applied to future multiple spacecraft missions [23].

1.2 Scope and Objectives

In this thesis a novel cooperative task-oriented fault-tolerant distributed computing (FTDC) architecture is proposed, which caters for high performance and reliability in systems on board spacecraft. In a nut shell, the architecture comprises two types of nodes, a computing node and an input-output node, interfaced together through a high-speed network with bus topology. To detect faults in the nodes, a fault management

scheme specifically designed to support the cooperative task-oriented distributed computing concept is proposed and employed, which is referred to as Adaptive Middleware for Fault-Tolerance (AMFT). AMFT is implemented as a separate hardware block and operates in parallel with the processing unit within the computing node. A set of metrics is designed and mathematical models of availability and reliability are developed, which are used to evaluate the proposed distributed computing architecture and fault management scheme.

The scope of the thesis is limited to a fault-tolerant technique for distributed computing employed in a single spacecraft only.

The main objectives of this research are:

- To assess existing approaches/methods/architectures for fault-tolerant distributed computing through investigation of the current state of the art.
- To propose a suitable fault management scheme for distributed computing on board spacecraft.
- To analyse, evaluate, test and demonstrate the proposed scheme through hardware implementation and a realistic space related case-study.

1.3 Methodology

We began our research by identifying existing distributed computing architectures in general, and fault tolerant schemes in particular. The architectures were evaluated in terms of their performance for reliability. Furthermore, shortcomings were identified, pros and cons compared, and finally it was observed that existing schemes were not efficient and suitable to meet our requisite performance and reliability. Therefore, a novel distributed fault-tolerant computing (FTDC) architecture, incorporating a new fault management scheme was designed, developed and finally implemented.

For validation of the proposed concept performance measuring metrics were identified and requirements were set that could serve as the appropriate criteria to assess operational success.

The proposed fault-tolerant distributed architecture was realised using system level hardware-software co-design principles and its performance was assessed by two

implementation approaches. Firstly, the distributed computing system was implemented and tested as a printed circuit board level design. Secondly, a novel MPSoC design was proposed, implemented and tested. Both distributed computing system implementations were evaluated for functional validity and performance.

To examine the FTDC architecture performance under a realistic space related distributed computing scenario a case-study application, representing a satellite Attitude and Orbit Control System (AOCS), was developed. The AOCS application was selected because it features a time critical task execution, in which system failure and reconfiguration time must be kept minimal. Based on the case-study application, it was demonstrated that the FTDC architecture is capable of fully meeting the desired requirements by timely migrating tasks to functional nodes and keeping rollback of task states minimal, which proves the advantages of the adopted cooperative distributed approach for use on board spacecraft.

1.4 Novelty Contributions

As a result of the research described in this thesis a novel concept for fault-tolerant distributed computing was developed and applied to a single spacecraft. The proposed scalable model is aimed at a single satellite subsystem as well as at the entire intra-satellite computing system. An assessment of the suitability of the approach to satellite on-board computing was carried out, which is accomplished for the first time. A demonstration of the fault-tolerant distributed computing concept was undertaken through a case-study aimed at the development of a new On-board Distributed Computer.

Specific novelty aspects of the work are as follows:

- A novel architecture for fault-tolerant distributed computing on board spacecraft is proposed, which is highly reliable and can provide high computing performance by running tasks concurrently on multiple nodes.
- A novel adaptive middleware design that is used for fault management of distributed system is proposed, designed and implemented.

- A novel MPSoC based approach to implement fault-tolerant distributed computing system is proposed, designed and implemented. An MPSoC based distributed computing system is designed, implemented and validated for the proposed fault management scheme.
- The proposed architecture and fault management scheme are validated by a case study of a new distributed design of satellite AOCS computer.
- Mathematical models for comparative evaluation of fault-tolerant computing system are developed.
- Novel algorithms for detection of silent data corruption for on board the spacecraft distributed computing are proposed and designed.
- Fault Injection mechanism, particularly suitable for assessment of distributed computing system is proposed, designed and implemented.

1.5 Thesis Structure

This thesis consists of nine chapters as shown in Figure 1.3. The structure of the thesis is carefully organized to show the complete picture from motivation to the final research outcome. The rest of thesis is divided into three parts – (1) background and related work, (2) research contribution of the thesis and (3) final conclusions and future work.

The next two chapters - Chapter 2 and 3 - present the background and research work done in the area of fault-tolerant computing and current applications. In particular, Chapter 2 discusses the basics of fault-tolerance techniques, fault detection methods, and communication protocols for fault-tolerant distributed computing in embedded systems. Chapter 3 gives a detailed overview of fault-tolerant computing for space applications. The current challenges and existing solutions are also discussed. It highlights the research gap and presents the research question that is addressed throughout the rest of the chapters.

The next five chapters (chapter 4-8) present the design, assessment and implementation of a novel reliable and efficient fault-tolerant distributed computing platform. Chapter 4 proposes a novel distributed computing architecture, where a

processor failure is resolved by migration of tasks to other processors. Chapter 5 presents a middleware design for fault management of the distributed computing platform. Fault management includes failure detection, failure coordination, and reconfiguration of distributed computing platform. This chapter includes algorithms, design and implementation details of the middleware.

In chapter 6, reliability and availability analysis of the proposed architecture and fault management scheme is presented. For reliability modeling, Markov models─ centralized, TMR, distributed system - were developed and compared. Following that fault management schemes - centralized and distributed - are analyzed and compared. Then functional verification is carried out by prototyping the fault-tolerant distributed system at a board level. Experimental results are reported and implementation issues are highlighted. Chapter 7 documents a Multiprocessor System-on-chip (MPSoC) based design and implementation of the distributed computing system. A separate dedicated hardware for the AMFT block and processing unit allows concurrent execution of the fault management functions and application tasks, thus achieving better reliability and performance. The outcome of this chapter is a reliable and efficient distributed computing system, particularly suitable for spacecraft on-board applications. Chapter 8 presents a case study of the satellite attitude and orbit control system (AOCS) distributed computer that is a most suitable example for the validation of proposed concept of distributed computing.

Chapter 9 summarizes the final results of this thesis. The research outcomes are assessed against the objectives. The key novelty contributions to the state-of-the-art are presented. Finally, future directions of the research are highlighted.

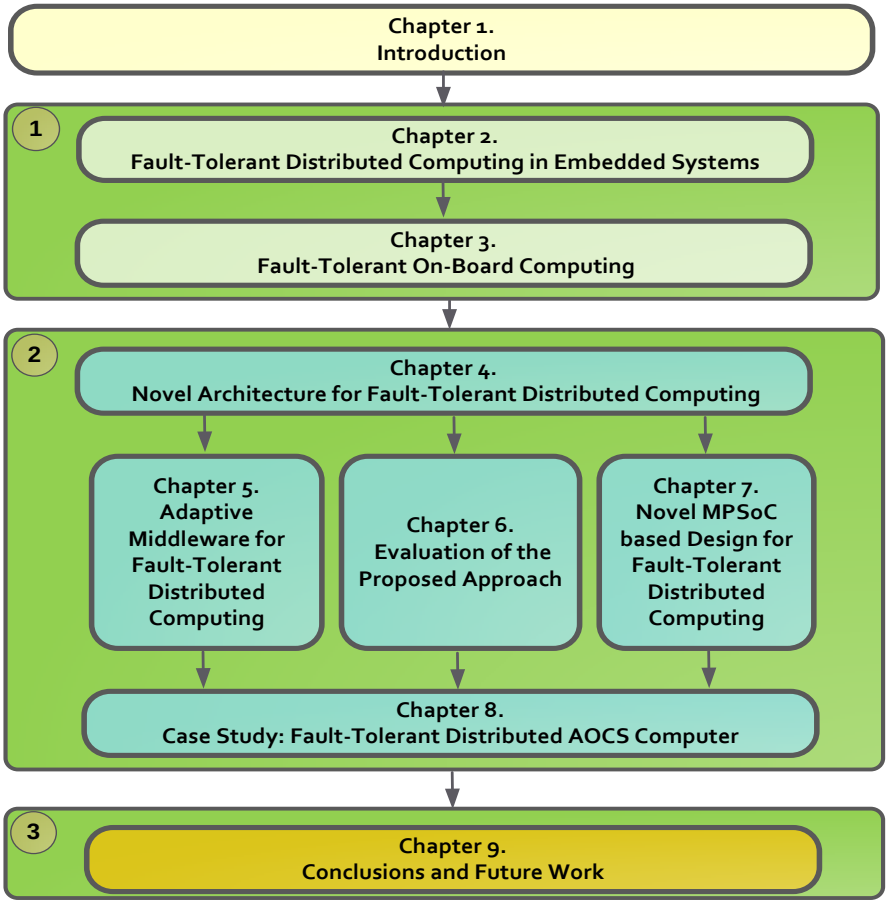


Figure 1.3: Thesis Organization

1.6 Publications

The results of this thesis were reported in the following publications, to each of which the author has made substantial contributions:

Conference Papers

1. M. Fayyaz, T. Vladimirova and J.M. Caujolle, "Adaptive Middleware Design for Satellite Fault-Tolerant Distributed Computing", in Proceedings of 7th ESA/NASA Adaptive Hardware and Systems Conference (AHS-2012), Nuremberg, Germany, 25-28 June 2012.
2. T. Vladimirova and M. Fayyaz, “Wireless Fault-Tolerant Distributed Architecture for Satellite Platform Computing”, in Lecture Notes in Computer Science, 2012, Volume 7425, pp. 428-436, Eds. G. Lee, D. Howard, and D. Ślęzak (Eds.), Springer-Verlag Berlin Heidelberg.

3. M. Fayyaz and T. Vladimirova, "Fault-Tolerant Distributed approach to satellite On-Board Computer Design," in Proceedings of IEEE Aerospace Conference, 2014, pp. 1-12.
4. M. Fayyaz and T. Vladimirova, "Detection of Silent Data Corruption in Fault-Tolerant Distributed Systems on Board Spacecraft," in Proceedings of 9th ESA/NASA Adaptive Hardware and Systems Conference (AHS-2014), 2014, pp. 202-209.

Journal Papers

1. T. Vladimirova and M. Fayyaz, "Fault-Tolerant Computing on Board Spacecraft using Distributed Multicore Processors, submitted to Acta Astronautica , Elsevier, 2015.
2. M. Fayyaz and T. Vladimirova, "Survey and Future Directions of Fault-Tolerant Distributed Computing on Board Spacecraft", submitted to Advances in Space Research, Elsevier, 2015.

Chapter 2

Fault-Tolerant Distributed Computing in Embedded Systems

This chapter presents a detailed review of fault-tolerant distributed computing in embedded systems. Section 2.1 covers the terminology and definitions used in fault-tolerant distributed computing. In section 2.2, a detailed review of fault-tolerance techniques employed in distributed computing systems is presented. Fault detection methods are covered in section 2.3. Issues related to communication among distributed nodes are covered in 2.4.

2.1 General Overview and Concept

2.1.1 Distributed Computing

Distributed Computing refers to any decentralizing of the computing power of a system. This means moving the centralized computing responsibilities away from a central location and distributing it between multiple locations, typically for some form of performance improvement or fault-tolerance purposes [24]. According to this definition, the computational problem is divided into tasks (processes) and is equally shared between the processors.

2.1.2 Distributed Embedded Systems

A distributed system is a collection of independent computing nodes, connected via a network, that appears to its users as a single coherent system [24]. A characteristic feature of a distributed system that distinguishes it from a centralized single computer system is the notion of partial failure. An important goal of distributed system is to recover from failures automatically without seriously affecting the overall performance. A distributed system should continue operations, even in the presence of a failure. A distributed system that provides this service is called a fault-tolerant distributed system. The same definition applies to a distributed embedded system. However, a distributed embedded system is a resource constrained specially designed system, usually placed inside or near the physical system that it controls or provides data to. It is constrained in terms of electrical power, computational performance, and physical size.

2.1.3 Fault-Tolerance

Fault-Tolerance is generally addressed via redundancy, i.e. providing backup resources that can be used in place of a failed resource. Fault-tolerance in a distributed system can be implemented at the architectural level, or at the node level. At the architectural level, failure of a node within a distributed system is masked by a redundant node. At this level, the failed node should display a simple failure mode (fail-stop). In the optimal case, a node exhibits only a fail-stop failure, i.e. the node is either operational or not. At the node level, the node implementation must ensure that the failure assumption that has been made at architectural level holds with a high probability [25].

2.1.4 Faults, Errors, and Failures

The terms '*fault'*, '*error'* and '*failure'* are extensively used in the context of fault-tolerant systems. A fault is a hardware or software defect that can lead to a system entering into an incorrect state. An error is a part of the system state which is liable to lead to system failure, while a failure is a state in which the system is restricted from

performing its required functions. When designing a fault-tolerant system, a designer makes some assumptions about the types of faults that must be handled. This is denoted as a system fault model. A fault model elaborates all the assumptions of a system failure. A designer will often design a system under the assumption that processors are failed in a fail-stop manner [26]. In general, the following types of faults are often considered [27].

- **Fail-Stop (Fail-Silent) Faults**: a processor stops producing outputs when it fails.
- **Byzantine (Malicious) Faults:** a processor sends erroneous output when it fails. Byzantine fault can be either symmetric or asymmetric.
    - **Fail Symmetric**: the fault results in the same erroneous value being sent to all other processors.
    - **Fail Asymmetric**: the fault results in different erroneous values being sent to other processors.

Faults may also be classified based on duration:

- **Transient Faults**: a processor fails and recovers after a short duration.
- **Permanent Faults:** a processor fails and disappears.
- **Intermittent Faults:** a processor fails and recovers sporadically.

2.1.5 Concept of Redundancy

Redundancy allows a computer system to work under the condition of faults or failures. A basic concept of redundancy is to provide alternative paths to allow the system to continue its operation, even in the presence of failures [28]. Redundancy can be implemented in either the time or spatial domain.

2.1.5.1 Time Redundancy

In time redundant systems (also called software redundancy), a software task is executed multiple times, consecutively to avoid temporary faults. It is used to detect transient faults in a software program [29].The disadvantages of this method include

performance loss and additional power consumption. Common examples of time redundancy are *N*-version programming [30], and redundant execution. In *N*-version programming, multiple versions of the same program are running sequentially to mask out a faulty version. In contrast, redundant execution can mask or detect transient faults by execution of the same program multiple times [31].

2.1.5.2 Spatial Redundancy

In spatial redundancy (also called hardware redundancy), physical spare resources are provided. Spatial redundant computers can be internal redundant, comprising internally redundant units [32], or externally redundant, comprising a set of two or more processors configured as dual modular redundant (DMR), triple modular redundant (TMR) or *N*-Modular redundant (NMR) configuration [33]. Spatial redundancy can be implemented as static redundancy, dynamic redundancy or hybrid redundancy.

Static redundancy relies on the fault masking approach. In this scheme, a set of multiple processors (e.g. triple or quad) are voted to mask single or double failures. Static redundancy is suitable for applications where maintenance during operation is impractical and is equally applicable to transient [34] and permanent faults [35]. In static redundancy [36], all processors are clock synchronized, processing the same input information and generating the same output data. The final output delivered to the target system, is derived from majority voting. In this scheme, it is assumed that no two processors can fail simultaneously. TMR computers are conceptually simple, but some issues arise in their implementation. These issues are due to the use of common circuits for clock synchronization circuit, voting, and common interfaces. A failure of common circuits in the TMR scheme can be catastrophic, leading to the failure of the whole computer. Therefore, these circuits have to be extremely reliable. Highly redundant implementations for these circuits can be used as adopted as in the Fault-Tolerant Multiprocessor (FTMP) [37] and Software Implemented Fault-Tolerant (SIFT) scheme [38]. An issue in TMR with repair computer (recoverable systems), which restricts its usage only to small duration missions, is the integration of a faulty processor after its successful recovery. There are two recovery techniques—rollback and forward recovery— which can be employed. Rollback Recovery is not a

preferable method for TMR computers, as it requires additional mechanisms for the saving and restoring of the execution context. [39]. Therefore, forward recovery to integrate with the other two running computers is employed. Forward Recovery is a difficult process, which requires all processor register values to be copied prior to reintegrating a recovered processor. Another disadvantage of the TMR redundancy scheme is its limited tolerance to a single processor failure. If one of the three processors fails, the fault-tolerance mechanism of the TMR computer is no longer effective. In order to overcome this problem, a hybrid redundancy is suggested [40, 41]. Hybrid redundancy includes the features of static and dynamic redundancy schemes. Hybrid redundancy is comparatively efficient in terms of fault coverage as compared to static and dynamic redundancy schemes; however it adds additional complexities in the computing system design to manage both types of redundancies.

In dynamic (standby) redundancy, a fault is first detected, and then a spare is substituted in its place. The following section explains the different types of dynamic redundancy schemes.

Simplex Processing with Backup Spare: This redundancy allows a program to run on a single processor while backup processors are available to take over the task load in case of a primary failure. Each processor has its own concurrent fault detection mechanism, which enables it to detect faults. Standby redundancy can be implemented as warm standby redundancy or cold standby redundancy.

In the case of warm standby redundancy, both the primary and redundant processors are powered [42]. Normally, only the primary processor executes tasks, while the redundant processor is in idle state. The downtime of the warm standby redundancy is considerably less because of its backup power up state. In cold standby redundancy, the redundant processor is placed in a power down state. In case of failure of a primary processor, the redundant processor is powered up. As the primary and redundant processors execution is not synchronized, a considerable amount of time is required for the redundant processor to reach a known state.

There are two main drawbacks of the standby redundancy. Firstly, a delay in switching the operation loses some of the computation, which gives a lower computational integrity. Secondly, due to single execution, it also has low fault

coverage. Nonetheless, it reduces electrical power consumption, and its simple design makes it favorable for long-life missions such as spacecraft and space rovers.

Duplicated Processing without Backup Spare: In the design of Dual Modular Redundant (DMR) computers, two processors run the same computation and their outputs are compared. On disagreement, diagnostics are executed to find the failed processor. To identify a faulty processor, it relies on a diagnostic test, which cannot ensure high computational integrity and fault coverage. A fault of an unknown nature can easily be missed, if it is not considered in the diagnostic test routine.

Later DMR designs include two self-checking processors. Both of them are running the same program. Each self-checking processor has its fault detection circuit, which can signal an error as soon as it appears. The outputs of the two processors are compared. On disagreement, the processor signals an error is ignored while the other completes its computation. This type of DMR computer has higher computational integrity due to its redundant execution. However, it has lower fault coverage because of the internal fault detection circuitry that may not detect all errors. In that case, an erroneous output could be delivered to the system.

2.1.5.3 Discussion

We conclude our discussion by comparing the two redundancy schemes as shown in Figure 2.1. The time redundancy approach is preferably employed for non-critical systems while spatial redundancy is employed for critical systems. The various forms of spatial redundancy have their advantages and disadvantages. In dynamic redundancy, all spare processors are operated in standby mode. Therefore, it requires less electrical power, which is very important for long-life applications. It does not require synchronization of the primary and spare processors. Design diversity between the primary and redundant processor is also possible. However, it uses Simplex processing, which provides poor fault coverage and computational integrity. On the other hand, static redundancy (TMR computer) provides better fault coverage and computational integrity but requires more electrical power and computational resources. Another drawback is when a single processor, out of three, fails, resulting in no further failure masking. Static redundancy is suitable for short duration applications (such as aircraft applications) that requires high fault coverage and able to bear high

electrical power consumption. The hybrid scheme, which utilizes the features of static and dynamic redundancy, includes spares to protect a system against more than one processor failure. However, designing such a system is significantly more complex. Its complex design enhances the design cost and introduces the possibility of more faults into the system [43-46]. All of the above redundancy based approaches can only protect a system against a single processor failure, and thus requires additional processors and support circuitry for enhancing reliability. Furthermore, these additions require an intelligent decision for integration/disintegration of a processor in existing systems.

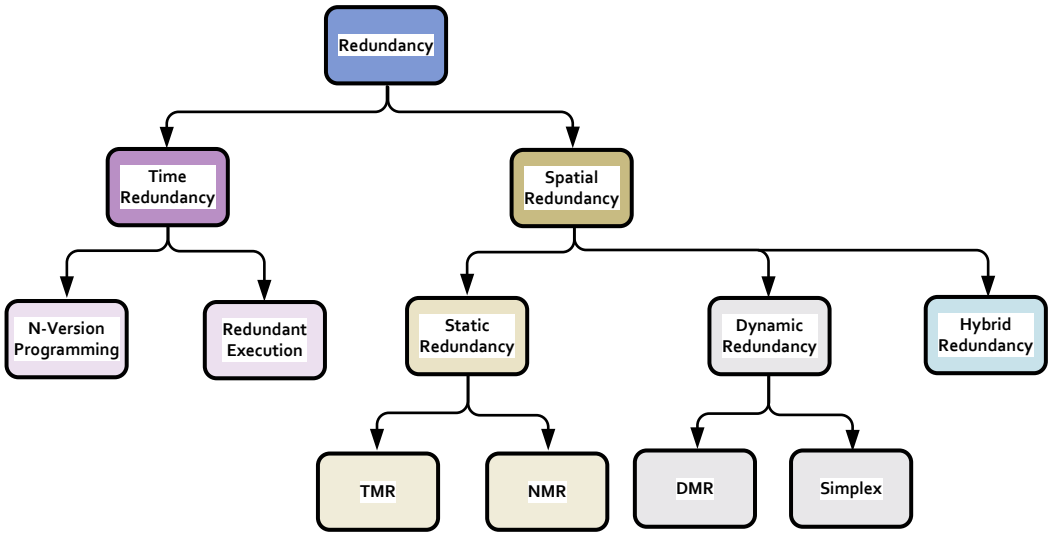


Figure 2.1: Redundancy Schemes.

2.2 Fault-Tolerance Techniques

In the previous section, we discussed that the redundancy based approaches are limited in terms of reliability. This section presents the various fault-tolerance approaches that utilize multiple processors for enhancing the reliability of computing systems. The main focus of this review is to analyse the techniques, particularly designed for reliable distributed embedded systems.

2.2.1 Replication

Replication involves information sharing to ensure data consistency between the redundant resources. These redundant resources can be hardware or software components. Replication is one of the most well-known solutions to fault-tolerance in distributed systems [47]. This technique is initially derived from highly reliable back-end servers where loss of service availability is very critical. Nowadays, it has widespread applications in distributed embedded systems. Replication has two main types called active and passive replication. The other variants are semi-active and semi-passive replications, which are derived from active and passive replications and retain their features, respectively.

2.2.1.1 Active Replication

In the active replication scheme [48], processes are replicated to multiple processors for fault-tolerance. The invocating process (client process) does not call a particular process. Instead, it addresses replicas as a process group. After sending a request to all replicas, the invocating process waits for a reply. If the replicas do not behave maliciously, then the invocating process can decide on the first reply. Otherwise, it waits for at least $k+1$ replicas in a k fault tolerant system. In the first case, the invocating process assumes fail-silent process failure while in the second case, it assumes Byzantine failure (behave maliciously when sick). The correct decision in the presence of a Byzantine failure is difficult, and various protocols are used [49-52]. To ensure the consistent replicas state, the totally-ordered multicast mechanism is used. This can be implemented using Lamport's logical clock that is suitable for small distributed systems. Most of the implementation techniques of active replicas [53, 54] assume partial synchrony of the underlying communication where the messages are communicated with certain time bound limits. In case of a large physical distributed system, partial synchrony cannot be achievable, and three-tier architecture is the only solution as reported in [55-57]. In three-tier architectures, instead of sending a request directly to replica's processes by the invocating process, an intermediate process is introduced for maintaining the consistency of replicas. The main advantages of the active replication scheme are its failure transparency and deterministic timing response

to the invocating process. However, it consumes more resources, as active replication requires processing at all nodes.

The practical example of the active replication approach is the design of Maintainable Real-Time System (Mars) [3]. Mars uses active redundancy for fault-tolerance whereby two or more components executes the same tasks. Communication between any two components is also protected against errors by sending messages twice. Components are self-checked and behave silently on the occurrence of a fault. This fail-stop feature restricts components to either sending correct message or no message. Mars components are arranged in a cluster. Communication between the different components is based upon the time division multiple access (TDMA) scheme.

Another example of active replication is the Delta-4 architecture [58], which consists of multiple computing nodes connected via a local area network (LAN). An individual node can be a uni-processor, a multiprocessor system or a specialized system comprised of array processors. Software components are replicated to multiple nodes to provide active redundancy against faults or failures. Each node has a network attachment controller (NAC) that provides services related to communication and message self-checking comparison. Also, the NAC provides multicast and fail-stop node operation.

Active replication is a useful scheme, which is employed in many applications. However, it has two main drawbacks: (1) it requires high resources due to redundant processing and (2) all requests have to be handled in a deterministic way. Furthermore, it requires voting among the replicas for systems in which byzantine failures can happen. [59, 60].

2.2.1.2 Passive Replication

In passive replication, also called primary-backup, only the primary node processes input messages and provides outputs. To make the replicas consistent, the internal state of the replicas is regularly updated from the primary replica. So, in the primary-backup scheme, contacting process communicates only with the primary node. If a primary node sends a reply immediately to the contacting process, it is called as a non-

blocking primary-backup scheme, while in a blocking primary-backup scheme, the primary waits for an acknowledgment from the backup nodes before sending a reply to the contacting process. In [61], trade-off analysis between the blocking and non-blocking scheme is presented. The response time for the non-blocking scheme is small in comparison to the non-blocking scheme. However, this is not always true, particularly in a scenario, when nodes are connected via a point-to-point communications network. In this case, a delay at the intermediate nodes causes an overall increase in the response time. Therefore, broadcast communication networks are preferred for achieving a small response time in non-blocking protocols.

In [62] a primary-backup scheme is proposed for real-time distributed systems, which unlike the active replication scheme, does not require a strong determinism. However, frequent state updates between the primary and the backups are necessary to achieve consistency among the replicas. To accomplish a timely response, a temporal consistency is suggested. Two objects or events are said to be temporally consistent with each other if their corresponding time stamps are within a predefined time interval. In a real-time primary-backup scheme the frequent state updates must be compliant with the predefined time bound of the application. In other words, a backup should have sufficient data information that can safely replace a failed primary node. Therefore, such a primary-backup scheme can be used in real-time distributed systems.

The major drawback of primary-backup replication is its slower response to failures. It is particularly the case when the primary replica crashes and a selection of a new primary is initiated.

2.2.1.3 Semi-Active Replication

In the semi-active replication scheme [63], which is also called “leader-follower”, only one replica, i.e. the leader, outputs messages, while the follower replicas perform the same computation autonomously as the leader does but do not produce output. However for the non-deterministic decision, they must follow the instructions from the leader replica, thus relaxing the requirement of determinism.

2.2.1.4 Semi-Passive Replication

A new style of replication, called semi-passive replication [60, 64, 65], is devised to overcome the slow response problem of passive replication. In semi-passive replication, the request is sent to all the replicas and only one replica processes the request. After processing the request, it generates a reply message for the client and an update message for the other replicas. Semi-passive replication is a variant of the passive replication and retains the characteristics of passive replication. However, it uses a rotating coordinator approach [66] for the selection of the new primary instead of group membership service. If the primary node crashes or a incorrectly suspected of having crashed, then the backup acts as the primary node.

Figure 2.2 shows the characteristic features of each of the replication techniques. The active replication technique provides a faster response to invocating process, but it requires a strong replica consistency and consumes more energy due to the execution of multiple replicas. On the other hand, passive replication requires less energy but it does not maintain a full consistency among the replicas. The semi-active replication scheme does not require a strong replica consistency, however, it consumes more energy. Similar to the passive replication scheme, the semi-passive replication scheme requires state messages for replica consistency. In addition, it has a lower response time to the invocating process as compared to the semi-active replication. To conclude, the replication based schemes are not efficient in terms of the utilization of resources, as they require excessive computing resources (processors, memory) for execution and maintenance of the replicas. This makes a replication based distributed system costly and inefficient for resource constraint applications.

2.2.2 Distributed Recovery Block

In the distributed recovery technique [67, 68], two copies of the same program are executing simultaneously on the processors of a node pair. A node pair is a set of dual redundant operational nodes. In a node pair, one node is active, called an operational node, while the other node is inactive, and called the shadow node. Under normal conditions, the active operational node executes a primary version of the tasks while the shadow node executes an alternative version of the same tasks. On each node,

correctness of the result is checked by an acceptance test. If the acceptance test is passed, executive layer routine outputs the results of the primary routine. On failure of the acceptance test inside the primary node, the shadow node is informed either by the primary node executive layer or by the shadow node time-out value (for the case where the primary node fails silently). In that case, the shadow node becomes an active node and sends output results. Similar to the redundancy based approaches, discussed in section 2.1.5, this technique has limited reliability and can only protect a computing system against a single processor failure.

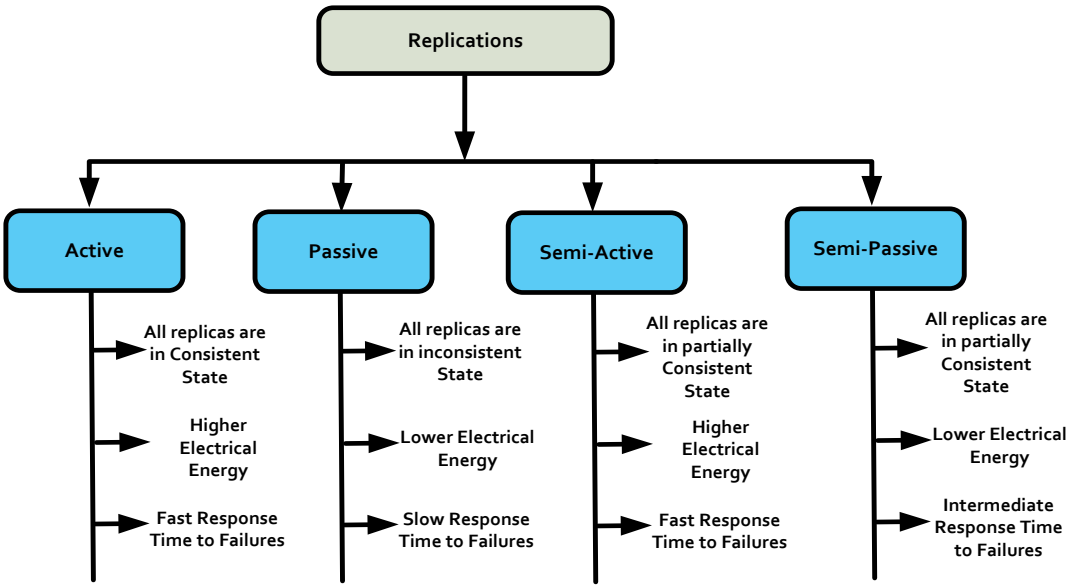


Figure 2.2: Comparison of Replication Techniques in Distributed Systems.

2.2.3 Redundant Execution

Redundant execution (also called time redundancy) is another fault-tolerance technique for distributed embedded systems [69]. Redundant execution can be done at the instruction level or the task level. At instruction level [70], each instruction of the executing program is duplicated and, after each duplicated instructions, results are compared for errors. On the other hand, at task level redundant execution, a software task is executed twice or more in time to avoid temporary faults. Contrary to replications, it does not require additional hardware to run the redundant copy. Instead, it uses extra time to do redundant execution of the same program. As the primary and

redundant execution of a program running on a similar hardware can only protect the system against the transient faults.

A single program executing multiple times reduces the overall computing performance. Additionally, it also consumes more electrical power. However, a recent form of redundant execution called multiplexed redundant execution, as suggested in [71] overcomes this problem to some extent. The basic scheme is the same as redundant implementation, but it uses chip multiprocessor (CMP) for the execution of leading and trailing threads.

2.2.4 Network Surveillance

Network Surveillance is a technique that uses a communication network for the fault detection and configuration of distributed components [72]. In its simplest form, network surveillance includes a master that periodically calls other nodes for the detection of a failure. The master confirms a node failure if it does not receive a reply message. In order to avoid a single point of failure, K. H. Kim and E. Shokri propose a decentralized approach, called periodic reception history broadcast (PRHB) [73]. In PRHB, each node broadcasts a periodic reception history, which is gathered during the last two TDMA cycles, that includes the health status of the available nodes. Disadvantage of the PRHB scheme is the large reconfiguration time of up to two TDMA cycles, which causes the scheme less responsive in case of a node failure. Both network surveillance schemes ─ simple master/slave and PRHB ─ are used for broadcast networks only. K.H. Kim and C. Subbaraman propose a scheme called supervisory-based network surveillance (SNS) for point-to-point networks [74]. This scheme utilizes two types of nodes; worker nodes and supervisor nodes. Worker nodes pass the health statuses of its neighbour nodes to a supervisor node, which sends status messages to all the other nodes in the network. In this scheme, each node has complete health information of all the other nodes irrespective of the availability of a direct connection. There are two main problems with this scheme: firstly, it requires election in case of a supervisor node failure, which can take a considerable time; secondly, messages traverse the network via several links in a store and forward fashion, resulting in an additional processing overhead on each node and adding extra fault sources.

2.3 Fault Detection Methods for Embedded Distributed Computing Systems

In fault-tolerant computing systems, fault detection is a very important element of the overall tolerance process. The fault-tolerant process cannot start unless a fault is detected. In distributed system, before applying prevention to tolerant a processor failure, a faulty processor must be identified. A fault in a processor can be detected by either hardware or software based detection methods. Hardware based fault detection requires extra hardware to detect faults and might not be feasible for distributed system comprising of the several processors. Therefore, software-based fault detection is usually preferable. The following section will critically review both methods from the point of fault detection of a processor in a distributed system.

2.3.1 Hardware Based Fault Detection

A comparison of two or more than two hardware computing elements is one such fault detection that relies on physical redundancy. In its simplest form, two elements are compared by (Exclusive OR) XOR operation. A mismatch of any of these indicates a fault that requires further diagnosis to find out the exact faulty element.

Monitoring for fault detection is used as an alternative to physical redundancy [75-81]. In monitoring, a separate hardware module called monitor is used to detect faults in the actual computing modules. A similar approach is employed in the Self-Testing and Repairing (STAR) computer that was developed by the Jet Propulsion Laboratory (JPL) [82]. This computer consists of multiple redundant units, connected via a 4-wire internal bus that is monitored by a special unit called Test and Repair Processor (TARP). The TARP is connected to the internal bus and uses error-detecting codes and status messages for fault detection of the computing units. On detection of a fault, it first rolls back the program and, if a problem persists, it replaces the faulty unit with a spare.

Watchdog Timers (WDT) are an additional monitoring based fault detection method that is widely used in embedded systems. A processor of an embedded system can go into an undefined state if an error appears in the program flow. The WDT

monitors a processor for a pre-defined timeout value. If there is no received signal from a processor during the timeout period, it signals the processor to go into its initial reset state or to run a processor in diagnostic mode. In some cases, it may switch to a redundant system. The WDT method is simple, but it is very obvious that malfunctions can happen, even if the processor generates the right timing signal for the watchdog timer [83]. During this situation, the simple watchdog timer is not very helpful into detecting system failures. In [84], a watchdog processor is dedicated to detect faults in the main processor. It splits the fault-detection process into a setup and a checking phase. In the setup phase, checked values for the detection of faults in the main processor are provided while, in the checking phase, the watchdog processor monitors the main processor against the checked values. These checked values can be control flow information, memory access behaviour, or the reasonableness of the results. In control flow monitoring, watchdog processor has to check signatures values and their associated relationship. During the program execution of the main processor, the watchdog processor computes the signature and compares it with a concurrently provided signature. It indicates an error if the two signatures are mismatched. In [85], authors propose a watchdog processor for a memory access behaviour that is called capability checking. In that case, the watchdog processor checks the memory accessibility at the processor/memory interface using physical addresses. If an illegal access to the memory is detected, the processor is informed, and a recovery process is initiated. In conclusion, watchdog based methods provide a basic level of fault detection and are limited to timing violation and illegal access. Therefore, the WDT based fault detection is preferably used in conjunction with other methods.

2.3.2 Software Based Fault Detection

Assertion is a basic technique for software fault detection, where a programmer inserts a small hand written code called an assertion for checking of the original program [86-90]. If a subprogram passes the acceptance test written as an assertion, it proceeds to the next subprogram. Otherwise, failure of the acceptance test is an indication of a fault. Assertions act as a barrier between the two sub-programs.

Unlike assertions, a separate non-distributed [91] and distributed monitor [92] is an alternative way of software-based monitoring. These software-based fault detectors as

demonstrated in [93, 94], they consume much higher resources than the execution of the task itself to support high-coverage of faults. The implementation of these methods, such as software-based monitors [91, 92] and software-based detectors [93, 94], sometimes consumes more resources than the application itself.

Another technique is called off-line periodic test. In which, the normal operation of the system is temporally suspended and a diagnostic program is used to monitor the system health. Contrary to off-line periodic checking, Software-based Self-Test (SBST), as suggested in [95], provides an online mechanism to self-test a system. In this case, a separate task is assigned to run with an actual program. This task periodically executes and its period controls the detection latency. For smaller values of the period, the detection latency is low.

Symptom-Based Fault Detection: A symptom is a departure from the normal function or execution of a computer system, indicating the presence of a fault [96]. Symptom-based anomaly detection techniques focus on monitoring key parameters within the software that can indicate abnormal behaviour due to either hardware or software faults. These key parameters are evaluated during runtime, maximizing the effectiveness while keeping the overhead to a minimum.

Man Li presented, in [97-99] a symptom-based anomaly detection system, named software anomaly treatment (SWAT). The system performs hardware and software anomaly detection by observing software behaviour. It employs four abnormal behaviours as symptoms of an anomaly: (i) fatal traps, (ii) abnormal application exits, (iii) hangs, (iv) high contiguous Operating System (OS) activities. The fatal trap is a common fault that is usually detected by the built-in detection mechanism of a commercial off-the-shelf (COTS) processor. Fatal traps detect faults like divide-by-zero, out of bound memory access, misaligned memory access, illegal instructions and watchdog expiration. Abnormal application exits are errors, which are not detected by hardware but are visible to the OS. The third abnormal behaviour symptom, hangs, is detected through a heuristic approach to monitoring all the executed branches in the application and OS. The fourth symptom is related to the time spent in executing the OS. If the execution takes longer than a certain number of instructions in the OS, then it is considered as a symptom of anomalous behaviour. The last two symptoms are effective but prone to false positives as their detection is based on heuristic

approaches. A similar fault detection method for multicore systems, named SymptomTM, is proposed by Gulay in [100]. This method uses transactional memory to isolate faults by first writing the results into a local hardware transaction memory and monitoring symptoms such as fatal traps. If there is a fatal trap, then the transaction is executed again, and if the fault persists, then the transaction is executed on another processor core. If the software runs correctly on the second core, then the first core is marked as damaged. If the transaction fails on the second core, this indicates a software error. Thus, faults are isolated, and a system can tolerate more latency, however, only faults that cause fatal traps are targeted.

Detection of conditional branching anomalies is addressed in a symptom-based soft error detection scheme named ReStore [101]. In this scheme, a checkpoint is created after every 10 to 1000 instructions and in case of soft errors, a rollback is executed. In case of a false positive the effect is a slight performance loss due to the repeated execution from the previous checkpoint. The proposed method makes use of built-in pipeline branch predictors. These predictors are highly accurate and can achieve up to 95 % accuracy [101]. To further reduce the overhead caused by the false positives, a confidence level indicator in the predictors is also utilized by setting a confidence threshold. Cache misses are normal behaviours; however, they can also be used for detection of anomaly symptoms. The existing hardware used for anomaly detection in [101] results in a large number of false positives, since it is not designed or optimized for this purpose.

Statistical anomaly detection schemes are suggested in [102, 103]. Finite state automata is used to define states of the software programs as runtime events. These states include program start, procedure start, loop start, compound statement start, program end, procedure end, loop end and compound statement end. In the training mode, the transition frequency of each event is calculated as a baseline. For example, assume that the frequency of a certain loop execution cycle is calculated during the test run, and its mean and variance values are x and y, respectively. If, during the software execution, the x or y value exceeds a certain threshold then, it will be considered as a symptom of an anomaly. This method can be effective. However, it requires dividing the program into states and an accurate calculation of the statistical parameters during test runs. Ensuring correspondence between the test runs and the

actual operation is critical to the success of the detection scheme. Also, dynamic operation environments can cause a large number of false positives.

Correctly detecting software crashes and hangs is very important for high availability systems. Nakka proposed three fault detection techniques to detect software crashes and hangs [104]. An Instruction Count Heartbeat (ICH) signal detects abnormal process termination and hangs. It monitors whether the processor executing the instructions has the right context by finding a particular fixed number of instructions in a fixed time. This functionality is already present in modern high-end processors. An Infinite Loop Hang Detector (ILHD) module detects hangs due to infinite execution in a loop. A compiler is used to instrument the entry and exit points in each loop with a different timeout for each loop. A Sequence Code Hang Detector (SCHD) module detects the infinite loop hangs due to illegal loops by maintaining a log of recent instructions and looking up the same instruction sequence. In this way, repeated instruction can be detected. However, the implementation of this technique on embedded processors requires additional hardware modules.

2.4 Communication Network

In fault-tolerant distributed computing, selecting the right communication network and protocol are essential. In general, the networks for critical systems have to meet the following broad functional requirements: fault-tolerant operation, determinism and reliable data delivery. In addition to these minimal requirements, features such as high-speed, multi-master, and power consumption are also important. Fault-tolerance is the most important feature for a mission critical distributed system. A single network failure ─babbling, network partitioning─ can be catastrophic for the entire communication process and result in the loss of the whole mission. Errors in a network such as an invalid message, a non-responsive message, and node conflicts are usually protected. The following section discusses these aspects of network communication protocols in the light of fault-tolerant distributed computing.

MIL-STD-1553 is a widely used protocol configured in bus topology and was initially developed for avionics systems. In MIL-STD-1553, three types of messages ─command, data, and status─ are used. A word is a smallest entity in a message and

contains 20 bits. Each transmitted word is protected by a parity bit for the detection of invalid messages. In addition to the parity bit, each word is also protected by a 3-bit sync pattern. The pattern for command and status words is identical, while a separate inverted sync pattern is used for data words. A failure of a non-responsive node is handled by the status messages. MIL-STD-1553 is a dual redundant bus, but during normal operation, data is only transmitted on one bus while the other bus is kept as a hot backup. The Bus Controller (BC) can use the other redundant bus when a babbling node on the primary bus prevents normal communication. In this case, the BC would send a stopping command for shutting down the babbling node's transmitter. The redundant bus can also be used for the normal communication in case of physical damage to the primary bus. In MIL-STD-1553, all the data movement is controlled by the BC, which ensures deterministic and real-time bus access. Although, MIL-STD-1553 proves it heritage in many avionics and space projects, but it has limited speed (up to 1 Mbps). Thus, it cannot meet the high-speed requirements of current and future applications. Although further researched [105, 106], no considerable performance in terms of speed is achieved. Another limitation of the MIL-STD-1553 bus is that it only supports master/slave communication model for distributed computing.

Controller Area Network (CAN) is an event-triggered network, originally developed for the automotive control applications. However, CAN and its variants have also been developed for other applications [107-109]. CAN is a multi-master, prioritized, short messaging, and medium speed data network. Reliability of a communication network is validated through its bit error rate, fault localization, and immunity to radiation. In the case of CAN, a 15 bit CRC and frame format/size checking is used for data integrity at the message layer. At the physical layer, two mechanisms─ bus monitoring and bit stuffing─ are implemented for the detection of errors. Each transmitter checks the transmitted signals on the network to ensure reliable communication. The CAN network is capable of switching-off a node if a node sends erroneous messages on the network greater than the pre-defined limit. The CAN communication protocol cannot, however, switch-off a babbling node while it is transmitting correct messages. While a CAN network is operated in a dual redundant bus topology, there is no provision in the protocol to switch to the redundant bus. Also, the CAN protocol uses priority-driven Carrier Sense Multiple Access/Collision

Detection (CSMA/CD) medium access scheme, so deterministic behaviour cannot be guaranteed. A modified CAN protocol that meets the real-time deadline of critical systems is a Time-Triggered Controller Area Network (TTCAN) [110]. TTCAN is highly reliable and suitable for critical applications [111]. It divides the bus access into multiple slots that guarantee real-time data delivery. TTCAN allows a node to write on the bus at a particular time. Therefore, it is relatively easy to detect the existence of babbling nodes by watching the relevant time slots [112].

Nowadays, Ethernet is rapidly emerging in embedded computing due to its widespread availability and cheaper cost [113]. Ethernet uses an event triggered Carrier Sense Multiple Access/Collision Detection (CSMA/CD) scheme for medium access, in which arbitration is based on a back-off mechanism. On contention, each node waits for a random amount of time and then attempt to re-transmit on the network. Due to this back-off mechanism for arbitration, the timeline for communication cannot be guaranteed. To ensure a timeline, TTTech suggested a new Ethernet protocol called the Time-Triggered Ethernet (TTEthernet) [114]. This protocol combines the event and time-triggered scheme to support rate constraints, best-effort and real-time traffics. Also, it uses redundant path, switches, and end systems to ensure fault-tolerant operation of the network. It concludes that a failure of a single node or messages in a network can be tolerated without affecting the application. Also, each node and the network switch are protected by guardians that ensure the communication compliance within the TTEthernet network, according to predefined parameters.

SpaceWire is a point-to-point standard that was developed to provide high data rate communication for on board space systems. Under this system, mass storage units, processing units and subsystems are interconnected via a SpaceWire router that allows multiple devices to communicate simultaneously. SpaceWire uses Low Voltage Differential Signalling (LVDS), which consumes very low electrical power at very high speed [115]. LVDS isolates its physical interface to avoid damages during a short circuit condition. SpaceWire supports group adaptive routing for isolating a failure link. The communication over SpaceWire is non-deterministic. Therefore a SpaceWireRT [116, 117] was proposed. SpaceWireRT allows a Quality of Service (QoS) layer over SpaceWire to support deterministic, reliable real-time data delivery.

Although the original SpaceWire protocol is non-deterministic, due to its simplicity, adaptable topology and high-speed communication, it has been used on many NASA and ESA missions.

Few other COTS communication protocols, such as I2C and IEEE-1394, are also suggested for network bus-based communication for various applications [118, 119]. To make these communications protocols fault-tolerant, necessary modifications are adopted in the X2000 program. However, due to lack of network layer support, these cannot be appropriate options for future scalable missions. Also, the limited speed of the I2C bus restricts its use for CubeSats and other similar space applications.

A network can be configured in different topologies─ such as bus, line, star, ring, mesh and point-to-point. The mesh topology has the best redundancy, even when a link is down. However, it becomes very complex as the number of nodes increases. Star is centralized and has a single point of failure; therefore it is not at all considered. The ring topology is more complex than the bus, and most of the communication protocols support a bus topology. Therefore, it is a more appropriate option for broadcast/multicast communication in fault-tolerant distributed computing systems.

To support current and future distributed applications, key features for each protocol are tabulated in Table 2.1. It includes features of fault-tolerance, high data rate, scalable topology, real-time and reliable data delivery, and multi-master support for comparison. This comparison shows that SpaceWireRT and TTEthernet are the two appropriate options for supporting all these features. However, TTCAN can also be used for low data rate applications.

Table 2.1: Comparison of Wired Communication Protocol

Parameters	***TTCAN Bus***	***MIL-STD-1553/1773***	***SpaceWireRT***	***TTEthernet***
Max. Speed	1 Mbps	1 Mbps	400 Mbps	10/100 Mbps
Power Consumption	0.75 W (COTs)/ 1W (RadCAN)	High	0.5 W	> 1 W
Topology	Bus	Bus	Point-to-point/Network	Bus/Network
Architecture	Multi-master	Master-slave	Any	Any

Max. Data/Packet	8 bytes	64 bytes	Variable	Variable
Real-Time Delivery	Deterministic	Deterministic	Deterministic	Deterministic
Fault-Tolerance	Good	Best	Good	Good
Scalability	Not	Not	Yes	Yes

2.5 Summary

Replication allocates the same tasks to multiple physical nodes. The disadvantage of this is the underutilization of computing resources for the sake of achieving a higher reliability. The techniques reviewed in the literature are not well suited for adapting to other applications.

One important aspect of the active replication-based system is that it uses the method of majority voting, in which a consensus is carried out to ascertain the most reliable outcome amongst each other. However, if one of the nodes fails (either through functional failure or damage), consensus cannot be established, which is a particular limitation only in a three node system. A consensus based system also suffers from a large inter-node communications overhead, which also demands higher processing power, indirectly consuming more power, and demanding a fast communication network. Therefore, fault-tolerance by replication can be suitable for a general purpose system, but not for a resource-constrained embedded distributed system.

Fault detection is an important element of fault-tolerant distributed computing, which is carried out commonly at two levels, hardware and software, both of which have been discussed. It is evident that both the hardware and the software level fault detection methods are imminent in a fault-tolerant computing system. It is observed that software-based fault detection is commonly used, since it utilizes fewer resources (and resources are scarce in embedded systems).

Distributed computing system requires some means of communication amongst its peer computing nodes. Communications protocols in light of distributed computing systems requirements were reviewed. The focus of this review was to analyse

protocols from three aspects— fault-tolerance, real-time and reliable data delivery— for the employment in embedded distributed computing systems.

In conclusion, replication is a tried and tested technique, which is widely used in distributed computing systems. However, emerging demands for high performance distributed computing require a fault-tolerance technique that utilizes the inherent availability of multiple processing units.

Chapter 3

Fault-Tolerant On-Board Computing

This chapter reviews the existing fault-tolerant computing schemes in space applications. In section 3.1, the general concept of space fault-tolerant computing is introduced. State-of-the-art fault-tolerant mechanisms employed in existing spacecraft and aircraft are surveyed in section 3.2, covering computing models, fault management schemes, and existing fault-tolerant distributed systems. In section 3.3, the importance and suitability of commercial off-the-shelf (COTS) wireless protocols are discussed in terms of spacecraft distributed computing. A brief overview of current non-distributed fault-tolerant systems, in terms of high-reliability and high-performance demands, is discussed in section 3.4. The main issues and research gaps in the existing fault-tolerant schemes are highlighted in section 3.5. A problem statement consisting of a main definition, fault model and performance metrics is defined in section 3.6.

3.1 Related Definitions

3.1.1 Spacecraft

Spacecraft is a vehicle that is designed to fly in outer space. It has the capability to travel in the free space while satellites are intended to orbit a planet. Both of these are designed to provide a particular service. Both are used for a variety of purposes

including communication, navigation, Earth observation, planetary exploration, transportation of humans and cargo. The internal functionality of each spacecraft/satellite is divided into platform (also called Bus System) and payload systems. The payload system is used to provide the intended service while the platform system is for the support of the payload system [120]. The spacecraft platform system consists of many subsystems including structure and mechanisms, thermal, attitude and orbit control (AOC), propulsion, power, telemetry and command, and on-board data handling (OBDH). Although, all subsystems of a spacecraft are important, the on-board data handling, AOC and the payload itself are very important from a computational point of view. All three are responsible carrying out computationally intensive tasks and huge data processing in the presence of severe environmental conditions.

3.1.2 On-Board Computer

The On-Board Computers (OBCs) of a spacecraft provide computational service to platform data processing, vehicle control (AOC) and payload data processing. These functions are very critical, so OBCs must be able to withstand the effects of thermal, mechanical, high energetic particle radiation and other environmental hazards. These implications significantly influence the design of OBCs and make them largely different from the traditional computers used on Earth [121].

3.1.3 Computer, Node, Unit, and Module

Throughout this chapter, four terms are often used to represents the physical entities of fault-tolerant computing systems. These terms are defined as:

- A ‘computing node’ is viewed as consisting of a processor connected to some network or communications medium.
- A ‘unit’ represents a computing entity within a node. According to this definition, a computing node can include multiple units.
- A ‘module’ can or cannot be a part of the computing unit. Each module is required to do some useful functions.

- A ‘computer’ consists of one or multiple nodes that are attached to each other by a communication network. This same definition applies to the fault-tolerant computer comprised of primary and redundant nodes.

3.2 Fault-Tolerant Computing for Aerospace Applications

This section reviews fault-tolerant computing from an aerospace application point of view. Existing fault-tolerant mechanisms are surveyed to get an idea of the current state of the art, covering computing models as well as fault management level schemes. Furthermore, a detailed design overview of fault-tolerant systems is also presented.

3.2.1 Computing Models

This section will present how the primary and redundant nodes of a fault-tolerant computer are connected to form a fault-tolerant computing architecture. There are mainly three architectures, which are explained in the following section.

3.2.1.1 Centralized Model

In a centralized computing model, all computing functions are executed on a single computer. This computer is internally redundant and usually designed using module-level redundancy [32]. In this computing model, only one module is active at any time, which can be replaced by the redundant module in case of a fault. For fault detection ─ arithmetic codes, product, and residue ─ are used. The reconfiguration module, which knows the health state of each module via a fault status signal, itself is triplicated. The reliability of the centralized computer depends on the failure rate of its components, whereas its computational performance depends on the processor’s operating frequency.

3.2.1.2 Cross-Strapped Model

The cross-strapping model is a widely used in on-board computing systems [122]. It is employed between two similar pairs of nodes or units as shown in Figure 3.1. Cross-

strapping is not dealt with at the internal node level of redundancy. It is an approach to provide a dual redundant communication path to ensure system reliability. In cross-strapping, multiple functional components/modules are connected in series to form a string or chain to achieve overall functionality. Components of one string are cross-strapped with the other string to build a cross-strapped computing system. This scheme is better in terms of failure isolation because it can isolate a unit without disabling the other units in a string. It is different from the non-cross-strapped case, where a failure of one computing unit will disable all the other units in the same string. However, the cross-strapped scheme has poor failure containment and fault detection, isolation and recovery (FDIR) testability [123]. A single failure may affect all the cross-strapped strings. In terms of FDIR testability, it is very difficult to test the overall cross-strapped computing system. Furthermore, each cross-strapped unit requires fault detection algorithm to isolate a particular unit.

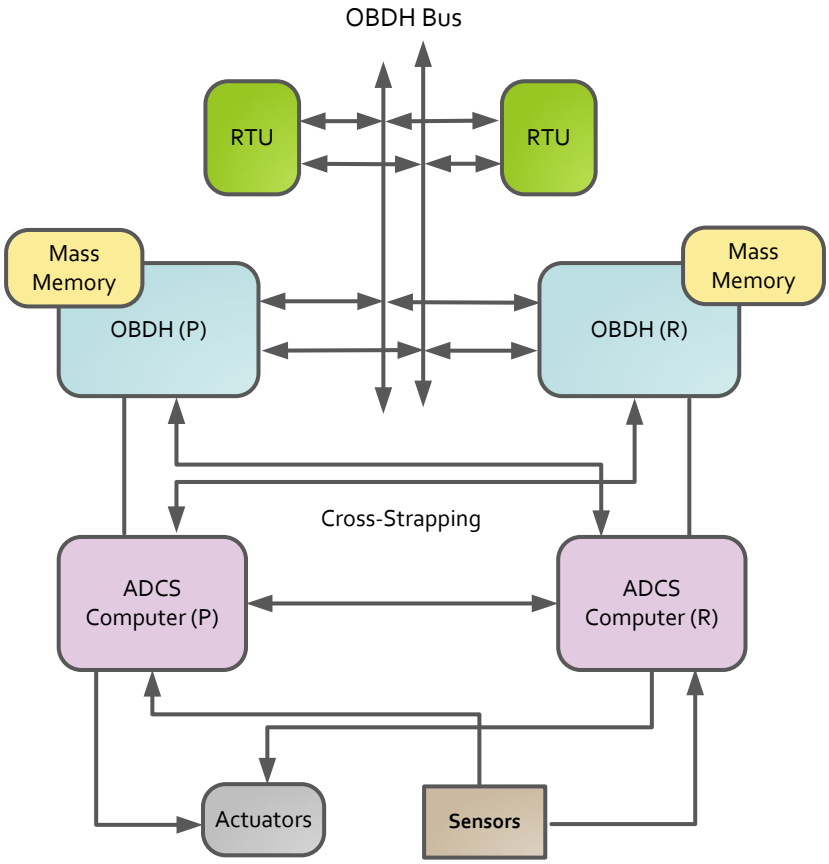


Figure 3.1: Cross-Strapped Satellite Platform Computing Model.

3.2.1.3 Distributed Model

A distributed model takes advantage of the underlying network. The three distributed computing models─ client-server, master-slave [124] and node pair [67]─ can be used for the implementation of fault-tolerant distributed systems. The client-server model is mostly used for general purpose fault-tolerant computing, where there is no need of absolute time bound, and enough computing resources are available. On the other hand, the master-slave design is widely employed in embedded fault-tolerant system. The main advantage of the master-slave model over the client-server model is a zero connection setup time. In the node pair model, a computing system consists of a single supervisor node and multiple operational nodes operating in a node pair configuration. A node pair is a set of dual redundant nodes. In a set pair, one operational node is active while its peer node is in a shadow mode, which operates on a failure of a primary.

3.2.2 Fault Management Scheme

Often a fault management scheme is identified using different terms, such as health management, fault detection, isolation, and recovery (FDIR), and redundancy management. In this thesis, we will use the term fault management scheme to represent all these terms. In general, the fault management scheme would apply to all parts of a spacecraft but here fault management at the architectural level to cater for a failure of a computing node is discussed. A computing node's failure can be handled by a manual or an autonomous fault management scheme. In the manual fault management scheme, a ground command is required for activation of the redundant node. The disadvantage of the manual fault management scheme is its long response time because of the long delay encountered during the communication and operator intervention. While an autonomous fault management scheme eliminates this delay by making local decisions on board. These decisions can be made by hard-coded or by table-driven algorithms. Hard-coded algorithms are coded as an integral part of the flight software and are verified in the same way as the flight code. Table-driven algorithms use a database for defining and monitoring parameter and, their failure pre-set thresholds. The table driven approach allows easier modification, however, full testing and verification are still required for each database change. There are four main

schemes based on the location of the algorithms: half satellite, centralized, decentralized, and hierarchical, which can be used for fault management of a computing system at the architectural level.

3.2.2.1 Half Satellite Fault Management Scheme

The half satellite fault management scheme is a simple form of fault management [125]. This scheme comprises of primary and redundant computing chains. The primary chain includes a set of primary computing nodes, while the redundant chain includes a set of redundant computing nodes. The scheme is not able to make decisions on isolation and reconfiguration of an individual computing subsystem. In case of a fault, it simply switches to the redundant computing chain irrespective of the remaining healthy computing nodes. Later on, the fault is analysed on ground, and possible commands for a particular configuration are initiated. This scheme is not a suitable option for current and future space missions, which are geared towards on-board autonomy.

3.2.2.2 Centralized Fault Management Scheme

In case of a centralized fault management (FM) scheme [126], all the functions related to fault detection, isolation and reconfiguration of a system are located on a single computing subsystem called on-board data handling (OBDH), as shown in Figure 3.2. This scheme is simpler in terms of the implementation of the fault management algorithms because all fault management related activities are executed on a single processor. Due to the centralized implementation of the fault management functions, it is much easier to verify the overall scheme. The main disadvantage of this scheme is that all the telemetry and telecommand signals are routed via a central computing node that introduces additional failures paths. Also, it may overload the central computing node.

3.2.2.3 Decentralized Fault Management Scheme

In the decentralized approach, the fault monitoring functions are moved to the individual computing nodes as shown in Figure 3.3. Each node monitors itself and passes the data to the centralized computing node that is the on-board data handling

(OBDH) node. The centralized computing node examines the telemetry data for detecting a fault in a node. If there is a severe fault, a particular node is replaced by a redundant node. However, on a minor fault, it is reported in the telemetry data for on-ground analysis.

The Modular Architecture for Robust Computing (MARC), proposed in [127], is based on a similar decentralized scheme. In this architecture, a Core Computing Module (CCM) is designated to run the main fault management algorithms. All the other computing nodes periodically send health telemetry data to CCM via SpaceWire [115] , which is used for detection of faults. On failure detection of a node, CCM reconfigures a redundant node to assign the lost tasks.

The decentralized FM scheme is better than centralized FM because it reduces the workload of the central computing node by shifting the monitoring functions close to the subsystems. It does not reduce the complexity but improves monitoring of parameters. However, similar to the centralized FM scheme, a failure of the centralized node can lead to the loss of the whole mission.

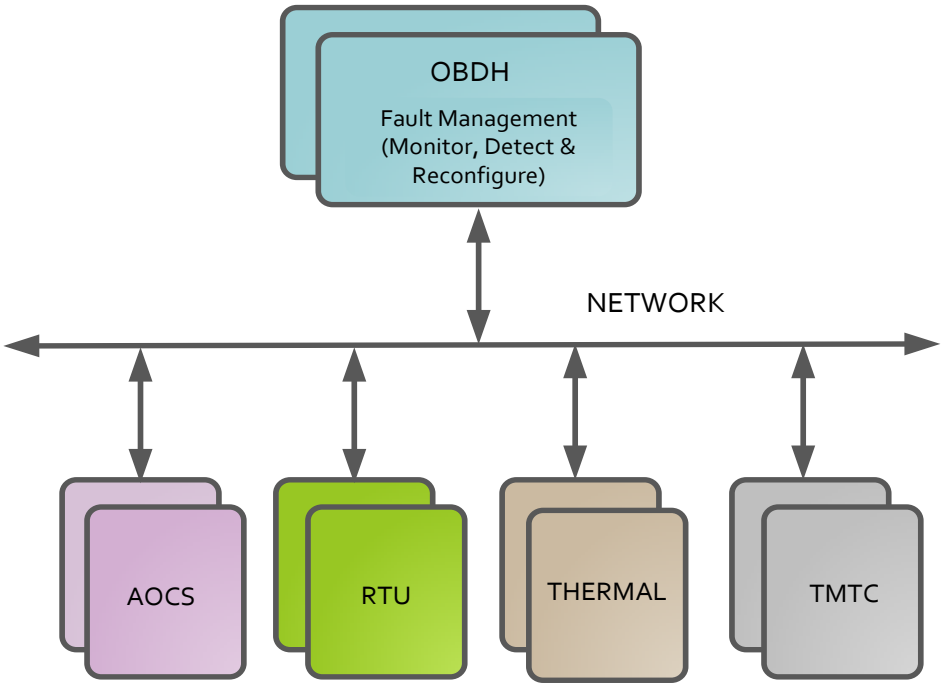


Figure 3.2: Centralized Fault Management Scheme.

3.2.2.4 Hierarchical Fault Management Scheme

In the hierarchical fault management scheme, shown in Figure 3.4, the system is divided into multiple levels [128]. Each level has its fault management mechanism for failure detection, isolation, and reconfiguration. Instead of a central controller, the hierarchical strategy spreads the fault management functions throughout the spacecraft. This off-loading of the fault management functions to multiple units results in a better performance. However, involving multiple levels of fault management makes the overall satellite system much more complex and vulnerable to failure, if not properly designed. Also, much more efforts will be required on the testing and verification of the fault management design.

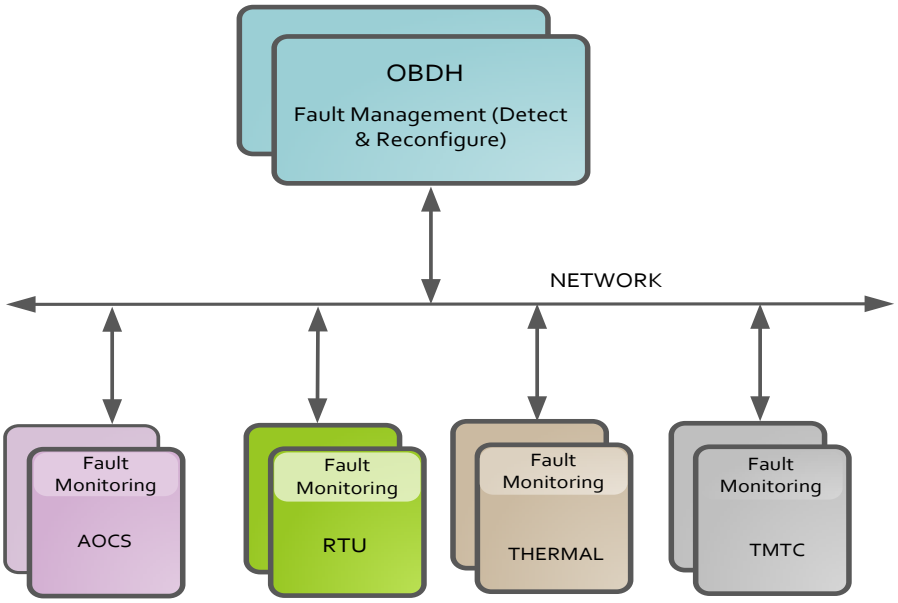


Figure 3.3: Decentralized Fault Management Scheme.

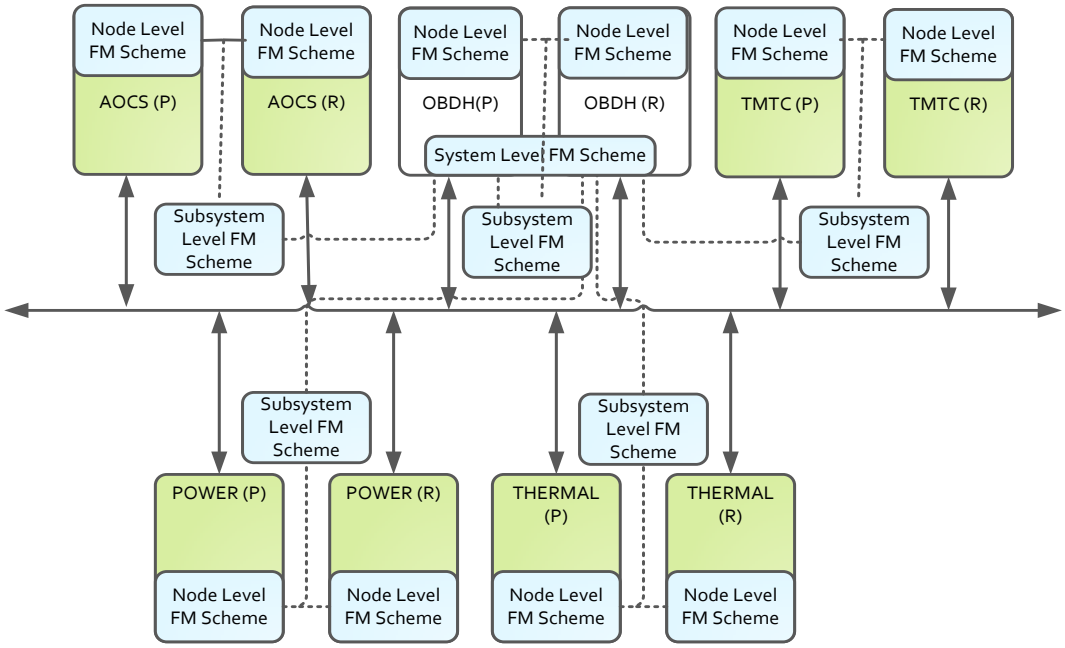


Figure 3.4: Hierarchical Fault Management Scheme.

3.2.3 Fault-Tolerant Computing Systems

Since 1960, various fault-tolerant computers have been designed and developed. In this section, we will first discuss centralized fault-tolerant computers. A centralized fault-tolerant computer executes a particular task set either by a Simplex execution, dual redundant execution or TMR execution. Then, we will discuss distributed fault-tolerant computing systems, where multiple sets of tasks are executed on different sites where each site is comprised of a single, dual or TMR computer.

3.2.3.1 Centralized Fault-Tolerant Computing Systems

The Apollo and SATURN V launch vehicle guidance computers were developed to use a static and dynamic redundancy scheme [33, 129]. Static redundancy has been adapted for the processors, while dynamic redundancy was used for the memory system. A processor set comprises three processors, running the same program and their outputs are voted to mask a single error. Two memories with error detecting codes were employed. If one memory fails, then a processor can access the other memory.

A self-testing and repairing (STAR) computer was developed by the Jet Propulsion Laboratory (JPL) [32]. The architecture of the STAR computer comprises multiple redundant functional modules, connected via a 4-wire internal bus and monitored by a special module called Test and Repair Processor (TARP). TARP monitors all the other modules by their messages on an internal bus. For detection, it relies on the arithmetic coding or uses simple comparison among the modules. Each transaction on the bus uses arithmetic codes (product code, residue code). In case of a failure, TARP first rollbacks a program in the faulty module. If the problem persists; it replaces the faulty module with a spare. The TARP itself has more than three redundant modules, but three of these are powered up at any time. The use of standby redundancy in the STAR computer is advantageous for long life missions. However, the overall design is much more bulky and consumes more electrical power due to its modular design, where each module has spares. Also, due to its Simplex execution, it can skip faults that are beyond the boundary of its self-detection logic.

The Fault-Tolerant Multiprocessor (FTMP) computer was designed for deep space missions in which maintenance is subject to a delay and a loss of control functions leads to a high cost in terms of life [37]. It is designed for a rate of 10^{-10} failures per hour due to random failures on ten-hour flights, where no on-board maintenance is available. The design consists of fully synchronous hardware units partitioned into processor cache modules, memory modules, and input-output modules, which communicate via redundant serial buses. All information processing and transmission is triplicated in the FTMP. A voter in each triad handles error correction and tolerance renewal. The tolerance renewal mechanism replaces a faulty module with a spare module. The FTMP redundancy scheme is not based on a simple TMR, it is a parallel hybrid redundancy, where any other similar module can substitute a major module.

The Software Implemented Fault-Tolerance (SIFT) computer was similar to FTMP in terms of functional specifications [38]. The two designs differ in the hardware and software fault-tolerance implementations. Unlike FTMP, SIFT mostly relies on software-based fault detection, correction and reconfiguration process. SIFT uses the concept of tasks execution in the form of iterations. A set of three processors runs the same task. Each processor executes tasks and places the data in its memory. After all the three processors place the output data in the memory, before the execution of the next task, all the three results are voted. If one of the three disagrees, a log event

is reported to the configuration module, which retires the faulty module and integrates a new module. The modules do not require a strict time synchronization. They only require different processors running the same iteration of the task within a 50 us boundary.

The space shuttle computer uses a hybrid redundancy approach, where each computing node communicates with the other computing node at the interface level for consistency, data voting, and synchronization [40]. This computer comprises of five general-purpose computing nodes. Four of these are reserved for flight-critical functions while the fifth is dedicated to non-critical functions. In normal operation, all four computing nodes are operated in a static redundancy mode whereby all nodes simultaneously process the same input and produce the same output. All computing nodes are loosely connected, and periodic messages are required to keep nodes within tolerable limits. For fault detection techniques like compare word testing, bus channel time out, self-testing and watchdog are used. On failure of two of the computing nodes, static redundancy is not possible, so dynamic redundancy mode is enabled. In this mode, only two nodes are included in the redundant set. Thus, failure identification is performed by self-testing of each machine. Due to the complex hybrid redundancy scheme and complex IOs design, the mean-time-to-failure (MTTF) for this computer is very small and it is not suitable for long-duration missions.

In [124, 130-132], the master-slave model is used for the implementation of a fault-tolerant computer for space applications, the design of which uses low cost embedded microcontrollers. Each embedded microcontroller module has three states; master, slave and off-line. In the initial configuration, one master, two slaves and one off-line embedded microcontroller modules are used. Master and slave modules have the same application program to provide masking against failures. Master and slaves are the active modules while off-line modules do not participate in the computation. During normal operation, the master sends computed result values to the slaves for voting whereas the slaves reply with an “agree” or a “disagree” decision. If any of the slaves disagree, the master sends an off-line command to that slave. In the off-line state, a slave can only execute diagnostic routines to detect the cause of failure. If an off-line slave cannot compute correct diagnostic computation, it turns off immediately. In case of a master failure, detection is a difficult process. If the majority of the slaves disagree with the master, then they confirm the master failure. In that case, a new

master is selected from the slaves. Thus, the master-slave model has a single point of failure. Also, critical functions such as clock synchronization, fault management, and consensus are all handled by a single computing node 'master', which potentially limits scalability and performance.

The Multicomputer Architecture for Fault-Tolerance (MAFT) was designed to provide extremely reliable computation in real-time control systems [133]. It divides each node into two separate processors: an Operational Controller (OC) and an Application Processor (AP). The OC handles the system executive tasks (the operating system tasks). These include inter-node communication and synchronization, data voting, task scheduling, error detection, and system reconfiguration. The AP runs only application tasks that include reading sensor data, executing control law functions and sending commands to actuators. In the MAFT computing system, application tasks are redistributed to account for changes after a node failure. Tasks are static to nodes, and can be run as an individual copy or through the use of replication. MAFT uses voting to detect a faulty task or faults / failures of a particular node. Also, a voted replicas system requires a minimum amount of healthy replicas to reach a consensus that is not possible if one of the nodes fails in a three node system. For communication, each node has its broadcast bus with the other nodes and number of communication buses depends on the number of nodes. On each additional node, an additional communication bus will be required which adds more overhead on the electrical power and weight.

A high assurance on-line recovery technology for an on-board computer design is presented in [134]. The on-line recovery computing system comprises multiple nodes, connected via a CAN network. The main objective of this computing system is to present a self-recovery mechanism of a computing node following a fault. In case of a fault, the faulty node can recover itself by only looking at messages on the communication network. It satisfies the requirements for a short control cycle and tries to maintain the degree of redundancy of the overall system. Similar to other approaches, this computer requires redundant computing nodes to mask a fault. Also, the computer uses a timeout for each computational step that requires perfect timing among the different nodes. Furthermore, the fault-tolerant scheme is not designed to work for Byzantine failures.

3.2.3.2 Fault-Tolerant Distributed Computing Systems

Distributed computing systems have gained significant acceptance in the area of mission critical applications. These systems offer substantial fault-tolerance features. Contrary to a centralized computing system, a single failure does not lead to the loss of the whole system [135].

Fault-Tolerant Distributed systems are classified into two main classes, (i) life critical and (ii) non-life critical systems as shown in Figure 3.5. Life critical systems involve human life directly or indirectly while the non-life critical do not. These are further divided into computationally critical, high availability, and high-performance systems. Computationally critical systems are those systems that require real-time hard or soft deadline for their operation. Missing a deadline can be catastrophic resulting in the loss of the whole mission. In high availability system, occasional loss of one computing unit is acceptable but the entire system outage is not acceptable. HPEC systems demand higher throughput but do not require a hard deadline. In the following section, we will review Fault-Tolerant Distributed Systems for computationally critical systems with particular emphasis on satellite systems [136].

The concept of FTMP [34] is extended to a distributed system in the Advance Information Processing System (AIPS) [137]. The computer system is divided into multiple sites. Each site consists of a triplex, duplex, or simplex configuration depending upon the criticality of its tasks. These sites are connected via an intercommunication network, which is also triplicated. Each processing site has its local clock for synchronization. Hardware voting is used throughout the system. Hardware voting within a site is easy and managed by the local clock. However, a synchronizer is required for triplicated data send between the processing sites.

In [82, 138], D. A. Rennels extended the master/slave approach proposing a distributed hierarchical computer, consisting of low-level and high-level computers. Although, the design is similar in nature to the master/slave approach in terms of physical architecture, it is different in terms of the functional behaviour. In the master/slave model, discussed in section 3.2.3.1, each computing node is assigned the same tasks, and the final output is delivered by the master only. In a hierarchical design, each low-level computer has its task set while the high-level computers store commands from ground, direct processing in the low-level computers and control the

communication on the network. For multiple low-level computers, one high-level computer is designated. For hardware redundancy, each computer (high, low) is duplicated. However, each duplicated computer is dedicated to a particular function and is assumed to run only a specific task set. In other words, it cannot reconfigure for other functions that make this approach inflexible.

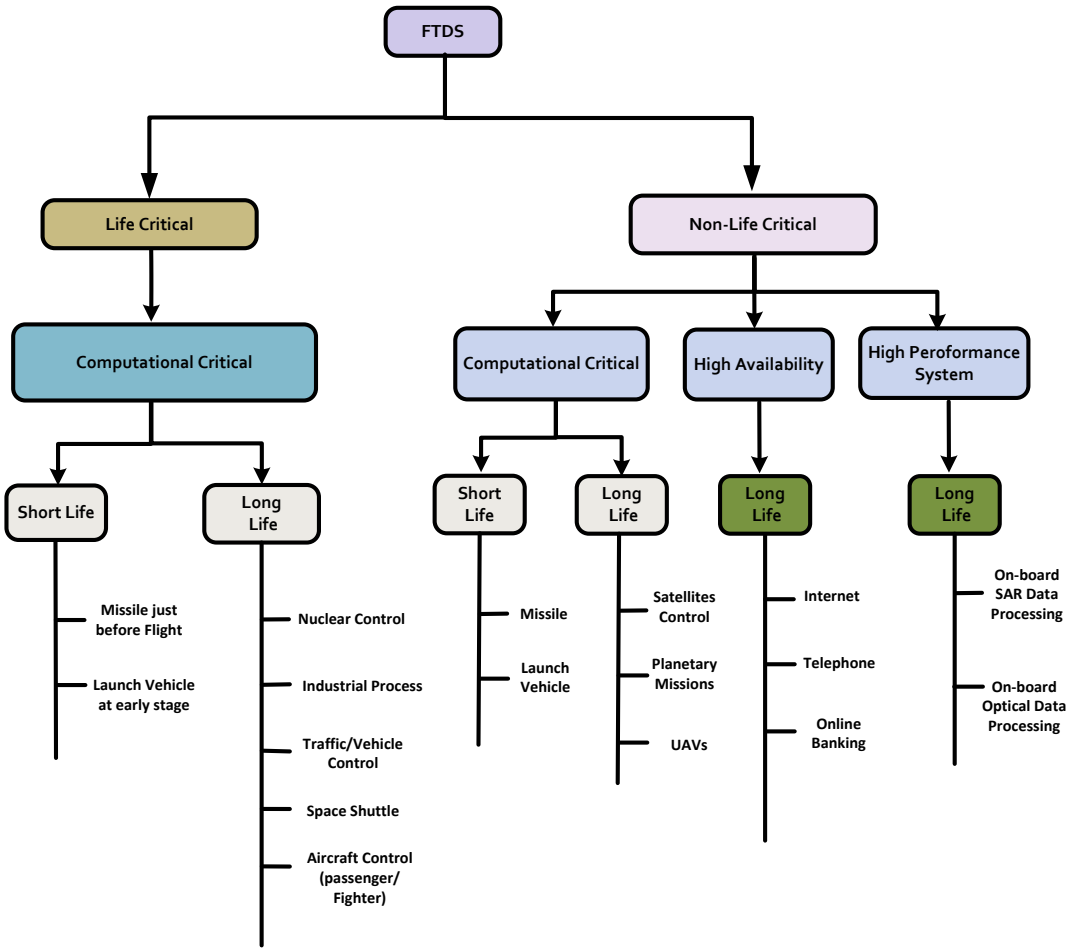


Figure 3.5: Classification of Fault-Tolerant Distributed Systems.

In 2000, NASA started to develop a low-cost distributed computing architecture for deep space applications [139, 140]. The main objective was to build multi-mission spacecraft systems using COTS technologies. Similar to the traditional computing architectures, the 'X2000' architecture is comprised of multiple nodes connected via two dual redundant networks in a bus topology (IEEE 1394, I2C). Due to COTS communication protocols in the design, extra efforts were required to implement a fault-tolerant network design. This design includes an enhanced fault detection and

recovery mechanism that includes fault isolation and recovery by design diversity. As the main focus of X2000 was to build a robust network, fault-tolerance is provided by the use of dual redundant nodes. The number of redundant nodes increases with the increase in actual computing nodes which is an underutilization of computing resources.

The design of the Maintainable Real-Time System (Mars) started in 1980 [3]. The first Mars prototype, which was targeted to real-time control applications, was developed in 1984 at the University of Berlin. Mars uses active redundancy for fault-tolerance whereby two or more components executes the same tasks. Communication between any two components is protected against errors by sending the messages twice. The components are self-checked and behave silently on the occurrence of a fault. This fail-stop feature restricts components to either sending the correct message or no message. Mars components are arranged in a cluster. Within each cluster, an interface component provides extensibility of the cluster. The communication between the different components is based upon the time division multiple access (TDMA) scheme. Although, Mars has useful features of self-checked components, all the redundant resources reside in the idle state and are only activated in case of a failure of their primary peer computing node.

The Delta-4 project [55] defines an open, fault-tolerant distributed computing architecture. It consists of multiple computing nodes connected via a local area network. An individual node can be a processor, a multiprocessor system or a specialized system comprised of an array of processors. Software components replicate to multiple nodes to provide active redundancy against faults or failures. Each node has a Network Attachment Controller (NAC) that provides services related to communication and message self-checking comparison. Also, the NAC provides a multicast and fail-stop node operation.

3.2.4 Discussion

It is evident from the summary of the reviewed centralized fault-tolerant computing systems in Table 3.1 that most of the computers, particularly designed for short duration applications (aircraft and space shuttle) rely on hardware and software TMR based approaches for fault-tolerance. Although, TMR has a better fault coverage, but

adding a spare node or reintegrating an existing node requires synchronization. Also, the use of TMR is more costly and consumes more electrical power. For long duration missions such as spacecraft, where a temporary pause in computation is tolerable, standby redundancy is considered a better option than TMR.

The reviewed distributed computing systems are summarised by rows 1 to 5 in Table 3.2, row 6 is related to the work proposed in this thesis. It is evident from Table 3.2 that for short duration missions, the highly redundant distributed system (AIPS) that includes DMR or TMR is the preferred option. While for long duration control applications, EDRB and triplication are the preferable methods. The third category of distributed systems is related to space applications where Simplex processing with a standby redundancy is used. Standby Redundancy requires less physical hardware resources and electrical power, both of which are limited on board spacecraft. As the space system is tolerable to lost computation without damage, the standby redundancy is the most appropriate option for these unattended systems.

It can be seen from Table 3.2 that distributed computing has been employed in spacecraft in one form or another. Especially, it can be noted that multiple processors are used to carry out a set of tasks in existing systems, however, the tasks are not executed in a collaborative way. The reliability is ensured via physical hardware redundancy of each processor. Therefore, if one processor fails its redundant backup takes over, and even if that fails, then there is no provision for transferring its set of tasks to another processor. In these circumstances, the spacecraft has to limit its functionality by operating in a safe mode. The distributed computing architecture summarised in the bottom row of Table 3.2 aims to address the above deficiencies. It is introduced in Chapter 4 and a system-on-a-chip multi-core implementation is described in Chapter 7. A distributed FDIR strategy is used, in which each node has its own FDIR mechanism, embedded inside a dedicated block. This approach provides architectural-level cost effective fault-tolerance by enabling tasks' migration among the computing nodes.

Table 3.1: Fault-Tolerant Centralized Computers.

Fault Tolerant Centralized Computer	**Redundancy**	**Redundancy Management**		**Communication Network**	**Application**
		Fault Detection	*Recovery & Reconfiguration*		
Apollo and SATURN V	Triple Modular redundancy at module level	Disagreement Detectors	Manual Commands	-	Short Duration missions (Launch Vehicles)
JPL STAR Computer	Simplex Execution with Spares	Arithmetic Codes, Comparison test	Test and repair Processor (TARP)	Internal 4-bit bus for communication of modules	Long-life missions (Spacecraft)
FTMP	Parallel Hybrid Triple Modular Redundancy at module level.	Hardware TMR Voting	Software to replace a faulty unit.	5x Redundant Buses with bus guardian	Aircraft
SIFT	Hybrid TMR redundancy at module level	Software TMR Voting	Software to replace a faulty unit.	5x Redundant Buses	Aircraft
Space Shuttle Computer	Hybrid redundancy (5 identical computers in a set)	Compare word test, bus-channel time-out test, self-test, watchdog timer	Manual Restoration	28 1-MHZ Serial Data Buses (23 shared, 5 dedicated)	Short Duration missions
COTs FT Computer by JPL	TMR Master/Slave with one spare	Message Timeout or mask error by majority voting	Multi-level Self-Recovery	Master channels, Slave Channels and Status Channels.	Long-life Missions (Spacecraft)
MAFT	Hardware / Software replication	Reasonable checks, Convergent Voting Strategy and ECC.	Operational Controller	Fully Connected Broadcast Bus Network	Aircraft
High Assurance Online Recovery	Multiple nodes with replication.	Message Timeout and Software voting	Self-Recovery	TDMA based over CAN	Short duration

Table 3.2: Fault-Tolerant Distributed Systems.

Fault-Tolerant Distributed Systems	**Site Redundancy**	**Redundancy Management**		**Communication Bus/Network**	**Application**
		Fault Detection	*Recovery & Reconfiguration*		
AIPS	Simplex, DMR, and TMR	Hardware TMR Voting, DMR Comparison	Primary / Backup global computers	Network, comprise of redundant links and switching nodes.	Short Duration missions
FTD Computer by JPL	Simplex for Low-Level and High-Level Computers	Arithmetic Codes	High-Level Computers	Multiple Redundant Buses	Long-life Missions(Spacecraft)
Mars	Hardware / Software replication with fail-silent node.	Self-Checking Nodes	-	TDMA over Ethernet with message redundancy	Process Control
Delta-4	Hardware / Software replication with fail-silent node.	Software Voting / System Monitoring	System Administration Software	Local Area Network (Duplex channels)	General purpose applications
EDRB FTD Computer	Extended Distributed recovery block	Acceptance Test, Message Timeout	Supervisor node	Dual redundant supervisor and node pair Networks.	Process Control
Fault-Tolerant Distributed Computing	Over-Provisioned Resources	Hybrid (Symptom-based Fault Detection + Monitoring)	Distributed Coordination	Dual redundant Time Triggered Networks	Space Applications

3.3 Wireless Protocols for Spacecraft Fault-Tolerant Computing

Systems such as traffic control, industrial automation, aerial vehicles, satellites and space shuttles heavily rely on data communications, both for normal operations as well as for diagnostics. Using wireless links instead of wired bus harnesses, has several advantages. Relocation of sub-systems becomes easy since there is no need for rerouting data cables. No special connectors are needed for additional diagnostics, and the data link is inherently immune to wear and tear. Furthermore, once the wireless link

is properly designed, system integration, testing, and operational diagnostics are faster and easier, which is a significant issue in critical systems comprising many integrated sub-systems.

Spacecraft, in particular, can benefit from using wireless communication links on board. Wireless interfaces can help reduce the overall mass, as the harness can weigh as high as 10 % of the total mass of the satellites [141]. Wireless links are less vulnerable to debris impact when these are deployed in LEO. It is because of the number of objects in these altitudes is drastically increasing with time [19, 20]. It is also relevant to other safety critical applications. A system failed in navy ships due to damaged harness and its cost analysis for its diagnostics is reported in [142]. Despite several benefits, wireless COTS protocols in its current form cannot be deployed for space fault-tolerant distributed systems. It is because COTS technologies are not inherently designed for critical applications.

Wireless links are inherently unreliable and often characterized by higher message loss. For reliable data delivery, forward error correction (FEC) and assured delivery are usually used. Additional bits in the form of FEC are appended with data for the correction of transmission errors while assured delivery assumes acknowledgment on each message from the recipient. FEC is better than the assured delivery because it not only assures reliable data delivery but also helps to maintain timeliness of the distributed application. FEC can only work if a packet receives at the receiver site. However, in case of packet loss, retransmission of data is essential. Packet loss may be worse when wireless distributed nodes place at a short distance in a close metallic structure of the spacecraft.

Electromagnetic Compatibility and Electromagnetic Interference (EMC/EMI) is another problem in the employment of wireless technologies for space applications. In [143], various wireless technologies for space applications are investigated. Among these, WiFi (IEEE 802.11) standard was analysed for satellite on-board communication for EMC/EMI at frequency bands of 2.4 GHz and 5.0 GHz. No interference was reported with the Telemetry/Telecommand S-band at 2.4 GHz, but it does interfere with the payload instrument, Doris, operated in the S-band range. Interference also occurred with the spaceborne Synthetic Aperture Radar (SAR) and X-band radar harmonics when operated in the 5.0 GHz band. It shows that the adoption of wireless

communication as OBDH bus requires careful frequency selection among the payload, TM/TC, and the wireless standard.

In fault-tolerant distributed systems, devices need to communicate with each other through the exchange of data which requires topology. The star topology in the infrastructure mode is proposed for Wi-Max/Wi-Fi [144] [145]. Star Topology is not suitable for fault-tolerant computing because all the computing nodes are connected via the access point rather than directly connected to each other. Another topology based on the tree structure as suggested for ZigBee [146], which can make direct communication possible but it may susceptible to a single point of failure. On the other hand, Mesh Topology avoids such single point of failure problems, is most suitable option for the spacecraft distributed computing.

In fault-tolerant distributed systems, a deterministic access on the network is essential which cannot meet with the non-deterministic CSMA/CA medium access protocol currently employed in wireless communications protocols. In IEEE 802.15.4 Standard, the channel is divided into slotted and non-slotted mode. In the slotted mode, the channel is accessed on a turn basis whereas non-slotted mode allows anyone access on the channel. The slotted scheme of IEEE 802.15.4 standard can be useful for the deterministic access.

Fail-stop node behaviour when sick is an important concern for critical applications. If this does not address properly, a faulty node can send erroneous messages that subsequently jam the rest of the nodes' communications. However, due to the availability of multiple channels in current wireless technologies, such failures can be handled very easily.

The IEEE-802.11 protocol [145] is the most researched and widely adopted to emerging applications. It has been used in military mobile ad-hoc networks, railways (ALARP) [147], aerospace, medical, and commercially almost every household and office. It has been evaluated for a constellation of satellites, operating in a network [148-152] and also proposed as the main data handling bus within the spacecraft [153]. Another wireless protocol, ZigBee [146] [154] is developed for very low power data sensing applications. ZigBee has been used in many applications. It includes Aerospace Wireless Sensor Network (AWSN) [155], Physical environmental and Physiologic [156], Electrical Ground Support Equipment (EGSE) [157], and Vehicle

Collision Avoidance System [158]. ZigBee is also found its usage in spacecraft telemetry/telecommand system [159], launch vehicles, and space shuttles system [160]. It is evident that ZigBee can deploy in critical applications. However, its adaptation requires further investigation of the requirements for critical systems

Bluetooth is analyzed as a replacement for existing CAN networks for intra-vehicle and inter-vehicle non-critical applications. Bluetooth is also suggested for combat vehicles as reported in [161] where wireless Bluetooth replaces wired communication between crew stations. A major hindrance to its adoption for critical applications is the connection setup time that in some cases is as much as 5-6 seconds. The main features of COTS wireless protocols are stated in Table 3.3. These features act as a basis for the selection of wireless protocol, over which further modifications can be made. The use of COTS protocols as a baseline design for fault-tolerant computing reduces the design cost and time.

Table 3.3: Features of Existing Wireless COTS Technologies.

Wireless Technologies	***WiFi (IEEE 802.11b)***	***WiFi (IEEE 802.11a/g)***	***ZigBee (IEEE 802.15.4)***	***Bluetooth (IEEE 802.15.1)***
Modulation	DSSS	OFDM	DSSS	FHSS
Encryption	Optional RC4(AES in 802.11i)	Optional RC4(AES in 802.11i)	AES Block Cipher(CTR, counter mode)	EO Stream Cipher
Topology	Infrastructure/ Ad hoc	Infrastructure/A d hoc	Star/ Mesh/Cluster Tree	Point-to-point, point-to-multipoint
Access Protocol	CSMA/CA	CSMA/CA	Slotted/Un-slotted CSMA/CA	Master/Slave
Transmission Range	30 m(indoor) @ 11 Mbps	30 m(indoor) @ 54 Mbps	10 to 100 m but for ZigBee Pro is 1500 m	1 to 100 m
Freq. Band	2.4 GHz ISM	2.4 GHz ISM (g) 5.0 GHz U-NII(a)	868 MHz Europe, 915 MHz USA/Australia; 2.4 GHz	2.4 GHz
Data Rate	11 Mbps	54 Mbps	20 to 250 Kbps	0.723 to 2.1Mbps
Power Consumption	< 1 W	< 1 W	< 1 mW	$\leq$ 100 mW
Channel BW	25 MHz	20 MHz	Multiple channels in each band	1 MHz/channel
Duplex	Half	Half	--	Half/Full

3.4 Modern Implementation Approaches to Fault-Tolerant Computing Systems

This section will overview other modern implementation approaches to fault-tolerant computing, particularly focusing on the design of space computing systems. It will briefly discuss reconfigurable computing systems, multicore systems and cluster computing systems. The main focus of this review is to highlight the advantages and disadvantages of each approach.

Reconfigurable Fault-Tolerant Computing: A reconfigurable computing platform is a platform that can be repaired or reconfigured. To achieve a reconfigurable computing platform, static random access memory (SRAM) based field programmable gate arrays (FPGAs) are used. The inherent reconfigurable feature of these FPGAs provides a computing platform that can be repaired to tolerate hardware failures. These FPGAs can accommodate redundant logic such as a processor, input-output blocks, and memory system, and can load a module to repair or upgrade an existing computing system. The reconfigurability feature of FPGAs is particularly suitable for remote systems, where it is hard to repair after the initial installation. The spacecraft is one such example of these systems [162].

In [163], a reconfigurable fault-tolerant (RFT) avionics system for a Nanosatellite has been proposed. A node based on an SRAM based FPGA was designed which allows switching between Simplex and TMR based redundancy scheme. The main objective is to save electrical power by switching different redundancy schemes. The selection of simplex or TMR scheme is based on the orbital parameters.

In [164], another method that includes Simplex, DMR and TMR redundancy scheme has been presented. Instead of using orbital parameters, selection of redundancy scheme is selected on the severity level of radiations which is monitored by a MicroBlaze processor. Each time a pre-set threshold is reached; configuration bits related to the particular redundant scheme are loaded. During the normal operation, it utilizes scrubbing to mitigate the effects of Single Event Upsets (SEUs).

FPGAs devices wear out with use and can fail due to two types of failure mechanisms ─ physical and functional failures. Physical failures are permanent and due to defect in processing, packaging, die attachment failure, bonding or particle

contamination. Functional failures are temporary or intermittent and due to striking of high energy proton or neutron. It is evident that fault-tolerance by a reconfigurable computing platform can only protect the system from functional failures or partial physical device failures. In the former case, scrubbing and in-circuit redundancy are employed. Scrubbing is an error correction technique, which is based on rewriting the FPGA configuration to avoid accumulation of errors induced by radiation [165]. In the latter case, the configuration affected by the partial device failure can be relocated. However, protection against a full device failure is essential, which can only be provided by a fault-tolerant distributed computing approach.

Multicore Fault-Tolerant Computing: Another recent trend for enhancing the reliability of on-board computing systems is the use of multicore or manycore processors [16]. Multicore processors are inherently redundant in terms of processor's cores, I/O, power supply pins and memory ports. Thus, a system can utilize these resources for improving the reliability of on-board computing systems. In a multicore system, multiple available cores are used for fault-tolerant computing. Either software or hardware provides fault-tolerance in multicore systems. Software-based fault-tolerance is preferably suitable for commercial-off-the-shelf (COTS) processors, where hardware modification is impossible. While the hardware based fault-tolerance is employed for custom design of multicore processors, which can be designed as ASIC (Application Specific Integrated Chip) or it can be implemented on an FPGA.

In the software-based fault tolerance, redundant execution approaches, which exploit the inherent replication of processor cores, are used [166-169]. Redundant execution of a process can be done in time or in the spatial domain. In the time domain, a single core is used to execute the multiple copies of the same process. The utilization of the single core for replication is simple, but it is limits to transient faults only. On the other hand, in the spatial domain, multiple cores are used to execute replicated processes. Each replicated process runs in a separate core and results from all are compared or voted to produce a result. A software implemented fault tolerance (SIFT) approach is demonstrated in Maestro [170], in which a processor consists of 49 cores interconnected via switch engines. The Control and Fault Management (CFM) software that is used to manage the cores collects the error messages and restarts the applications runs on three cores. CFM sends a heartbeat message to the external

hardened state machine, where they are voted. If any two of the three fails to deliver an 'OK' heartbeat message, the whole system is restarted by the external sequencer.

Common circuitry of COTS multicore processors, such as clock and control, are not inherently designed to be fault-tolerant. Also, a fault in a single core can stop the whole chip. Furthermore, shared memory among the cores is a potential bottleneck to achieve high performance [171] [172]. Lastly, heat dissipation per small chip, particularly in the presence of vacuum is another issue [173].

To solve these problems, various solutions have been proposed, which include the custom design of multicore processors. For fault containment, one approach is to isolate all the cores completely on a chip. Each core has its memory controller and input-output connection. This approach is efficient for fault containment, but hard partitioning of resources such as cache, memory controller, and input-output significantly reduce the overall performance. To overcome this problem, another partially partitioned design for the multicore processor is proposed in [167]. This design divides the overall resources into multiple groups, and each group is bound to share its resources. For fault-tolerance, such as in the case of DMR or TMR implementation, each computing core must be from a different group to isolate primary from the redundant core.

A non-shared memory based architecture of the multicore processor is presented in [170]. This architecture eliminates the shared memory access latency. In this architecture, each core has its dedicated memory and sharing of data accomplishes via message passing over a network among the cores. For fault-tolerance, TMR based software replication is suggested.

Instead of considerable efforts to solve multicore issues, heat dissipation is still a problem, particularly for space applications. Also, a single node of a multicore processor is a single point of failure. Therefore, an approach that exploits features of the multicore processor and avoids the problems of a single point of failure and heat dissipation is essential.

Cluster-based Fault-Tolerant Computing: The NASA's New Millennium (NMP) ST8 project was aimed to develop a COTS-based dependable multiprocessor systems [174-177]. The architecture consists of dual redundant system controller and *n* data computing nodes; all are connected to a dual redundant gigabit Ethernet. The whole

assembly forms a cluster computer for parallel data processing. All type of messages (data, control), a single Ethernet, is used which can potentially delay the cluster's reconfiguration process in case of failure.

One such example of Linux-based Beowulf cluster computer for terrestrial applications is proposed [178]. In [179], a similar Beowulf cluster approach is adopted for the satellite imaging payload application. This cluster computer comprises of 20 StrongARM controllers that are connected by four FPGAs. This cluster is merged as a single computing node to form a homogenous cluster and require some mechanism to dissipate a large amount of heat. Heat dissipation is a serious issue in embedded clusters architectures for on-board computing systems. It will require system designers to solve problems ranging from how to house, power, and cool the machine.

Remote Exploration and Experimentation project (REE) is one such effort, particularly designed for deep space missions [180, 181]. The main goal of the REE project is to develop a low cost very high-performance computer system comparable to a supercomputer for space applications. The main motivation behind this project is to provide on-board autonomy so that more science objectives can achieve with the help of low-cost commercial-off-the-shelf components. Unlike fault-tolerant systems, it is allowed to fail occasionally similarly to the sample data computation systems. It is primarily designed for science data processing rather than the mission critical and hard real-time data processing; therefore software based triple mode redundancy for fault-tolerance was used.

3.5 Issues of Current Fault-Tolerant Computing Approaches

Legacy fault-tolerant computing systems aimed at space applications were designed with high reliability as an uncompromised objective. Employing a redundancy scheme is a common practice. It is evident that physical redundancy alone cannot be a cost effective solution to achieving higher reliability as well as High Performance Embedded Computing. Also, recent failures of On-Board Computers in GOCE-2013 [182] and Phobos-Grunt-2012 [183] revealed that centralized and dual redundant computers may fail resulting in the loss of the complete mission. Other issues related to computational integrity, adaptability, resource underutilization, provision of task

migration, and isolation of the input-output interfaces also demand a new approach to designing fault-tolerant computing systems for space applications.

Computational Integrity: In a fault-tolerant computing system, computation integrity refers to a loss free computation. Computational integrity is maintained by periodic storing of the task states during the normal operation that are provided to a redundant node in case of failure of a primary node. In a review of the state-of-the-art schemes and methods, only a few fault-tolerant schemes consider state storing while most of them do not. However, due to a large amount of primary to redundant switching and state transfer time, a considerable amount of computation is lost that would result in compromised computational integrity.

The two main approaches to computational state storing use: i) a separate internal module attached to the system bus, ii) a centralized module connected to a network, can be used [121]. The need to design a separate internal module adds additional design complexity in terms of isolation between the powered and unpowered nodes, particularly in case of a cold redundant system. On the other hand, a centralized module attached to the network provides a complete isolation between the primary and the redundant node, but the state transfer time due to the communications is considerably high. Furthermore, both approaches can cause a single point of failure.

Adaptability: The second important issue is the lack of adaptability in current fault-tolerant systems. In current on-board systems, redundant resources are fixed to a particular subsystem and sharing of these resources among the different subsystems is not possible. Emerging demands for High Performance Embedded Computing require a fault-tolerance technique that utilizes the inherent availability of multiple processing units. In other words, instead of placing idle resources for each subsystem, resources need to be shared for the purpose of high reliability, performance, and computing load balancing.

Also, demands of space missions ─ space probes, space robotics ─ vary throughout the mission life. In some phases, High Performance Embedded Computing is required for a very short duration, such as in the case of a Lander. Autonomous landing requires High Performance Embedded Computing for real-time range and range rate estimation algorithms, as well as terrain visualisation and trajectory calculations. Additional dedicated processors are used for this computation. However,

as soon as the Lander completes its landing manoeuvre, these computers are put in the idle state and are not utilizable for another purpose. The provision of dedicated computing resources for such a short duration is not a good solution because it increases the cost, weight and size of the spacecraft. The same demand can be met with the idle redundant computing resources if a system is provisioned to adapt to such scenarios.

Furthermore, current computing systems are not capable of adapting their operation to achieve thermal balancing or compensation against environmental radiation. This makes thermal and radiation design of High Performance Embedded Computing systems rather complex.

Inefficient Utilization of Resources: In the state-of-the-art redundant space fault-tolerant computing systems, discussed in section 3.2.3, resources are reserved for fault-tolerant operation. For example in the TMR computer, a set of three processors is running the same tasks to achieve high reliability. It results in a resource utilization efficiency of only 33 % because three processors do the work of one processor. Compared to TMR, the internal redundant centralized and dual redundant standby computers are relatively more efficient, where one processor is dedicated to the execution of the tasks, while the other processor in the idle state, consuming only 50 % of the total available resources.

On the other hand, physically distributed computing systems, which are usually comprised of three nodes requires three redundant nodes. In this scenario of three nodes, three redundant nodes are in the idle state, thus making the system inefficient in terms of resource utilization. It is because computing resources are grouped in a pair of two nodes (primary, redundant), and a node failure is only masked with its peer redundant node. The reliability of a distributed system can be enhanced if the system is provisioned to reconfigure all available nodes in case of a failure of any of the distributed nodes.

No provision for Task Migration: Task migration is supported in general purpose desktop based distributed systems. However, there is no such technique which could be directly employed in safety critical distributed embedded systems. The lack of the task migration capability for such critical systems binds the computing tasks with a

particular node, thus making a distributed embedded computing system inflexible to achieving reliability and performance simultaneously.

Isolation of Input-output Interfaces: In fault-tolerant computing systems, both the primary and the redundant computing nodes need to be connected with the various input-output signals. These signals correspond to the sensors, actuators or health status of the different modules. In case of a failure, a switch-off is essential for isolating a faulty primary node. Although standard methods are employed for input-output isolations of the primary and redundant node [184, 185], the power-off faulty node can accidently be back-powered from the common logic signals, thus causing unexpected behaviour in the running node.

3.6 Problem Definition

Distributed computing is achieved via the use of distributed systems to solve computational problems. In a distributed system, a bigger computation problem, such as satellite attitude and orbit control, is divided into *v* tasks, and each processor is assigned a subset of the *v* tasks. All *v* tasks are executed collaboratively to produce the final result.

On board spacecraft, each task has to be executed timely, and in a coherent manner. For instance, consider an example of an AOCS system. In some missions, AOCS handles firing of the thrusters at exact times. The time slot is limited, and the task has to be executed within the time available, otherwise the wrong firing of the thrusters would result in the satellite moving in an unwanted orbit, and may cause a collision with existing satellites. This emphasises the following points:

- Tasks must be executed timely.
- No task can be left incomplete. Otherwise, the overall outcome will be incorrect, leading to erroneous computational results, which can cause a disaster.

Consider a set of tasks being executed on the distributed computing system shown in Figure 3.6. Firstly the tasks will be distributed amongst the processors by some algorithmic means and then each processor will perform its tasks. Eventually, the system output is the overall outcome from all processors. In a normal operation, all processors would be working as desired. Now consider a scenario, when one or more

processors fail, as depicted in Figure 3.6, leading to an erroneous result. In order for the system to be able to recover from this unacceptable situation the following is required:

- A fault detection (FD) mechanism must be in place.
- The fault should be isolated, i.e. the faulty processor should be correctly identified, and removed from the system.
- Finally the system should be reconfigured via re-allocation and migration of the tasks of the failed processor(s) to other healthy processors.

The needs stated above necessitate the incorporation of a fault management capability. The existing fault management schemes, reviewed in this chapter, do not fulfil the requirements of high-performance mission critical embedded systems. To address this gap in the present state-of-the-art, in this thesis a novel approach to fault management is proposed, the fault model and performance metrics for which are detailed below.

Fault Model: We define the targeted fault model that is representative of typical problems occurring in distributed computing systems in the presence of failures. We assume that failures could manifest themselves as temporary or permanent processors failures. We assume that the underlying network is reliable and messages are shared via synchronous communication semantics. We consider the following processor failures.

- ***Processor Fail-Stop or Crash Failures:*** A processor may crash, and once it crashes, it can be restarted and reintegrated. The crash failure of a processor in our synchronous model of communication is detected by timeout messages.
- ***Processor Crash and Restart:*** In this scenario, a processor restarts after having a crash failure. The crashed processor loses its current state, but its operation is resumed from a known state after restart.

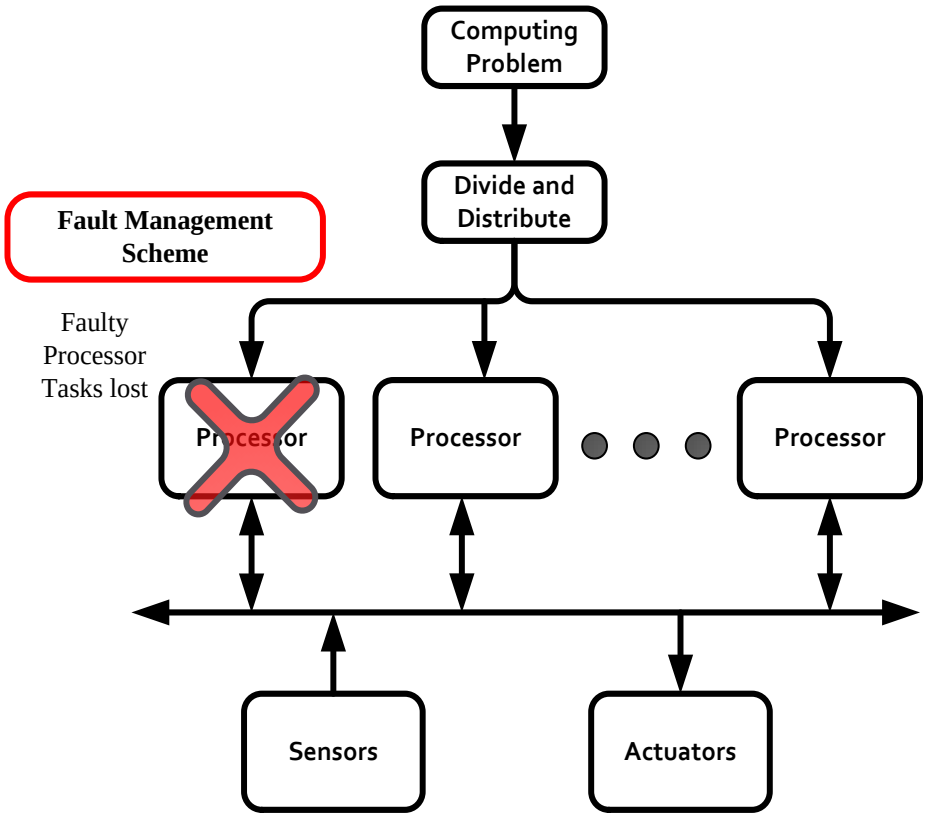


Figure 3.6: Operations of a Distributed Computing System under Fault

Performance Measures: For evaluation of the proposed fault management scheme, reliability and availability metrics are identified as detailed in chapter-6. The performance of the fault management scheme will be measured by the fault coverage and computational integrity of the system as follows:

- ***Fault coverage:*** We define fault coverage in terms of qualitative and quantitative parameters. The qualitative parameter specifies the type of the faults (transient, permanent) that the system can handle. The quantitative parameter expresses the conditional probability that the system will recover appropriately in the event of a fault occurrence of a particular type. For each fault type, its quantitative parameter gives a measure of how well the fault protection mechanism work.
- ***Computational integrity:*** Computational integrity has two main components, (i) the time period when the computation is not available, referred to as *reconfiguration time*, and (ii) the degree of protection of critical state data Δ_{SD} ,

referred to as *check-pointing* and *state rollback*. Both are defined and explained in chapter-6.

3.7 Summary

In this chapter existing redundancy schemes, architectures and fault management approaches in computing systems were reviewed and analysed. It is concluded that for long term missions, it is appropriate to use the standby redundancy scheme due to its reduced electrical power and area. An on-board distributed architecture requires a redundancy management scheme to make the computing system fault tolerant. Various redundancy management schemes were critically reviewed— Half Satellite, Centralized, Decentralized, and Hierarchical—commonly employed in space computing systems, highlighting the advantages and disadvantages of each scheme. It was concluded that the decentralized and the hierarchical scheme have an edge over the other schemes in terms of reliability. However, current redundancy based designs, where a task is bound to execute on a primary node or its peer redundant node, can only utilize the primary and redundant node computing resources. Also, they do not allow other subsystems to make use of their idle redundant resources either for performance or fault-tolerance purposes. In future computing systems, where a computing system is comprised of several computing nodes, subsystem level redundancy in the form of primary/redundant node is not feasible. If the same legacy approach is adopted, it will be very expensive and would result in underutilization of computing resources.

Conventional space-borne fault-tolerant systems— centralized, and distributed— were also systematically reviewed and analysed. It was concluded from this study that the proposed system approaches were limited and could not be used to directly enhance high performance and reliability in mission critical embedded systems. Furthermore, significant issues of current fault-tolerant computing systems were highlighted and discussed.

An overview of wireless protocols and their suitability for on board spacecraft fault-tolerant computing was also presented in view of the benefits that intra-satellite wireless communication can bring to distributed systems. Two COTs wireless protocols (WiFi and ZigBee) were found suitable for further improvement and their

deployment as a fault-tolerant network. A brief review of modern implementation approaches to fault-tolerant computing systems was also presented. The suitability of each design was critically reviewed by presenting advantages and disadvantages.

To summarise, legacy computing architectures and approaches are designed to meet only one objective, either performance or reliability. No single computing system meets both objectives simultaneously. The outcome of this review emphasized the importance of the current research topic of task oriented fault-tolerant distributed computing and highlighted the need for a feasible, efficient solution that is not available in the known literature and engineering practice. Following from that a new approach to fault tolerance management in distributed systems was conceived, which requires architecture level changes and enhancements, thus leading to a novel distributed computing architecture and fault management scheme.

Chapter 4

Novel Architecture for Fault-Tolerant Distributed Computing

In this chapter, the proposed architecture for fault-tolerant distributed computing is presented. The architecture covers the hardware as well as the software part. In section 4.1, preliminary details of the architecture are introduced while the system hierarchy of the proposed architecture is discussed in section 4.2. Details of the design of the distributed computing node are presented in section 4.3, while the input-output node is discussed in section 4.4. Nodes are connected via a network that is discussed in section 4.5. The software stack of the distributed computing node is presented in section 4.6, which primarily includes the software design of the processing unit and the fault management block. Details of the fault management scheme are covered in section 4.7, while section 4.8 describes algorithms proposed for fault detection.

4.1 Introduction

The proposed architecture addresses High Performance Embedded Computing requirements and the need for a fault-tolerant capability of the current mission critical applications as discussed in Chapter 1. High Performance Embedded Computing is achieved via the incorporation of multiple processors, which communicate with each other over a network. The multiple processors work in collaboration and, therefore, the system falls under the category of collaborative distributed computing systems. To

make the distributed architecture resilient to failures, a Fault Management Scheme must be incorporated as stated in section 3.6.

The three main components of the architecture are:

1. Nodes,
2. Communication Network, and
3. Fault Management Scheme.

Any processor attached to the communication network is termed as a node. A node is further distinguished based on its functionality as: (i) a distributed computing node and, (ii) an input-output (I/O) node. The distributed computing node handles the computation, whereas, the I/O node allows the interface to sensors and actuators for acquisition and commanding.

The reason for including a separate I/O node is primarily, to permit access to all distributed computing nodes, as needed by the system. For instance consider the example of AOCS, where, sensors and actuators need a direct dedicated interface to its processors. In our case, sensors and actuators are attached to the network, not directly to processors. Therefore the I/O nodes provide the necessary conversion of the dedicated interface to the network interface. Removing the direct interface in our architecture addresses several issues, namely, the question of isolation on interfaces, back-powered, and lack of adaptability as discussed in section 3.5.

As discussed in section 2.4, a network can be implemented, using four main topologies, i.e. the mesh, star, ring, and bus. The bus topology is more appropriate because of its simple design and widely used in spacecraft network. Therefore, we opt for a bus topology in our architecture.

To make a distributed computing architecture fault-tolerant, a fault management scheme is required. From the literature as discussed in section 3.2.2, it was evident that there are two broad schemes in use i.e. centralized and decentralized. Centralized scheme is a single point of failure. Therefore, it is not considered further. In the decentralized scheme, decision power is available with a few distributed computing nodes. To elaborate this point, consider a hypothetical scenario of 10 nodes. With the decentralized scheme, suppose the decision for the fault management scheme handles on only three nodes. These nodes monitor all other nodes, and itself, for faults, by

reading node's health status. The node's health parameters comprise temperature, voltage, current values, which are included in a message being sent via the communications network, thus has a time delay. The message is read, and if it shows an error, a fault is detected. A decision is made to isolate the faulty node. This decision is broadcasted to all nodes (broadcast). The faulty node is isolated by powering it off, whose control is with the decision node. Thus, the system recovers via reconfiguring itself.

A considerable time will be required to reconfigure the system (complete the whole FDIR process) due to inherent communication delays over the network. If the decision-making control is distributed to all nodes, the communication delay can automatically be eliminated, as a node would be able to isolates itself in the case of a fault. Other nodes could reconfigure themselves as soon as the node is isolated. Furthermore, the distribution of the decision making control to all nodes makes the system more reliable than a decentralized scheme.

With this argument, we proceeded to implement a Distributed Fault Management Scheme, which was not readily available in the literature. This scheme is distinguished from the Decentralized one, in the sense, that the responsibility for making decisions is not limited to a few nodes but is available to all nodes.

Another distinguishing point is the integration of the proposed new fault management scheme with the corresponding processor. It is possible to provide this functionality within the processor or use a separate hardware block outside the processor. We opted for the second option, details of which are discussed in section 5.3.2.

4.2 System Hierarchy of the Proposed Architecture

The generic view of the proposed distributed computing hardware architecture for Fault-Tolerant Distributed Computing (FTDC) is depicted in Figure 4.1. The architecture can be extended to variable depth hierarchies, starting from the top level to the group level. At the top level, it is comprised of multiple groups connected via switches. The role of the switch is to route the data from one group to another group.

Also, it divides the physical network into multiple physical networks enhancing the communication bandwidth per node.

A computing group comprises multiple nodes (processing / IO nodes), connected via two separate networks using a bus topology. Figure 4.2 shows a group view of the architecture, where two dual redundant networks, main network and AMFT network, are used. The main network is used for communication between the processing units, while the AMFT network is dedicated to communication for the purpose of fault management. Separate networks do not only reduce the response time in case of a node failure but also provide better performance for the applications tasks. A group executes the tasks corresponding to one (or more than one) spacecraft subsystem (e.g. OBC/OBDH) and is responsible for their successful execution.

For input-output (I/O) operations such as acquiring data from sensors, and commanding actuators, no direct I/O is routed through the distributed computing nodes. It is essential to off-load the I/O operations from the distributed computing node for achieving high reliability as described earlier. Therefore, all sensors and actuators are accessed via dedicated input-output nodes.

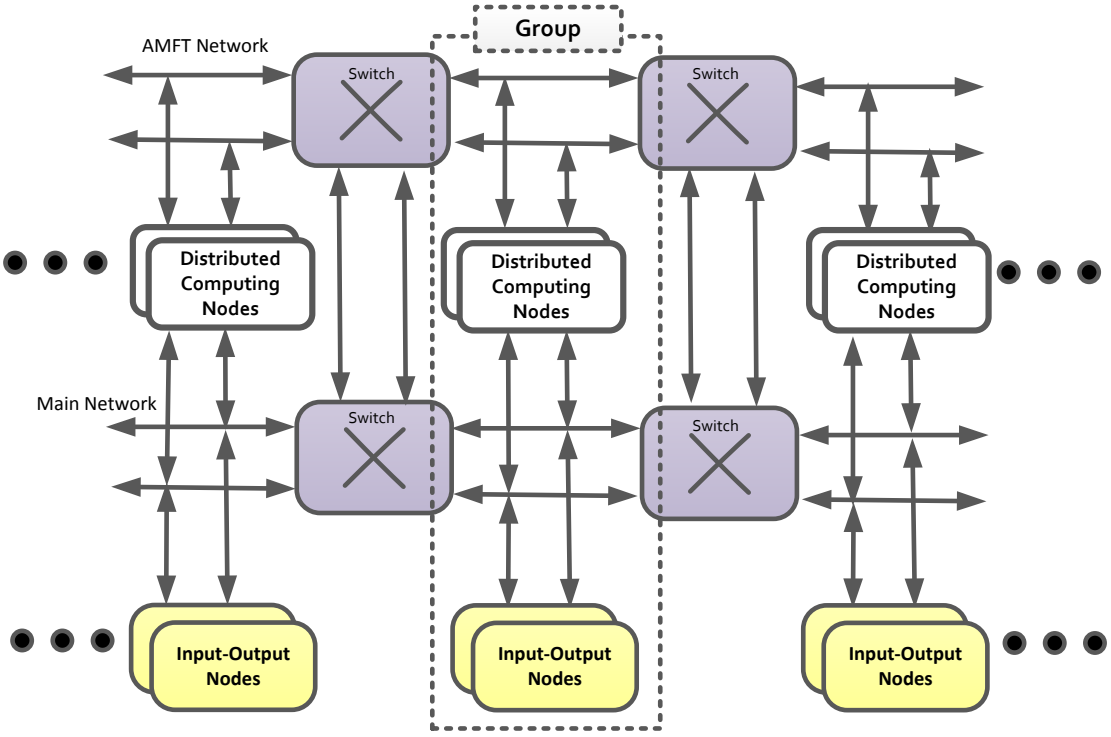


Figure 4.1: Hardware Architecture for Fault-Tolerant Distributed Computing.

4.3 Distributed Computing Node

In the proposed architecture, a Distributed Computing Node (DCN) consists of a Processing Unit (PU) and a Fault Management (FM) block as shown in Figure 4.3. The Processing Unit runs the actual application tasks while the Fault Management block is used for the fault detection, isolation, and reconfiguration of the distributed system. Unlike other techniques, we adopt a different approach whereby the fault management block is implemented on a separate physical medium, outside the host processor and is connected to a different network. In this way, the fault management block retains the detection, isolation and reconfiguration functions in case of faults in the processing units. Also, being detached from the processing units, it does not interfere with real-time requirements.

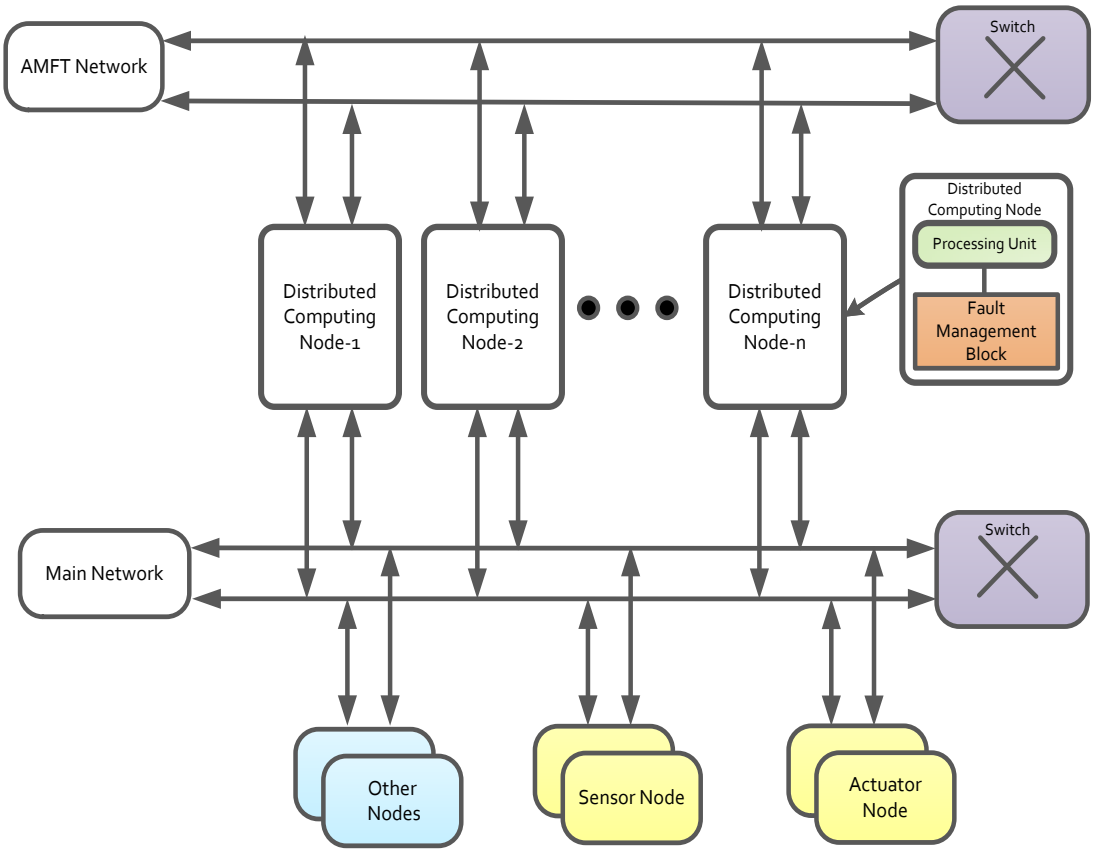


Figure 4.2: A Group View of Architecture for Fault-Tolerant Distributed Computing.

A Processing Unit consists of a processor that can be a single core or a dual-core processor. The memory system for each DCN includes a boot Read-only Memory

(ROM), Non-volatile Random Access Memory (NVRAM), Static Random Access Memory (SRAM), and Error Correcting Code (ECC) memory. The boot ROM is used to store the boot image that is needed for the initial booting of the DCN. The main program is stored in the NVRAM while the data is stored in the SRAM. An ECC protects the data stored in SRAM. Internal Random Access Memory (RAM) on each Processing Unit is used for faster data access to its processor.

As shown in Figure 4.3, the Fault Management block requires analogue and digital signals for failure monitoring. Analogue signals monitoring include temperature, current, and voltage of DCN while the digital signals monitoring includes watchdog and memory error status signals. On detection of a fault of a Distributed Computing Node, the Fault Management block generates a shutdown signal to turn off the node power, which can be turn-on later via a ground command.

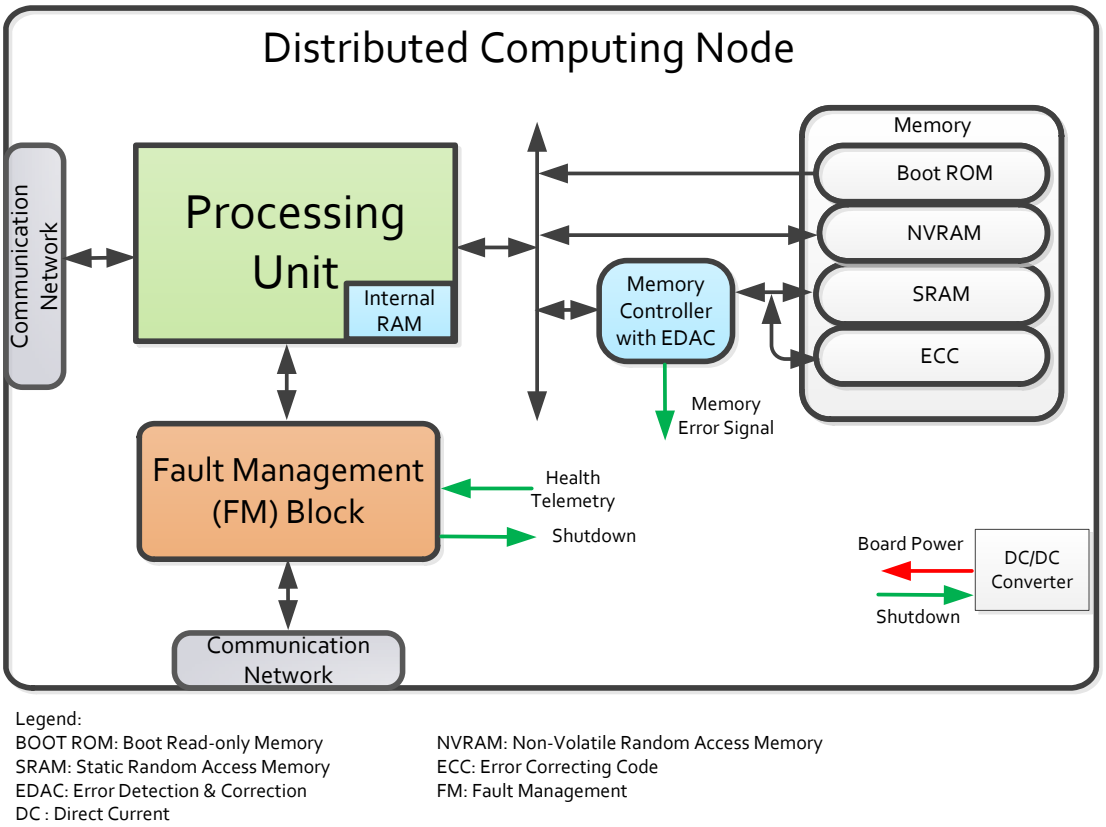


Figure 4.3: Fault-Tolerant Distributed Computing Node.

4.4 Input-Output Node

Input-Output (I/O) node is used for the purpose of sensor data acquisition and actuators commanding. Failure in the Input-Output node can be catastrophic because it is the only interface that connects the Distributed Computing Node with the sensors and actuators. Therefore, the design of the Input-Output node should be highly reliable and should have a compact size.

Figure 4.4 shows the block diagram of the proposed design for the I/O node. The design comprises a processor, Error Detection and Correction (EDAC) module, a triple modular voter for program memory, watchdog timer (WDT), an oscillator (OSC.) and a communication controller. The memory used for the program storage is based on erasable programmable read-only memory (EPROM)/electrical erasable programmable read-only memory (EEPROM) technologies. These technologies are susceptible to total ionising dose [186, 187] and may cause a functional failure. Therefore, a triple modular redundant design for the program memory is suggested that can mask such a failure. The data memory is based on the SRAM technology is more vulnerable to soft errors caused by Single Event Upsets (SEUs) [188, 189]. Soft errors are temporary and cannot cause a functional failure of the memory. Therefore, an EDAC module is included to detect and correct these soft errors in the data memory. On each data word read operation, the EDAC module checks and corrects a single bit error caused by SEUs while, during a write operation, each data word with its checksum is written to the data memory.

The network controller is attached to the processor for communication on the network. The type of the controller depends on the network and will be discussed in section 4.5. The Input-Output node runs small software routines for acquiring sensors data and commanding actuators. A watchdog timer handles hangs in the software routines. Analogue inputs correspond to the analogue sensor's input data while the digital IOs are used for the digital sensor's input and actuator's commanding. Both analogue inputs and digital IOs are routed to the glue logic that supports the necessary conversion circuitry before connecting it to the processor.

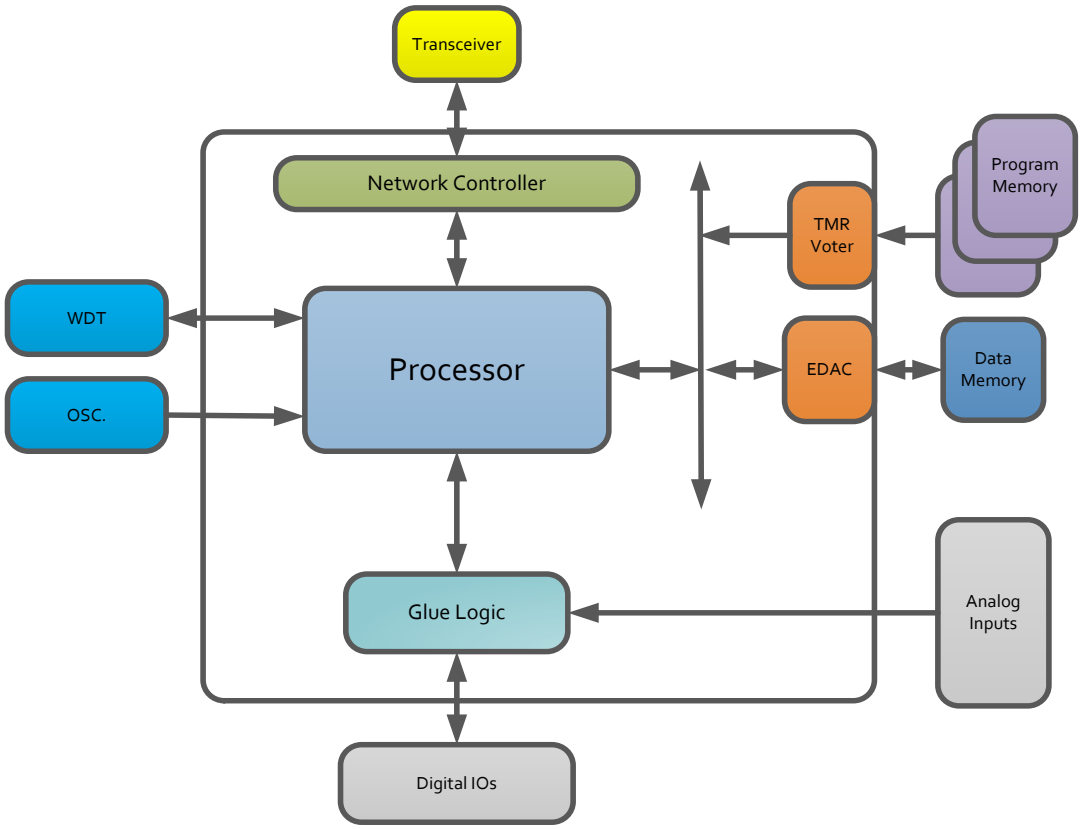


Figure 4.4: Design of Input-Output Node

4.5 Communication Network

As shown in Figure 4.5, the communication network of the proposed architecture is comprised of dual redundant networks, which are connected by multiple switches to combine two networks together to achieve a scalable fault-tolerant design. The purpose of the separate networks is to provide a dedicated bandwidth for the fault management functions and application tasks e.g. OBC/OBDH. Both networks are configured in a bus topology. To ensure deterministic access on the network, the Time Division Multiple Access (TDMA) protocol is proposed, where each bus has its own time-triggered communications Scheduler for the bus access. The redundant network is provided for fault-tolerant purposes and is only activated in case of a failure of the primary network.

The main reasons for the use of switches in the proposed architecture are: (i) to provide a separate collision domain for each group, (ii) for on-board clock synchronization and time distribution and furthermore, (iii) switches allow easier

expandability to achieve a scalable distributed computing system. The rationale behind (i) and (ii) is explained next. In the proposed hardware architecture, the bus topology of the network is a shared communication channel. If multiple computing groups are connected to such a communication medium, the effective available bandwidth per node will be decreased. To avoid this problem, switches are used to partition the overall network allowing a separate collision domain for each computing group. Also, switches are used to maintain clock synchronization and time distribution to all the nodes of a distributed system.

The communication Scheduler of each network (in a bus topology) consists of 64 time slots, as shown in Figure 4.6. Slot number 0 is reserved for the distribution of the network time and slots number 1-61 are reserved for the communication of the DCN tasks, while slots number 62-63 allow communication among two groups via the network switches.

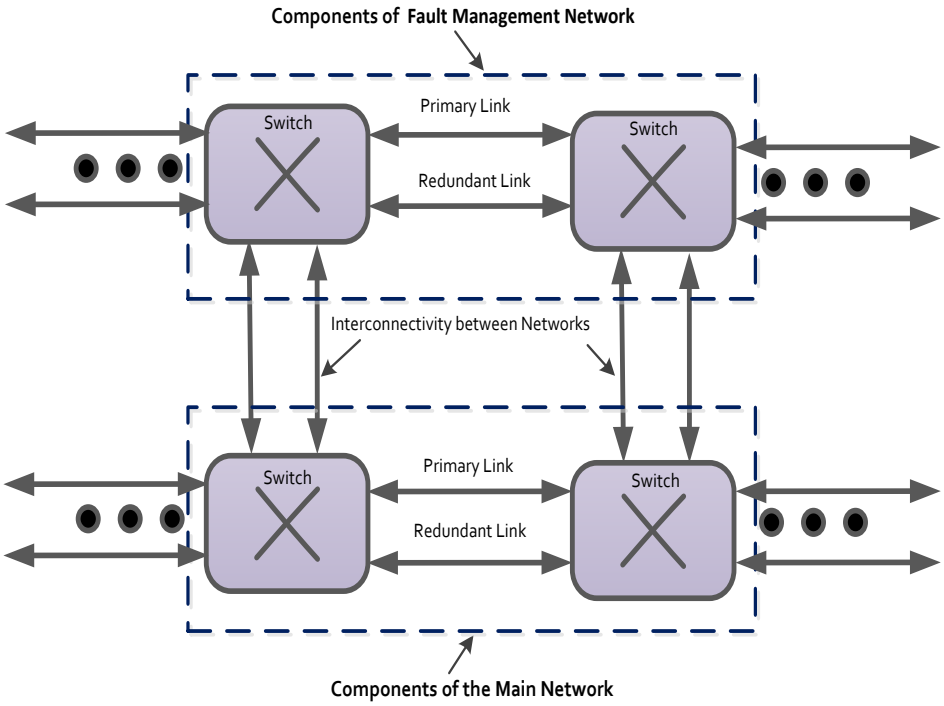


Figure 4.5: Network for the Proposed Architecture.

To fulfil the requirements, TTEthernet is selected as the most appropriate network technology because of its inherent features as discussed in section 2.4. The main characteristics of the TTEthernet are derived from the Ethernet and a large set of protocols are available, which can be tailored for adaptation to space applications.

Furthermore, high speed, multi-master, and embedded in a large number of devices, makes it a suitable choice for the future on-board computing networking. TTCAN can also be used for the proposed architecture, but its limited speed of 1Mbps cannot meet the high-speed demands of future applications. However, for low-speed network demands, it can also be employed.

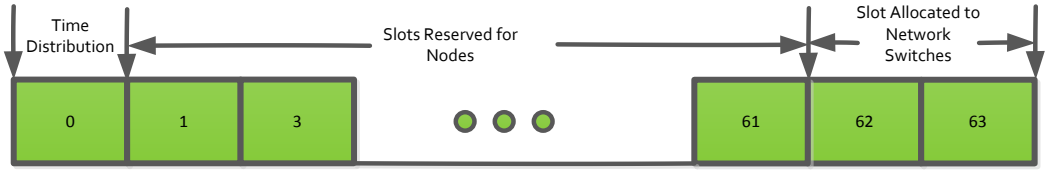


Figure 4.6: Time Slots for Network Communication in Bus Topology.

4.6 Software Stack

The software stack for a DCN, supporting fault-tolerant distributed computing is shown in Figure 4.7. It comprises two main parts: (i) a Processing Unit and (ii) a Fault Management software block. The software of the Processing Unit is further divided into the following layers:

- Application layer: This layer includes functions that facilitate the implementation of a distributed application. The details of these functions are given in section 4.6.1.
- Fault detection layer: This layer handles the detection of faults in the application software. It detects a fault and passes its information to a fault management block via a software monitoring (SM) interface. The detail of fault detection layer is covered in section 4.8.
- Both the above layers run on top of a Real-Time Operating System (RTOS) such as FreeRTOS or threadX. The use of RTOS makes it easier to manage resources and schedule tasks timely. Otherwise that would require considerable programming efforts.
- The application tasks are distributed to multiple nodes that require communication among themselves. For this purpose, the communication on the main bus is made

via a message passing interface (MPI). This interface provides send/receives functions to communicate on the bus.

The software stack of the Fault Management block consists of three layers. The function of each layer is explained below:

- The first layer corresponds to the implementation of the Fault Management functions. It is responsible to detect a failure via either the hardware monitoring (HM) interface or via the software monitoring (SM) interface.
- The second layer supports task migration. It manages task state data, node/task tables, and coordination among the fault management layer and the communication layer.
- The third layer handles communication among the Fault Management blocks. It allows each Fault Management block to access the bus in its dedicated time slot.

Details of the Fault Management (also called AMFT) algorithms and their software implementation are given in Chapter 5.

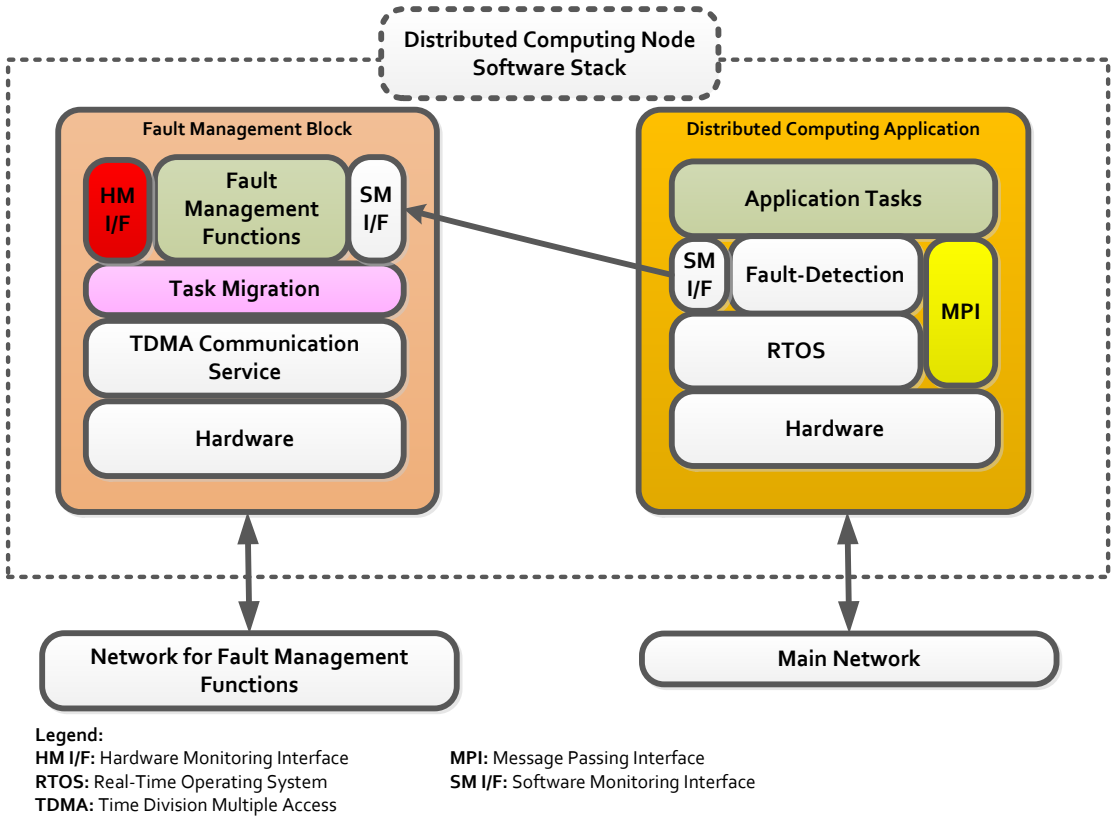


Figure 4.7: Software Stack.

4.6.1 Distributed Computing Application

A distributed application decomposes into smaller tasks that are distributed to the multiple processing units. The code for every task is present on every DCN, but a task only activates/runs on a single node at any one time. A simplified block diagram for the application software is shown in Figure 4.8, which shows a top-level view of the DCN application layer. The distributed application comprises application tasks and support tasks. Application tasks represent the actual distributed application and the total number of tasks to be executed by the system can be varied, as well as the characteristics of each task. The main task characteristics are periodicity, duration and state data length. The state of a task comprises a set of values that must be preserved for future execution of the task. Support tasks are the tasks which provide support in terms of communication and activation/deactivation of the main tasks.

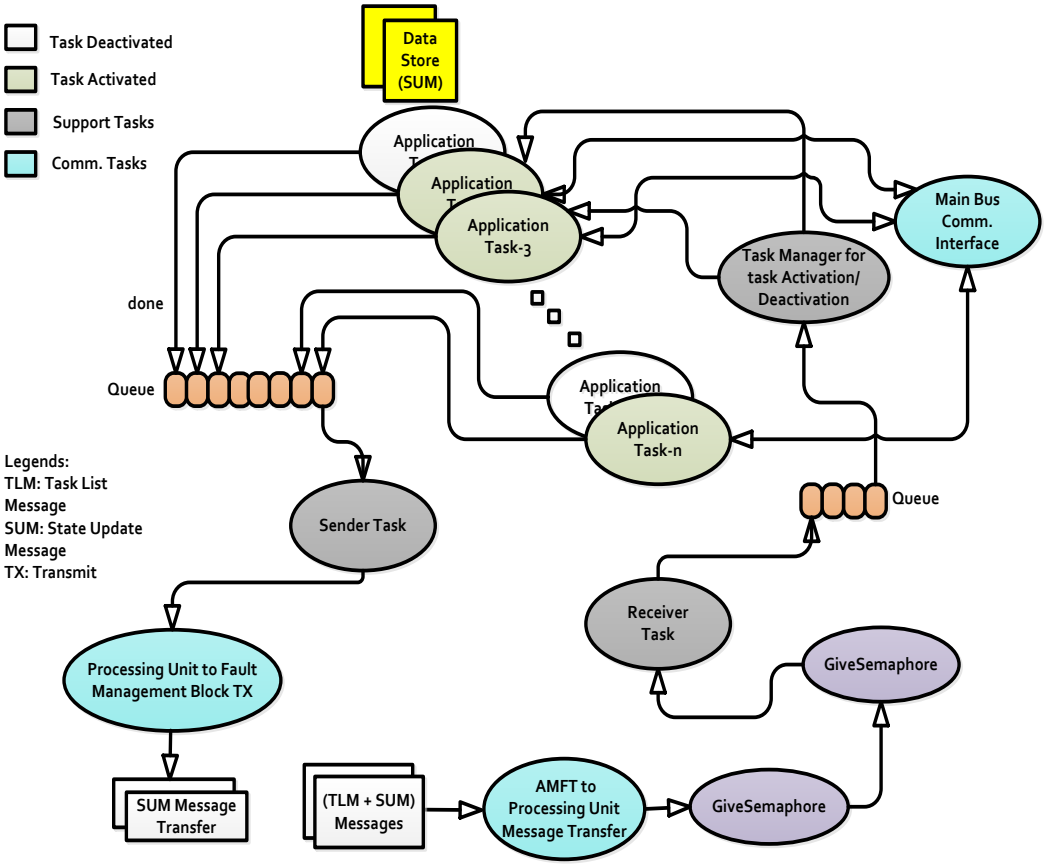


Figure 4.8: Distributed Application Software Block Diagram.

4.6.1.1 Application Tasks

Each application task is used to represent an activity to be carried out by the distributed system, to fulfil the intended system functionality such as OBDH, AOCS or payload functions. Whenever a task runs, it output results in the form of state data values, which are stored, and the support task 'sender task' is informed via a queue to send a message to the fault management block regarding the updated data values.

For the purpose of the prototyping in section 7.4, the particular function performed by each application task is not taken into account. It is sufficient that each task is required to run for a given duration, with a given periodicity, which may be different for each task. Also, each task has a "state" which is updated when the task executes. It may be, for example, the previous values of the AOCS angles or velocities that are required for future calculations. In prototyping, the state data Δ_{SD} consists of a series of bytes, and the operation performed is to increment the value of each of these bytes one by one each time the task executes.

To enable task states to be transferred between nodes for task migration using State Update Messages (SUMs) are used. The task state is saved to a location which is accessible by the AMFT Sender task that creates and sends the SUMs containing the state data, Δ_{SD}. Once the application task updates its state, it notifies the AMFT Sender task that a new state is available, so the state can be sent to the other nodes. It is also required that the task state data is initialized within its assigned memory locations prior to starting a task on the node following migration of the task from a failed node. This is done by the application Task Manager task. A template pseudo code for an application task is given below:

Pseudo Code:

```
1  Application Task (task i) {
2      while(1){
3          wait for x seconds
4          update state data, i.e. increment each byte comprising of state
5          send message to AMFT Sender task: new state update message
6          }
7  }
```

4.6.1.2 System Support Tasks

System support tasks provide support to the application tasks. They consist of communication tasks (sender and receiver tasks) and an activation/deactivation task. Communication tasks handle the communications with the AMFT and the distributed processing unit while the activation/deactivation task controls the execution of the application tasks based on the requests of the AMFT unit.

Activation/deactivation: The Task Manager task activates and deactivates application tasks as required, based on task lists received from the AMFT Receiver task. For each application task in the received task list that is not already running on the node, it initializes the task's state data Δ_{SD} using the data received in the State Update Message following the Task List Message, and then starts the application task. It stops any tasks that are running but are not included in the received task list. For the transfer and receipt of state data values and Task List Messages (TLMs), various queues are used. Data queues enable safe communication among the two tasks.

Pseudo Code:

```
1  Task Manager Task(task list) {
2     while(1){
3        wait for message from AMFT Receiver Task: task list message
4        Receive task list message and state update message
5        for each application task in the task list
6        {
7              if the application task is not already started
8              {
9                    Get state from SUM (if any)
10                   Initialize task state data
11                   start the task
12             }
13       }
14       for each task not in the task list that is running
15       {
16             stop the task
17       }
18 }
```

Communication Tasks: The communication tasks handle the transfer of data. There are two primary interfaces that handle communication.

Processing Unit Network Communication: This interface is used for the communication of application tasks over the main network. It is a software interface that is accessible from each of the application tasks. It enables all tasks to execute in parallel by exchanging data messages to complete an overall task. The following shows the pseudo code for this task.

Pseudo Code:

```
1   Message Passing Interface Task(task i, data) {
2       while(1){
3           Msg = formMessage(task i, data)
4           send(Msg)
5                   if(msgReceive == true){
6                           place in buffer reserved for task i
7                           indicate the relevant task i for received message
8                   }
9           }
10  }
```

AMFT and Processing Unit Communication: This software interface sends and receives data between the processing unit and fault management block (AMFT). This interface depends on the physical hardware interface between the Fault Management block and a Processing Unit. It is mainly includes two tasks; AMFT Receiver Task and AMFT Sender Task.

AMFT Receiver Task: The AMFT Receiver task waits for data to be received from the AMFT via the Processing Unit-AMFT interface. If this data is a followed by State Update Messages, these data are passed to the Task Manager for activation/deactivation of tasks. The following shows the pseudo code for this task.

Pseudo Code:

```
1   AMFT Receiver Task{
2           while(1){
3                   wait for message from AMFT
4                   if message is task list message
5                   {
6                           receive task list and state update messages
7                           send messages to Task Manager
8                   }
9           }
10  }
```

***AMFT Sender Task*:** The AMFT Sender task sends State Update Messages to the AMFT, created from state data sent to it from application tasks. The following shows the pseudo code for this task.

Pseudo Code:

```
1   AMFT Sender Task {
2           while(1){
3                   wait for message from Application task: new state data
4                   create state update message
5                   send state update message
6           }
7   }
```

The detail of the application software implementation is given in Appendix E.

4.7 Fault Management Scheme

In the proposed architecture, fault management functions are distributed locally to each node as shown in Figure 4.9 and implemented as a separate block named Adaptive Middleware for Fault Tolerance (AMFT). This distribution of fault management functions allows each node to detect and isolate its faults locally without

the intervention of a centralized/decentralized node. It reduces the amount of time involved in the decision, thus making the system more responsive by reducing the reconfiguration time as desired.

To achieve high reliability, a hybrid fault detection method based on hardware and software is proposed in this thesis. This method covers hardware as well the software faults. In case of hardware fault detection, the health status of each node is monitored to identify a fault condition. Health monitoring includes observing the analogue (temperature, current and voltage) and digital signals (watchdog, IO) of each node. For fault detection, these monitored values are compared with pre-defined values and statuses. Violation of any pre-defined thresholds or combination of these is considered as an indication of a fault.

In case of software fault detection, symptom-based anomaly detection methods are used. In our case, we applied this method to detect faults in the application software as well as hardware faults propagated to software, as described in section 4.6. Details of this method are given in section 2.3.2. The symptom-based detection method has a limitation which is that it does not protect the processor from silent data corruption errors. Therefore to overcome this, new algorithms for silent data corruption are proposed and explained in section 4.8.

The next step after detection is fault isolation. Fault isolation means to disconnect the faulty node from the rest of the system. There are two options either to power off the faulty node or to disconnect it from its interfaces (IOs). We adopted the former approach for its simplicity.

Now that the fault is isolated, the tasks must be migrated to other nodes within a minimal period of time. Each task should be executed starting from the point where the processing was interrupted by the node failure. This information is stored in the form of information about program states. The corresponding technique, which is called program checkpointing, is computationally intensive and requires a very high speed communication network to transfer frequent checkpoints. Another method is to monitor the outcomes of each task and store the state information, which is called data checkpointing. In this method, the task is not executed from the point where it was interrupted, in fact, the task is re-executed. During the re-execution the task know its data, whose information is provided in the form of data checkpointing. This method is

much simpler than program checkpointing, in terms of a reduced amount of state variables that need to be stored and, therefore, it was adopted in the proposed scheme.

In the suggested scheme, the migrated task outcome is passed to other nodes via the AMFT network, where the checkpointed data (states) is being maintained. The other node has access to the task outcome, which serves as its input now to execute the migrated task. However, when the fault occurred, the task outcome was not updated, and, therefore, the other node gets the previous task outcome and executes the migrated task based on that. Therefore, after the first task execution on the new node there will be a slight deviation from the actual task outcome and results, which may also affect any dependant tasks. Task migration for inter-dependant tasks is application specific, and we consider this issue in our case study for AOCS in section 8.5.1.

During subsequent re-executions the migrated tasks are able to compensate for any deviations caused by the node failure and normal operations are resumed. This functionality (correct state storage/management) is carried out by AMFT. The details of the AMFT design and implementation are given in chapter-5.

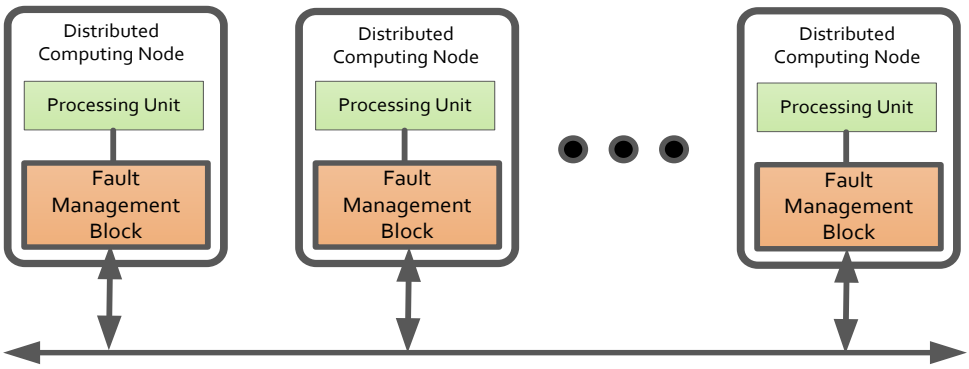


Figure 4.9: Fault Management Scheme.

4.8 Fault Detection

The symptom-based fault detection approach to detect software faults is adopted among the various methods as discussed in Section 2.3.2. It requires fewer resources because current processors are inherently designed to detect symptoms. However, this method is limited to known symptoms only. This includes fatal traps─illegal memory access, misaligned memory access, illegal instruction execution. Symptom-based

detection method could not handle errors caused by silent data corruption (SDC). SDC errors are those errors, which could affect the program flow and its data contents, if they persist. This may cause a system to produce the wrong output. To detect these errors, two algorithms are proposed. The first one is a selective duplication algorithm that protects the software against transient SDC errors. The second algorithm detects permanent SDC faults by explicitly moving data patterns in storage / functional elements.

4.8.1 Transient SDC Error Detection

In the transient SDC detection algorithm, proposed here, it is assumed that the application program consists of multiple functions, where each function is protected against SDC errors. The objective is to detect a fault in the processor microarchitecture, and so it is further assumed that the memory is fully protected. Figure 4.10 shows the description of the algorithm. In this algorithm, all global and formal variables, including the loop index involved in the computation, are stored on a spare storage before being used for execution. Run the program for both the formal and stored variables. If they produce different results, a transient SDC error in the computation of microarchitecture is reported. Contrary to full duplication, the adopted selective duplication method only selects a particular part of the program and its variables at any time. Once the program executes, the storage for duplication is released. This method improves performance and consumes small data memory.

Selective Duplication Algorithm:

```
1   spare storage 's' = copy read variables related to function A
2   call function A with stored variable set 's' '
3   functiona A (formal variable 'f'){
4      execute function A on formal variables set 'f'
5      compare both execuation ( formal set 'f' and stored values 's')
6     if (∀ 'f' ≠ ∀ 's')
7       return sdc_error;
8    else
9      retuen 0;
```

Figure 4.10: Algorithm for Transient SDC Errors.

4.8.2 Permanent SDC Fault Detection

This section presents an algorithm for detection of permanent SDC faults. These faults include stuck-at-bit and bridging faults. Stuck-at-bit faults are those faults where a bit is stuck at logic one or zero, irrespective of the actual value of the element. While a bridging fault is a crossing of two signals that results in an 'OR' or 'AND' logic operation, which forces adjacent bits to change to either logic '1' or logic '0'. Bridging faults are difficult to detect because they depend upon the physical routing of the connections. To detect these faults, the following data patterns are selected.

- For stuck-at-bit faults, data patterns of all zeroes '0x00000000' and ones '0xFFFFFFFF' are used. These patterns can easily detect errors in registers/ latches caused by bits stuck at zero and one.
- For bridging faults, data pattern of '0x55555555' and '0xAAAAAAAA' are used. This pattern of alternating zeroes and ones is suitable to detect the crossing of bits.

Figure 4.11 shows the proposed algorithm for detection of permanent SDC faults. The input of the algorithm is the contents of functional elements such as registers, latches and arithmetic logic unit (ALU), while the output of the algorithm is the faulty components. In an iteration of the algorithm, storage/functional elements are checked by explicitly moving a pattern. The element is considered faulty if the defined conditions in the algorithm are not met. This process repeats for the rest of the data patterns with all storage /functional elements. At the end of the algorithm, numbers of identified faulty elements are returned to the calling function as a signature value. In order to improve the performance of algorithm, it is recommended that storage registers─R1, R2, A, B, data_pattern─ and a comparator, which are being used in the execution of the algorithm, should be triplicated.

Detection of Permanent SDC Faults:

1	Input = $Microarchitecture\ Elements\ (Registers, Latches, ALU\ etc.)$
2	output = $Faulty\ Elements$
3	$signature \leftarrow 0;$
5	$FUNCTIONAL\_ELEMENT \leftarrow ALU \quad STORING\_ELEMENT \leftarrow REGISTER/LATCH$
6	$for\ i = 1\ to\ i < data\_pattern$

```
7        for j = 1 to j <  num_elements
8            if(element[j] == STORING_ELEMENT)
9                element[j] ← data_pattern[i]
10               if(element[j] ≠ data_pattern[i]
11                   signature + +;
12           else if(element[j] == FUNCTIONAL_ELEMENT)
13               A ← data_pattern    B ← data_pattern
14                   R1 ← 0; R2 ← 0; // registers
15                   R1 ← A ^ B; R2 = A ~^ B;
16                   if( R1 ≠ 0x00000000 || R2 ≠ 0xffffffff)
17                       signature + +;
18   return signature;
```

Figure 4.11: Algorithm for Permanent SDC Faults.

4.9 Summary

A novel distributed computing architecture is proposed in this chapter, which addresses the primary objective of providing both reliability and High Performance Embedded Computing for mission critical applications. It overcomes the issues of current fault-tolerant computing approaches, described in section 3.5. The main features of the architecture are its cooperative distributed behaviour, distributed nodes, communication bus, and a new fault management scheme. The system has a hierarchal structure, which is modular and scalable.

The core components are the DCNs, which are segregated into separate application and fault management functions. This enables the high-performance activities carried out by the processing unit and the fault management to be executed in parallel. Other peripherals interface to the DCNs via I/O nodes.

Two communication networks are used, separating the network for processing and fault management, thus incorporating high reliability, without compromising performance. Standard high data rate (reliable) networks are identified as candidates for the network implementation. The network follows a deterministic behaviour by deploying a TDMA scheme.

A novel fault management scheme is proposed, where tasks are seamlessly migrated to other nodes in the case if one of the nodes fails. To implement these features, an AMFT block is proposed, the design of which will be discussed in the next Chapter 5.

The design of the whole software architecture is described, which enables the correct functionality of the proposed architecture for a given distributed embedded computing scenario. Faults occurring in the application are also detected, for which new algorithms have been designed.

To conclude, a reliable High Performance Embedded Computing architecture is proposed. This architecture is further assessed and analysed in Chapter 6.

Chapter 5

Adaptive Middleware for Fault-Tolerant Distributed Computing

In this chapter, an adaptive middleware is proposed to implement the fault management functions as briefly described in section 4.7. The goals of the middleware design are discussed in section 5.1. In section 5.2, algorithms for the middleware functionality to enable distributed computing are presented. Details on the design of the middleware will be described in section 5.3. In section 5.4, failure scenarios in case of distributed processing are covered. The implementation of proposed middleware for the fault management scheme is described in section 5.5. Section 5.6 is devoted to general discussion. Section 5.7 concludes the chapter by presenting contributions towards the state of the art.

5.1 Design Goals

Middleware manages interactions between the application and the underlying system software such as the Operating System and the device driver [190], thus acting as an intermediate layer. In general, it provides Quality of Service (QoS) management, fault-tolerance, resource allocation and timeliness guarantees to distributed applications., Each middleware component is specifically designed to support a particular application.

The various implementations of middleware design for distributed fault-tolerant systems are proposed as part of the ISIS, Mars, Delta-4 and Mach OS projects [191], [3], [192],[193]. These middleware have been designed to suit the replication based approach (see section 2.2.1) and require a large memory size. The object-based middleware implementation─ referred to as fault-tolerant Common Object Request Broker Architecture (FT-CORBA)─ was proposed by the Object Management Group (OMG). The design of FT-CORBA was aimed to operate in a client-server model and support active and passive replication styles. Another implementation of middleware, focused on providing an adaptive failover strategy and overhead management approach is described in [194]. This middleware is designed for passive replication to handle soft real-time applications. As discussed in section 2.2.1, replications cannot meet the emerging demand for high-performance distributed computing. These demands require a fault-tolerance technique that utilizes the inherent availability of multiple processors. Unlike redundancy, fault-tolerance by task migration is a new concept for critical distributed embedded systems. It is a promising technique that can provide a balanced approach to high-performance and high reliability under the constraints of limited resources.

The proposed fault management scheme uses a middleware concept and falls in the category of embedded system employing distributed computing with fault tolerant capabilities, meeting soft real-time requirements, which are essential in space applications. Contrary to the traditional fault-tolerant middleware for distributed systems as discussed earlier, the proposed adaptive middleware for fault-tolerance (AMFT) is novel, and it is the first effort to support task migration for distributed computing applications.

The objective is to design a middleware that should have the following features.

- The middleware design should be adaptive, up to a considerable extent without comprising system reliability and deterministic behaviour. The design should support static, adaptive behaviour, whereby for each possible failure scenario, a distributed system configuration is pre-stored in the form of tables.
- Instead of the replication approach, discussed in section 2.2.1, tasks are migrated to compensate for failures in the distributed system. It eliminates the amount of

computational resources used during the normal operation by executing only one copy of the task at any time.

- The middleware should be able to support state resumption of a task when it is migrated to another node in the case of a failure.
- A TDMA based communication protocol is implemented in the middleware to support a deterministic channel access and a bounded distributed system reconfiguration time in the event of a failure.
- Due to its deterministic communication protocol, the middleware can run on top of any wired or wireless communication protocols.

5.2 Algorithms

Algorithms that are based on the master-slave approach for fault detection, isolation and reconfiguration of a distributed system were presented in [195]. Due to a single point of failure, the algorithms were modified, and a new set of algorithms based on a distributed coordination were developed [196, 197]. In the distributed coordination approach, each AMFT has a complete knowledge of the working nodes in a group. As described in section 4.2, a group is a set of nodes working collaboratively to accomplish a bigger task. Within a group, each AMFT has a node table stored in its local memory. This node table contains an entry for each distributed computing node with the current operating status of that node (active or inactive) and the relative communications time slot for that node to communicate on the AMFT network. Each AMFT maintains its node table based on the messages it receives from other group members via the AMFT network. All messages are multicast, so all group nodes connected to the network are expected to receive these messages and update their node tables accordingly. On receipt of a HeartBeat Message (HBM) from an AMFT block, it is known that the sending node must be active. Conversely, a node is determined to be inactive if it sends no message during its allocated communications slot or if it sends a message in another slot or sends a fault message.

The functionality of the AMFT is divided into four different phases; start-up, normal operation, AMFT fault handling, and task migration. After the start of a node, the start-up phase is the first phase where a node knows about the rest of nodes and

connects itself with the other computing nodes. In normal operations, each node runs tasks and sends messages to other nodes about its health status in the form of HeartBeat messages. The third phase is fault handling phase that is only activated on detection of a failure in a node. Following the fault handling phase, a node entered into the task migration phase where it migrates the tasks of the faulty node to other nodes. To accomplish these phases, algorithms are proposed and explained in section 5.2.1 to 5.2.4.

5.2.1 Start-up

Figure 5.1 shows the AMFT start-up algorithm. Upon start-up of the AMFT block, it attempts to establish a connection with its processing unit. If the processing unit is not available (e.g. it is switched off or has failed), the AMFT will enter a fault handling mode, using the fault handling algorithm as shown in . If a connection is established successfully, the AMFT network communications slots for each node are read from the Node Table stored in local memory. These time slots are relative and, therefore, provide no absolute timing information for inter-AMFT communication. An additional step is required for a node to determine when it is allowed to transmit on the AMFT network, and this is the final stage of the start-up algorithm. The AMFT listens for heartbeat messages from other nodes on the AMFT network for a time equal to one complete communications cycle. If a heartbeat message is received, this provides the absolute timing information required. When combined with the relative slot times stored in the Node Table, the AMFT has knowledge of the absolute communications time slots for every node. If no heartbeat message is received during the listening time, the AMFT assumes no other nodes are active and transmits its heartbeat message immediately. The start-up time depends on the number of nodes and slot time per node. The more the node or slot time, more will be the start-up time.

5.2.2 Normal Operations

Figure 5.2 shows the AMFT algorithm used during normal operation. During normal operation, in every communications cycle, the AMFT sends its heartbeat message via the AMFT network during its communications slot. Each AMFT block is restricted to send messages to other AMFTs in its communication time slot. If a fault has been

detected in the AMFT's processing unit, a fault message is sent instead of a heartbeat message. AMFT also listens for messages received from other AMFTs in their allocated communications slots, and updates its Node Table based on whether a particular node has sent a heartbeat message, a fault message or no message at all. If a node's status has changed, i.e. it has just become active or inactive, tasks will need to be migrated based on the new group configuration using the task migration algorithm. Communications slot timing within the AMFT is also updated when messages are received from other nodes, to maintain synchronization between all the nodes. Following its heartbeat message, each AMFT sends a State Update Message containing the most recent state data for each task being executed on the AMFT's corresponding processing unit.

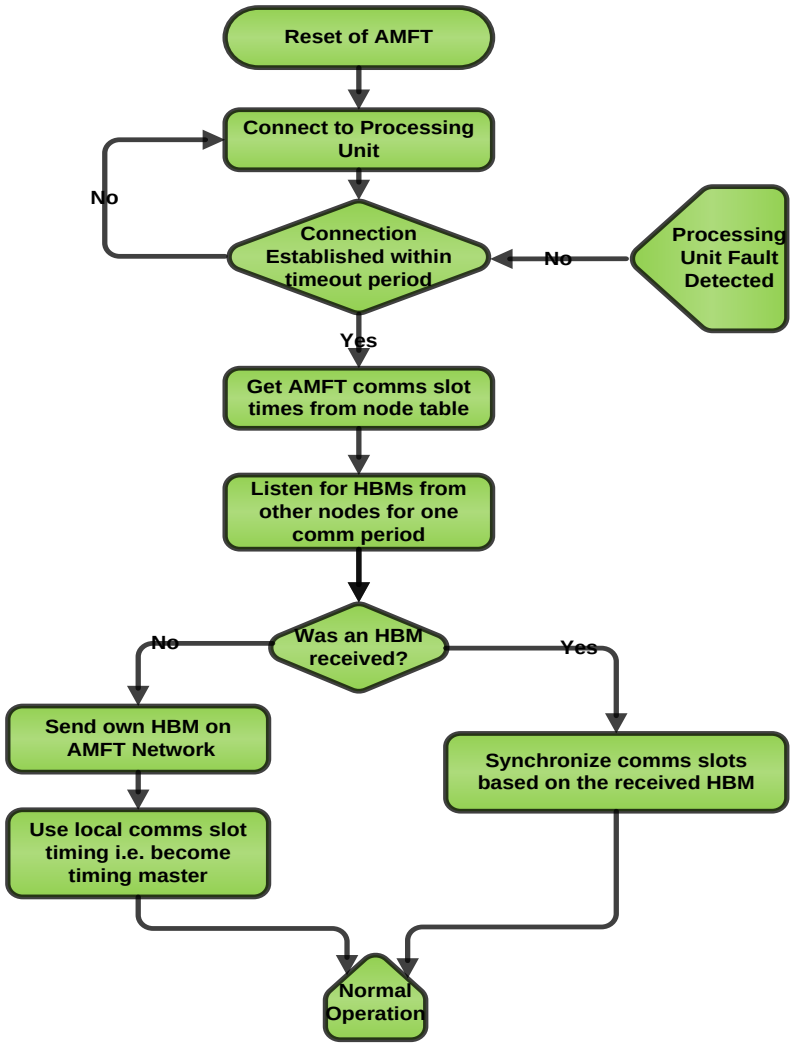


Figure 5.1: Algorithm for AMFT Start-up.

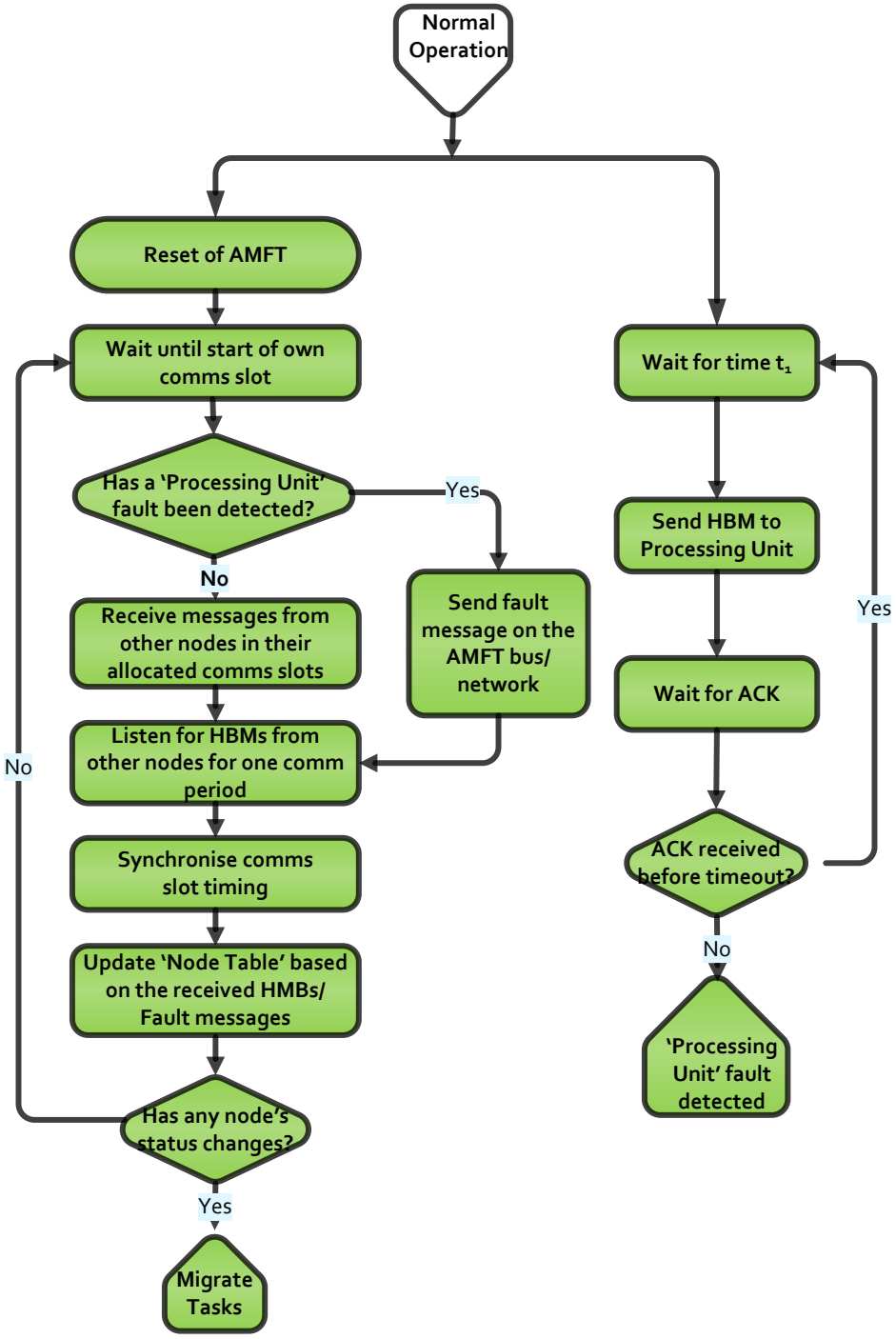


Figure 5.2: Algorithm for AMFT Normal Operations.

5.2.3 AMFT Fault Handling

Figure 5.3 shows the fault handling algorithm. The fault handling algorithm is used when a fault has been detected in the AMFT's processing unit. Processing unit itself

uses symptom-based fault detection mechanism [139]. An interface between the processing unit and AMFT is used to indicate a fault condition inside the processing unit.

The first step taken following fault detection is to isolate the processing unit from the network so that it cannot interact with the rest of the system. The exact actions to be taken will depend on the nature and severity of the fault, e.g. double bits error in program memory, failure of on-board WDT, and extreme condition on temperature. Once a severe fault is detected, campaign for the task migration is immediately started.

It then sets a flag to indicate that the AMFT should send a fault message on the AMFT network rather than a heartbeat message so that the other nodes are aware of the fault. Once the fault flag has been set, measures may be taken to attempt recovery of the processing unit. If autonomous recovery steps are permitted, then these may be attempted first. However, it may be preferred to take manual recovery steps following an investigation of the fault, which may be carried out sometime after the fault occurs. If the processing unit is recovered, either autonomously or manually, processing unit is reconnected to the network. The AMFT's fault flag is reset so that heartbeat messages are again transmitted on the AMFT network, and the node can be reintegrated into the group. However, a failed node is powered off after the recovery attempt.

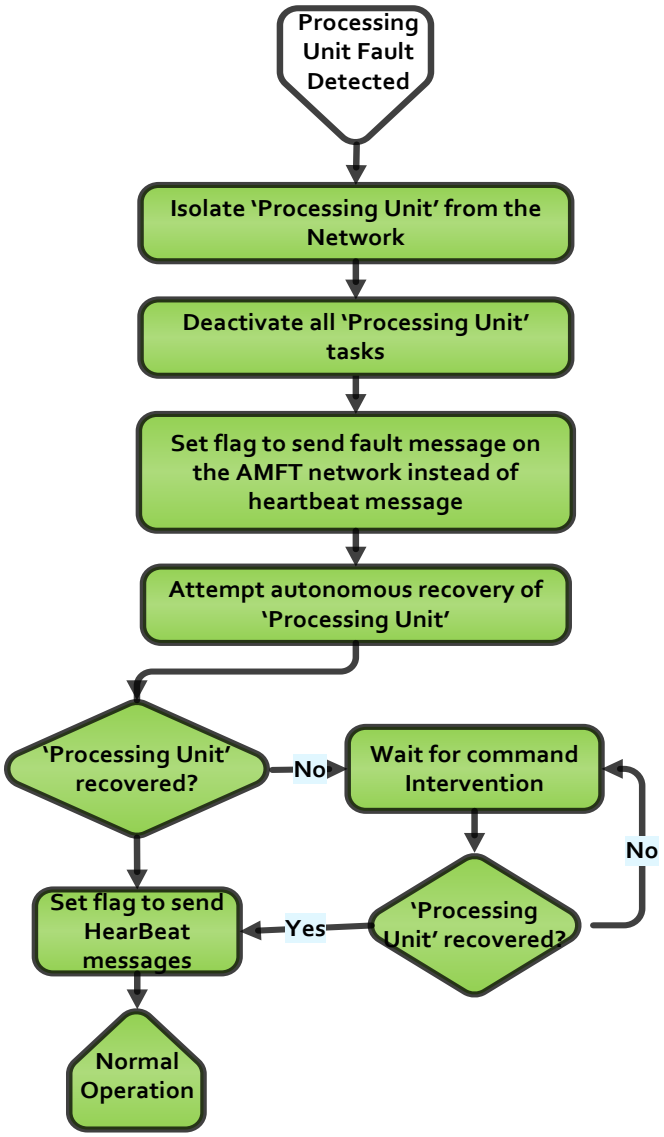


Figure 5.3: Algorithm for Fault and Recovery Handling.

5.2.4 Tasks Migration

The migration of tasks to nodes within the distributed architecture is achieved through the use of Task Migration Tables. The allocation and scheduling of tasks across the nodes are determined through analysis at design time, and the resulting distribution of tasks to nodes is stored in a table that is accessed at run-time. An alternative approach would be to determine dynamically the distribution of tasks to nodes and scheduling at run-time, but a static approach requires less processing and provides greater assurance that the system can meet real-time deadlines. One Task Migration Table is created for

each possible group configuration. The complete Task Migration Table is stored in each AMFT unit's local memory.

Figure 5.4 shows the algorithm used for task migration. It is entered into task migration whenever a change is detected in the operational status of a node (e.g. if a fault message has been received from an AMFT via the AMFT bus). The Task Migration Table to be used is selected based on the current operational status of all nodes. By reading the table, each AMFT can then determine which tasks are to be executed by its processing unit. An AMFT unit then informs its processing unit of the tasks to be executed by sending it a Task List Message (TLM). It also sends the most recent state data for the tasks in the task list, so that the processing unit can execute the tasks with their latest states.

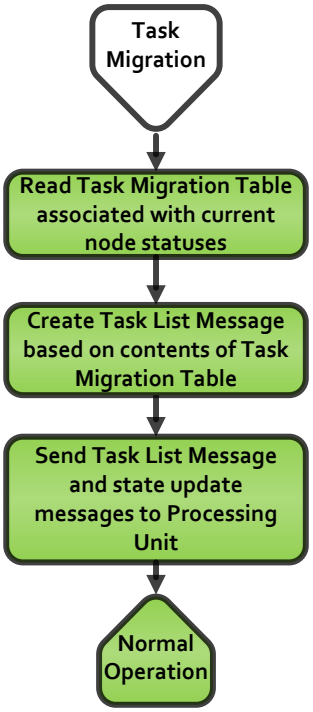


Figure 5.4: Task Migration.

5.3 AMFT Design

This section describes the design of the AMFT, a middleware that is proposed for the on-board spacecraft distributed computing.

5.3.1 Top-Level Design

Figure 5.5 shows the top-level design of the AMFT, which consists of four main modules; 1) Fault Monitoring and Detection Module 2) Target Fail-Over Node Selection 3) State Checkpointing 4) Communication Module.

Fault Monitoring and Detection Module: The Fault Monitoring and Detection module is used to detect the failure of the processing unit within each distributed node. As shown in Figure 5.5, this module is used to monitor the software and hardware faults of the processing unit. For software faults, symptom-based detection method, as described in section 4.8, is proposed to be implemented inside the processing unit that inform the AMFT for any abnormality via a software monitoring (SM) interface. This interface can be a shared memory interface or a simple UART interface. Also to the software faults, this module also monitors the temperature, current, watchdog and double bit errors signals. The final decision of a node failure depends on the fusion of information provided by software and hardware fault detection.

Target Fail-Over Node Selection Module: This module is capable to select the target node for migration of the tasks. A selection of the target node for task migration is static [198], whereby the preconfigured tables are used. Based on the fault of a particular node, this module determines the tasks that need to be run by its associated processing unit. After this, a TLM is sent to the processing unit for execution of the tasks.

State Check-pointing Module: State check-pointing is a mechanism in which a consistent state of a task is stored. If a task or its associated processor fails, then the task needs to be started on another node. In that case, the stored state helps the task to resume its execution from that point onward rather than reset from the initial point. Inside the AMFT block, a module State Checkpointing is dedicated to this type of functionality. This module handles sending, receiving and storing the check-pointing information. This module receives checkpointing information from its associated processor and AMFT network. This information is stored on each node and later transmitted to other AMFTs.

Communication Module*:* The communication mechanism adopted inside the AMFT over the AMFT network is TDMA. Each node has its time slot for communication on

the AMFT network, which is allocated at each node start-up. TDMA mechanism ensures deterministic system behaviour in case of a failure. It also eliminates the need for underlying deterministic communication protocol that enables the AMFT to use with any non-deterministic protocols.

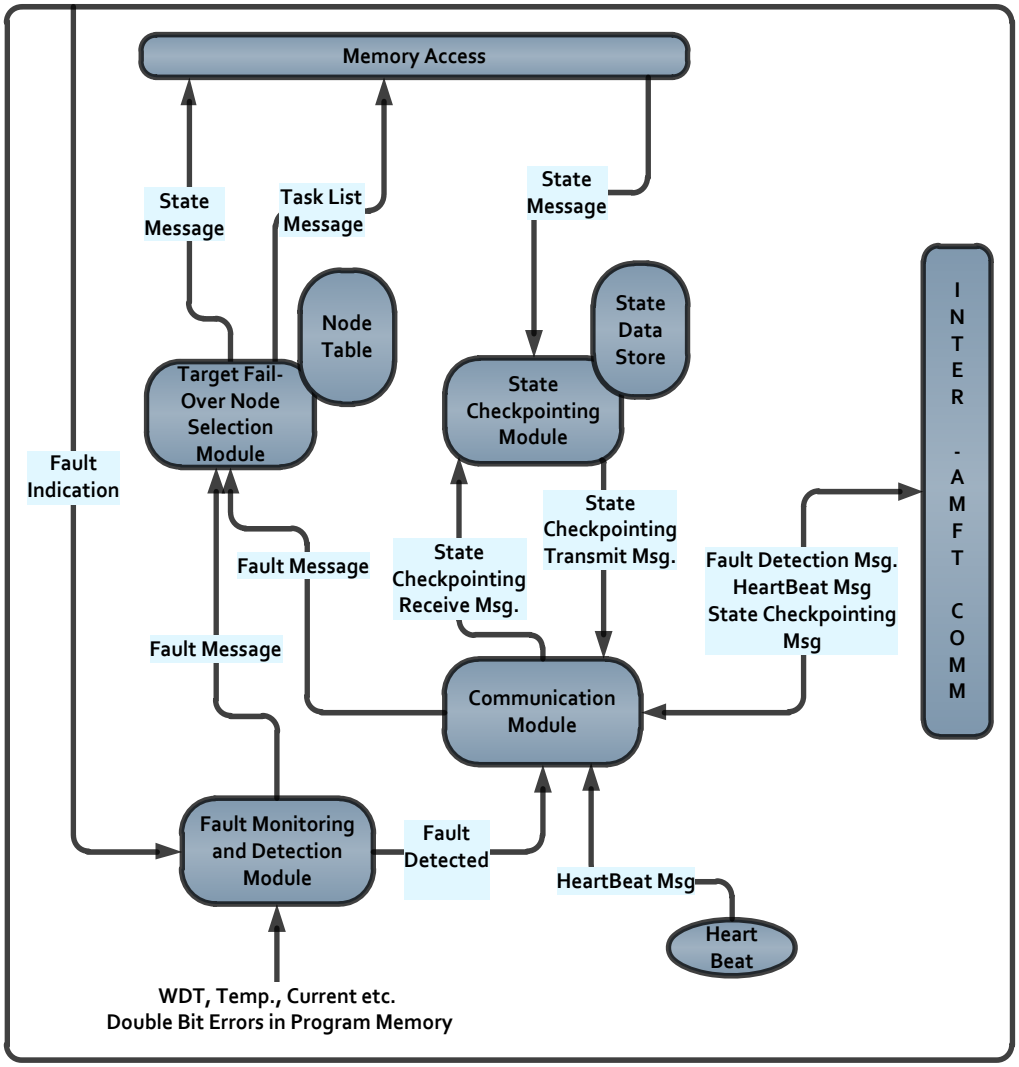


Figure 5.5: AMFT Top-Level Design.

5.3.2 Implementation Approaches

AMFT is particularly designed to support distributed computing of resource constraint distributed embedded systems, such as spacecraft distributed computing. AMFT is attached to each distributed processing unit as a separate hardware block, or it can be integrated within a Multiprocessor System-on-chip (MPSoC) of a distributed node as

shown in Figure 5.6. An MPSoC based approach is more beneficial because it requires less area, size, mass and also consumes less electrical power.

As mentioned earlier, the client-server approach is not suitable for resource constraint distributed embedded systems due to its connection setup time, It also allows only one way service from server to client and its server can act as a single point of failure. Therefore, a distributed coordination approach was adopted for the communication among the AMFT blocks via a separate network. This distributed coordination is accomplished through consistent messages among the AMFT blocks which will be explained later in this Chapter.

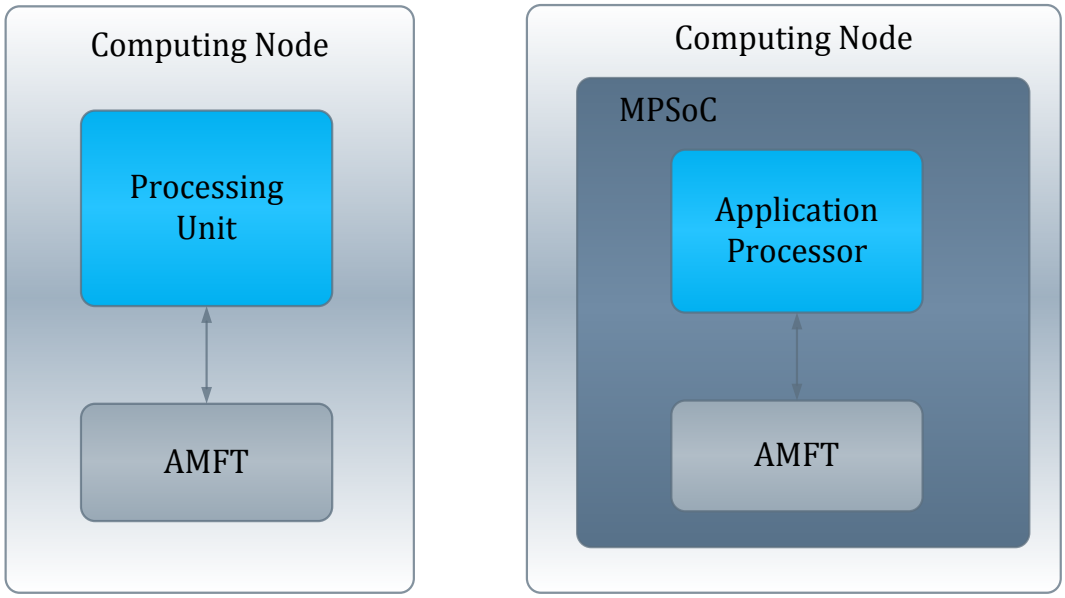


Figure 5.6: AMFT Block: Implementation Approaches.

5.3.3 AMFT Messages and Formats

This section describes the messages transferred between the AMFT blocks via the AMFT network, and between the AMFT and the processing unit. The following are the messages:

- HeartBeat Message (HBM)
- Fault Message (FM)
- State Update Message (SUM)

- Task List Message

HeartBeat Message: A HeartBeat Message is transmitted by a node's AMFT block via the AMFT network at the start of the node allocated communications slot. Its purpose is to indicate the other nodes that the local node is active.

Format. Table 5.1 shows the format of the HeartBeat Message. Table 5.2 provides a description of each field in the message.

Fault Message: A Fault Message is sent by an AMFT unit via the AMFT at the start of the node allocated communications slot instead of a HeartBeat Message to indicate the other nodes that a fault has occurred in the node processing unit.

Format. Table 5.3 shows the format of a Fault Message. Table 5.4 provides a description of each field in the message.

Table 5.1: HeartBeat Message Format.

Bits	**0** **7**	**8** **15**
Field	Node Identifier	Heartbeat code (0x48)

Table 5.2: HeartBeat Message Fields.

Field	**Length**	**Description**	**Value**
Node Identifier	8 bits	Unique identifier of the sending node	0 to 255
Heartbeat code	8 bits	Constant value identifying message as a HeartBeat Message	0x48

Table 5.3: Fault Message Format.

Bits:	**0** **7**	**8** **15**
Field:	Node Identifier	Fault code (0x46)

Table 5.4: Fault Message Fields.

Field	**Length**	**Description**	**Value**
Node Identifier	8 bits	Unique identifier of the sending node	0 to 255
Fault code	8 bits	Constant value identifying message as a Fault Message	0x46

State Update Message: State Update Messages (SUMs) are transmitted to provide the most recent state data Δ_{SD} from an application task. The exact nature of the state data will be task-specific, but may include values for quantities such as angles or velocity data to be used in future calculations. SUMs are sent from a processing unit to its associated AMFT block. SUMs are then passed between AMFT block via the AMFT network after every HBM to ensure all nodes have up-to-date state data. They are also sent from the AMFT to the processing unit, if necessary when task migration is required, to ensure a node taking over responsibility for executing a task uses the most recent state data Δ_{SD}. A SUM contains state data Δ_{SD} for a single task, consisting of a set of state values. The length in bits of each state value depends on the nature of the data, e.g. a Boolean or a double-precision floating point value. Therefore the delimitation of state value fields within a SUM will vary for each task, and the correct interpretation of the SUM depends on each node having knowledge of the state values format for each task. The SUM contains a Task Identifier field enabling a node to determine the correct state values format for that task, and to correctly store the values in the correct memory locations for use by the mission task.

Format: State Update Messages are sent between AMFTs on the AMFT network, and also between an AMFT and its processing unit. For the messages sent between an AMFT and its processing unit, a Node Identifier field is not required, so these State Update Messages do not include this field. Table 5.5 shows the format for a State Update Message sent on the AMFT network. Table 5.6 shows the format for a State Update Message sent between an AMFT to the processing unit while Table 5.7 shows the message format for a processing unit to AMFT. Table 5.8 provides a description of each field in the message.

Task List Message: A Task List Message (TLM) consists of a set of Task Identifiers, indicating the set of tasks to be executed by a processing unit. The AMFT block always transmits this message to the processing unit.

Format: Table 5.9 shows the format for a TLM. Table 5.10 provides a description of each field in the message.

Table 5.5: State Update Message Format for Inter-AMFT Communication.

Bits:	**0 7**	**8 15**	**16 23**	**24**			**variable**
Field:	Node Identifier	SUM code (0x53)	Task Identifier	State value 0	State value 1	…	State value n

Table 5.6: State Update Message for AMFT and Processing Unit Communication.

Bits:	**0 7**	**8 15**	**16**			**variable**
Field:	SUM code (0x53)	Task Identifier	State value 0	State value 1	…	State value n

Table 5.7: State Update Message Format: Processing Unit and AMFT Communication.

Field:	**Sync Byte-1 (0xEB)**	**Sync Byte-2 (0x90)**	**Message Length**	**SUM code (0x53)**	**Task Identifier**	**State value 0**	**State value 1**	**…**	**State value n**

Table 5.8: State Update Message Fields.

Field	**Length**	**Description**	**Value**
Sync Byte-1	8 bits	Required for Packet Synchronization	0xEB
Sync Byte-2	8 bits	Required for Packet Synchronization	0x90
Packet Length	8 bits	Packet Length for Processing Unit-AMFT	0 to 255
Node Identifier	8 bits	Unique identifier of the sending node	0 to 255
SUM code	8 bits	Constant value identifying message as a State Update Message	0x53
Task Identifier	8 bits	Unique identifier of the task to which the state update applies	0 to 255
State value fields	8 bits	Set of bytes containing the state data for the task	Variable

Table 5.9: Task List Message Format.

Bits:	**0 7**	**8 15**	**16 23**	**24 31**	**Task Data variable**	
Field:	TLM code (0x54)	Number of tasks in TLM	Task Identifier 0	Task Identifier 1	…	Task Identifier n

Table 5.10: Task List Message Fields.

Field	Length	Description	Value
TLM code	8 bits	Constant value identifying message as a TLM	0x54
Number of tasks in TLM	8 bits	The number of Task Identifier fields following this field	Variable
Task Identifier fields	8 bits for each field	Set of fields containing the unique identifier of each task to be included in the task list	0 to 255

5.3.4 AMFT Tables

There are two types of tables inside the AMFT: (i) Node Table and (ii) Task Migration Table. The following explains each of these:

Node Table: The Node Table contains a single entry for each DCN present in the Fault-Tolerant Distributed system. Each entry contains the Node Identifier, and the node's communication slot start time, which are both fixed at design time. The status of the node (active/inactive) which is updated by the Target Fail-Over Selection Module within the AMFT when a node's status changes (e.g. when a node fails). The format for the Node Table is given in Table 5.11, which shows a communication slot time of 1000 ms and status of the nodes ('0/1'). The communications slot time of 1000 ms is for demonstration purposes only and the actual slot time depends on the application tasks period and their data sizes. Further details on the selection of the communications slot time is given in section 8.4.1.

Table 5.11: Node Table.

Node Identifier	Communications Slot Start Time (ms)	Active (1) / Inactive (0)
0	0	1
1	1000	1
2	2000	0

Task Migration Table: A Task Migration Table maps each mission task to be executed by the Fault-Tolerant Distributed system, to one of the distributed computing nodes. One Task Migration Table exists for each possible active node set, e.g. one table contains the task migrations for the case where all nodes are active. Another table contains the task migrations where one of the nodes is inactive, and so on. Each table contains one entry for each application task referenced by its Task Identifier, and a corresponding Node Identifier indicating the node on which the task is to execute. Multiple tasks may run on a single node, but at any time only one instance of a given task will be executed by the Fault-Tolerant Distributed system. The Target Fail-Over Selection Module within the AMFT uses the Task Migration Table corresponding with the current node statuses to inform its processing unit of which application tasks it must run. The format of the Task Migration Table is shown in Table 5.12, with example entries. These entries represent a particular scenario for three nodes and five tasks distributed system. This shows that in the presence of all three nodes, task-0 and 2 are assigned to node-0 while the task-1 is assigned to node-1 and task-3 and 4 are assigned to node-2.

Table 5.12: Task Migration Table.

Task Identifier	Node Identifier
0	0
1	1
2	0
3	2
4	2

5.4 AMFT Scenarios and Network Communication

This section describes the working of different components of the middleware in terms of fault detection, isolation and tasks migration.

Normal Operation: During the normal operation of AMFT as shown in Figure 5.7, each node's AMFT is bound to send a periodic HeartBeat message in its own time slot via the AMFT network for the health of each node. Inside the AMFT, Communication

module handles sending and receiving these HeartBeat messages. A reception of HeartBeat message indicates that a node is healthy, and no configuration is required while a missing HeartBeat message indicates that node is failed. Also to HeartBeat, Communication module also sends and receives the tasks state data messages that are periodically checkpointed from processing unit by the state check-pointing module. The mechanism inside the state check-pointing module discards the previous state's values when it receives new state values. The details on these task's state data values depend on the nature of running task while the number of time slots depends on the number of distributed computing nodes of a distributed system.

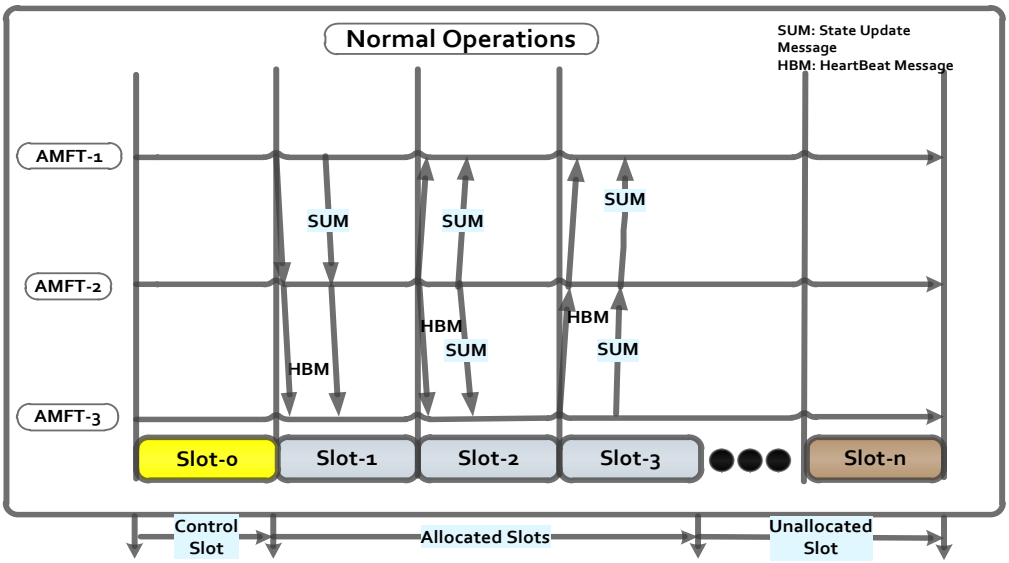


Figure 5.7: Network Communication in case of Normal Operations.

Processing Unit Fault: On the occurrence of a fault, AMFT communicates this information to the Communications Module and Target Fail-Over Node Selection module. On receipt of fault message from 'Fault Monitoring and Detection' module, communication module sends a fault message on the network as shown in Figure 5.8. All the healthy nodes of a group, which can share the tasks load, receive this message and reconfigure its associated processing unit based on the node table entries. On the healthy nodes, each AMFT 'Target Fail-Over Node Selection' module sends Task List Message and state data values for the lost tasks predefined to be shared with this processing unit. Then processing unit starts the lost tasks with the provided state values. While on the faulty node, all tasks assigned to its processing unit are stopped.

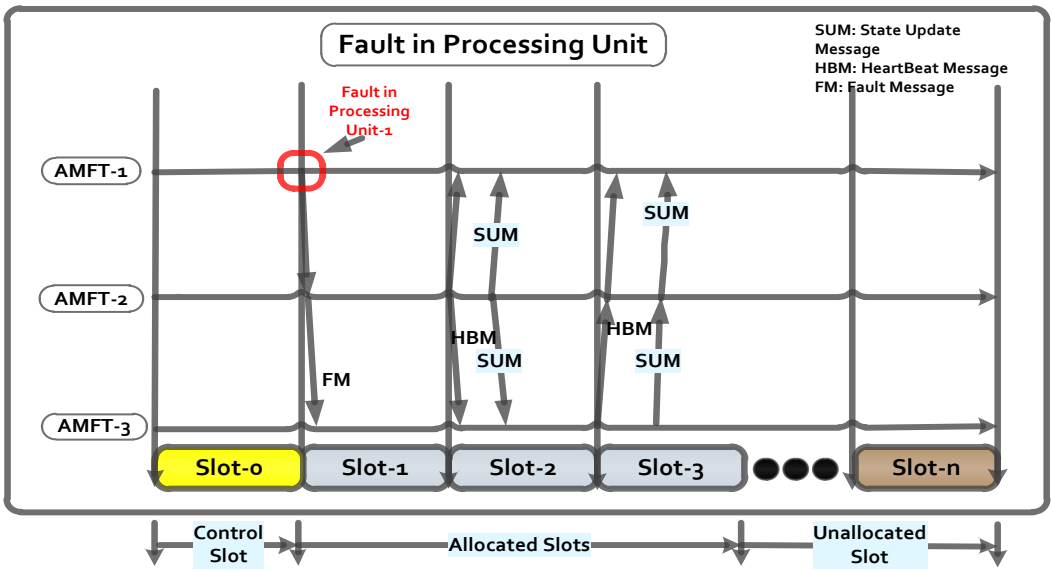


Figure 5.8: Network Communication in case of Processing Unit Failure.

AMFT Failure: Figure 5.9 shows a scenario when one of the AMFTs fails. During normal mode, the AMFTs exchange periodic HBM messages to indicate their presence. If there is no HBM from one of the AMFTs during its respective slot, the receiving AMFTs consider it to be a failed AMFT, and reconfiguration process is started.

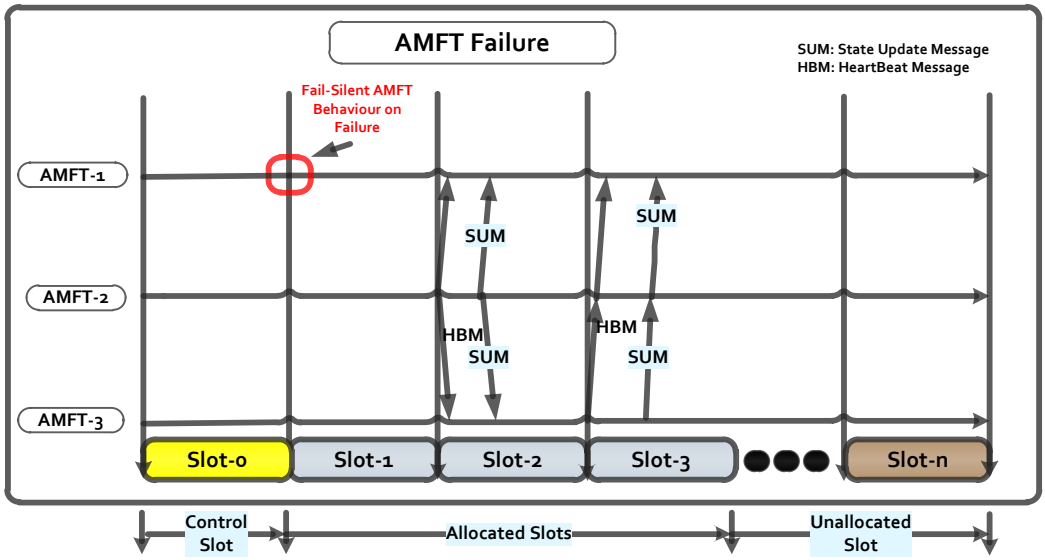


Figure 5.9: Network Communication in case of AMFT Failure.

5.5 AMFT Software Structure

The AMFT functionality, based on the fault management algorithms, presented in section 5.2, was mapped to software for the purpose of prototyping the proposed fault management scheme. The structure the AMFT software implementation is shown in Figure 5.10. The design of the middleware is mapped to several tasks whereas 'FDIR', 'TargetFailOverNodeSelection', and 'TDMA communication' are considered as the most important tasks. FDIR task complements the functionality of Fault Monitoring and Detection module of the top-level design of AMFT. As shown in Figure 5.10, the FDIR task is connected to the SM and HW interface message queues for monitoring the abnormal condition in a DCN. Data from SM and HW interfaces is passed to FDIR via software queues. FDIR task analyses the data and detects a fault if the pre-stored threshold limits for the monitored data is crossed.

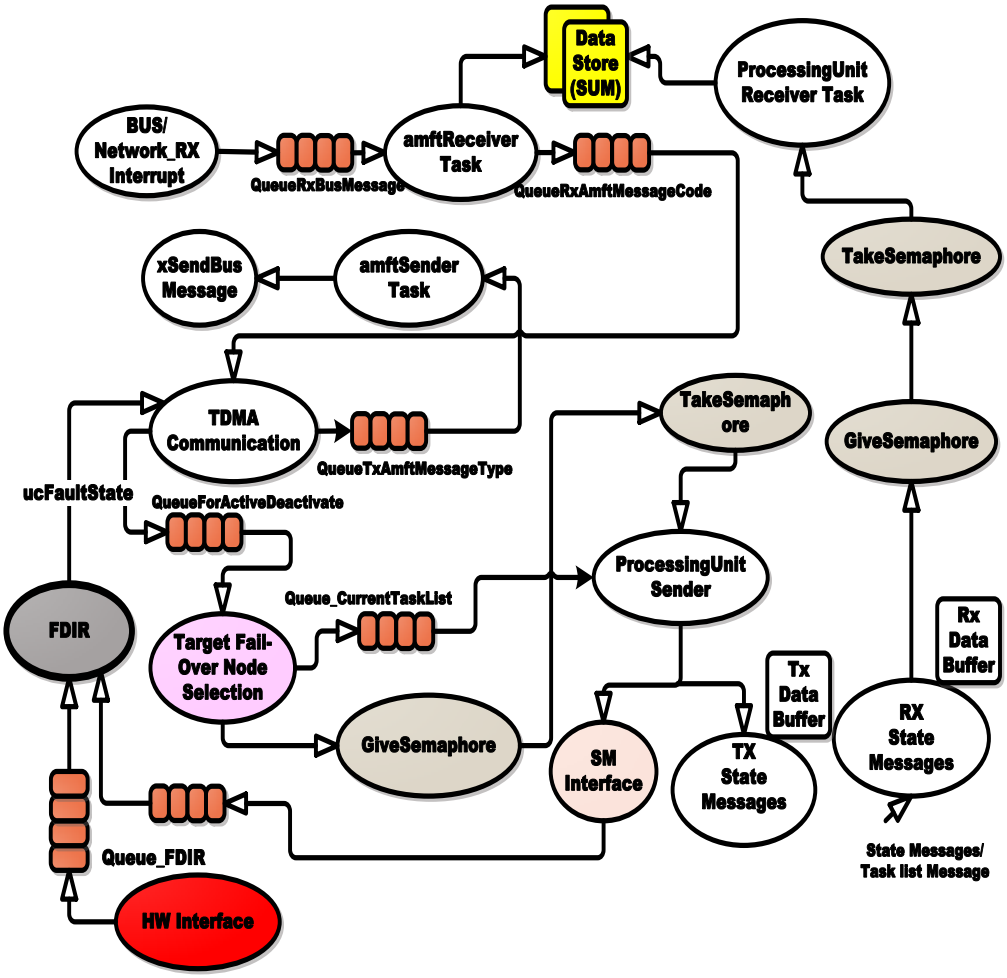


Figure 5.10: AMFT Software Implementation.

For simplicity and portability, the AMFT functions were implemented as a representative task of an operating system (OS). Table 5.13 shows the footprint comparison among the various Real-Time Operating System (RTOS). The real-time operating system 'FreeRTOS' [188] was selected for the prototyping, because of its small footprint and free availability for many embedded processors. However, the middleware functions can be ported to any operating system or can be implemented as a standalone application. In addition, these functions can also be implemented as hardware modules.

Table 5.13: Footprint Comparison for Real-Time Operating Systems.

Real-Time Operating System (RTOS)	***Footprint (kB)***	***License Type***
VxWorks	Code Size Min: 36/Basic:150/Full:250 [199]	License required
Windows Embedded CE	400 (minimum code size) [200]	License required
QNX	7-204 [201]	License required
uCLinux	2 MB (ROM)/4MB (RAM) [202]	Open Source
FreeRTOS	4.4 (code size)/200 bytes (data size) [203]	Free for Educational and Commercial use.
RTEMS	Basic: 64-128 (code size)/Complex: 512 (code size)	Free for Educational and Commercial use.
XilKernel	12-20 (code size) /46.5-59 (data size) [204]	Free (Integrated with Xilinx Embedded Development Kit)

5.5.1 FDIR Task

Description—The FDIR task monitors for processing unit faults. If a fault is detected, a message is sent to the other AMFT blocks, via the AMFT Comms task, indicating that the unit has failed. Attempts are then made to recover the processing unit.

The process of fault detection, isolation and recovery is implemented by the FDIR task inside the AMFT. FDIR informs other modules about the health of processing unit by a status flag. Other modules inside the AMFT read the status flag and sends messages accordingly. If there is a fault (software or hardware), FDIR detects a fault and switches its mode to faulty '1' and starts the recovery process. Meanwhile, communication service of the AMFT associated with the faulty processing unit reads

the status flag and starts sending Fault Detection message on the AMFT network. Afterward, an attempt for the autonomous recovery of the faulty processing unit is made if recovered; the processing unit is reintegrated into a distributed system. If not recovered, a faulty node (Processing Unit + AMFT) is shutdown. The pseudo code for the FDIR task is as follows:

Pseudo Code:

```
1   FDIR Task {
2           while(1){
3                   wait for fault detection
4                   set fault flag
5                   autonomous recovery
6                   if recovered {
7                           reset fault flag
8                           reintegrated
9                   }
10                  else{
11                          shutdown
12                  }
13          }
14  }
```

5.5.2 Target Fail-Over Node Selection Task

Description— The "TargetFailOverNodeSelection" responds to messages from the AMFT Comms task indicating a change in the status of a node. It updates the Node Table to account for the node status change. It then reads the Task Migration Table associated with the updated group configuration. Based on the contents of the Task Migration Table, it sends a Task List Message, It follows by State Update Messages for each task in the task list, to the "Processing Unit Sender Task" to be sent to the Processing Unit. The pseudo code for the "TargetFail-Over Node Selection Task" task is as follows:

Pseudo Code:

```
1   Target Fail − Over Node Selection Task {
2       while(1){
3           Wait for message from AMFT Comm.Task for
                Node Status change
4           Modify Node Status within Node Table
5           Read Task Migration Table associated with current node set
6           Create task list message containing tasks assigned to this
            Node
7           Message to Processing Unit Sender Task
8           Send Task list Message to Processing Unit
9           For each task within the task list {
10              Get Stored SUM for the task
11              Send SUM for the task to Processing Unit
12          }
13      }
```

5.5.3 AMFT Communications Task

Description—The "AMFT Comms task" handles the communications with other AMFT blocks via the AMFT network. It includes a start-up period during which any HeartBeat Messages from already-transmitting nodes are detected. The local node's communication slot timing is then synchronized with the received HBMs during this phase. If no HBMs are received, i.e. no other nodes have been activated, then the AMFT Comms task immediately sends the local node's own HBM. It sets the communications slot timing based on the time at which it sent its HBM. In this way, the communications slot timing is always determined by the first node to start up. The AMFT Comms task handles the overall communications, e.g. slot timing, but the actual sending and receiving is delegated to the AMFT Sender and AMFT Receiver tasks, which pass information to and from the AMFT Comms task using message queues.

After the start-up phase, the AMFT Comms task enters an infinite loop of sending its HBM and SUMs during its communications slot. In each subsequent communications slot, receiving an HBM and SUMs from the other nodes. On receipt of an SUM, the relevant state data is updated within the AMFT's local memory. During each communications period, the local communications slot timing is updated based on the HBM reception time for the node with the lowest ID in the group (if the local node does not have the lowest ID).

If the AMFT Comms task receives a message from the FDIR task indicating a fault in the Processing Unit, the AMFT Comms task stops sending HBMs and instead sends a fault message via the AMFT bus. It does this until the FDIR task indicates recovery of the Processing Unit.

The AMFT Comms task informs the Task Migration Manager task that a group update is required in the following circumstances:

- A fault message is received from a node that is marked as active in the Node Table;
- An HBM is not received from a node which is marked as active in the Node Table, during its allocated communications slot;
- An HBM is received from a node that is marked as inactive in the Node Table. This may occur, for example, when a failed node has been recovered.

Pseudo Code:

```
1   AMFT Comm.Task {
2       Read Node Table to determine slot times for all nodes
3       Listen for HBM on the AMFT bus for a time Comm. Cycle
4       If HBM is received
5           use this to determine the start time of the Comm.Cycle
6       else
7           send HBM and set start time of the Comm. Cycle to
            reflect send time.
8       while (1)
9           wait until start of this Node'sComm.Slot
10          If FDIR task has indicated, there has been a Processing
```

```
                Unit fault, and has not yet subsequently recovered {
11                      Send Fault Message
12              }
13              else
14                      Send HBM
15                      for each SUM Stored in memory{
16                              Send SUM
17                      }
18              }
19              for each other node in distributed system{
20                      wait for HBM
21                      if HBM received{
22                              if HBM is from node with lowest ID{
23                                      update local time with synchronize with
                                        time HBM received.
24                              }
25                              if node sending HBM is marked inactive in
                                the Node Table{
26                                      message to Task Migration Manager
                                        node ID activated
27                              }
28                              wait for SUMs
29                              for each SUM received{
30                                      stored the SUM in memory
31                              }
32                      else if fault message received or no HBM within
                        comm. slot from node marked as active in Node Table{
33                              message to Task Migration Manager
                                node ID deactivated
34                      }
35              }
36      }
37  }
```

5.5.4 AMFT Receiver Task

Description: The AMFT Receiver task processes messages received on AMFT network from other AMFT units. The messages are received from the network interrupt routine via a message queue. If the message received is a Heartbeat message or a Fault message, this information is sent to the AMFT Comms task. If the message is an SUM, the state data within the message is stored in memory. The pseudo code for the "AMFT Receiver Task" is as follows:

Pseudo Code:

```
1  AMFT Receiver Task {
2      while(1) {
3          receive message from the CAN interrupt routine
           if state data is expected i.e. continuation of previous
           state update message {
4              stored the state data
5          }
6          else {
7              if the message is HBM or fault message
8                  send a message to AMFT comm. Task:
                   HBM fault message received
9          }
10         else if the message is a SUM {
11             get task ID from message and check expected message
               length for the SUM
12         }
13     }
14 }
```

5.5.5 AMFT Sender Task

Description: The AMFT Sender task sends messages on the AMFT network to other AMFT units. The AMFT Comms task specifies a message type to be sent (communicated to the AMFT Sender task via a message queue). Based on the type, the AMFT Sender task sends a message using the correct message format. This includes

the current state data for an application task if the message to be sent an SUM. The pseudo code for the "AMFT Sender Task" is as follows:

Pseudo Code:

```
1   AMFT Sender Task {
2       while(1) {
            receive message from the AMFT comm.Task
            send a message of type < type >
3
            create and send a message of the requested type
4       }
5   }
```

5.6 Discussion

In this section, we discuss the salient features that make the proposed AMFT design different and more efficient than the existing middleware.

Employed middleware, as stated in the literature, uses client-server model [205-208]. In the client-server model, the client sends a request to the server asking for a service and the server process the request and returns a reply. There are two drawbacks to this approach: (i) for each request, the client has to establish a connection before sending a request to server, (ii) the client can request but the server cannot. Since a two-way communication is required for real-time applications, this connection arrangement causes an additional time delay, therefore, the typical client-server model was not suitable for our application. However, it can be employed for desktop distributed computing systems.

In middleware designs [3, 62, 191, 193], processes are replicated for fault tolerance in distributed systems. If the middleware of the running process fails to send a ping message within the timeout interval, then the process is considered as failed and campaign for the failover target node selection is launched. This failure can be a middleware failure, or it can be a process failure. The behaviour of this type of failure can be fail-silent or Byzantine. Current middleware failure's assumption of fail-silent

is simple and does not consider hardware failures (processor failure common in space). Therefore, we adapted an additional level of fault detection as deemed essential to ensure the fail-silent node behaviour.

Applications require data consistency among the primary and backup computing nodes. This is useful to start the process or task from that point onward when it is started on another node. This feature is very difficult to achieve, and usually the fault tolerant middleware designers ignore this feature as mentioned in [208, 209]. In [62], where an exact match (Complete Task State – State Data) between the primary and redundant data contents is suggested. It is not necessary because it wastes network resources, particularly in case of large size data in distributed systems. In our approach, the State Data is reduced to necessary state information, and therefore reducing the communication overhead required for state information management, between AMFTs.

It is important to state here, that the middleware is designed to cater to desktop applications, whereas the proposed design is for embedded applications. JAVA language is commonly used for programming, which is inherently not efficient for real-time applications. Therefore, we used C programming language for its known heritage.

5.7 Summary

The chapter discusses the proposed new fault management scheme, which is named Adaptive Middleware for Fault-Tolerance (AMFT), in depth. Novel algorithms for fault detection, isolation and task migration are developed. The algorithms provide fault-tolerance in a distributed system by tasks migration rather than task replication. This feature improves the overall cost efficiency by making the system reliable with little resource utilization.

AMFT runs on a separate hardware module, alongside the main processing unit, thus enabling local decision control in case of fault occurrence. AMFT utilizes pre-assigned node and task list tables, ensuring high reliability, in contrast to dynamic allocation.

AMFT is capable of monitoring both hardware as well as software faults, working in conjunction with the application software in DCN. The software faults are supervised by the application software via a special software monitoring interface (SM). The hardware is monitored via analogue and digital signal monitoring.

The software code for the main AMFT operations is described. The beauty of AMFT is that it can be hardwired as a VHDL module. The analysis of the architecture is carried out in chapter 6. The physical implementation and its issues are highlighted in Chapter 7.

Chapter 6

Evaluation of the Proposed Approach

The design of the proposed fault tolerant distributed (FTDC) architecture and the functionality of the fault management scheme were explained in chapter 4 and chapter 5 respectively. This chapter is dedicated to the analysis and evaluation of the proposed approach. The first part of the chapter presents analysis of the architecture and the fault management scheme. The second part of the chapter provides a functional verification through prototyping at board level of the system. In section 6.1.1, performance metrics are defined, the proposed architecture is analysed and compared with a centralized and a TMR-based systems and fault management schemes are analysed in terms of their performance. In section 6.2, details on the functional verification are presented, in which experimental results are reported and implementation issues are highlighted. Section 6.3 concludes the chapter.

6.1 Dependability Analysis of the Distributed Computing Approach

"The notion of dependability covers the meta-functional attributes of a computer system that relate to the quality of service a system delivers to its users during an extended interval of time" [210]. A task-oriented distributed computing system provides the necessary services for execution of an application, which is represented by a set of tasks. Reliability and availability are essential criteria to judge the fault-

tolerance performance of a computing system. In this section, a reliability evaluation of the computing architectures is presented. These architectures are analysed and compared by their developed mathematical Markov models. The fault management scheme is also analysed in terms of reliability and availability.

6.1.1 Performance Metrics

Since there were no performance metrics, metrics most suited to dependability analysis were identified. The performance of the computing architecture and fault management scheme is evaluated by analysing two main parameters: reliability and availability.

6.1.1.1 Reliability

Reliability *R* is the probability of a system to produce the correct (acceptable) output for a specified period of time *t*. The reliability of a system, which is used as a performance metric in computing systems, degrades with system-level failures. Thus, a reliable computing system should be able to recover from a failure condition. The system reliability *R* is an exponential function [28], which decreases with time *t* as shown in Figure 6.1. An expression for $R(t)$ is given by:

$$R\ (t) = \ e^{-\lambda t} \tag{6.1}$$

where

λ	Failure rate per unit time *t*
t	Time duration, measured as per requirements (ranging from sec, min, weeks to years)

6.1.1.2 Availability

Availability represents the probability that a system is operational during a given period of time. Availability *A* is measured by the ratio of the uptime (during which the system is operational) to the total time including the downtime (repair time) and can be expressed by [28]:

$$A = \frac{Uptime}{Uptime + Downtime} \tag{6.2}$$

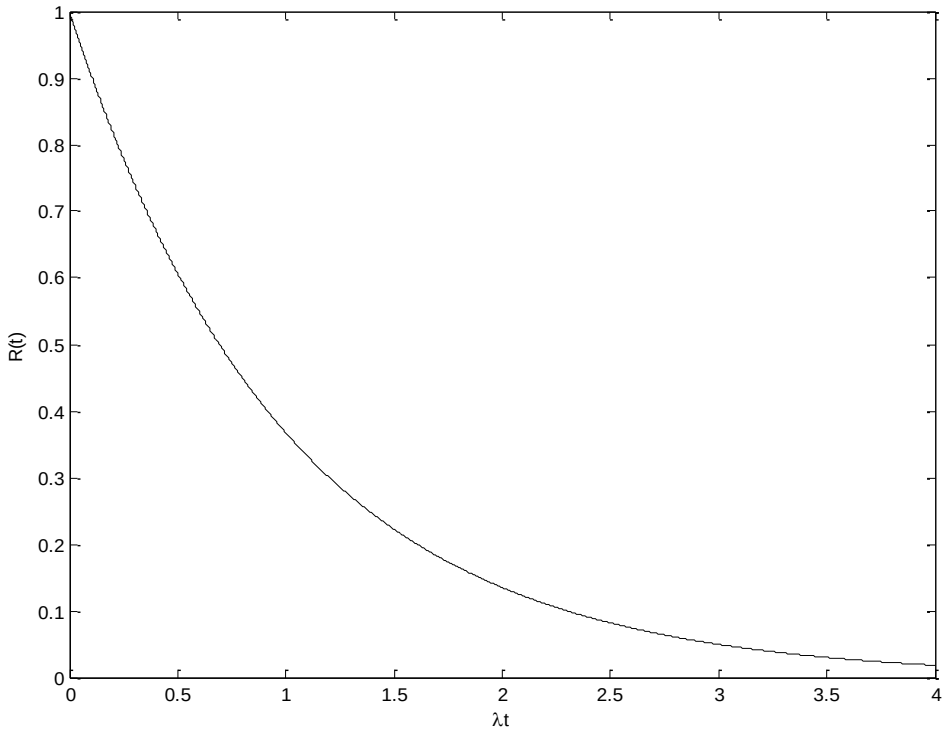


Figure 6.1: System Reliability.

6.1.2 Reliability Analysis of Computing Architectures

In this section, we analyse and compare three computer system configurations: (i) centralized, (ii) TMR and (iii) the proposed FTDC and evaluate their reliability. The centralized and the TMR-based systems are chosen for the comparison because these are state-of-the-art computing systems, which are widely used in mission critical applications and, in particular, on board spacecraft, as discussed in 3.2.3. The reliability derivation for satellite OBCs is presented in Appendix B, using the Bernoulli distribution, which is a simplified method and suitable for computing systems with independent computing nodes. However, if the nodes are statistically dependant and the failure or repair process of any one of them is dependent on the state the other, a more sophisticated technique, which is able to incorporate these dependencies, is required. In

this section, a more elegant approach to reliability analysis of computing systems, based on the Markov mathematical model is presented, where the fault model includes failure detection, isolation and repair processes. A node in the model (i) is considered dependent and (ii) can fail with a failure rate of λ and a repair rate of μ. A three years mission lifetime is assumed with a failure rate λ of $1x10^{-3}$ per hour and a recovery rate μ of $1x10^{-5}$ per hour.

6.1.2.1 Centralized Computing System

A Markov model for the centralized computing system with repair is shown in Figure 6.2. It consists of two states; S_0 and S_1. The transition probability from state S_0 to S_1 is given by $\lambda.\Delta t$, where λ is the failure rate, and Δt represents the time interval in which failure is probable. Similarly, transition probability from state S_1 to S_0 is given $\mu.\Delta t$ where μ represents the repair rate, and Δt represents the time interval in which repair is probable. Each Markov equation represents a particular state of the system and is expressed by the differential equations (6.3) and (6.4) below.

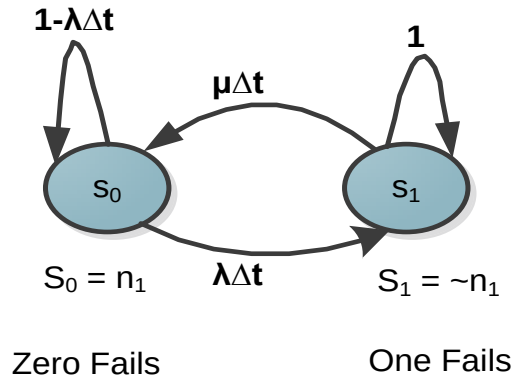


Figure 6.2: Markov Model for Centralized System.

$$\frac{dPs_o(t)}{dt} = -\lambda\, Ps_o(t) + \mu\, Ps_1(t) \tag{6.3}$$

$$\frac{dPs_1(t)}{dt} = +\lambda\, Ps_o(t) - \mu\, Ps_1(t) \tag{6.4}$$

Applying the Laplace transform to Equation 6.3 and 6.4 results in:

$$sPs_o(s) - ps0(0) = -\lambda\, Ps_o(s) + \mu\, Ps_1(s) \tag{6.5}$$

$$sPs_1(s) - ps1(0) = \lambda\, Ps_o(s) - \mu\, Ps_1(s) \tag{6.6}$$

Initially, the centralized system is in working state S_0, So, $ps_0(0)$ =1 and $ps_1(0)$ =0. By substituting initial values and rearranging the Equations 6.5 and 6.6, we obtained linear equations, Equation 6.7 and Equation 6.8, which can be easily solved for $Ps_0(t)$ and $Ps_1(t)$:

$$(s + \lambda)Ps_o(s) - \mu\, Ps_1(s) = 1 \tag{6.7}$$

$$(-\lambda)Ps_o(s) + (s + \mu)\, Ps_1(s) = 0 \tag{6.8}$$

Equation 6.7 and 6.8 can also be represented in a (2x2) matrix form, as follows:

$$\begin{bmatrix} s+\lambda & -\mu \\ -\lambda & s+\mu \end{bmatrix} \begin{bmatrix} Ps_o(s) \\ Ps_1(s) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{6.9}$$

Solving the above equations yields an expression for the reliability of the centralized system, $R_{cent\_sys}(t)$, given by Equation 6.10.

$$R_{cent\_sys}(t) = \frac{1}{(\lambda + \mu)}\left[\mu + \lambda e^{-(\lambda+\mu)t}\right] \tag{6.10}$$

6.1.2.2 Triple Modular Redundant System

A Markov model for the TMR-based system is shown in Figure 6.3. This model comprises three identical nodes n_1, n_2, and n_3 with the same failure rate λ and recovery rate μ. Initially, all the nodes are in state S_0, and the probability of the transition from state S_0 to S_1 is given by $3\lambda.\Delta t$. The probability of each state is represented by three Markov equations as follows:

$$\frac{dPs_o(t)}{dt} = -3\lambda\, Ps_o(t) + \mu\, Ps_1(t) \tag{6.11}$$

$$\frac{dPs_1(t)}{dt} = +3\lambda\, Ps_o(t) - (\mu + 2\lambda)Ps_1(t) \tag{6.12}$$

$$\frac{dPs_2(t)}{dt} = 2\,\lambda\,Ps_1(t) \tag{6.13}$$

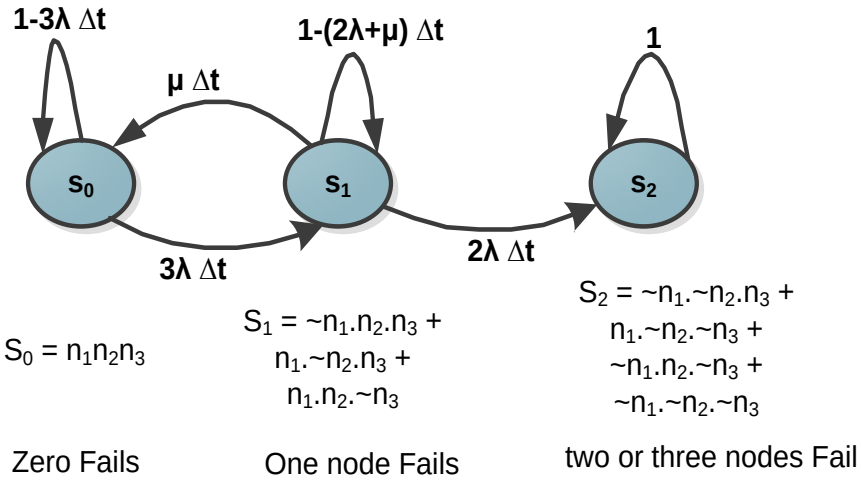


Figure 6.3: Markov Model for TMR-based System [211].

Applying the Laplace transform to Equation 6.11, Equation 6.12, and Equation 6.13 gives:

$$sPs_o(s) - pso(0) = -3\lambda\,Ps_o(s) + \mu\,Ps_1(s) \tag{6.14}$$

$$\mathrm{sPs_1(s) - ps1(0) = 3\lambda\,Ps_o(s) - (\mu + 2\lambda)\,Ps_1(s)} \tag{6.15}$$

$$sPs_2(s) - ps2(0) = 2\lambda\ Ps_1(s) \tag{6.16}$$

$$\begin{bmatrix} s+3\lambda & -\mu & 0 \\ 3\lambda & s+\mu+2\lambda & 0 \\ 0 & -2\lambda & s \end{bmatrix} \begin{bmatrix} Ps_o(s) \\ Ps_1(s) \\ Ps_2(s) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{6.17}$$

Solving the equations 6.14 to 6.17 for the probabilities of the operational states, $Ps_0(t)$ and $Ps_1(t)$, gives the reliability of the TMR system, $R_{tmr\_sys}(t)$, which is represented by Equation 6.18.

$$R_{tmr\_sys}(t) = \left(3 + \frac{\mu}{\lambda}\right) e^{-2\lambda t} - \left(2 + \frac{\mu}{\lambda}\right) e^{-3\lambda t} \tag{6.18}$$

6.1.2.3 Fault-Tolerant Distributed System with Two Nodes

A Markov model for a two-node FTDC system, which uses a task migration scheme, is shown in Figure 6.4. This model comprises two identical nodes n_1, and n_2. Initially, both nodes are in state S_0. Upon a failure, the transition from state S_0 to S_1 is activated and the tasks running on the faulty node are migrated to the healthy node. In state S_1, one of the nodes is working and sharing the workload of the faulty node tasks. The probability of each state is represented by Equations 6.19, 6.20 and 6.21 below:

$$\frac{dPs_o(t)}{dt} = -2\lambda\, Ps_o(t) + \mu\, Ps_1(t) \tag{6.19}$$

$$\frac{dPs_1(t)}{dt} = +2\lambda\, Ps_o(t) - (\mu + \lambda) Ps_1(t) \tag{6.20}$$

$$\frac{dPs_2(t)}{dt} = \lambda\, Ps_1(t) \tag{6.21}$$

Applying the Laplace transform gives:

$$sPs_o(s) - pso(0) = -2\lambda\, Ps_o(s) + \mu\, Ps_1(s) \tag{6.22}$$

$$sPs_1(s) - ps1(0) = 2\lambda\, Ps_o(s) - (\mu + \lambda)\, Ps_1(s) \tag{6.23}$$

$$sPs_2(s) - ps2(0) = \lambda\ Ps_1(s) \tag{6.24}$$

$$\begin{bmatrix} s+2\lambda & -\mu & 0 \\ 2\lambda & s+\mu+\lambda & 0 \\ 0 & -\lambda & s \end{bmatrix} \begin{bmatrix} Ps_o(s) \\ Ps_1(s) \\ Ps_2(s) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{6.25}$$

Solving the above equations for the probabilities of the operational states, $Ps_0(t)$ and $Ps_1(t)$, gives the reliability of the distributed system, $R_{Dis\_sys(t)}$, as follows:

$$R_{Dis\_sys}(t) = \left(2 + \frac{\mu}{\lambda}\right) e^{-\lambda t} - \left(1 + \frac{\mu}{\lambda}\right) e^{-2\lambda t} \tag{6.26}$$

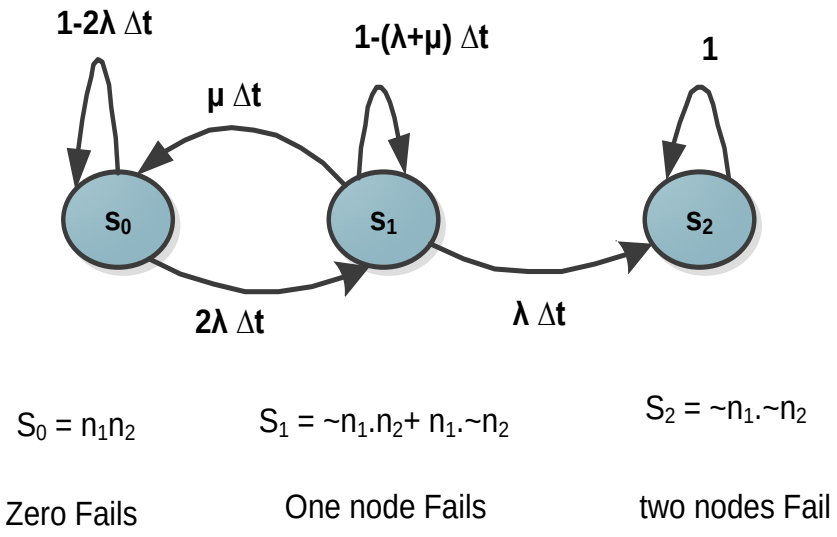


Figure 6.4: Markov Model for a Two-Node Distributed System.

6.1.2.4 Fault-Tolerant Distributed System with Three Nodes

A Markov model for the three-node distributed system is shown in Figure 6.5. Compared to the two nodes distributed model, one additional state is added to represent the third node in the system. State S_0 represents normal operation while state S_3 represents a complete system failure. One and two nodes failures are represented by state S_1 and S_2 respectively. The corresponding Markov equations (6.27) – (6.30) are given below:

$$\frac{dPs_o(t)}{dt} = -3\lambda\, Ps_o(t) + \mu\, Ps_1(t) \tag{6.27}$$

$$\frac{dPs_1(t)}{dt} = +3\lambda\, Ps_o(t) - (\mu + 2\lambda) Ps_1(t) \tag{6.28}$$

$$\frac{dPs_2(t)}{dt} = 2\lambda\, Ps_1(t) - (\mu + \lambda) Ps_2(t) \tag{6.29}$$

$$\frac{dPs_3(t)}{dt} = \lambda\, Ps_2(t) \tag{6.30}$$

Applying the Laplace transform gives:

$$sPs_o(s) - pso(0) = -3\lambda\, Ps_o(s) + \mu\, Ps_1(s) \tag{6.31}$$

$$sPs_1(s) - ps1(0) = 3\lambda\, Ps_o(s) - (\mu + 2\lambda)\, Ps_1(s) \tag{6.32}$$

$$sPs_2(s) - ps2(0) = 2\lambda\ \ Ps_1(s) - (\mu + \lambda)\, Ps_2(s) \tag{6.33}$$

$$sPs_3(s) = \lambda\, Ps_2(t) \tag{6.34}$$

$$\begin{bmatrix} s+3\lambda & -\mu & 0 & 0 \\ 3\lambda & s+\mu+2\lambda & 0 & 0 \\ 0 & -2\lambda & s+\mu+\lambda & 0 \\ 0 & 0 & -\lambda & s \end{bmatrix} \begin{bmatrix} Ps_o(s) \\ Ps_1(s) \\ Ps_2(s) \\ Ps_3(s) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.35}$$

Solving the above equations for the probabilities of the operational states, $Ps_0(t)$, $Ps_1(t)$, and $Ps_2(t)$, gives the reliability of the distributed system with three nodes, $R_{Dis\_sys}(t)$, expressed by Equation 6.37.

$$R_{Dis\_sys}(t) = Ps0\ (t) + Ps1(t) + Ps2\ (t) \tag{6.36}$$

$$R_{Dis\_sys}(t) = \left(1 + \frac{\mu}{\lambda}\right) e^{-3\lambda t} - \left(3 + \frac{\mu}{\lambda}\right) e^{-2\lambda t} + \left(3 + \frac{\mu}{\lambda}\right) e^{-\lambda t} \tag{6.37}$$

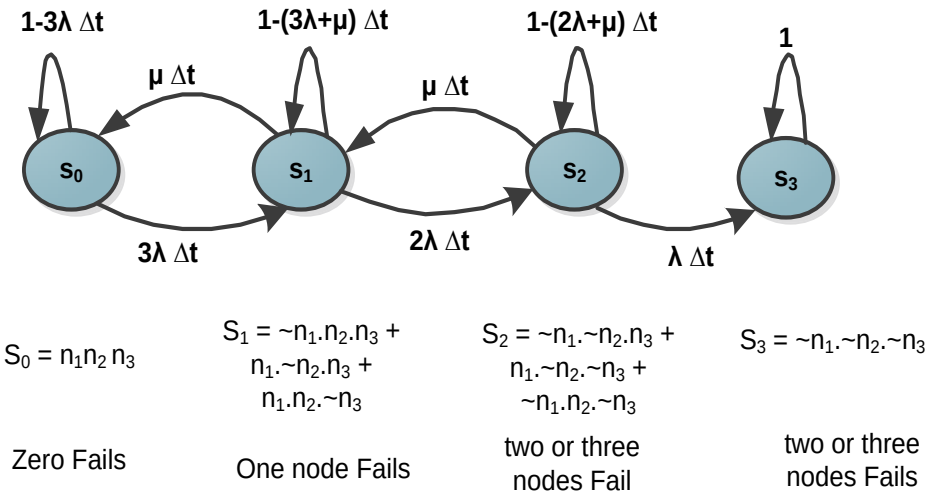


Figure 6.5: Markov Model for a Three-Node Distributed System.

6.1.2.5 Discussion

The reliability for each system (Centralized, TMR, and FDTC) is calculated using equations 6.10, 6.18, 6.26 and 6.37 and presented graphically in Figure 6.6. From the graphs in Figure 6.6 it can be seen that the proposed scheme is more reliable than the

other two systems. The centralized system's reliability graph crosses that of TMR after about $0.6x10^4$ hours. This is due to the TMR system inherent dependency on three nodes and a voter circuit - as soon as one node fails, the reliability performance falls drastically. One of the reasons that the proposed FTDC system outperforms the others is due to the fact that the lost tasks are compensated for by migrating tasks from a faulty node to healthy nodes, whereas, this functionality is not present in the other systems. It can also be seen from the Figure 6.6 that the reliability of the distributed architecture increases with the increase of the number of nodes. It is also evident that the reliability degrades relatively more gracefully compared to the other systems, which is desired in mission critical systems.

Figure 6.7 shows the relative improvement in reliability values of the FTDC system compared to centralized and TMR-based systems. First the centralized system is compared with the two and three nodes distributed system and their relative improvement in reliability values (marked with rectangles and circles) are plotted as shown in Figure 6.7. Then, the TMR-based system is compared with two and three nodes distributed system and their relative improvement in reliability values (marked with triangles and asterisk) are plotted as shown in Figure 6.7.

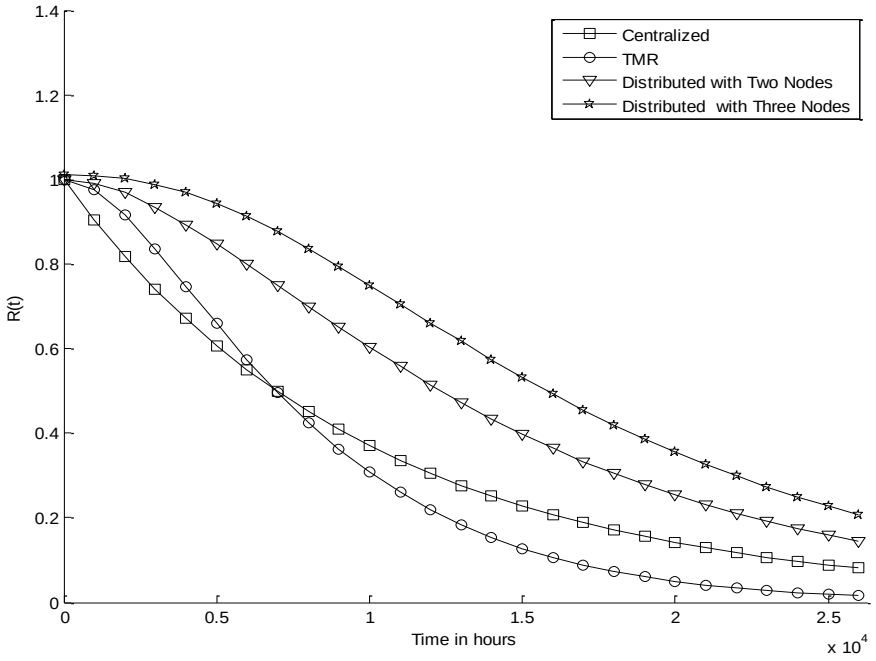


Figure 6.6: Reliability Curves for Centralized, TMR-based and Distributed Systems.

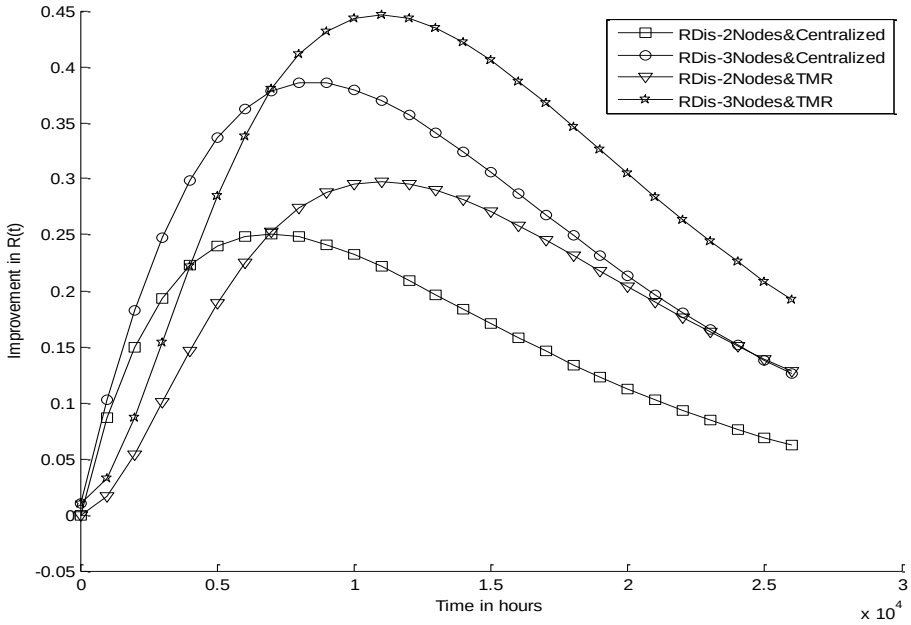


Figure 6.7: Relative Improvement in Reliability Values for Distributed Computing Approach.

6.1.3 Fault Management Scheme Analysis: Distributed vs Centralized

In this section, the proposed fault management scheme is compared with the centralized scheme of section 3.2.2.2 in terms of reliability and availability values. A three-year mission lifetime with a failure rate λ of $1x10^{-3}$ per hour is assumed. For a fair comparison, equal reliability of each node was assumed.

Both schemes include a number of computing nodes, which could correspond to an individual subsystem or a set of subsystems, such as OBDH, Thermal, Telemetry/Telecommand (TMTC) subsystems. However, in the case of the centralized scheme, only a single computing node and its pair redundant peer are responsible for the overall fault management, while in the proposed approach, each individual computing node is responsible for its fault detection and the reconfiguration of the system by migrating tasks to other nodes. In the proposed fault management scheme, each computing node is attached to an AMFT block to serve as distributed fault management scheme. By distributing the fault management to each individual computing node, much higher reliability can be achieved as it is evident from the following evaluation.

6.1.3.1 Reliability Assessment

In order to simplify the reliability evaluation, a straightforward reliability assessment method is adopted in this section, whereby the system is broken down into sub-systems, represented as blocks in a block diagram. The blocks are connected either in series, parallel or a combination of both. The Reliability is the probabilistic description of the success of the system. In other words, the assessment is carried out by determining the relationship of how the node failure, which is dealt with by the fault management functions, affects the complete system. The detailed derivation of each block connection (series, parallel) is given in Appendix B.

6.1.3.1.1. Centralized Fault Management Scheme

Consider a traditional centralised on-board computing system, which consists of a central node and a set of other nodes, with all nodes being dual redundant. The central node is responsible for the fault management (FM) functions. Upon detection of a node failure the central node reconfigures the system by switching off the primary node and switching on the redundant node of the failed dual redundant pair. In this scheme, the reliability of the complete system depends on the reliability of the central node. Therefore, it can be represented as follows;

$$R_{sys\_cent\_fm} = R_{cent\_node} \tag{6.38}$$

Whereas, the reliability of the central node depends on its configuration and in general, is expressed as,

$$\mathrm{R}_{cent\_node} = 1 - (1 - \mathrm{e}^{-\lambda \mathrm{t}})^{\mathrm{m}} \tag{6.39}$$

where $(1 - \mathrm{e}^{-\lambda \mathrm{t}})^{m}$ is the failure probability of *m* identical parallel nodes. Therefore, *m* = 2, for the case of a dual redundant central node. Hence the centralized scheme reliability depends on the reliability of the dual redundant central node configuration. As the centralized node manages all the fault management functions, therefore its failure could be catastrophic for the whole computing system. However, for *m*=3 in a centralized fault management scheme, additional two redundant computing nodes will be required which increases the overall system cost. The detailed derivation of reliability is carried out in Appendix B.

6.1.3.1.2. Distributed Fault Management Scheme

In the proposed distributed fault management scheme, unlike centralized scheme, the fault management functionality is not limited to a single node but distributed to an *m* number of nodes in the computing system. The distributed scheme can be represented as parallel reliability blocks; therefore, the reliability is increased and is directly dependent on the number of nodes in the system. Therefore, unlike centralized fault management scheme, where $m = 2$, depends on the configuration of the centralized node only, the proposed scheme reliability depends on the total number of available nodes in the system. The same formula as given in equation 6.39 is also applicable to the distributed fault management scheme, but *m* represents the total number of nodes in a system.

6.1.3.1.3. Discussion of Reliability Results

Figure 6.8 shows the reliability curves that were obtained from the reliability Equation 6.39. The reliability curve R_1 as shown in Figure 6.8 is plotted for the centralized fault management scheme where m=2 is assumed.

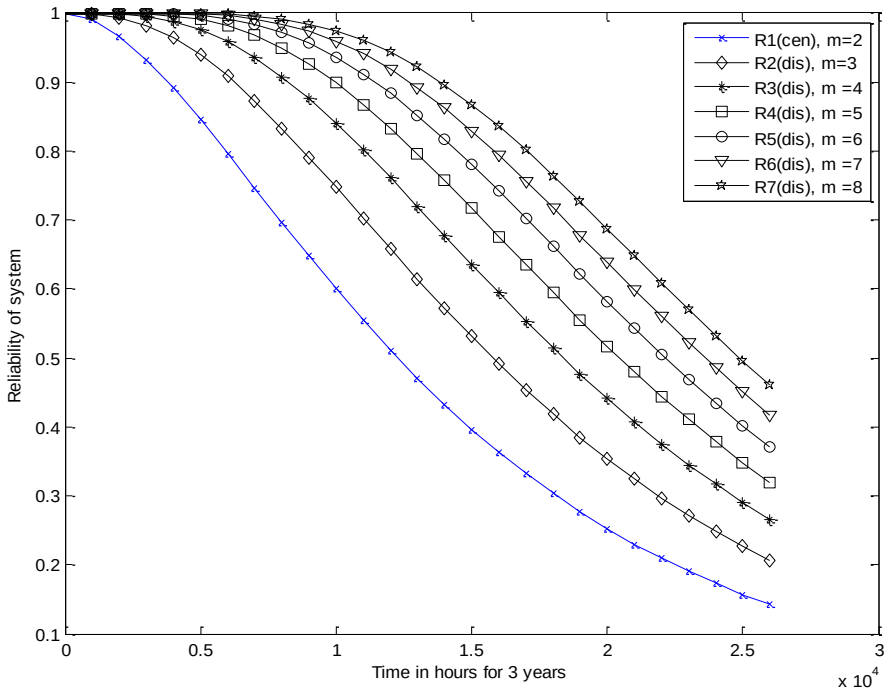


Figure 6.8: Comparison between Centralized and Distributed Fault Management Scheme.

In the case of centralized fault management scheme, as the centralized node is mainly responsible for the fault management functions, so $R_{sys\_cent\_fm}$, is severely degraded by the reliability value of the centralized node. On the other hand, the reliability for the distributed scheme is much higher because of the absence of a centralized node. The distributed scheme reliability is plotted for different *m* values (*m*=2, *m*=3, *m*=4, *m*=5, *m*=6, *m*=7, *m*=8) as shown in Figure 6.8. It is evident from the reliability curves in Figure 6.8 that the reliability of the distributed scheme is higher in comparison to the centralized scheme and depends on the value of *m*.

6.1.3.2 Availability Evaluation

In this section, a set of stochastic models was developed for the availability evaluation of the distributed fault management scheme. The models, which are based on the Extended Deterministic and Stochastic Petri Nets (EDSPN) method, are designed with TimeNet v4.02 [212]. We choose Petri Nets, because it allows to represent structural modelling of complex systems easier. Also, Petri Nets can express statistical dependency that cannot be expressed by reliability block diagrams (RBD) or fault-tree analysis [213].

Availability models for both the centralized and distributed fault management schemes were developed for comparison. Assumptions for the failure recovery and failure transition apply. The followings terms are used for the description of models:

- P_{robust}: Normal working state of a node.
- $P_{failure}$: Failed state of a node.
- $t_{recovery}$ (t_R): The time for a node to recover from a failed state to a healthy state.
- $t_{failure}$ (t_F): A failure transition time (t_F) from a normal operation to a failed state.
- t_{switch} (t_S): Switching time from a primary to a spare and vice versa.

6.1.3.2.1. Centralized Fault Management Scheme

As shown in Figure 6.9, the centralized fault management availability model comprises a central node and a subsystem. The centralized node is the main node responsible for the fault management functions, while the primary and spare represent the nodes of an on-board computing subsystem. The details of the transition times (t_R,

t_F, t_S) and trigger condition are given in Table 6.1. The values of the transition times are extracted from the open literature and are based on the previous experience.

For a fair comparison, the centralized node is considered more reliable, and its failure transition time t_F, from robust state to faulty state, is four times that of the other nodes. Also, it is assumed that the centralized node is hot redundant with 2:1 redundancy. To represent hot redundancy of the centralized node, two tokens in the P_{robust} state are used.

As shown in Figure 6.9, each subsystem robust state contains only one token. In case of a failure of the primary, a spare token is moved to the primary space to cater for the effect of the failure. This operation requires a switching time t_S, equal to 0.3 min. On recovery of the primary node, the spare token is moved back to the node-3 space.

The centralized fault management scheme is available, if the total numbers of tokens in the P_{robust} state are greater than or equal to two as expressed by:

$$\mathrm{A}_{cent\_FM\_scheme} = \mathrm{P}[\ \#P_{cent} + \#P_{node} \geq 2] \tag{6.40}$$

Table 6.1: Parameters for the Centralized Fault Management Scheme Model.

Centralized Node	
P_{robust}	Robust state of the centralized node representing two tokens due to its hot redundant configuration.
$P_{failure}$	Failure state of the centralized node
t_R	Recovery transition time $= 10^6\ msec.$
t_F	Failure transition time = 4 x 2628000 min
Subsystem (Primary and spare node)	
Node − 2	
P_{robust}	Robust state of the node − 2 representing one token
$P_{failure}$	Failure state of the node − 2
t_R	Recovery transition time $= 10^6\ msec$
t_F	Failure transition time = 2628000 min.
Node − 3	
P_{robust}	Robust state of the node − 3 representing one token
$P_{failure}$	Failure state of the node − 3
t_R	Recovery transition time $= 10^6\ msec.$

t_F	Failure transition time = 2628000 min.
Miscellaneous	
t_1	*Conditional transition* $(\#P_{spare} \geq 1)$ *of token from spare node to primary node on a failure of primary*
t_2	*Conditional transition* $(\#P_{primary} \geq 1)$ *of token from primary node to spare node on a recoveryof primary*
t_s	Switching time = 0.3 min. from primary to redudant node and vice versa.
#	Number of tokens

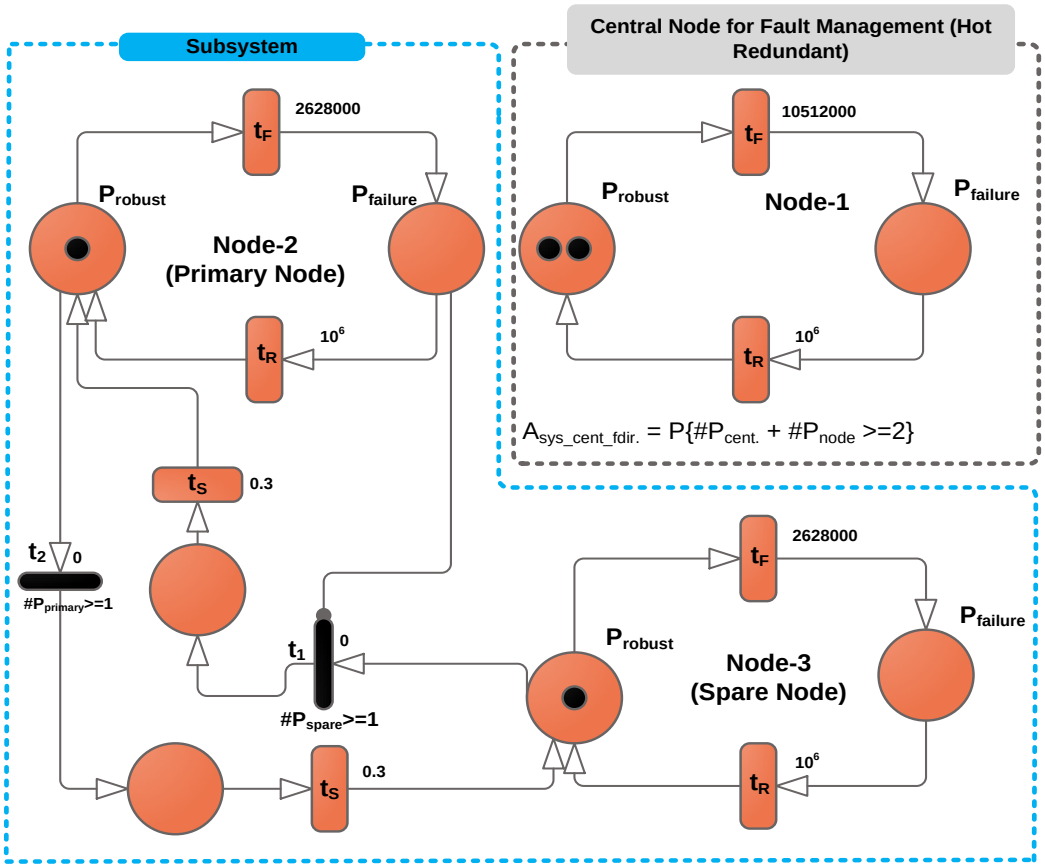


Figure 6.9: Availability Model for Centralized Fault Management Scheme.

6.1.3.2.2. Distributed Fault Management Scheme

Figure 6.10 shows the availability model for the proposed distributed fault management scheme, which comprises three nodes. Two tokens represent each node, one represents its active processor core, while the other one represents its spare

computing resources. These spare computing resources can be a complete spare core, or over-provisioned resources for a computing node. During normal operation, the active core token executes the tasks workload. On the failure of one of the nodes, the spare computing resources are utilized for the migration of the faulty node's tasks. The details of the transition times (t_R, t_F) are given in Table 6.2.

The proposed scheme is available if the total numbers of tokens in the P_{robust} state are greater than or equal to three as expressed by:

$$\mathrm{A}_{Dist\_FM\_scheme} = \mathrm{P}[\ \#P_{node-1} + \#P_{node-2}\#P_{node-3} \geq 3] \tag{6.41}$$

Table 6.2: Parameters for Proposed Distributed Fault Management Scheme Model.

$\boldsymbol{Node-1}$	
P_{robust}	Robust state of the node $-$ 1, representing a token
$P_{failure}$	Failure state of the node $-$ 1
t_R	Recovery transition time $=$ 10^6 $msec$
t_F	Failure transition time $=$ 2628000 min.
Node $-$ 2	
P_{robust}	Robust state of the node $-$ 2 representing one token
$P_{failure}$	Failure state of the node $-$ 2
t_R	Recovery transition time $=$ 10^6 $msec$
t_F	Failure transition time $=$ 2628000 min.
Node $-$ 3	
P_{robust}	Robust state of the node $-$ 3 representing one token
$P_{failure}$	Failure state of the node $-$ 3
t_R	Recovery transition time $=$ 10^6 $msec$.
t_F	Failure transition time $=$ 2628000 min.
Miscellaneous	
#	Number of tokens

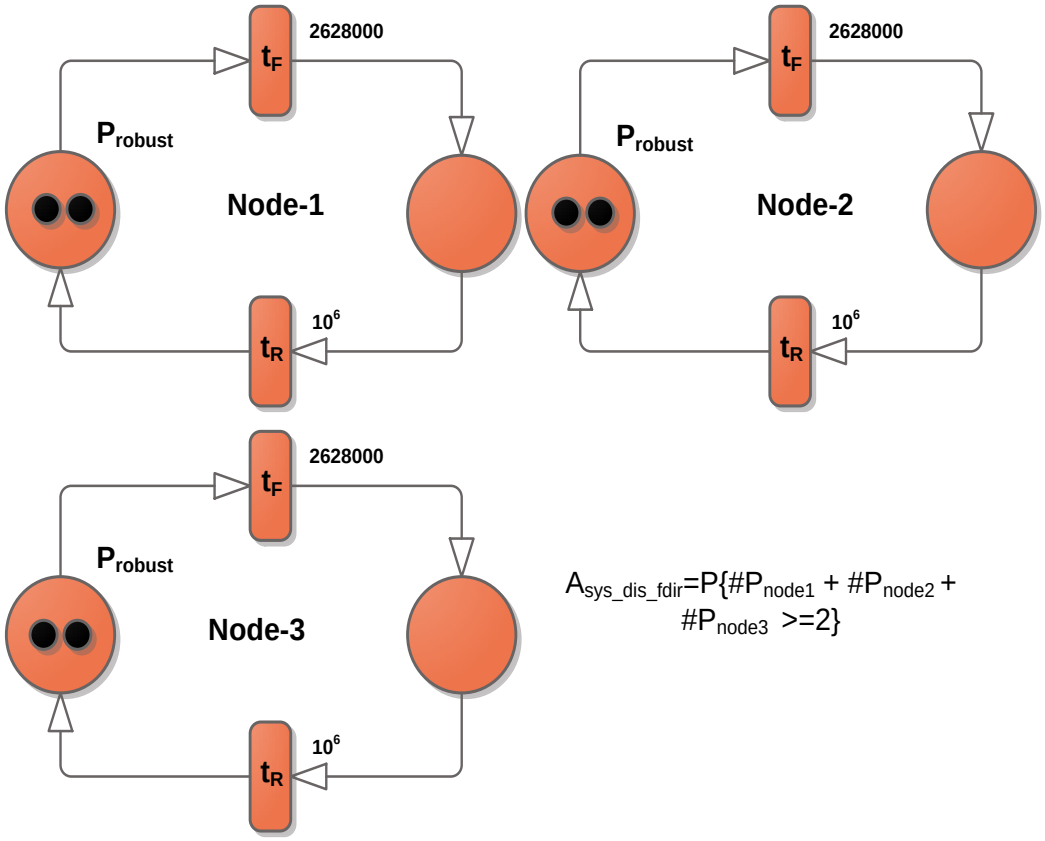


Figure 6.10: Availability Model for Distributed Fault Management Scheme.

6.1.3.3 Discussion of Availability Results

To compare the two fault management schemes, fair transition times and triggering conditions are assumed. Both models are run for several iterations with a different number of nodes. Based on the obtained availability results shown in Table 6.3 it can be concluded that the distributed scheme has a higher availability value compared to the centralized scheme.

This is obvious, because of the availability of spare computing resources at each node in the distributed case. This allows to immediately take over the lost tasks while in centralized method availability largely depends upon redundant physical nodes. It can further be concluded that the availability values increase with the number of nodes while, in the centralized scheme, availability decreases with the increase in the number of nodes.

Table 6.3: Availability Values for Centralized and Distributed FM Schemes.

	Availability of Fault Management (FM) Scheme	
Number Of Nodes	*Centralized FM Scheme*	*Distributed FM Scheme*
3	0.87174681	0.99240057
5	0.73916847	0.99909247

6.2 Functional Verification

The proposed FTDC architecture was verified functionally through prototyping, which was carried out at a board level using commercial off-the-shelf microcontroller boards and purpose-built software. A 3-node FTDC system was implemented and tested. This section provides details of the implementation setup and the testing results.

6.2.1 Distributed System Performance Metrics

The performance of the proposed FTDC architecture is evaluated in terms of Reconfiguration Time and State rollback (state age).

6.2.1.1 Reconfiguration Time

Definition: The time required to migrate tasks and resume execution following a node failure is termed as the "reconfiguration time". This is measured from the time at which a fault first happened in a node, to the time at which all the tasks of the faulty node are made runnable on other nodes.

The reconfiguration time following a node failure, t_{Reconf}, is stated as a sum of the following components.

$$t_{Reconf} = t_D + t_{FM} + t_{TX} + t_{TM} \tag{6.42}$$

Where t_D is the fault detection time; t_{FM} is the time that starts from fault detection to begin sending a fault message on the AMFT network; t_{TX} is the time required to transmit the message; and t_{TM} is the time taken for the other nodes to receive the fault

message and schedule the migrated tasks. These timing parameters are defined as follows:

$$t_D = t_{FD\_Period} + t_{FD\_Processing}$$

$$t_{FM} = (n-1) * tcs$$

$$t_{TX} = \frac{msg\_length(\ bits)}{speed(bits/sec)}$$

$$t_{TM} = tx_{AMFT2PU} + t_{sch.}$$

where

t_{FD_Period}	$Period\ for\ fault\ Detection\ task$
$t_{FD\_Processing}$	$Processing\ time\ for\ fault\ Detection$
n	$Number\ of\ Nodes$
tcs	$Duration\ of\ a\ Time\ Slot$
$tx_{AMFT2PU}$	$Time\ required\ for\ transmitting\ data\ from\ AMFT\ to\ attached\ processing\ unit.$
$t_{sch.}$	$Time\ required\ to\ schedule\ the\ migrated\ tasks.$

Adhering to the time slot-based communications on the AMFT network, fault messages can only be sent during the allocated slot. It is, therefore, possible that the time before a fault message is sent, t_{FM}, may be up to (n-1) t_{CS}. The reconfiguration time, which depends on t_{FM}, may, therefore, be large if the time slot duration is large. The system reconfiguration time should be as small as possible.

6.2.1.2 State Rollback

Definition: State rollback is caused by a temporary pause in the computation during the migration process, during which a task may rollback to a previous state when it restarts on another node. It is represented by equation 6.43.

$$n_{rollback} = \frac{t_{Reconf}}{T} \tag{6.43}$$

$n_{rollback}$ = Number of task′s State rollback

t_{reconf} = Reconfiguration Time

$T =$ Task Period

State rollback is unitless, and ideally it should be minimal. Consider an example,

Where,

$$t_{reconf} = 30 \text{ ms}, \; t_{task\_period} = 10 \text{ ms}$$

$$n_{rollback} = \frac{t_{Reconf}}{t_{task\_period}} = \frac{30}{10} = 3$$

6.2.2 FTDC Prototype Design

As shown in Figure 6.11, the board-level prototyping design consisting of three distributed nodes. The processing unit and AMFT in each DCN are implemented on two separate COTS microcontroller boards interconnected via a point-to-point interface (PPIF) which is used for the exchange of various messages between the Processing Unit and AMFT.

6.2.3 Distributed Node Prototype

The implementation of the distributed node was carried out using two STM3240G-Eval boards [214] as shown in Figure 6.12. The STM3240G board includes an ARM® CortexTM-M4F 32-bit microcontroller, memory, and peripherals. ARM CortexTM-M4F 32-bit microcontroller has a rich set of peripherals and also includes floating point unit that can be used for real numbers operations. One board is dedicated to the application tasks while the other board is used for implementation of AMFT functions. The communication between the processing unit and its associated AMFT block is through a UART interface (the PPIF). Processing Units and AMFTs have two separate CAN buses for communication.

The objective of the AMFT block is to monitor the health of the processing unit (Critical IOs, Memory Error, Watchdog timer (WDT)) and to indicate a fault condition. For such monitoring, on-chip ADC and IOs of the AMFT's microcontroller were used. Also, there is a UART (Universal Asynchronous Receiver Transmitter) interface between the processing unit and AMFT that is used for the transmission of State Update Messages and Task List Messages.

For the development of the application and the AMFT functions, the IAR Embedded Workbench (v6.3) [215] was used. First software code of an application, comprised of five tasks was developed, then the AMFT functionality, as described in Chapter 5, was coded too. The both software were developed in C using the FreeRTOS Operating System. The software was downloaded to the STM3240 boards, and testing was carried out as explained in the following section.

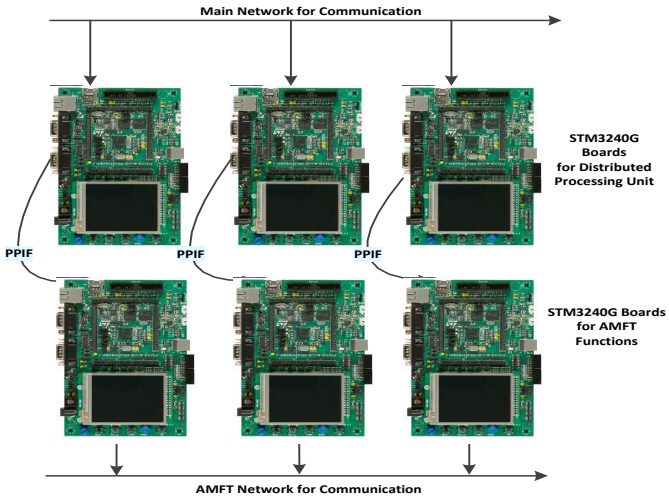


Figure 6.11: Board Level Design of Distributed Computing System.

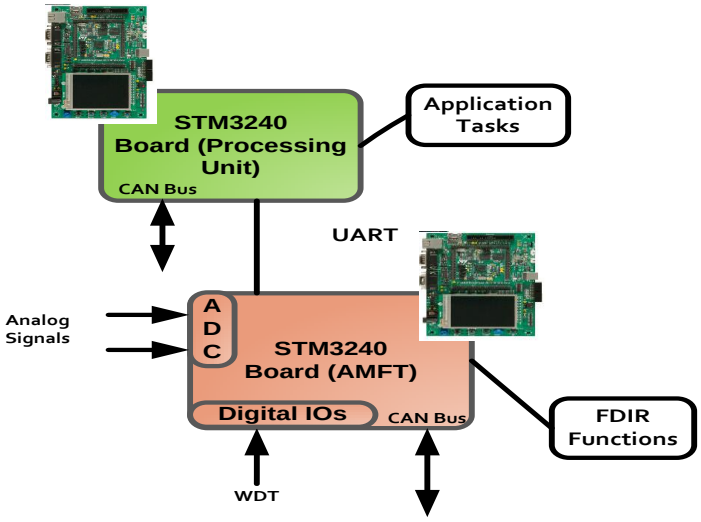


Figure 6.12: Board Level Implementation of Distributed Computing Node.

6.2.4 Distributed Computing Node Testing

Functional testing was done to verify the processing unit and AMFT functionality. Before experimenting with the overall fault-tolerant distributed system, each processing unit and AMFT were individually tested. The UART interface is used to generate test vectors and monitor the status of both the application and AMFT software. Test vectors are commands given in Table 6.4, which are required to test each testing scenario. For functional validation, status is generated which is observed in a terminal (HyperTerminal) running on a remote PC. These status messages are programmed within the application software and are named software instrumentation.

A similar process is used for the AMFT functional testing; however, its critical internal variables are also monitored. For this purpose, a JTAG interface is used along with the IAR workbench. IAR supports a live watch window facility, which allows us to monitor the internal registers of a processor running on AMFT. In this manner, we can pinpoint the exact state of the AMFT process, and also debug when necessary.

In the integrated testing, AMFT and the Processing Unit are tested as one unit. Test vectors are generated by emulating CAN messages (Heartbeat / SUM) and the states of AMFT and the Processing Unit are monitored on the Saleae Logic Analyzer [216]. The faults are injected manually on the board by pressing a button on development boards.

Table 6.4: Test Vectors.

Testing Scenario	**Test Vectors**	**Interface**
Processing Unit only	Task List Messages	UART
AMFT only	CAN emulated message (Heartbeat and State Update Messages)	CAN bus
Integrated (Processing Unit and AMFT)	CAN emulated message (Heartbeat and State Update Messages)	CAN bus

Processing Unit Functional Testing: In the first scenario, the Processing Unit functionality is verified by activating/deactivating tasks. As shown in Figure 6.13, the experimental setup for the testing includes a STM3240G-Eval board and a laptop. The actual application software was run on the STM3240G-Eval board, while the laptop

was executing the hyper-terminal software that allows us to give commands for activation and deactivation of the tasks. The successful activation and deactivation of a single, dual and five tasks are shown in Figure 6.14, Figure 6.15, and Figure 6.16 respectively.

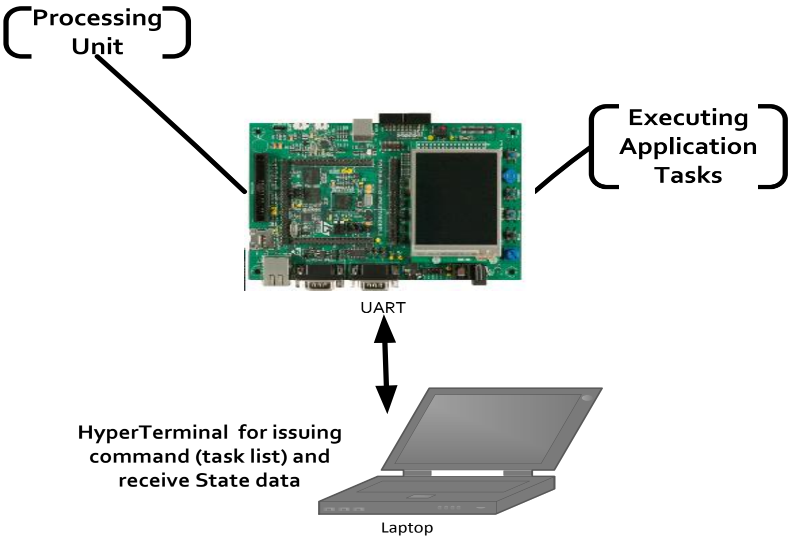


Figure 6.13: Setup for Testing of Processing Unit.

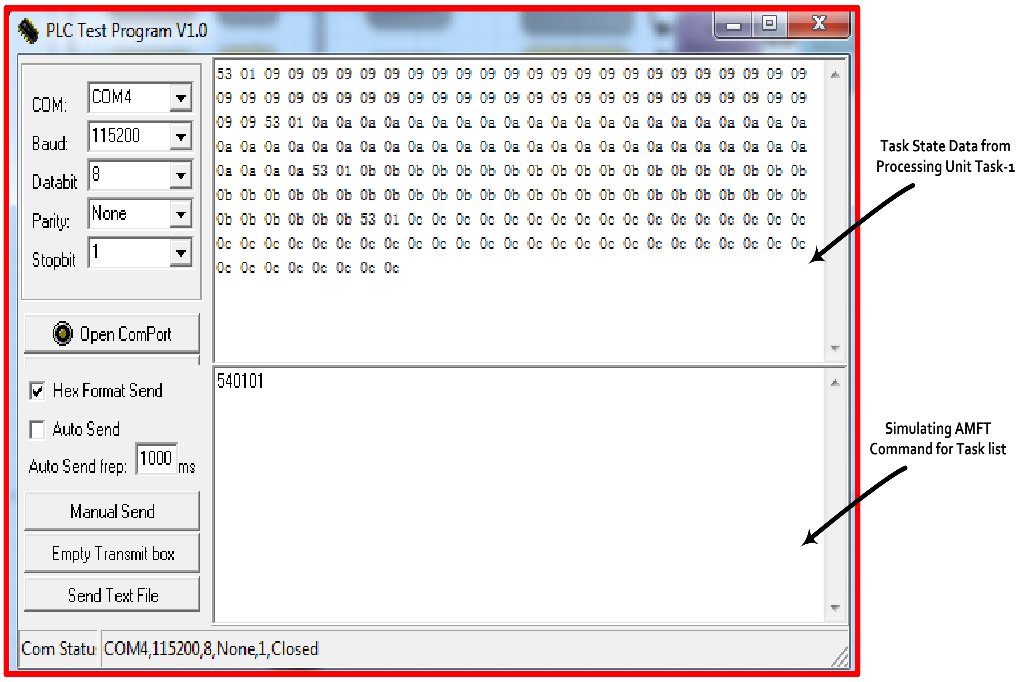


Figure 6.14: Processing Unit Functional Testing with Task-1.

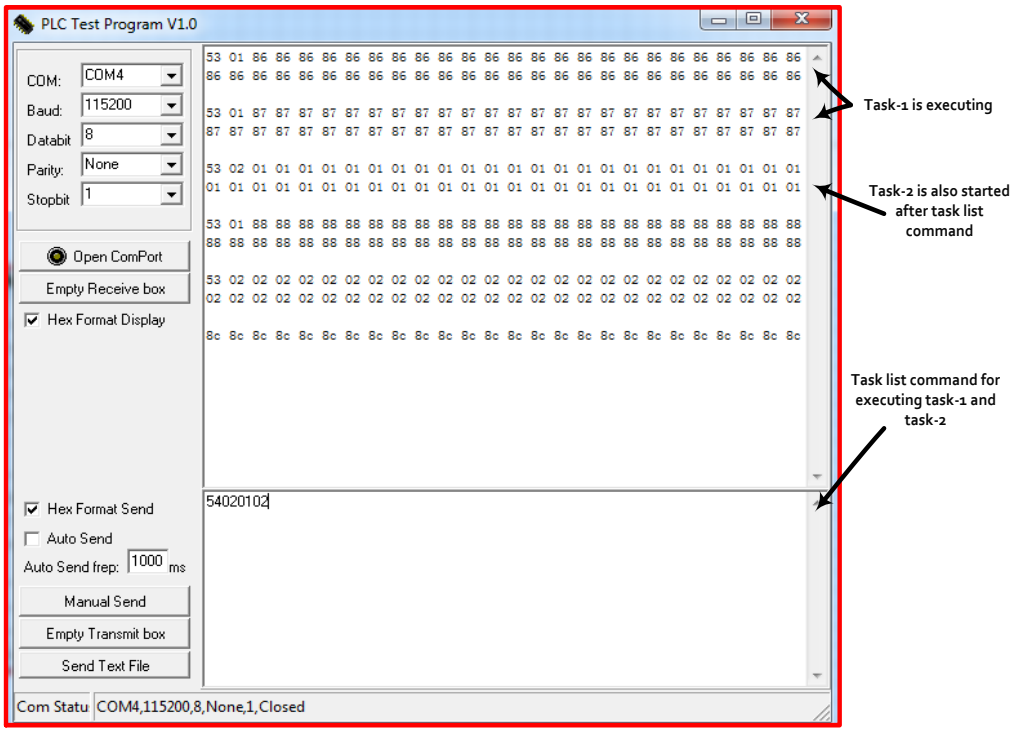


Figure 6.15: Processing Unit Functional Testing with Task-1 and Task-2.

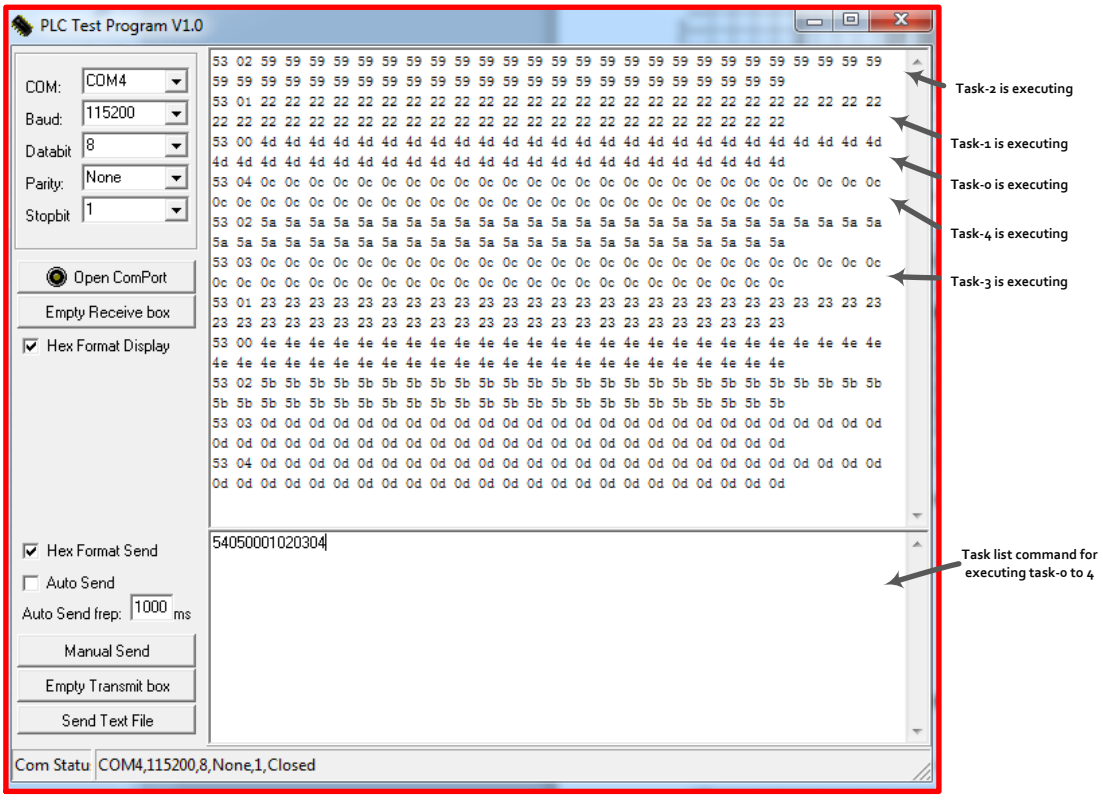


Figure 6.16: Processing Unit Functional Testing with Task-1 to 5.

AMFT Functional Testing: In the second scenario, an experimental setup was developed to test the AMFT functional behaviour as shown in Figure 6.17. To emulate the faults caused by WDT and memory (digital IOs), two buttons were used. One button was used to inject a fault while the other was used to remove a fault. A potentiometer was used to emulate the node temperature variation that indirectly provides node health information. On either of the failures (over temperature, WDT, memory error) AMFT detects a failure and deactivate tasks running on the processing unit. To capture the internal behaviour of the AMFT, variables corresponding to state data Δ_{SD}, node status (ucFaultStatus) and task allocation were observed in the live watch window as shown in Figure 6.18 (IAR debugging). This depicts the true internal state of the AMFT, when no fault. Figure 6.19 shows a situation when a fault is injected. On a fault, AMFT deactivates all the tasks and changes its node's status to faulty (ucFaultStatus=0x01).

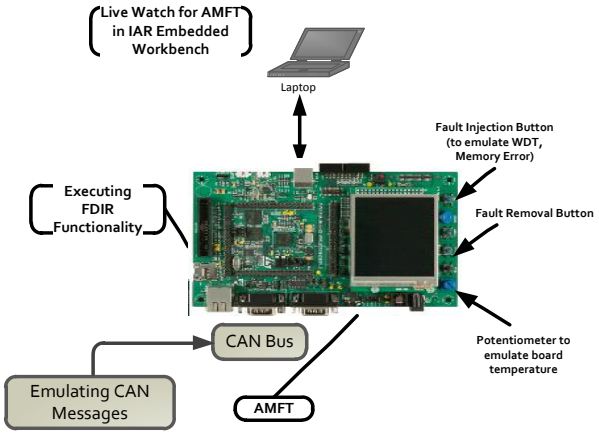
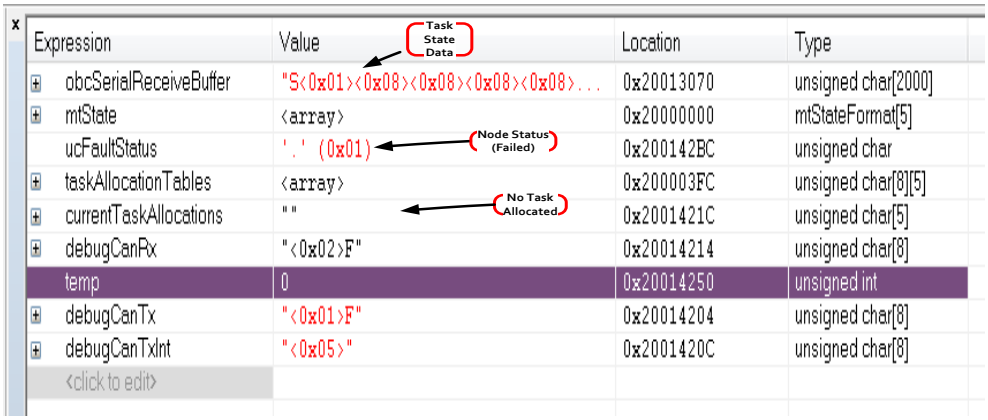


Figure 6.17: Setup for Functional Testing of AMFT Unit.

Expression	Value	Location	Type
obcSerialReceiveBuffer	"S" (Task State Data)	0x20013070	unsigned char[2000]
mtState	<array>	0x20000000	mtStateFormat[5]
ucFaultStatus	'.' (0x00) (Node Status (Healthy))	0x200142BC	unsigned char
taskAllocationTables	<array>	0x200003FC	unsigned char[8][5]
currentTaskAllocations	"<0x01><0x01><0x01><0x01><0x01>" (All Task Allocated)	0x2001421C	unsigned char[5]
debugCanRx	"<0x02>F"	0x20014214	unsigned char[8]
temp	0	0x20014250	unsigned int
debugCanTx	"<0x01>H"	0x20014204	unsigned char[8]
debugCanTxInt	"<0x0e>"	0x2001420C	unsigned char[8]
<click to edit>			

Figure 6.18: AMFT Memory View Captured by IAR, when node was healthy.



Expression	Value	Location	Type
obcSerialReceiveBuffer	"S<0x01><0x08><0x08><0x08><0x08>...	0x20013070	unsigned char[2000]
mtState	<array>	0x20000000	mtStateFormat[5]
ucFaultStatus	'.' (0x01)	0x200142BC	unsigned char
taskAllocationTables	<array>	0x200003FC	unsigned char[8][5]
currentTaskAllocations	""	0x2001421C	unsigned char[5]
debugCanRx	"<0x02>F"	0x20014214	unsigned char[8]
temp	0	0x20014250	unsigned int
debugCanTx	"<0x01>F"	0x20014204	unsigned char[8]
debugCanTxInt	"<0x05>"	0x2001420C	unsigned char[8]
<click to edit>			

Figure 6.19: AMFT Memory View Captured by IAR, when node was faulty.

6.2.5 Fault-Tolerant Distributed System Prototyping

Table 6.5 shows the configuration setup for the implementation of the distributed system with three nodes, five mission tasks and 1000 ms slot time for TDMA communication over Controller Area Network (CAN) at 1Mbps. It is prototyped with six STM3240G [214] evaluation boards as shown in Figure 6.11. Three STM3240G evaluation boards are used to implement the processing units (top of the picture) and three boards are used to implement the AMFT block. All three boards of the AMFT blocks communicate through a CAN bus interface referred to as AMFT network while the STM3240G containing the application software communicates through another CAN bus interface called main network. The communication between the Processing Unit and its associated AMFT block is through a UART interface (the PPIF).

Software Setup: The AMFT block running on each board STM3240G implements the algorithms as described in section 5.2. We use FreeRTOS for the AMFT implementation to exploit parallelism in software. The software stack includes applications software and fault management software as outlined in section 4.6. The application software mainly includes five application tasks: Attitude determination & control, power management, thermal management, payload management, telemetry/telecommand. However, the total number of application tasks to be executed by the system can be varied, as well as the characteristics of each task. The main task characteristics are periodicity, duration and state data length. The "state" of a task comprises a set of values that must be preserved for a future execution of the task. The application tasks are all periodic, similar to many spacecraft on-board computing tasks

[217]. For the purposes of this prototyping effort, each application task is periodic with a period of 1000 ms [218] and includes 150 bytes of state data Δ_{SD}. The main option requiring a trade-off to be made is the number of distributed nodes to be used. For the on-board computing architecture, increasing the number of processing nodes will increase the reliability of the system. This is traded against the increased resources required for additional nodes, such as power consumption and mass. For prototyping purposes, the tasks are mapped to a three node distributed computing system. The communications slot time depends on the nature of the tasks, which is explained in section 8.4.1. The fault management functions are implemented according to the software structure, introduced in section 5.5. For the software development, the embedded workbench (v6.3) from IAR Systems aas selected. The software development is written in the C programming language.

Table 6.5: Configuration Setup for Prototyping of FTDC System.

Parameter	***Value***
Number of Nodes	3
Total Number of Mission Tasks	5
Communication Slot Time (ms)	1000 ms
TDMA Cycle Time	(1000 x 3) ms
Internode Communication	CAN Bus, 1 Mbps

6.2.6 Experimental Results

Key aspects of the distributed system that have been investigated including the ability to migrate successfully tasks and resume execution following the failure of a node. The reconfiguration time required to migrate tasks is an important performance parameter that is recorded for each possible scenario. In addition, during the migration, state rollback for each task is also recorded.

6.2.6.1 Scenario-I: Processing Start-up

The time required after the power-up to the final execution of the tasks is called Start-up time. For the measurement of this time, two pins are used as follows:

- AMFT Board: ARM CORTEX STM32F407 GPIO (GPIOA-PIN-7)
- Processing Unit Board: ARM CORTEX STM32F407 GPIO (GPIOA-PIN-8)

An oscilloscope monitors the time duration between the two pins just after the Start-up. Table 6.6 shows the start-up time measurements for the three nodes.

Table 6.6: Scenario-I: Results on Start-up Time Measurements.

Node	**Time (seconds)**
1	***6***
2	5
3	5

6.2.6.2 Scenario-II: Failure

6.2.6.2.1. Failure of One Processing Unit

The AMFT block performs monitoring of the processing unit failures. To emulate hardware failures within the processing unit, we use two I/O switches on the AMFT board to represent failure within a processing unit. Also, two GPIO pins are used for the measurement of reconfiguration time as follows:

- AMFT Board: ARM CORTEX STM32F407 GPIO (GPIOA-PIN-7)
- Processing Unit Board: ARM CORTEX STM32F407 GPIO (GPIOA-PIN-8)

Failure of a processing unit is inserted by pressing one push button on the AMFT board. The AMFT detects this failure and toggles the output to PIN-7 to indicate a failure condition. This is the start of the reconfiguration time. The reconfiguration ends when the failed processing unit tasks are successfully migrated to other units. We toggle the output of another pin on the healthy processing units to indicate that tasks have been successfully migrated. The time between these two conditions is observed by oscilloscope and is given in Table 6.7. We observed that the reconfiguration time is always less than the TDMA cycle (3000 ms).

In case of a failure, ideally each task has to resume its state from the point at which execution ended on the failed processing unit. However, due to the TDMA cycle time

of 3000 ms, it is not possible to resume the state from that point onward if a task period is short. This effect is worse for tasks having very short periods.

6.2.6.2.2. Failure of AMFT Block

In this scenario, an AMFT is considered to be failed in a fail-stop manner. During normal operation, each AMFT is required to send Heart Beat Messages to the other AMFTs to communicate its presence. If there is no Heart Beat Message from an AMFT when it expects it, that AMFT is considered to be failed and a reconfiguration of the distributed system begins which ends with the successful migration of tasks to other processing Units.

As can be seen from Table 6.8, the reconfiguration time and task state rollback are similar to those obtained for the OBC unit failure case above.

Table 6.7: Scenario-II: One Processing Unit Failure.

Processing Unit Failure	***Reconfiguration Time (ms)***	***Migrated Mission Task***	***Task State Rollback***
Processing Unit-1	1110	Mission Task-1 Period = 1000 ms	1 to 3
	840	Mission Task-2 Period = 1000 ms	1 to 3
Processing Unit-2	2020	Mission Task-3 Period = 1000 ms	1 to 3
	2100	Mission Task-4 Period = 1000 ms	1 to 3
Processing Unit-3	2800	Mission Task-5 Period = 1000 ms	1 to 3

Table 6.8: Scenario-II: Failure of AMFT Block.

AMFT Unit	***Reconfiguration Time (ms)***	***Task State Rollback***
AMFT-1	≈ 200 to 2300	1 to 3
AMFT-2	≈ 1000 to 2180	1 to 3
AMFT-3	≈ 300 to 2900	1 to 3

6.2.6.2.3. Failure of Two Processing Units

The simultaneous failure of two processing units is considered in this scenario. In this case, we simultaneously insert a failure in both Processing Units. The failure insertion is emulated by pressing two push buttons on the two AMFTs. Similar to the one processing Unit failure scenario, the reconfiguration time is observed by oscilloscope using the same pins, and the LCD display observes the state age. The results of our observations are again the same as those presented for the one processing unit and AMFT failure scenario.

6.2.6.3 Scenario-III: Recovery

If a processing unit is recovered after failure, it can be reintegrated into the distributed system. We emulate the recovery process by pressing another button on the AMFT. When the button is pressed, the AMFT assumes that its processing unit has been recovered following a failure and starts the reconfiguration process by sending a Heartbeat Messages to the other AMFTs via the CAN Bus.

6.2.6.3.1. Recovery of One OBC Unit

This scenario represents the recovery of one processing unit after a failure. The recovery process starts by pressing a button on the AMFT. This shows that its associated AMFT is recovered. After this, the AMFT starts sending Heartbeat messages to the other AMFTs. The other AMFTs update their node tables and send updated task lists to their Processing Units. Similarly, the recovered node's AMFT sends an updated task list to its recovered Processing Unit. The results obtained for reconfiguration time and task state rollback are similar to those obtained in the case of one processing unit failure scenario.

6.2.6.3.2. Recovery of One AMFT Unit

The recovery of the AMFT unit after a temporary fault starts by sending a Heartbeat Message on the CAN network. After receiving the Heartbeat Message, each of the other AMFT units updates its node table and sends an updated task list to its associated processing unit. The recovered AMFT also sends an updated task list message to its associated processing unit. The test was carried out by removing a held reset button on the AMFT. This process reintegrates the AMFT into a distributed system, and tasks related to its associated processing unit are reallocated back. The results obtained for

the reconfiguration time and state age after recovery of the AMFT are shown in Table 6.9.

Table 6.9: Scenario-III: Recovery of AMFT Block.

Recovered OBC	***Reconfiguration Time (ms)***	***Task State Rollback***
AMFT-1	≈ 560 to 1000	1 to 3
AMFT-2	≈ 456 to 2750	1 to 3
AMFT-3	≈ 1106~2226	1 to 3

6.2.6.3.3. Recovery of Two Processing Units

This is a scenario when both of the processing units are recovered simultaneously after a failure. For performing such a test, the recovery of two processing units is emulated by simultaneously pressing the recovery button on each of the two node's AMFT boards. The important aspect of the result is the same reconfiguration time irrespective of the number of nodes as shown in Table 6.10.

Table 6.10: Scenario-III: Simultaneous Recovery of Two Processing Units.

Recovered Processing Unit	***Reconfiguration Time (ms)***	***Task State Rollback***
Processing Unit-1&2	≈ 1000 to 2080	1 to 3
Processing Unit-1&2	≈ 1300 to 3000	1 to 3
Processing Unit-1&2	≈ 1100 to 2620	1 to 3

6.2.7 Implementation Issues

During the implementation of the board level design of distributed computing systems, following critical issues were identified.

- **Computational Performance:** Standalone processors interconnected at board level require more size, area and electrical power resources and are also computationally inefficient because of the insufficient resource sharing and limited interconnect speed.

- **State Transfer Issue:** The state of the task, as described in section 5.3.3, is necessary for its resumption when a task is migrated to another node. The size (in bytes) of the state data Δ_{SD} depends upon the nature of the task - it may be of a few bytes or may be of several bytes. In board level design as shown in Figure 6.20, the state data $\Delta_{SD}$of a task need to be stored in two places before being transferred to the other nodes. First, the data is stored in the local memory of the processing unit and secondly, in the local memory of the AMFT. In addition to the dual storage, the transfer of data consumes physical (CPU bus bandwidth, DMA, UART) and computational resources that affect the performance of the actual application.
- **Task Scheduling Issue:** Each node is pre-configured to share the computing workload of the failed node as described in section 5.2.4. When a task is migrated to another node, it needs to be added into the existing scheduler of the OS. All tasks are statically pre-scheduled for all possible scenarios. It was observed that adding a task in a running schedule, disturbs the overall Scheduler time, a known issue among the research community [219]. Although our pre-scheduled method succeeded in addressing the possible scenarios, it was observed, that the scheduler needs a standalone processing.

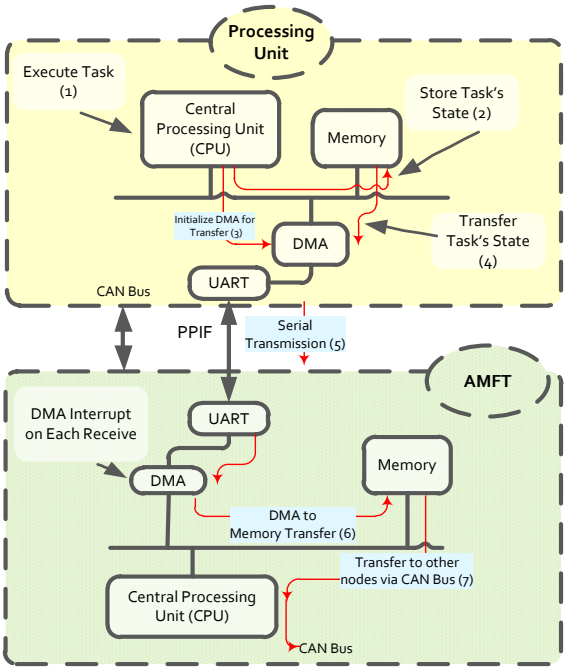


Figure 6.20: Task State Data Flow.

6.3 Summary

Fault-tolerant systems are of a complex nature and require to be analysed before development. In the first part of this chapter, a novel approach based on mathematical models was developed to analyse the reliability and availability of fault-tolerant computing systems. The results on the reliability showed that the proposed distributed computing approach is more reliable in comparison to the centralized and TMR based systems. This is due to the fact that the proposed distributed computing architecture is reconfigurable which allows task migration in the case of a processor failure. Fault management schemes were also compared and analysed in terms of reliability and availability. It is evident from the results that by distributing the fault management functions, much higher reliability and availability values can be obtained.

The second part of the chapter presents a functional verification of the FTDC system, which was carried out through quick prototyping at board level, providing a proof of concept for the proposed architecture. The testing setup allows observations of the system behaviour at run-time. It verifies the proposed approach at the functional level, including the tasks migration in case of a node failure or following a recovery. It proves that the tasks are being added to or dropped from the existing scheduler. In addition, it validates that the AMFT is reconfigured to account for the changes in the FTDC. Furthermore, the prototyping allowed initial debugging of the software. It also helped to identify implementation issues which were not foreseeable at the architectural level design stage.

Chapter 7

Novel MPSoC Based Design for Fault-Tolerant Distributed Computing

In this chapter, a novel multi-processor system-on-chip (MPSoC) based design for the proposed fault-tolerant distributed computing approach is presented. The design is targeted at modern FPGAs, which incorporate hard processor IP cores and programmable logic fabric on the same chip. The design is an upgraded version of the FTDC concept presented in Chapter 4, providing new enhanced features, which are enabled by the technology. It serves also as validation of the proposed approach and is used in the space related case-study in Chapter 8. In Section 7.1, the need and benefits of the MPSoC design is discussed. Section 7.2 presents details on the MPSoC design covering operational scenarios, block diagram and selection of FPGA. The hardware design of the MPSoC is presented in Section 7.3. The MPSoC software implementation is elaborated in Section 7.4. The MPSoC fault injection mechanism is discussed in Section 7.5. Experimental setup and results on the MPSoC based design are reported in Section 7.6. A CubeSat payload based on the MPSoC is presented in Section 7.7.

7.1 Why MPSoC Design?

It is evident from the reviewed literature that fault-tolerance against permanent failures of distributed system is provided through redundant resources. This replication of the

hardware resources incurs a very high cost for fault-tolerance. This cost can be reduced by utilizing over-provisioned resources in each node as presented in Section 6.2. However over-provisioning resources on a node is not sufficient enough for migrating all the faulty node's tasks. This limitation of resources forces us to migrate tasks to multiple nodes, which creates problems of inter-tasks communication dependencies and enhances the communication on the network. Due to inter-tasks communication, the performance of the overall application can be severally degraded. However, a multicore processor based distributed system design naturally eliminates this problem by migrating the tasks of the failed node to a single core. In addition, MPSoC reduces the damage by utilizing the power of multicore embedded processors. So this design is more cost effective than the others where a complete physically redundant node is used for fault-tolerance purposes. Furthermore, in an MPSoC design each distributed computing node acts as a target fail-over node for the other node, achieving much higher reliability.

The number of cores in a multicore processor depends on the requirements. Each additional core provides more computing resources, which can be utilized for computational performance or fault-tolerance. However, managing a large number of cores requires more design efforts and must be justified in terms of complexity and heat dissipation.

An MPSoC may be developed as an Application Specific Integrated Circuit (ASIC) chip or by using an FPGA. Both approaches have their advantages and disadvantages. ASIC is much faster but it is inflexible for design changes. On the other hand, the FPGA based approach is more flexible in terms of prototyping and also allows in-orbit reconfiguration. Therefore, the FPGA based approach is opted for in the design of the fault-tolerant distributed architecture, proposed in this thesis. Furthermore, a hybrid FPGA that includes a multicore processor and programmable fabric is selected. Hybrid FPGA allows mapping of fault management functions in the programmable logic separate to the Processing Unit. Thus it defines a clear boundary between the Processing Unit and the fault management functions, which makes the design more reliable. Additionally, the programmable fabric can also be used for hardware acceleration of the computational intensive functions. The use of a hybrid FPGA in space depends on the underlying technology and criticality of the mission. Although, in

the past, FPGAs have been used in space for many spacecraft missions, the SRAM based FPGAs are vulnerable to space radiations, particularly the Single Event Upsets (SEU). An SEU can flip a bit either in the configuration or data memory which may cause catastrophic effect. Although, the FPGA based approach is adopted but at this stage, if required, the same design can be used as a way to prototype an ASIC.

The first academic MPSoC design realised a real-time embedded system for automotive applications, which was prototyped on an FPGA in 2009 as part of the GENESYS project funded by the European commission [210].

7.2 Description of the MPSoC Based Fault-Tolerant Distributed Computing Design

A proof of concept for the FTDC architecture and preliminary functional testing was achieved by the board-level implementation, presented in section 6.2. The intended functionality was achieved, however, in section 6.2.7 additional issues that would improve the performance, were also observed, which are taken care of in the MPSoC design.

Figure 7.1 shows a distributed computing system, each node which is implemented as an FPGA based MPSoC. The design of the MPSoC consists of a hard multicore processor (in this case dual core) and main memory, as well as soft middleware IP and modules implemented on the FPGA programmable fabric. The dual-core processor is used for the execution of the application tasks. During normal operation, one of the cores of the dual-core processor runs a fraction of the application task set, while its associated core is idle. The idle core can share the workload in case of failure of a node in the distributed system. The shared memory handles the communications between the two cores for resource sharing. The middleware for the fault management functions, as detailed in Chapter 5, is implemented as a soft hardware IP. A system bus interface for the middleware is provided to access the memory attached to the multicore processor. This interface allows the middleware to retrieve data in case of a failure. Two separate networks are used – main network and AMFT network. The former is used for the communications between the processing units while the latter is used for the communications of middleware blocks.

To proceed with the MPSoC design and implementation, operational scenarios are specified, followed by a Block Diagram to address physical interfaces, and functional mapping. Then a suitable target FPGA is selected, to map each of the functions described in the Block Diagram. To have a functional MPSoC, the implementation follows a prescribed flow and ECAD tool suite associated with the particular FPGA. Finally, each module is implemented and integrated. The functionality is validated by performing a series of tests. The implementaion of the MPSoC is characterised by parameters, such as electrical power and logic resources consumption, which should ideally be as minimal as possible.

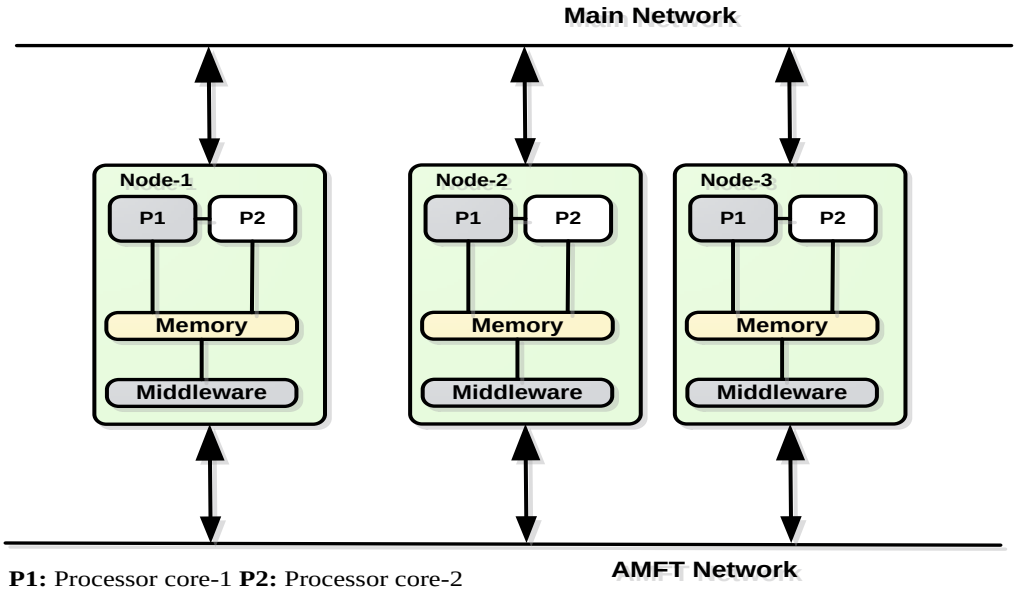


Figure 7.1: Distributed System Configuration.

7.2.1 MPSoC Operational Scenarios

7.2.1.1 Normal Scenario

The first scenario represents the normal behaviour of the MPSoC operations as depicted in Figure 7.2. A number marks each operation for this scenario. A complete data flow is marked from 1 to 3 (red colour circles) representing reception side, message into the system. Data flow from 1 to 4 (blue colour rectangles) represents 'transmit direction'. In the receive direction, first CAN messages (Heartbeat, Fault,

and State Data) were emulated to be sent to the MPSoC under test via CAN AMFT interface. These messages are periodic and follow the TDMA scheme. Then received messages are separated into Heartbeat and state update message. A correct receive of heartbeat message indicates a healthy node that does not require any update in the node table. The receive data in state message is moved to main memory, indicating other node state update.

On the transmit side of the node, each task writes its state to the main memory. Later, this state is read by AMFT and then it is sent to the CAN bus. During the whole process, task execution, task state data Δ_{SD} and hardware signals were monitored. Each task is instrumented to send data on a serial console for monitoring and debugging purposes during task execution. The hardware signals of the advanced high-performance bus (AHB), advanced extensible interface (AXI) and advanced peripheral bus (APB) were also monitored via on chip JTAG on a separate PC, as shown in Figure 7.2.

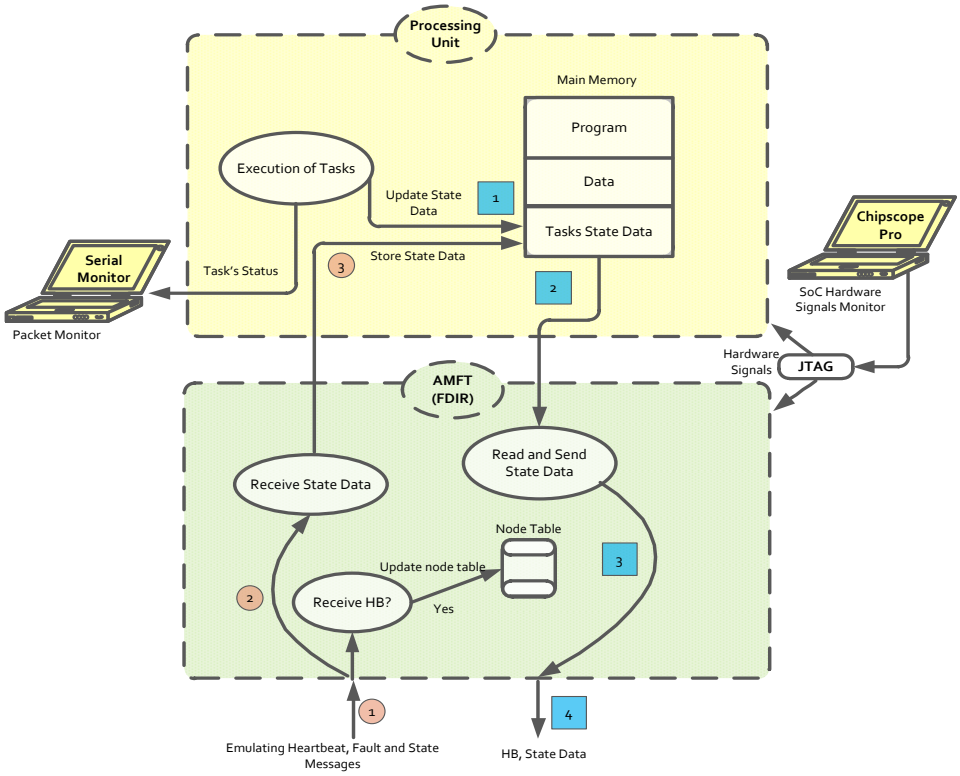


Figure 7.2: Data Flow in a Normal Scenario.

7.2.1.2 Task Migration Scenario

This scenario represents the effect of a node failure on the other node. The task migration scenario, as marked from 1 to 7 steps, is shown in Figure 7.3. The failed node stops sending HB, which indicates a fault condition in that node. To observe the same scenario, CAN emulator acts as a failed node and stops sending messages on the bus. Inside the AMFT, node failure was observed via no HB message within allocated slot. The first action after no HB, an entry for that particular node was immediately removed by updating node table. Afterward, action for task migration was started. This being follows a task list preparation and its transmission to add/drop operation. Add/drop operation adds tasks into the Scheduler, and simultaneously taking spare processor core out of sleep mode. The complete operation of task migration was monitored by hardware monitoring and software packet capturing. To validate the correct behaviour of this scenario, following test points were observed.

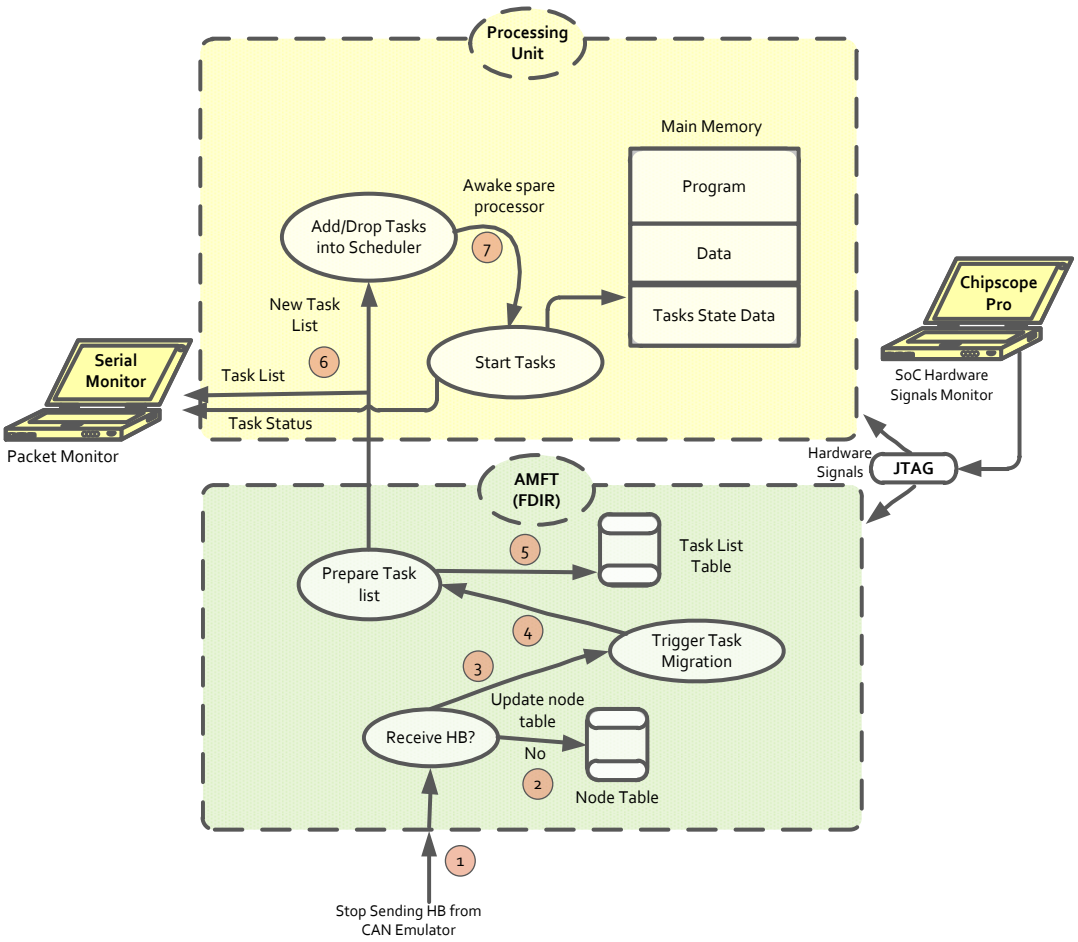


Figure 7.3: Task Migration Scenario

- Task list from AMFT is sent to the processor, which emulates failed node tasks. This task list is captured on the fly and routed to the serial console for monitoring AMFT behaviour.
- Each task is instrumented to send messages on the serial console to indicate its state during execution.
- Task migration also involves hardware signal monitoring such as AHB, APB activities to ensure all operations. This operation was done using Chipscope Pro as shown in Figure 7.3.

7.2.1.3 Fault Detection and Isolation

This scenario represents a situation, where a node detects its failure and isolates it from the rest of the system. Figure 7.4 illustrates this scenario. The health of each node in the MPSoC implementation consists of one analogue signal (emulating node temperature) and two digital signals (emulating WDT and main memory error). Both analogue and digital signals are constantly read and checked to determine fault in the node. This is carried out in terms of defined limits of temperature values, as soon as a high or low temperature is reached, a fault signal is generated. WDT status is also read, if a predefined state (set to low in our case) is received then the processor is deemed faulty, or in an undetermined state. A fault is injected by intentionally exceeding one of the signal limits, manually.

To verify the correct behaviour of AMFT for this scenario, Task List Message (TLM), task status, and hardware signals are monitored. On a fault, an empty TLM that was generated for processing unit was observed. On receipt of the TLM, all tasks were immediately deactivated and observed task status was stopped. Also, shutdown signal was also observed as marked 7 in Figure 7.4.

The detailed verification is carried out as a case study in chapter-8 where a distributed OBC was designed, implemented and tested.

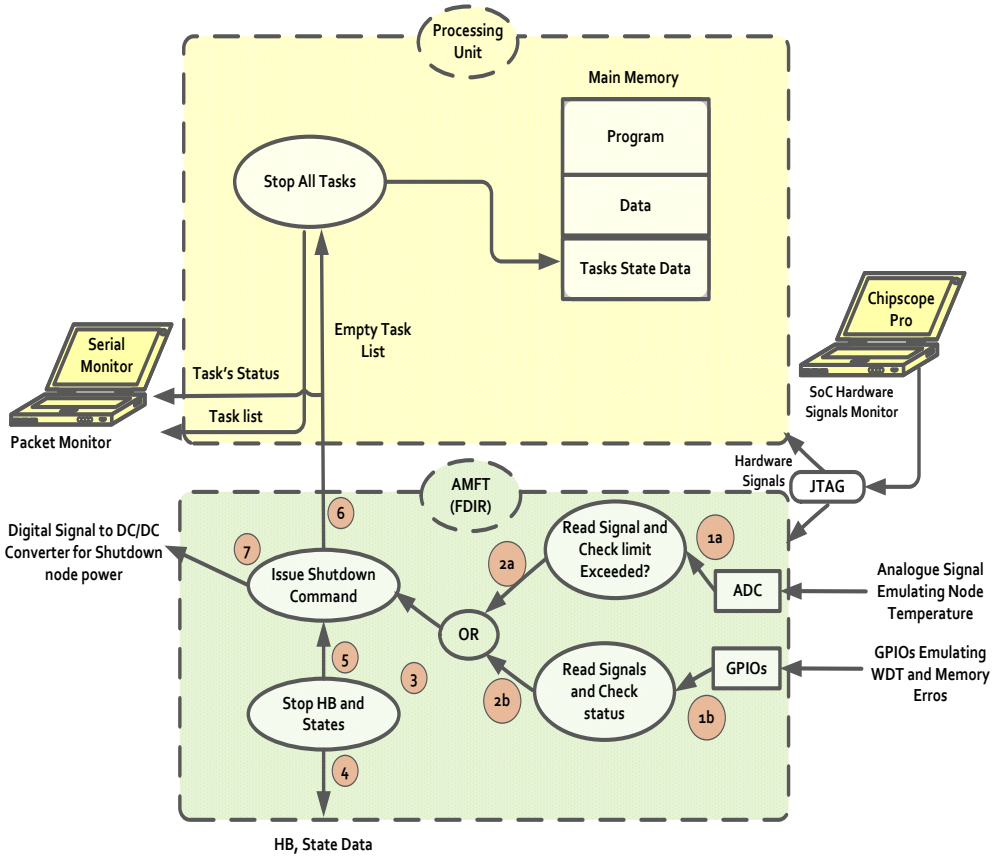


Figure 7.4: Fault Detection and Isolation Scenario

7.2.2 MPSoC Block Diagram

The functional implementation of proposed approach as an MPSoC design is shown in Figure 7.5. It represents a block diagram of a distributed computing node, in which each function of the proposed approach is mapped.

In this design, the single processor of the board level implementation is replaced with dual processors. One of the processors is usually executing the application tasks while the other is reserved to share the task load of the faulty node. In this scheme, each processor has its own task scheduler, thus eliminating the problem of scheduling.

An AMFT block is connected to the on-chip system bus that connects it to main memory for accessing state data directly. This MPSoC design reduces the time and resources to transfer state information between the AMFT and processing unit.

The memory scheme of the MPSoC comprises an internal and an external memory. The internal memory is used for faster access to shared data between the

processors, while the external memory, accessed by AMFT and the processors via the memory controller, is used to store the application data.

For connecting peripherals, two separate buses (peripheral bus-1 and peripheral bus-2) are suggested, which are attached to various on-chip devices. An analogue-to-digital converter (ADC) is included to monitor the node health for the purpose of detecting a failure of the node.

Two network interfaces are used for external communication of the processing units and AMFTs.

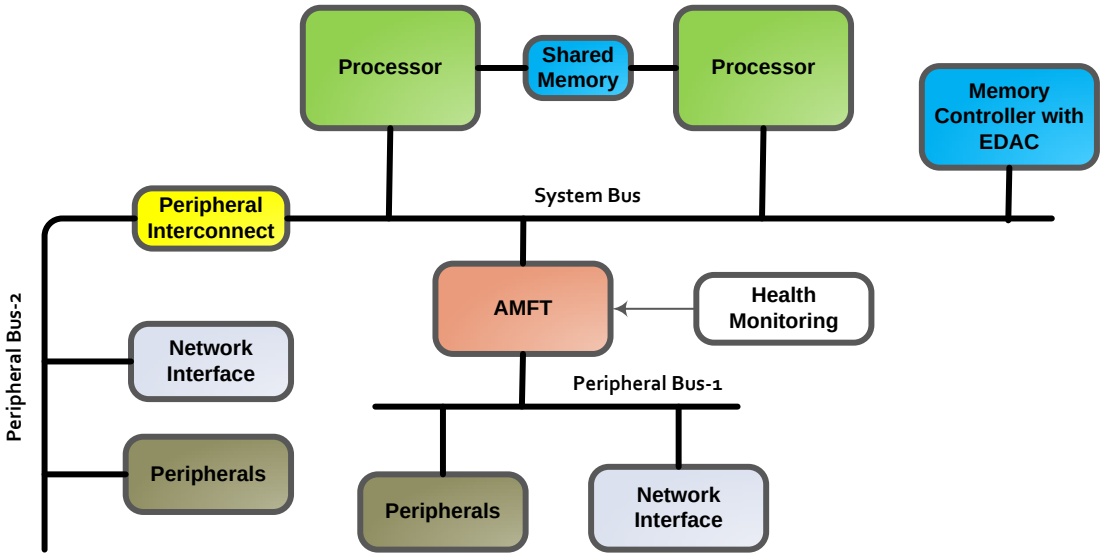


Figure 7.5: Block Diagram of the MPSoC Design.

7.2.3 Selection of FPGA Based MPSoC Device

There are primarily three FPGAs manufacturers who support SoC-based design implementation, namely, Xilinx, Microsemi, and Altera. Each manufacturer has its set of design tools that support synthesis, which varies in complexity and usage. Furthermore, all of them support a number of on-chip functions that vary between families of FPGA. The requirements of proposed MPSoC design are on-chip hard processor IP and mixed-signal processing. Another important selection criterion is the availability of Intellectual Property (IP) cores. We compared three FPGAs most suited to our requirements as tabulated in Table 7.1. Of the many choices, the Xilinx FPGA

is selected, due to its known heritage in space applications. Also, a vast majority of the available IP cores, experience in its design tools and lastly for its mixed-signal processing capability.

Another aspect is the choice of implementation i.e. the use of higher language used in programming of our functionality. We opted for utilizing readily available IP cores from Xilinx. The available IP cores helped to reduce significantly the design effort in mapping the distributed computing functions. This is the key reason for the successful implementation of the proposed MPSoC scheme on time. The selected Xilinx FPGA and its development tools are listed in Table 7.2.

Table 7.1: SoC FPGAs

	Xilinx Zynq-7000 FPGAs	***Altera SoC FPGAs***	***Microsemi SmartFusion2 FPGAs***
Processor Type	ARM Cortex-A9	ARM Cortex-A9	ARM Cortex-M3
Single or Dual Core	Dual	Single or Dual	Single
FPGA Fabric and Logic Density	Artix-7, Kintex-7, 28 K to 444 K Logic cells	Arria, Cyclone V, 25 K to 462 K Logic Elements	Fusion2, 6 K to 146 K Logic Elements
External Memory Error Correcting Code (ECC)	Yes	Yes	Yes
On-Chip RAM	256 KB, no ECC	64 KB with ECC	64 KB, no ECC
Floating-Point Unit/NEON Multimedia Engine	Yes	Yes	Not Available
Analog Mixed Signal	2 x 12 Bit, 1 MSPS Analog-to-digital converters (ADCs)	Not Available	Not Available

Table 7.2: Design and Development Tools and Target Board for MPSoC Implementation

Device/Tool	***Description***
FPGA	Xilinx Zynq-7000 EPP FPGA (XC7z020clg484-1) [220]
Implementation Tools	Xilinx Platform Studio v 14.5, Software Development Kit (SDK) v 14.5
Debugging	Chipscope Pro [221]
Development Board	ZEDBOARD [222]

7.3 MPSoC Hardware Implementation

Figure 7.6 shows the MPSoC design implementation. The upper part of the MPSoC implements the design of the processing unit that maps its functions to the processing system of the Zynq FPGA. The processing unit consists of two hard ARM Cortex A9 processor IP cores. Both cores act as a single distributed processing unit - one is active while the other is in sleep mode. Each processor core is running an OS, for which a readily available FreeRTOS [223] was used. The software of each processor follows the design stack described in section 4.6. To exchange messages among the two processors, the shared memory of the processing system was used. Central interconnect was used to allow access to the main memory by both processor IP cores and AMFT, thus allowing sharing of main memory.

The lower part of Figure 7.6 is the programmable logic where the AMFT is implemented. AMFT is realized on a Xilinx soft processor MicroBlaze. The MicroBlaze processor is connected to block RAM (BRAM) and advanced extensible interface (AXI) interconnect. BRAM contains the program and data for the AMFT software while the AXI-Interconnect acts as a bus, allowing MicroBlaze access to peripherals (timers, UART, GPIOs, CAN). All peripherals interrupts are routed via a central interrupt controller that selects an interrupt source for the processor. For self-monitoring of the distributed computing node, on-chip ADC (a hard macro) and GPIOs (emulating watchdog timer, memory error) are used.

For debugging purposes, a debug module, allowing on-chip software debugging of MicroBlaze via JTAG interface was used. The circuit diagram of the MPSoC implementation is shown in Figure C.3 (Appendix C). During the implementation, logic resources, and electrical power consumption were measured.

7.3.1 Logic Resources

After the final mapping, placement and routing of the design, the resource utilization for MPSoC was determined. These comprise the processing unit and the AMFT resources. The former mainly consist of the multicore processor, while the latter are the form of slice lookup table (LUT), slice registers, and memory as reported in Table

7.3, which shows that very few resources for the AMFT implementation are required. A detailed report for logic resources is given in Appendix C.

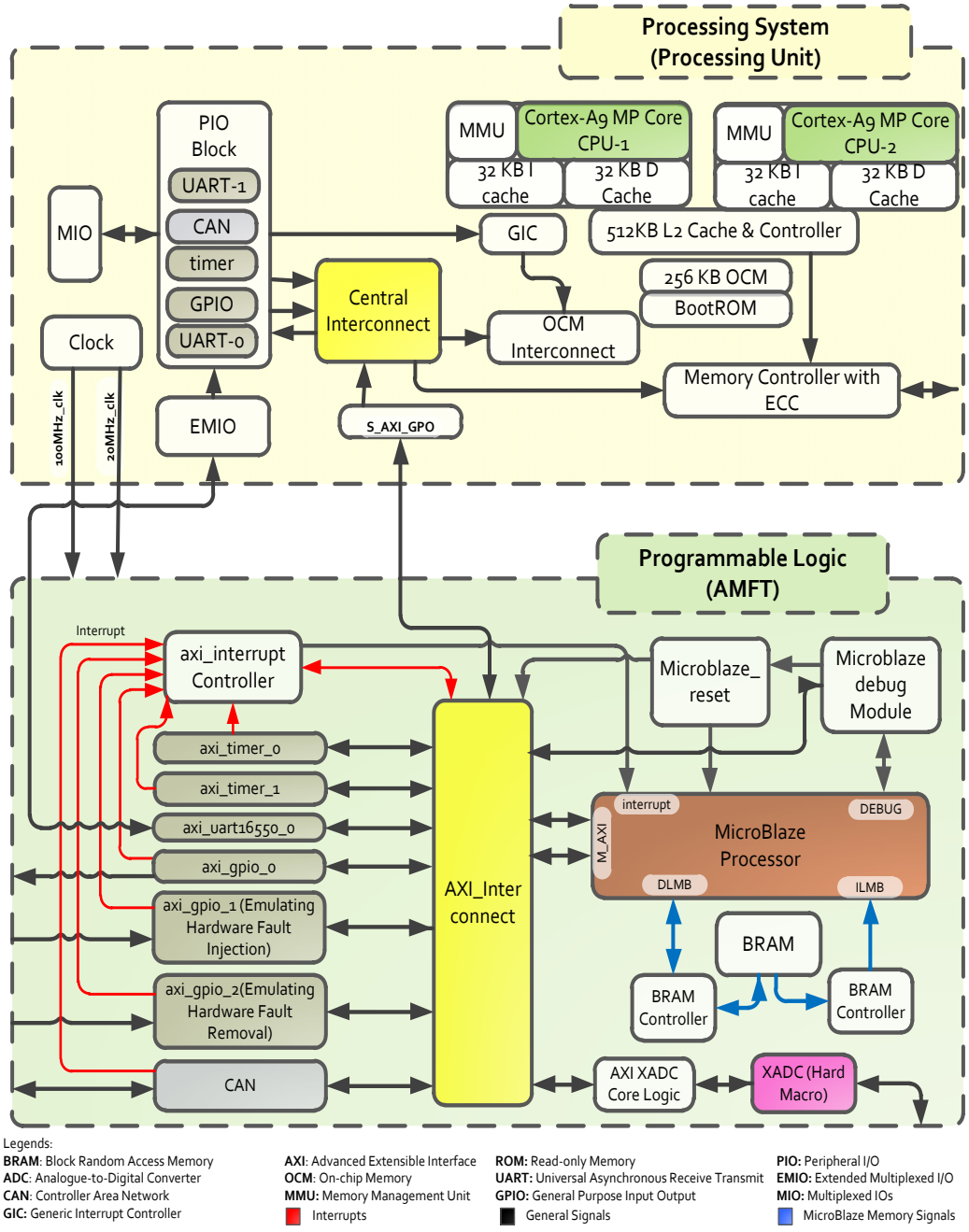


Figure 7.6: MPSoC based Implementation of a Distributed Computing Node.

Table 7.3: Logic Resources.

Computing Unit	*Resource*	*Utilization*	*Available*	*Utilization %*
Processing System (PS)	Multicore Processor	2	2	100

Programmable Logic (PL) (AMFT)	Slice LUTs	4855.0	53200.0	9
	Slice Registers	4663.0	106400.0	4
	Memory	18	140	13

7.3.2 Electrical Power Consumption

The electrical power consumption for the implementation of MPSoC was estimated using the Xilinx XPower tool [224], as shown in Figure 7.7. Electrical Power was obtained for the Zynq (Z-7020) Artix-7, 28 nm technology [225]. It is evident from the results that a small amount of electrical power, approx. 180mW, is required for the AMFT, while 1384mW is needed for the processing unit, in which the main power is consumed by the DDR memory.

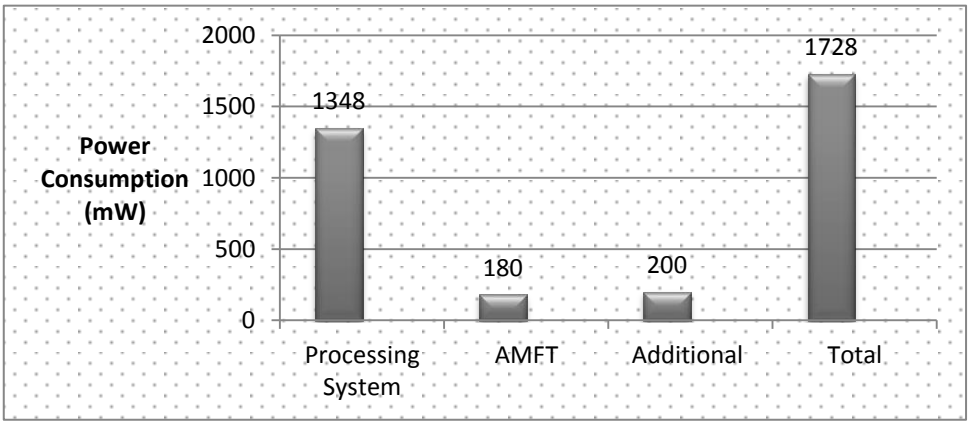


Figure 7.7: MPSoC Electrical Power Consumption.

7.4 MPSoC Software Implementation

This section describes the MPSoC software development, which consists of two parts: (i) a processing unit application and (ii) an AMFT application.

7.4.1 Application Software

The processing unit functionality is implemented in C, as a FreeRTOS application executing on the multicore ARM processor on the Zynq FPGA. The main purpose of this application is to control the execution of the computing tasks as requested by the

unit's associated AMFT. The processing unit application consists of the following main functional modules:

- *AMFT Sender* and *AMFT Receiver*, which handle the communications with the AMFT unit.
- *Mission Task Manager*, which controls the execution of the mission tasks based on the requests of the AMFT unit.
- *Mission tasks* – in the distributed system prototype, the total number of mission tasks to be executed by the system can be varied, as well as the characteristics of each task. The main task characteristics are periodicity, duration, and state data Δ_{SD} length. The "state" of a task comprises a set of values that must be preserved for future execution of the task. The mission tasks are all periodic, similar to many spacecraft on-board computing tasks. Each mission task performs an operation which involves incrementing each byte of its state data.

Each processing unit in the distributed computing system has an identical implementation, and the software code for every mission task is present in every processing unit. While the code for every task is present, tasks are started and stopped (by the Mission Task Manager) at run-time as required so that a different sub-set of tasks executes on each processing unit.

The processing application-specific code is located in 8 source files, as listed in Table 7.4. This shows the application-specific source files. Also, the FreeRTOS source code and development board support files are required. Most of the application-specific source files also have a corresponding header file, which has the same name as the source file name (e.g. amftReceiverTask.c has a corresponding header file amftReceiverTask.h). In most cases, the header files contain only function definitions. Figure 7.8 graphically shows the organization of the software, logically grouping files into related sets.

Table 7.4: File list for Processing Unit Application.

File	***Contents***
amftReceiverTask.c	AMFT Receiver task
amftSenderTask.c	AMFT Sender task

ftdc_lcd.c	Code to control output to the board's LCD
main.c	main function; initialization code
missionTaskManagerTask.c	Mission Task Manager task
missionTasks.c	Code for all mission tasks
missionTaskStates.c	Mission task state data
serial.c	UART interrupt handler

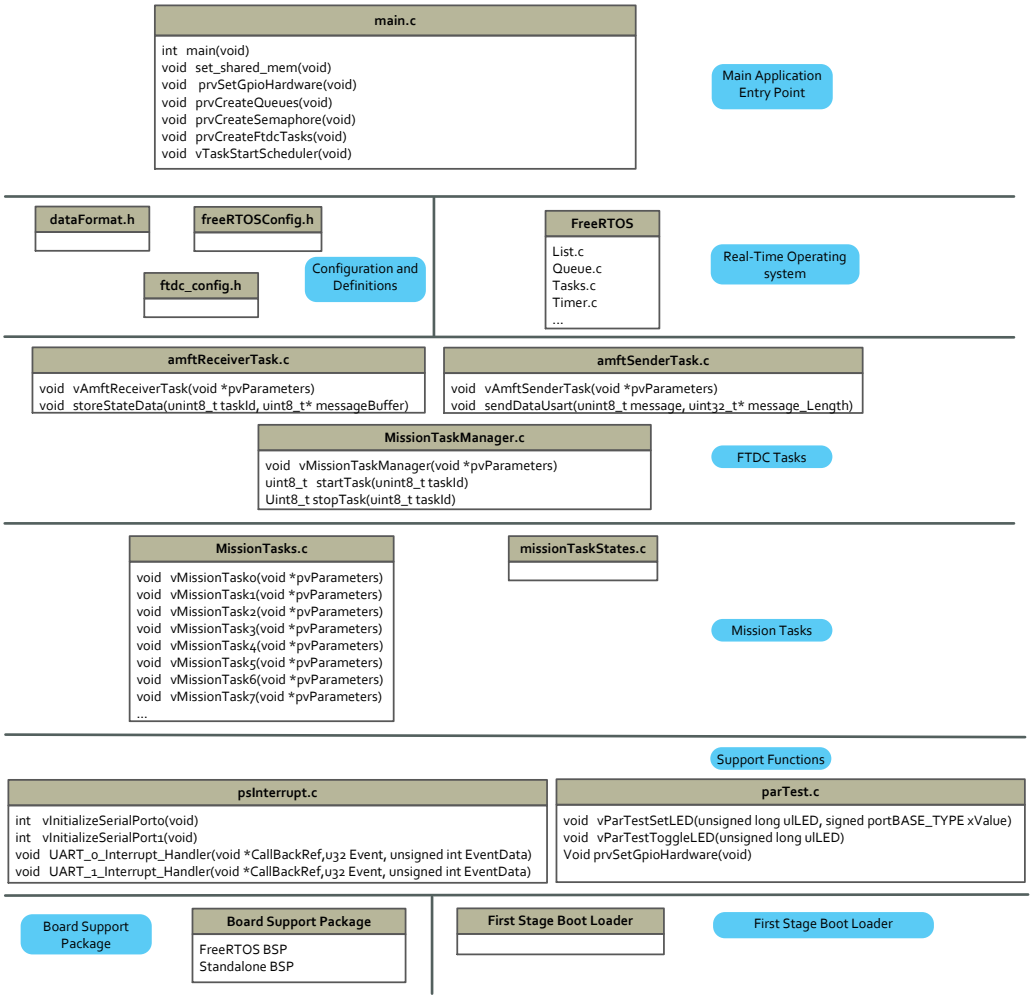


Figure 7.8: Application Software Structure.

7.4.2 AMFT Software

The AMFT is a FreeRTOS-based software application created in C, which executes within the MicroBlaze on each MPSoC board. The implementation of AMFT within each MPSoC board is identical except for a stored node identifier that is used to identify each AMFT uniquely.

The AMFT monitors for faults in its associated processing unit and handles the distributed functionality such as migrating tasks following a node failure. The AMFT application implements the AMFT algorithms as described in section 5.5. Figure 7.9 shows an overview of the AMFT application structure. The main modules are:

- *obcSender* and *obcReceiver Tasks*, which handle the communications with the processing unit.
- *AMFT Sender* and *AMFT Receiver*, which handle the low-level communications with the other AMFT units via the CAN bus.
- *AMFT Comms*, which handles the high-level communications with the other AMFT units, such as communications slot management.
- *Task Allocation Manager*, which handles determining which mission tasks should be executed on the node's processing unit.
- *FDIR*, which monitors for faults in the processing unit.

The overall AMFT functionality, i.e. the algorithms described in section 5.2, is implemented through the functions of the above modules and the interactions between them.

The AMFT application-specific code is located in 13 source files, as listed in Table 7.5. Also, the FreeRTOS source code and development board support files are required. Most of the application-specific source files also have a corresponding header file, which has the same name as the source file name (e.g. amftCommsTask.c has a corresponding header file amftCommsTask.h). In most cases, the header files contain only function definitions. Figure 7.9 graphically shows the organization of the software, logically grouping files into related sets.

Table 7.5: File list for the AMFT Application

File	***Contents***
amftCommsTask.c	AMFT Comms task
amftReceiverTask.c	AMFT Receiver task
amftSenderTask.c	AMFT Sender task
AdcInterrupt.c	Interrupt handler for reading ADC channels data
com_can.c	Interrupt Handlers for CAN transmit and receive, and

	initialization code for CAN interrupts.
com_ser.c	Interrupt Handler for UART, and initialization code for UART interrupts.
parTest.c	Initialization code for LEDs
fdirTask.c	FDIR task
ftdc_config_data.c	Global variables used for system configuration
main.c	main function; initialization code
obcReceiverTask.c	OBC Receiver task
obcSenderTask.c	OBC Sender task
taskAllocationManagerTask.c	Task Allocation Manager task

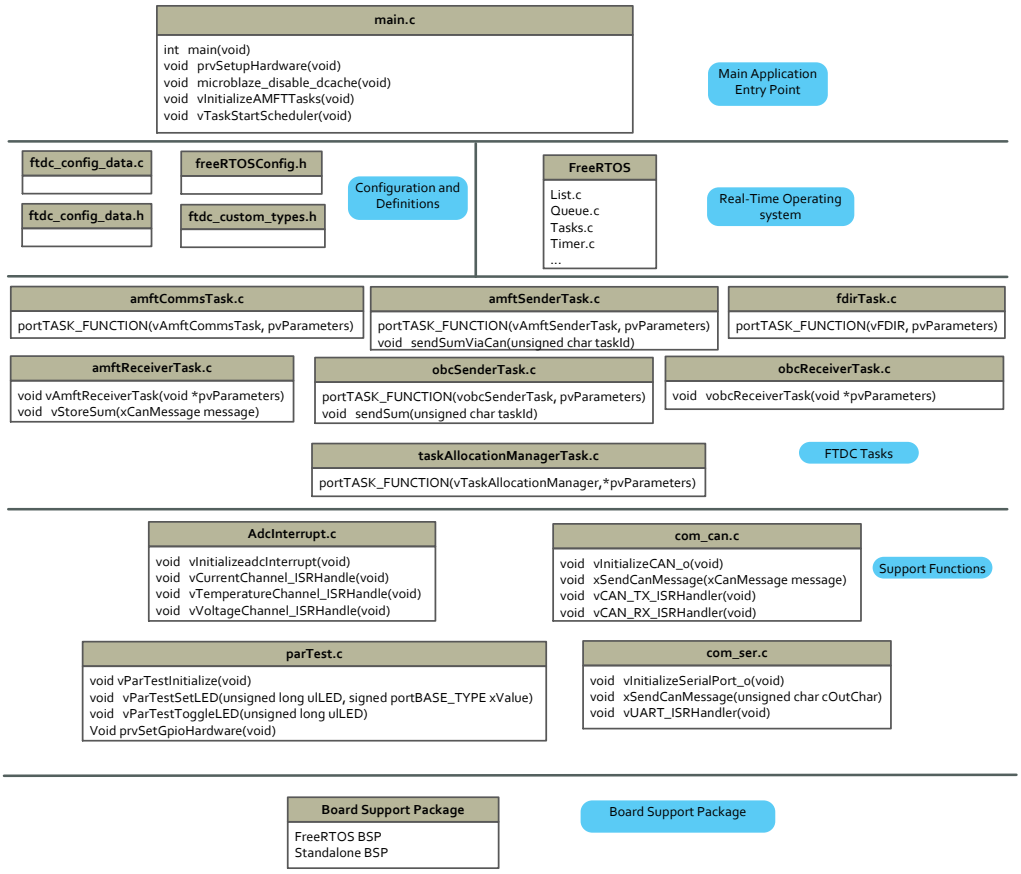


Figure 7.9: Structure of AMFT Software

7.4.3 AMFT Software Overhead

The overhead for the implementation of AMFT in terms of code memory and data memory is shown in Table 7.6. These sizes were obtained for the case-study of AOCS application presented in Chapter 8. It concludes that the code memory size remains

fixed, however, the size of the data memory and non-volatile stable storage depends upon the checkpointed data volume.

Table 7.6: Overhead of AMFT

S. No.	*Component*	*Required Size (KB)*	*Implementation Size (KB)*
1	Code Memory	12.316 KB	16 KB
2	Data Memory	5.81 KB	8 KB
3	Non-Volatile Stable Storage	0.2 KB	1 KB

7.5 MPSoC Fault Injection Mechanism

A symptom-based approach for the detection of software faults is proposed in section 4.8. To test the fault detection methods in the case of the MPSoC design, a new fault injection mechanism was required which is capable to inject transient as well as permanent faults in the MPSoC. Both types of faults are defined in section 2.1.4. Given this requirement, a fault injection mechanism was developed, which is capable to inject faults in any of the distributed computing nodes. This mechanism is not only helpful for the validation of the fault detection algorithms, but it is also useful for the overall validation of proposed FTDC approach. The following sections present the details of the both fault injection mechanisms.

To test the approach to software-based fault detection of section 4.8, a mechanism to inject transient faults was developed, as shown in Figure 7.10. This mechanism comprises of two parts. The first part is running on the host computer and was developed as a Windows Form Application in Visual Studio 2013. This provides a Graphical User Interface (GUI) to inject a fault in the processing unit software. A snapshot of the GUI is shown in Figure 7.11, which allows a user to select a distributed computing node, processor core, registers for the injection of faults. After selecting parameters, the user can send a command by pressing a ‘send’ button. Due to the bus architecture, this command message is broadcast to all processing units of the FTDC system. However, the UART of each processing unit can discard the message at

a hardware level without interrupting the processor, if the address of the processing unit inside the message is not matched.

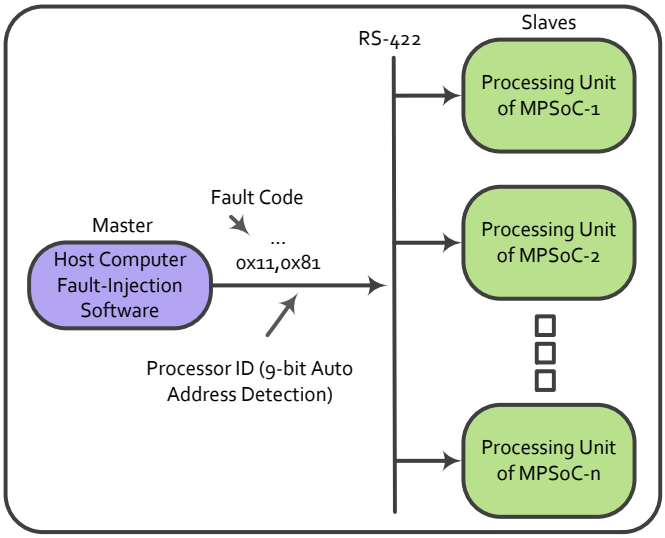


Figure 7.10: Fault Injection Mechanism.

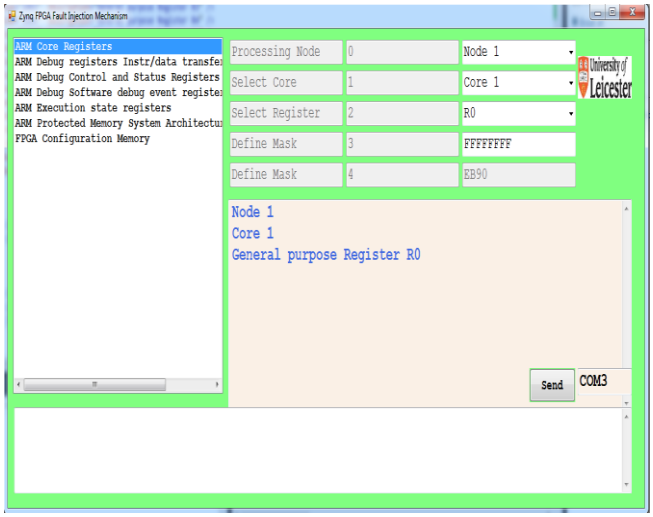


Figure 7.11: Host Software for Fault Injection.

7.5.1 Transient Fault Injection

To inject transient faults in the running application software, the mechanism shown in Figure 7.12 was developed. The mechanism can inject faults in the processor registers, peripheral registers and the data content of the main memory. This mechanism

receives a fault injection command via the RS-422 interface from the host computer software. After the command reception, the command message is placed on a queue 'QueueforFaultMsg' for later use by the fault injection task. The fault injection task 'TaskforFaultInject' is a low priority task that executes only, if no other task is ready to execute. The execution time for the interrupt service routine (ISR) is carefully controlled, and it does not last for more than 150 ns. This small period of time does not affect the execution of the actual application tasks. The execution of 'TaskforFaultInject' injects a fault in the processing unit of MPSoC. Once the fault is injected, the detection method, proposed in section 4.8, gets activated and detects the fault anomaly.

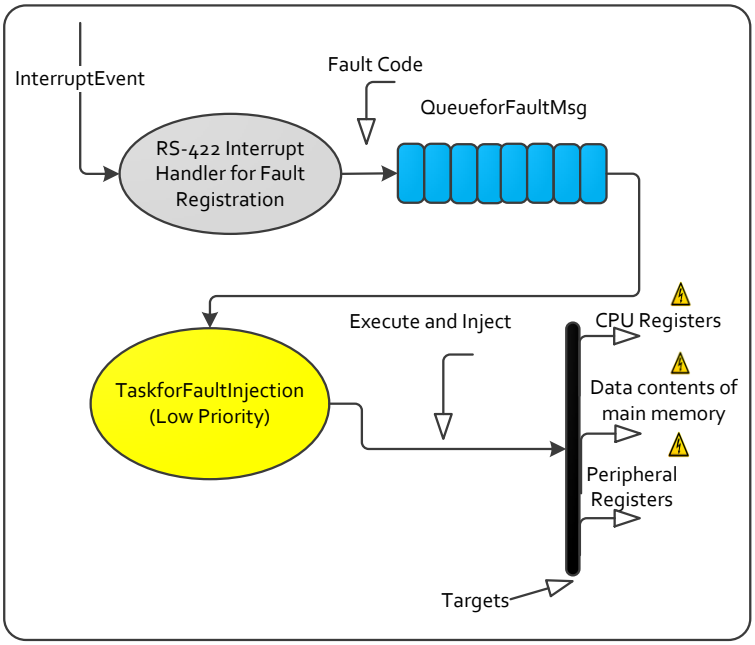


Figure 7.12: Transient Fault Injection Mechansim.

7.5.2 Permanent Fault Injection

A permanent fault causes an error that leads to the failure of the system. In case of the MPSoC distributed computing system, there can be various reasons for permanent faults. The common causes are the malfunctioning of memory, processor and watchdog timer (WDT). The malfunctioning of the components can be caused by internal structural failures, particularly a fault in the processor registers. To develop

such a fault injection mechanism to cover all components is a very laborious task, and so in this thesis, we only cover permanent fault injection-related failure of a processor. This mechanism is essential for the validation of the permanent SDC fault detection algorithm, presented in section 4.8.2.

Fault injection of permanent SDC errors is not possible in the hard processor. Therefore, a soft IP MicroBlaze processor was used for that. To test the proposed permanent fault detection method, a complete embedded system based on the MicroBlaze processor was designed and implemented on the programmable logic side of the Zynq FPGA, as shown in Figure 7.13. In order to inject a fault into the configuration memory of the implemented system, an existing Soft Error Mitigation (SEM) controller from Xilinx was integrated with the design to provide access to the internal configuration access port (ICAP) of the Zynq FPGA. This core is capable to receive commands via a simple UART. Also, it is designed to connect with the ICAP interface for the injection of a fault at any location of the configuration memory of the Zynq FPGA. The detailed circuit diagram is shown in Figure C.4 (Appendix C).

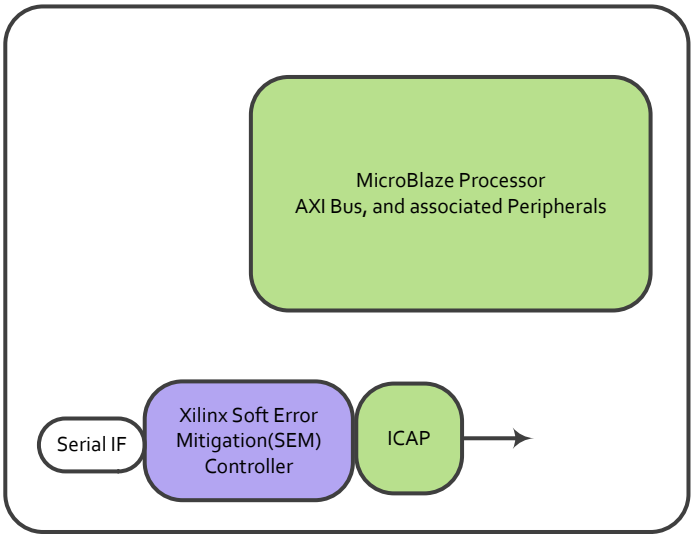


Figure 7.13: Permanent Fault Injection Mechanism.

7.6 Experimental Setup and Results

Experimental Setup: The experimental setup for the distributed system demonstrates the required system behavior, i.e. tasks are migrated to healthy computing nodes when

a node fails and upon recovery. The details of the configuration for the setup are given in Table 7.7. It comprises three distributed nodes, implemented on the Xilinx Zynq FPGA devices of three ZEDBoard, connected via communication networks as shown in Figure 7.14. Each node is implemented as an MPSoC that includes the Zynq ARM dual-core processor and MicroBlaze processor. The application software runs on the ARM dual-core processor, while the fault management software runs on MicroBlaze. Both the application and fault management software are implemented in C using FreeRTOS.

Table 7.7: Prototyping System Parameters.

Parameter	***Value***
Number of Nodes	3
Mission Task Set	Simulated
AMFT Communication Slot Time (ms)	100 ms
AMFT TDMA Cycle Time	(100 x 3) ms
Inter-AMFT Communication Network	CAN @ 1Mbps
Main Bus/Network	CAN @ 1Mbps
Development Board	ZEDBoard

Mission Task Set: For the purpose of this prototyping effort, a simulated mission task set, based on representative spacecraft on-board computing tasks, was implemented, which is summarized in Table 7.8. The main task characteristics that were considered were task period *T* and state data size ts_{size}. We assumed that all tasks *v* are periodic and requires executing after a certain period. Furthermore, the worst case execution time (WCT) of each task (v_1, v_2 …) is less than or equal to its period *T*. The state of a task comprises a set of values that must be check-pointed for the future execution of the task. It is similar to a priori knowledge, which is required to get the current output values from a task. The task state size ts_{size} and the value of the state depend on the nature of the task. In the mission task set, we assume that all tasks are periodic with a different task period, T , and state data sizes ts_{size}. The data values for each task state are initially equal to zero. Each mission task periodically updates its state data and

outputs a condensed form of the current value of the state data on the serial terminal (the condensed form is generated by summing all the bytes comprising the state data).

Rationale and Assumptions: The proposed approach deals with fault-tolerance at an architectural level, where the failure of an entire computing node is compensated for. Fault-tolerance within a node and at the communication level is not considered. We assume that the network and the implementation of the AMFT are fault-tolerant. A failure of a node can be a temporary failure caused by single event effects (SEEs), or it can be a permanent failure caused by malfunction of electrical components. Also, we assume that the implementation of the AMFT block is dual-redundant, and its failure behaviour is fail-silent. Furthermore, a failure of an AMFT results in a failure of the complete node. However, the failure of a processing unit is handled by its associated AMFT.

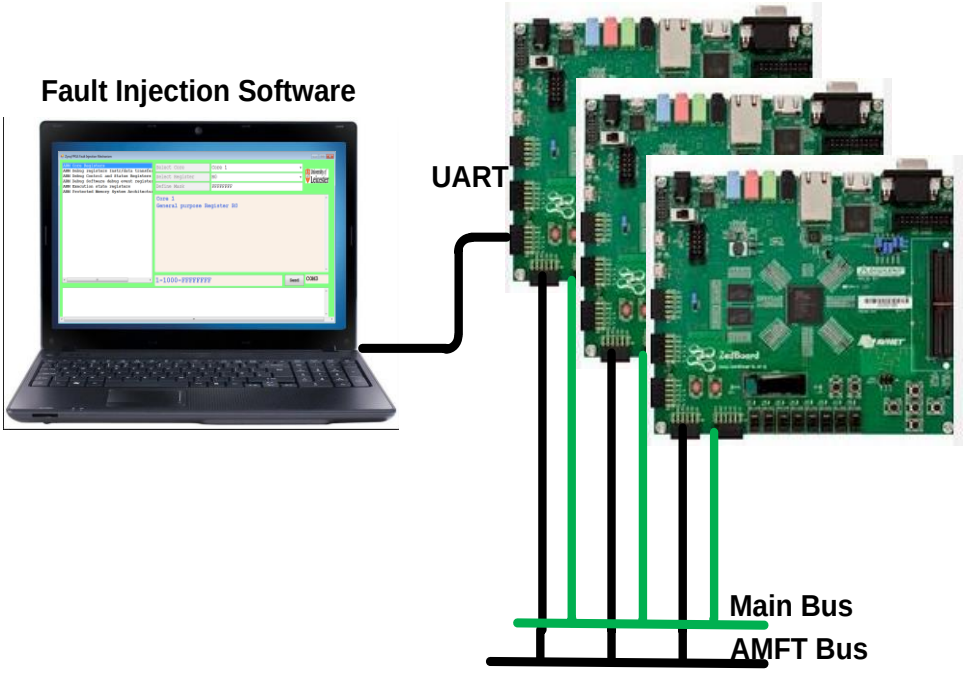


Figure 7.14: Experimental Setup.

Table 7.8: Mission Task Set

Mission Task (v)	*Task Period (T) (ms)*	*State Data Bytes* (t_{s_size})	*Initial State Values* (ts_0)

Task-1	50	100	0…0
Task-2	100	200	0…0
Task-3	150	300	0…0
Task-4	200	400	0…0
Task-5	300	500	0…0

Experimental Results: During the experimental testing and validation of the MPSoC design fault detection latency, reconfiguration time, $t_{Reconf.}$, and number of state rollbacks, $n_{rollback}$, were observed.

Fault Detection: The fault detection algorithms presented in section 4.8 were implemented and tested by injecting transient as well as permanent SDC faults. For that the fault injection mechanism, described in section 7.5 was used. In addition to the SDC fault detection, the AMFT FDIR task, which monitors physical signals; temperature, voltage, current and WDT for the failure detection of each distributed node, is also tested. The faults were injected into the various components of an MPSoC node, and the detection latency was measured as shown in Figure 7.15. The lowest detection latency value was observed in the case of CPU registers when a transient fault was injected into the CPU registers. This is due to the transient fault detection algorithm, proposed in section 4.8.1, which immediately checks processed variables for errors. For all other faults, the detection mechanism was implemented as a separate task. The detection latency of all other faults depends on the period of their execution. The execution period of the fault detection task t_{FD_Period} for various components such as peripheral registers, data contents of memory, temperature/voltage/current signals, watchdog timer, CPU microarchitectural elements, was carefully selected, considering the severity nature of each fault.

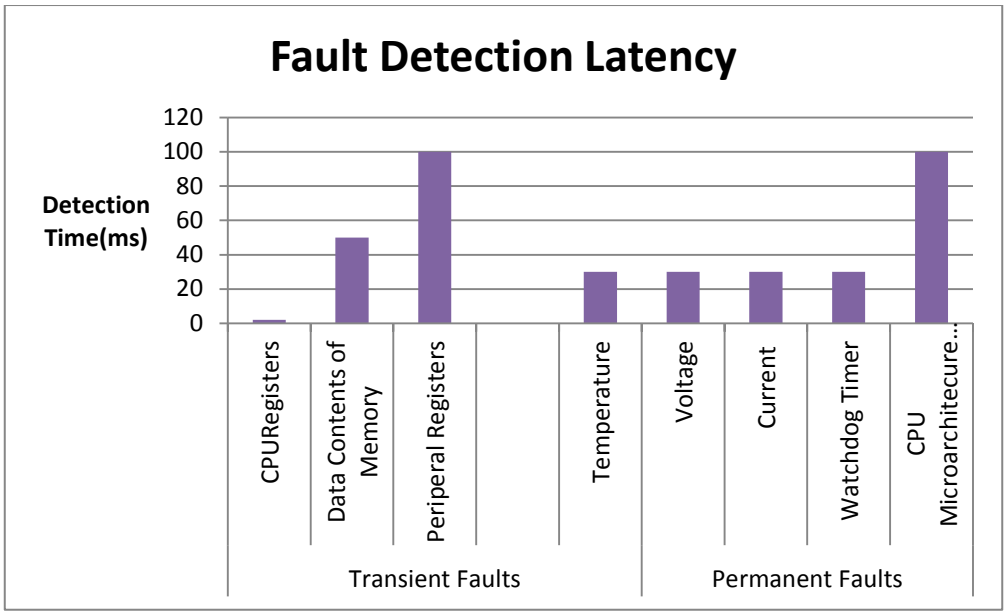


Figure 7.15: Fault Detection Latency.

The fault detection latency results showed that maximum detection latency was observed for the peripheral registers and CPU microarchitecture elements faults that should be less than or equal to 100 ms. The fault detection latency values are highly important as they affect the distributed system reconfiguration time, when tasks are migrated to other nodes.

***Reconfiguration Time*:** The reconfiguration time $t_{Reconfig}$, defined in section 6.2.1.1 is the time required to configure the distributed system. It includes detection time and migration time. Figure 7.16 shows the measured reconfiguration time for the defined configuration system setup and simulated task set. It is evident from the results that the reconfiguration time is variable and does not depend on the state data size. However, the state data size indirectly affects the communications slot time, which increases the fault message transmission time. For the particular settings of 300 ms communication cycle and 100 ms slot time, the fault message time can vary from 50 to 300 ms. This is due to the slot-based communication on the AMFT network, where fault messages can only be sent during the allocated slot.

The Permanent Fault detection time is the second highest value that directly contributes to the reconfiguration time. The value of the permanent fault detection t_D depends on the execution period of the detection task, t_{FD_period} , and the fault

detection processing time, $t_{FD\_Processing}$. For different faults, the execution period t_{FD_period} was different, therefore it can vary from 20 to 105 ms.

The third time value that contributes a small increase to the reconfiguration time is the 'Scheduling and Task Start time t_{TM}. When a task is migrated to another node following a failure, it needs to be scheduled and started. The value of t_{TM} depends on the processor speed, current workload, and the scheduling scheme used. Figure 7.15 shows that this time is also variable and can vary from 2 to 4 ms.

The upper bound value of the reconfiguration time $t_{Reconfig}$ is fixed and should always be less than 400 ms for this particular system configuration. This value of the reconfiguration time $t_{Reconfig}$ is very less as compared to traditional standby redundant computing systems used in space applications. In traditional systems, the switching time from a failed computer to the redundant computer is around 3000 ms [226].

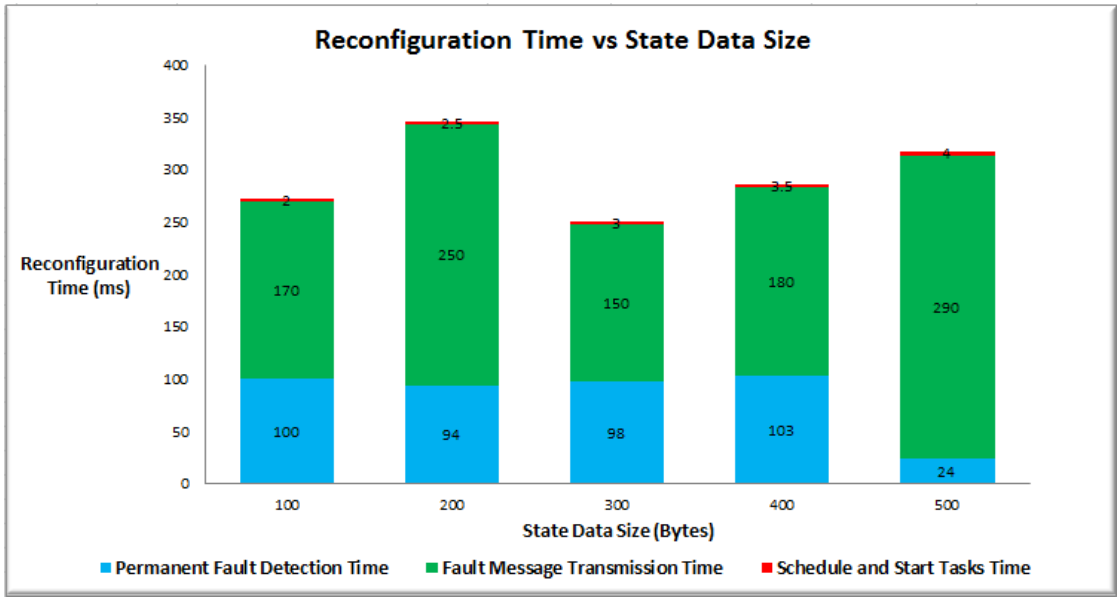


Figure 7.16: Reconfiguration Time.

State Data: The state data Δ_{SD} represents the task's input values, which are required in order for the task to generate output values, whenever it is executed. Larger state data size, ts_{size}, affects the transmission time, t_{TX} , and correspondingly the communication time slot, t_{cs} , of the TDMA cycle. Figure 7.17 shows the measured values for the transmission time t_{TX} and correspondingly the minimum slot time tcs. This shows a

linear relationship between the state data size ts_{size} and the transmission time t_{TX}. We also observed that a node which shares computing load of a failed node will require more transmission time t_{TX} on the communication network due to more number of states. Therefore, a suitable value for the slot time t_{cs} is selected to cover all possible failure scenarios. Equation 7.1 is used for the theoretical calculation of transmission time t_{TX} on the AMFT CAN network.

$$t_{TX} = \left(\frac{1}{R}\right) * TB * OB + IFS * NF \tag{7.1}$$

where

$R = CAN\ Speed\ in\ Bits/s$
$TB = Transmission\ Bits$
$OB = Overhead\ Bits$
$IFS = Interframe\ Spacing$
$NF = Number\ of\ frames/message$

State Rollback: During the normal operation of the proposed FTDC system, task's states are stored on each node. It is essential to resumes a task when it is started on another node following a failure. This ensures the integrity of a fault-tolerant computing system. During the experimental evaluation, state rollbacks $n_{rollback}$ for the different tasks were observed as shown in Figure 7.18. These observations were made for the earlier defined mission task set. We observed that state rollbacks $n_{rollback}$ are varied with the task period *T*. For large task period, *T*=300 ms, the number of state rollbacks was small and equal to one state rollback only. However, for a small task period *T*=50 ms, the number of state rollbacks was large and equal to seven rollbacks.

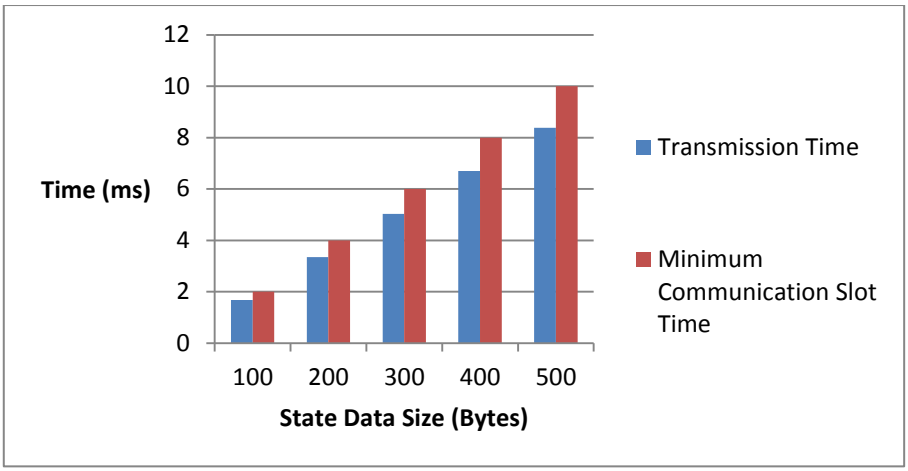


Figure 7.17: State Data Size, Transmission Time, and Communication Time Slot.

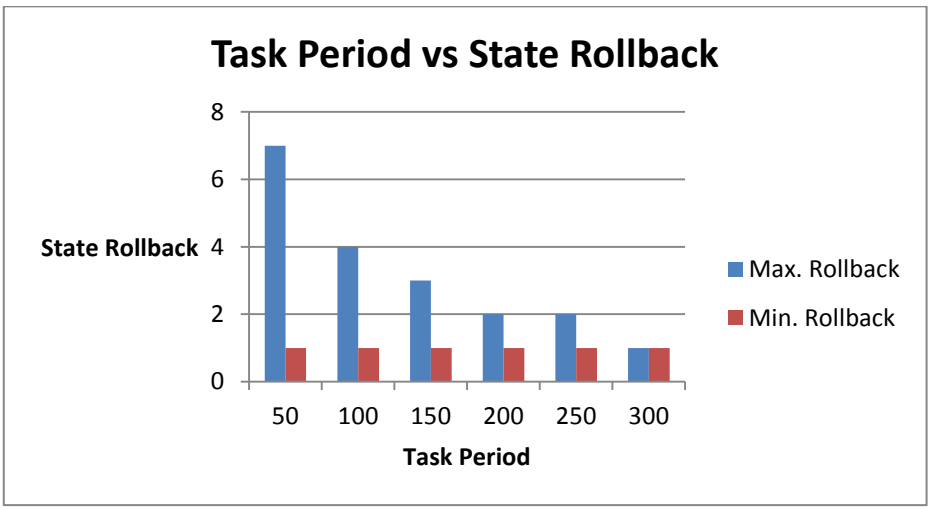


Figure 7.18: Number of State Rollbacks and Task Period.

7.7 Multiprocessor System-on-chip for a CubeSat Mission

CubeSat is a miniature satellite. It has a mass equal to 1 kilogram and volume equal to 1000 cm^3. It is mainly used for technology demonstration and educational purposes. Although its size is small, it requires all the computing functions of a normal satellite. These functions include satellite control, data handling, power/thermal management, and the ground communications. Each of these is usually implemented as standalone printed circuit boards. This makes the CubeSat overcrowded and very small physical

space is left for the actual payload equipment. Also, it is not possible to make CubeSat design fault-tolerant by providing extra redundant computing units.

To avoid these problems, the proposed approach to fault-tolerant distributed computing could be applied to CubeSat. In this section a novel multiprocessor system-on-chip CubeSat (MPSoC-CubeSat) design for fault-tolerant distributed computing is proposed. This design implements all the functions on a single chip, reducing its size significantly in comparison to the traditional board-level design. In addition, it allows task migration to make the system fault-tolerant. The design of the MPSoC-CubeSat is shown in Figure 7.19.

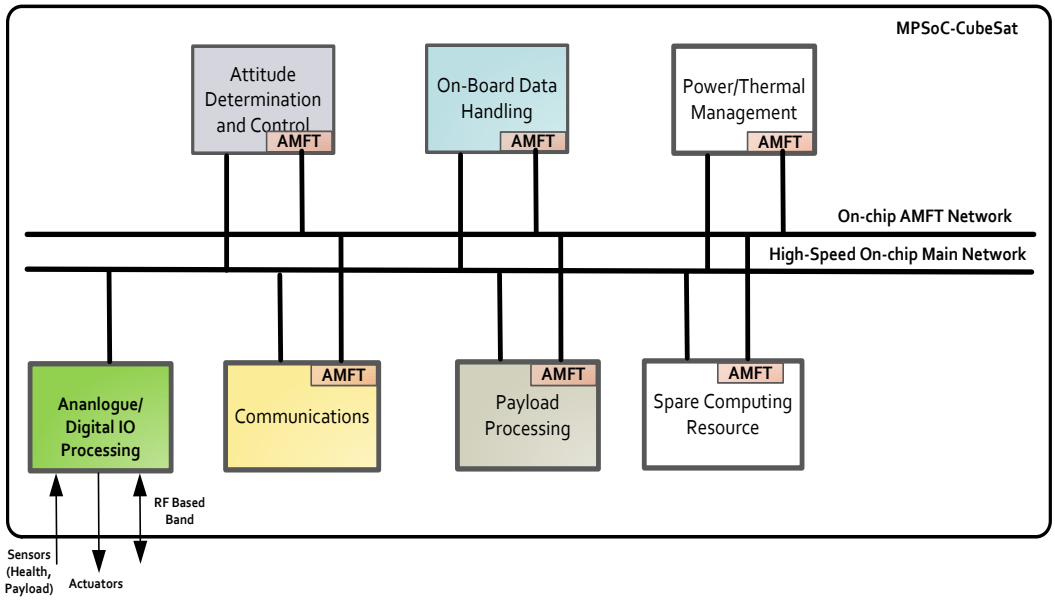


Figure 7.19: Design of Multiprocessor System-on-chip CubeSat (MPSoC-CubeSat).

It comprises multiple computing modules and communication networks, housed on a single chip. These modules are similar to the nodes in the proposed FTDC approach. One of the on-chip networks is used for the communications of the computing modules while the other is used for the communications of the AMFT blocks. All the IOs are attached to a single input-output module for digital and analogue data. In the MPSoC-CubeSat, the functionality of the AMFT block is similar, except the decision for the chip power-off, which is handled by the OBDH only.

As the whole satellite computing functions are integrated on a single chip, it reduces the physical size and electrical power consumption. Unlikes, traditional CubeSat design, MPSoC-CubeSat design allows rapid integration and modification adapting to the needs of a particular mission.

7.7.1 Implementation of MPSoC-CubeSat PCB Design

In lieu of a prospective Leicester CubeSat mission, a Printed Circuit Board (PCB) complying with its specifications was designed. The PCB design was developed in Altium Designer v10.2 [227] and main features of the PCB design are stated as follows:

- A 12 Layer PCB which complies with CubeSat specifications.
- The design adapts our proposed MPSoC scheme, i.e. one MPSoC-CubeSat on a single PCB.
- The majority of components and PCB layout are space qualified.
- It is a standalone unit, with own power regulators, SDRAM (16x128)x2=512MB, Flash Memory 512Mb, CAN 2561 interfaces, USB 2.0 to UART bridge.
- The main clock is 33.333 MHz. The internal PLL of FPGA takes the main clock and converts it to 666.67 MHz. The AMFT processing unit takes a separate clock of 100 MHz clock.
- MPSoC has dedicated ADC interface using analogue inputs and sensor's data processing.
- The interface of 20 I/O available for additional daughter cards, e.g. for SpaceWire and WiFi modules.
- Debugging and Programming interface via JTAG.

The PCB Layout and its Bill of Materials (BoM) is given in Appendix D.

7.8 Summary

A novel MPSoC based approach to fault-tolerant distributed computing system was proposed for space applications. The proposed MPSoC design was implemented and

tested for functional validation, for which appropriate experimental setup was developed. This experimental setup allowed observations of the scheme behaviour at run-time. For instance, the tasks are being added, dropped or reconfigured and faults injected at software and hardware level.

A key observation was the behaviour of the proposed system under a failure scenario of the computing nodes. In the traditional redundant system, a failure of a computing node can cause degradation or be catastrophic for the whole distributed system. This failure behaviour depends on the criticality of the functions it holds. The MPSoC design reduces this damage to a minimum level by utilizing the power of multiple processors. Thus, making a distributed computing system computationally efficient and more reliable without the need for a spare computing node for fault-tolerance. This reduces the overall design cost and makes the overall system economical. In addition to tolerating faults, an essential aspect of the proposed architecture is its ability to resume operations on a healthy node with a recent state data values.

The traditional space computing redundant system requires time-consuming switching operation from primary to a redundant node in case of a failure. It can take several minutes if ground commanding is used. The obtained experimental results on the reconfiguration time are promising. The small, deterministic value of the reconfiguration time showed that a failure could be immediately masked by migrating tasks to another healthy node. It reduces the system downtime, thus improving the overall availability of the system, an essential feature of space computing systems.

Chapter 8

Case Study: Fault-Tolerant Distributed AOCS Computer

In this chapter, performance analysis of the proposed approach to fault-tolerant distributed computing (FTDC) is carried out under near realistic constraints and requirements using a time critical space application, namely satellite Attitude and Orbit Control System (AOCS). First, the developed AOCS application is specified and its task set is defined and adapted to measurable parameters. Then the requisite AOCS task set is mapped on the distributed computing nodes and the AMFT system is configured. The AOCS is implemented using three MPSoC based nodes using the MPSoC design in Chapter 7, which are configured to operate as a distributed computing system. Experimental results are obtained, which are analysed and evaluated by a MATLAB AOCS model. The SDC fault detection algorithms, proposed in section 4.8 are also analysed and compared for performance and efficiency.

8.1 Attitude and Orbit Control System

AOCS controls the attitude and orbit of a spacecraft. Attitude control refers to the system which checks and corrects the orientation of a spacecraft with reference to an inertial frame of reference, whereas orbit control refers to a system that checks and sets the desired spacecraft position in orbit.

The AOCS computer receives data from sensors and executes complex algorithms to determine the desired orbit and attitude. It generates appropriate commands for the actuators to compensate for the errors in the desired and actual orientation and position, as shown in Figure 8.1. This set of operations is carried out in a control loop during the entire life of the spacecraft. Also, the AOCS computer has an external interface to accept commands from On-Board Data Handling (OBDH) Computer as well as from the Ground Station. AOCS also generates its state (current mode of operation) in the form of telemetry data, which provides information about the spacecraft orientation, as well as position in orbit.

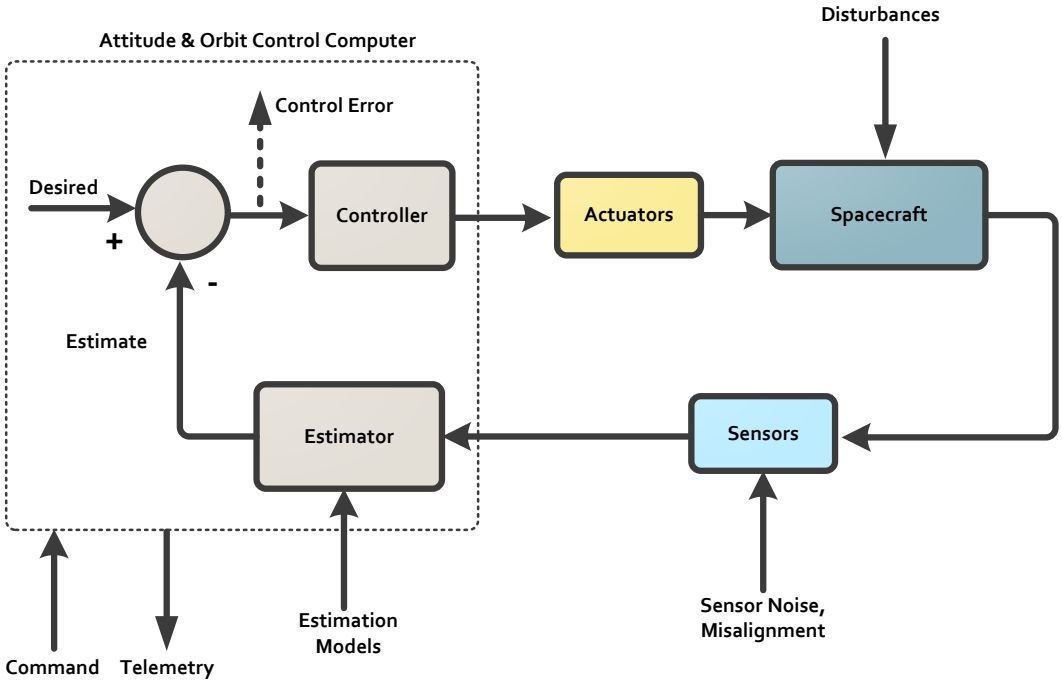


Figure 8.1: Block Diagram for Attitude and Orbit Control System.

8.2 Rationale for Distributed AOCS

The main requirements which demand a distributed AOCS are as follows:

- In a single spacecraft mission, advanced attitude and orbit control for future spacecraft are needed to provide improved pointing accuracy, reliability, faster response, and on-board autonomy. These demands cannot be met with a centralized design.

- In a multi-spacecraft mission, to fly spacecraft in a formation, a distributed spacecraft control is essential. A Spacecraft Formation Flying (SFF) mission consists of a set of satellites, flying in a close configuration. Their geometry is accurately measured and controlled, requiring the distributed spacecraft to exert collaborative control of their relative positions and orientations. Contrary to a centralized control, distributed maintenance of the spacecraft formation is more reliable because it reduces the delay in performing the control of formation geometry and eliminates the chances of a single point of failure. Despite being very challenging, formation flying missions are the only solution to achieving a high-quality resolution synthetic aperture that is otherwise impossible to achieve. Table 8.1 lists a few Spacecraft Formation Flying missions.

Table 8.1: Spacecraft Formation Flying Missions.

Mission	***Launch Date***	***Formation Type***	***# of Spacecraft***	***Application***
Prisma [228]	June 2010	Trailing formation with 10 cm distance to each other.	2	Autonomous Formation Flying Demonstration
TanDEM-X [229]	June 2010	Trailing formation with 250/500 m distance to each other.	2	High-Resolution Interferometric SAR
Proba-3 [230]	Scheduled to be launched in 2018	Trailing formation with 25 to 250 m distance to each other.	2	Formation Flying Technology Demonstration

8.3 Design of a Distributed Attitude and Orbit Control

The design of a distributed AOCS starts from the requirement specifications that follow design processes, application structure, task set and its mapping to actual physical nodes. The requirement specifications include parameters, such as accuracy, operational modes, reliability, and computational performance. The functional design processes involves mapping of the requirement specifications into functional design units.

8.3.1 Requirement Specifications

- AOCS shall maintain fine pointing attitude accuracy during the payload operations, while during the rest of the time it shall maintain coarse pointing accuracy. Also, AOCS shall be able to correct the orbit, whenever required.
- AOCS should be computationally efficient and able to perform autonomous attitude and orbit adjustment without ground control.
- AOCS shall accept ground commands to perform attitude and orbit correction. It shall also provide attitude measurement via telemetry data. Furthermore, it shall provide information via telemetry to allow diagnostic on board the spacecraft.
- AOCS shall be able to reconfigure itself in the case of a node failure. This requires that the migration of tasks of a faulty node should be handled timely and reliably.

8.3.2 AOCS Sensors and Actuators

Based on the requirement specifications in section 8.3.1, three attitude sensors, one orbit sensor and two types of actuators were selected as detailed in Table 8.2. These are the minimum sensors and actuators set, i.e. essential to meet the AOCS requirement specifications.

Table 8.2: AOCS Sensors and Actuators.

Function	***Sensors/Actuators***
Attitude Measurement	Sun Sensor, Magnetometer and Rate Gyro
Orbit Measurement	GPS
Attitude Control	Magnetorquer
Orbit Control	Thrusters

8.3.3 Functional Design Processes

In our design process, AOCS application consists of four main processes, as shown in Figure 8.2. The *input process* is responsible for reading and formatting the sensor data in such a way that it is acceptable for the next process. The *determination process*

estimates the spacecraft orbit and attitude by using sensor data, an estimation algorithm, and available models. The output of this process is an estimate of the position in orbit, angles, and angular rates. The next process is the *controller process* that can adapt to multiple configurations depending upon the selected mode of operation. In our AOCS application design, in total six operational modes of the controller process are possible as detailed in Table 8.3. The essential modes are detumbling, payload, normal and orbit control. The detumbling mode is used for initial rate reduction after the spacecraft separation from the launcher while the payload mode is used for the fine pointing of the spacecraft. If the error is within the predefined limits, a normal execution mode is used. The final process is the *output process* that handles writing commands for actuators.

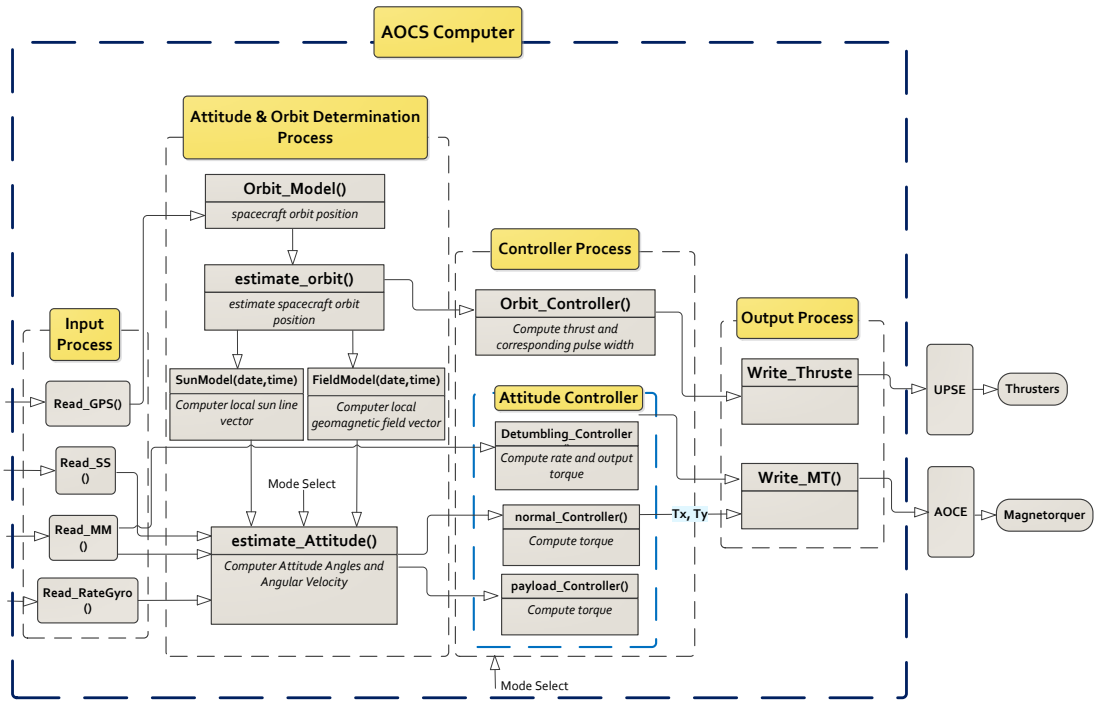


Figure 8.2: Design Processes for Attitude and Orbit Control Application.

Table 8.3: AOCS Modes of Operations.

Mode of Operation	***Mode Description***
Standby mode	Only Telecommand and Telemetry tasks are active. There is no attitude and orbit control, so all the AOCS units are powered off except the computer itself. This mode is typically used, when the satellite is still inside the launcher or during the first and second stage after launch.

Detumbling mode	This mode is typically activated after separation from the launcher and used for angular velocity rate reduction. The b-dot controller is used for this mode.
Payload Mode	This mode is used for fine attitude adjustment for the payload operation.
Safe Mode	Entered, when there is no reconfiguration possible.
Normal Mode	In normal mode, if the error is within predefined limits, no actuation is provided.
Orbit Control Mode	This mode is used when orbit change is required.

8.3.4 Distributed AOCS Software Structure

The distributed AOCS software structure mainly consists of three parts: application software, support software and fault management software as shown in Figure 8.3. The support and fault management software for the distributed AOCS computer are implemented as presented in section 4.6.1 and section 5.5, respectively. The design and implementation of the AOCS application, represented as a suitable task set, which meets the requirements specified in section 8.3.1 and their corresponding mapping to distributed computing nodes, are explained in the following section.

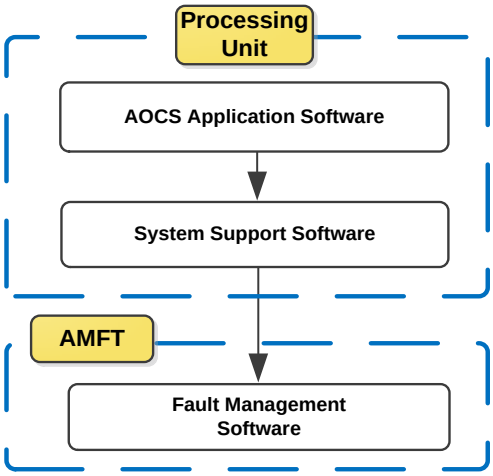


Figure 8.3: Distributed AOCS Software Structure.

8.3.4.1 AOCS Task Set

The AOCS computer is responsible for execution of complex algorithms. The algorithm can be divided into a set of the task, from a computation point of view. These tasks are computationally intensive, requiring high reliability, high availability,

and real-time operation throughout their execution. An unfinished task leads to a failure of AOCS, which jeopardises the space mission, and may cause an in-orbit collision with other spacecraft. The importance and criticality of the AOCS computer in spacecraft is the main reason for choosing it for the validation of the proposed approach. Based on the requirement specifications and chosen sensors/actuators, the AOCS requisite tasks are derived and measurable parameters are assessed as summarised in Table 8.4.

Table 8.4: Characteristics of Distributed AOCS Task Set.

AOCS Task	***Typical Task Period, T (ms)***	***State Data Size ts_{size} (bytes)***	***Operational Modes***	***Task Number***
Attitude Measurement and Anomaly Check	50-300 ms	6 bytes/sensor	Detumbling (Only B Field) Normal and Payload Mode	Task#1
Attitude Determination	50-300 ms	Attitude angles and angular rates (24 bytes)	Normal Mode and Payload Mode.	Task#2
Attitude Control	50-300 ms			Task#3
- B-dot		Control Torque, Tx, Ty, Tz (12 bytes	- Detumbling	
- PD		Control Torque, Tx, Ty, Tz (12 bytes)	- Normal Mode and Payload Mode.	
Orbit Estimation & Control	10000 ms	24 bytes	Orbit Control Mode	Task#4
Telemetry	1000 ms	60 bytes	All	Task#5
Telecommand	Sporadic	8 bytes	All	Task#6

The common properties of the tasks are:

- ***Periodic:*** all tasks are executed periodically with time periods *T* , which are stated in Table 8.4. The Telecommand task is an exception.
- ***Deadline:*** each task has to be executed within a specified time, considered its deadline, which is determined from the period *T*.
- ***Critical:*** no single task can be left unexecuted.

- ***State Data:*** State data Δ_{SD} represents the result on completion of a task, which has a certain data length, as shown in Table 8.4 and as explained in section 5.3.3.
- ***Inter-dependant:*** the outcome of one task can be an input to another task.
- ***Modes:*** different modes demand separate tasks and, therefore, have to be addressed separately.

The above parameters allow us to quantify measures to determine the performance of the proposed approach. The rest of this section presents a functional description of the AOCS application tasks.

Attitude Measurement and Anomaly Check Task: Firstly, this task is responsible for taking attitude sensor data measurements and passing the values to the *attitude determination task*. Sun and magnetometer sensors are used for the attitude measurements. A sun sensor measures the components of the sun vector 's' in body frame 's_b', while a magnetometer measures the components of the field vector 'm' in the body frame, m_b. Secondly, this task detects and isolates the anomalies in the magnetometer and sun sensors. It can detect an anomaly by a single and a multi-sensor consistency check.

Pseudo Code:

```
1   vAttitudeMeasurement_ AnomalyCheckTask {
2       Initialize variables
3       while(1){
4           s_b = read_ss()
5           m_b = read_mm()
6           rate_b = read_gyrorate()
7           ss_ok = anomaly_CheckSS(s_b)
8           mm_ok = anomaly_CheckMM(m_b)
9           rate_ok = anomaly_CheckGR(rate_b)
10          Switch(mode){
11          case DETUMBLING:
```

```
12                          send_{main_network_Msg}(m_b)
13                          send_{amft_Msg}(m_b, mode); break;
14                  case NORMAL&PAYLOAD:
15                          send_{main_{network_{Msg}}}(s_b, m_b, rate_b)
16                          send_{amft_Msg}(s_b, m_b, rate_b, mode); break;
17                  default:
18                  }
19          }
20  }
```

Attitude Determination Task: The purpose of this task is to obtain the spacecraft attitude. The task uses measurements of sensors and mathematical models [231] to collect components of vectors in the body and inertial reference frames. The components of these vectors are utilized by the attitude algorithm to obtain the spacecraft attitude in the form of Euler's angles or quaternions. Attitude determination algorithms are broadly classified into non-recursive and recursive methods. Non-recursive algorithms do not require a priori estimate and determine the attitude based on the current measurements only. These algorithms require minimum two measurement vectors to determine the complete attitude. Examples of non-recursive algorithms are Triad, Quest, and Davenport's q-method. Recursive algorithms utilize past information and current measurements to obtain attitude. The most commonly used recursive algorithm is Kalman filtering. In our implementation of distributed AOCS, a triad algorithm was selected for the attitude determination task. This task is periodic and its period of operation depends on the period of the control cycle. The typical value of the task's period is given in Table 8.4. The specification of the attitude task is as follows:

Pseudo Code:

```
1   vAttitudeDeterminationTask {
2       Initialize variables
3       while(1){
            [time, orbit_position] = GPS()
4           s_i = sun_Model(time, position)
5           m_i = IGRF_Model(time, position)
6           Switch(mode){
7           case PAYLOAD:
8               R^bi = triad(s_b, s_i, m_b, m_i)
9               send_{main_network_Msg}(R^bi)
10              send_{amft_Msg}(R_bi); break;
11          case NORMAL:
12              R^bi = triad(s_b, s_i, m_b, m_i)
13              send_{main_network_Msg}(R^bi)
14              send_{amft_Msg}(R^bi); break;
15          default:
16          }
17      }
18  }
```

```
1   void triad(s_b, s_i, m_b, m_i){
2       t_1b = s_b ;
3       t_2b = (s_b x m_b) / |s_b x m_b| ;
4       t_3b = t_1b x t_2b;
```

```
5        t_1i = s_i ;
6        t_2i = (s_i x m_i) / |s_i x m_i| ;
7        t_3i = t_1i x t_2i;
8        R^bt = [t_1b, t_2b, t_3b];
9        R^it = [t_1i, t_2i, t_3i];
10       R^bi = R^bt x R^it = [t_1b, t_2b, t_3b] [t_1i, t_2i, t_3i]^T;
11   }
```

Attitude Control Task: The *attitude control task* is responsible to calculating the applied torque force to spacecraft in order to correct its orientation. The input to this task can come from the *attitude determination task* or directly from sensors depending upon the mode of operation. This task is operational in different modes. A wide variety of controllers have been used to control the spacecraft attitude. These include B-dot [232], Constant Gain [233], Proportional Integral Derivative (PID) [234], Linear Quadratic Regulator [235], and non-linear H∞ controller. For our design of distributed AOCS, we use a B-dot controller and a PD controller. The B-dot controller is employed for the detumbling mode, while the PD controller is used for the payload pointing of the spacecraft. Both controllers are implemented in the *attitude control task*, one of these is activated based on the AOCS mode of operation. The *attitude control task* is periodic and its typical period *T* of execution is given in Table 8.4. The following shows the specification of the task:

Pseudo Code:

```
1   vAttitudeControlTask {
2       Initialize variables
3       while(1){
4           Switch(mode){
5               case DETUMBLING:
6                   B^b_{t-1} = m_b;   % Magnetic field at t − 1
7                   delay (Δt); % delay of delta t
```

```
8                   B_t^b = m_b ;  % Magnetic field at t
9                   B^dot = [B_t^b − B_{t−1}^b] / Δt ; % Rate of change in B
10                  u = [M_x, M_y, M_z]^T
                        = − [k_1 B_xdot , k_2 B_ydot , k_3 B_zdot ]^T ; % torque, u
11                  send_{main_network_Msg}(u) % Send msg for u on main network
12                  send_{amft_Msg}(u); break; % Send msg for u on AMFT network
13              case NORMAL&PAYLOAD:
14                  % cal. difference in angles
15                  α_e = [φ − φ^d ; θ − θ^d ; Ψ − Ψ^d ]^T
16                  % cal. difference in angles
17                  ω_e = [ω_x − ω_x^d ; ω_y − ω_y^d ; ω_z − ω_z^d ]^T
18                  % K_p and K_d are gain matrices for PD Controller
19                  u = −K_p * α_e * K_d ω_e  % PD Controller torque vector u
20                  send_{main_network_Msg}(u) % Send msg for u on main network
21                  send_{amft_Msg}(u); break; % Send msg for u on AMFT network
22                  default:
23          }
24      }
```

Telemetry Task: The *telemetry task* gathers health and AOCS parameters. This task is also of a periodic nature and its telemetry parameters depend on the algorithms. The details of the telemetry parameters are given in Appendix E. On execution, it acquires the telemetry data and stores them into the $TMList$ data structure, which is transmitted to the main and the AMFT networks.

Pseudo Code:

```
1  vTelemetryTask {
2      Initialize variables
3      while(1){
4          TMList = acquireTM(); % TelemetryData Acquisition
5          send_{main_{network_{Msg}}}(TMList) % Send msg for TMList on
           main network
6          send_{amft_{Msg}}(TMList); % Send msg for TMList on AMFT network}
7      }
8  }
```

Telecommand Task: The *telecommand task* is a sporadic task. Each command can be issued from the OBDH computer or directly from ground via a Telecommand decoding system. The pseudo code for the *telecommand task* is given below. As soon as the command arrives, it is received by all distributed computing nodes via the main network. The node responsible for the execution of the *telecommand task* executes it and sends a successful execution message on the AMFT network. All the other nodes, remove the stored command. If a command does not execute within the allocated time $(t + dt)$, the next node responsible for the command executes it.

Pseudo Code:

```
1  vTelecommandTask {
2          Initialize variables
3          while(1){
4              if( TC_{EVENT} == true){ % Telecomamnd Received Event
5              execute(TC);; % Execute TC
6              send_{amft_{Msg}}(Successful);; % TC Successful, Invalidate TC
7          }
8  }
```

Orbital Estimation & Control Task: Satellite orbit drift due to atmospheric drag and gravitational pulls. Orbits can be determined and corrected by ground-based tracking systems [236, 237], or this can be done on board spacecraft [238]. The former method has a disadvantage over the later one, because it requires a ground intervention to control the orbit of a satellite. However, the later method requires complex orbit determination and control algorithms to be processed on board the spacecraft that are difficult to run on centralized low-performance computers. Contrary to a centralized, a distributed AOCS computer can run these complex algorithms easily because of the inherent computational power of multiple nodes. In the design of the distributed AOCS computer, the *estimation and control task* is responsible for correcting the spacecraft orbit. First, this task determines the spacecraft orbit by extended Kalman filtering (EKF) and then forces are applied by delta-V actuators. The period of the orbit estimation and control task is very large and the exact value depends on the altitude and mission. A typical value of the period *T* for this task is given in Table 8.4.

Pseudo Code:

```
1   vOrbitEstimation&ControlTask {
2       Initialize variables
3       while(1){
4           % Orbit Estimation
5           rawGpsdata = GpsMeasurement();
6           prefilteringAndSampling(rawGpsdata, IGS);
7           statesEstimate = orbitFiltering();
8           orbitPridicted = orbitPrediction();
9           % Orbit Control
10          u = −kx
11  }
```

Now that the AOCS Task Set has been described, their mapping to the distributed computing nodes is described next, in which three nodes are employed for evaluation purposes.

8.3.4.2 Tasks-to-Nodes Mapping

In order to distribute the AOCS task set across the FTDC system, they have to be mapped correctly to the individual computing nodes. The reason being, that one task outcome may be used as an input to another task and, therefore, task execution requires this inter-dependency information to be taken care of unmistakeably.

In our case, there are six tasks listed in Table 8.4, which comprise the AOCS Task Set and 3 distributed computing nodes are used in the implemented FTDC system. Figure 8.4 shows how the Task Set can be mapped to these nodes. The mapping decisions are described below:

- Task # 5 and Task #6 being critical and are mapped to all nodes.
- Task #1 is mapped to node 1. Task #2 cannot execute till Task #1 has completed.
- Task #2 and Task #3 are mapped to node 2. Task #3 cannot execute till Task #2 has completed.
- Task#4 is mapped on node 3.

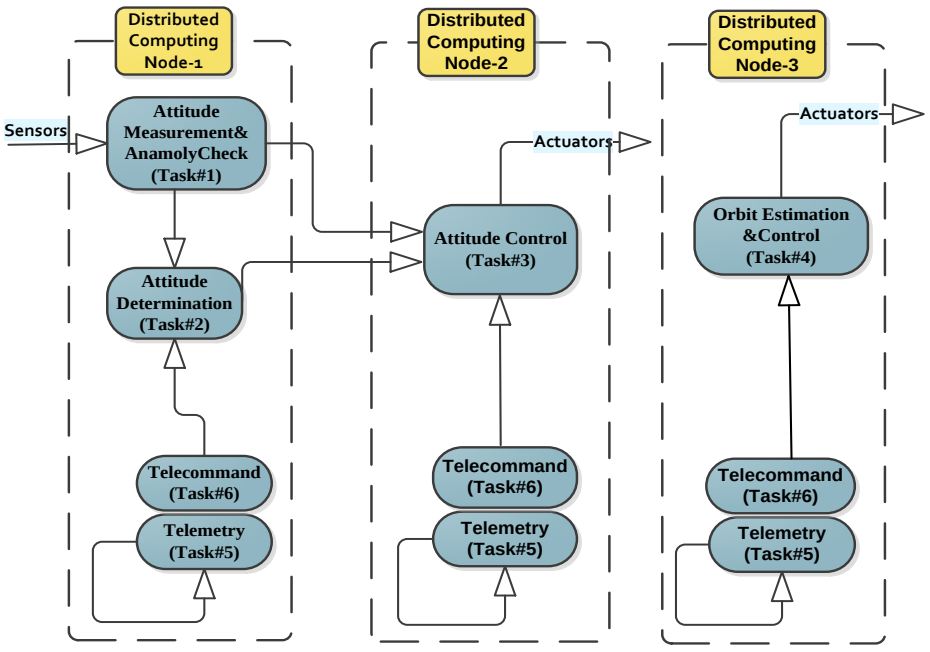


Figure 8.4: Mapping of AOCS Tasks.

8.4 Distributed AOCS Computer Implementation and Testing

The fault-tolerant AOCS computer consists of three distributed nodes, which are based on the MPSoC design presented in 7.3. Each processing unit communicates with its peers via a separate network, from the network used by AMFT. The targeted network high data rate time-triggered network (TTEthernet) was not available and therefore, the lower speed CAN network was used. Since CAN is not time triggered, and to make its behaviour deterministic, the TDMA scheme was used. A time slot is allocated to each node, and only that node can communicate during that time slot, i.e. transmit its information. The time slots are repeated after a set time, known as the TDMA cycle. In the TDMA cycle, the slot time is based on the size of the data in bits, and the data rate of the communication network, as well as the minimum time, a node can wait before its information becomes invalid. The details of the fault-tolerant distributed AOCS system parameters are given in Table 8.5.

Table 8.5: Distributed AOCS System Parameters.

Parameter	**Distributed AOCS Computer**
Number of Nodes	3
Task Set	AOCS
AMFT Network Slot Time (ms)	30, 300
AMFT Network Speed (Mbps)	CAN @ 1Mbps
Main Network Slot Time (ms)	100
Main Network Speed (Mbps)	CAN @ 1Mbps
Development Board	ZedBoard

8.4.1 System Configuration

The following sections discuss the slot allocation separately for the processing unit and the AMFT block.

TDMA Slot Time: AMFT has to communicate its information to its peers, and the shared information is the task outcome in the form of the state data Δ_{SD}. For instance, from Table 8.4, the largest state data size ts_{size} is that of Task #5, which is 60 bytes.

That state data has to be passed to its peer node within the shortest time possible, which is determined based on the communication speed as follows. The controller area network (CAN) used for the AMFT communication is limited to 1 Mbps. According to Equation 7.1 to transmit state data size ts_{size} of 60 bytes on CAN, it requires a transmission time t_{TX} of 1002 µsec. Based on that, a slot duration t_{cs} of approximately 2 ms is sufficient to transmit the state data ts_{size} of 60 bytes, ensuring that the data is reliably delivered before the start of the next slot.

The second communication parameter is the slot repetition time, which is derived from the tasks period *T*. For example, from Table 8.4, the minimum task period corresponds to 50 ms, which means that the data must be delivered within 50 ms time interval before the next execution of the same task. A repetition period, equal to the task period *T*, helps to maintain the updated data state of a particular task during the task migration process.

AMFT Configuration: As the AMFT is deployed in a distributed computing scenario, additional information has to be incorporated in the AMFT. This information is in the form of configuration tables, i.e. Node and Task Migration Tables, which are stored in the AMFT as described in Section 5.3.4.

- ***Node Table:*** This table holds information about the active and non-active nodes in the system. In the initial state, it is assumed that all nodes in the system are active. and the number of the nodes in the distributed system are known. As soon as the connection is established, the node table is updated, based on the current status of each node. The table is also updated in case of a node failure, as necessary.
- ***Task Migration Table:*** This table holds critical information about the initial Task Mapping, as well as possible migration scenarios of tasks in case of faulty nodes. Therefore, each AMFT knows how to migrate tasks when a certain node fails.

In this case study, all possible scenarios of node failures are defined statically (off-line). Similarly the above two tables are both set off-line, supporting all scenarios, which are needed to assess the performance of the proposed architecture.

8.4.2 Experimental Results

During the testing of the distributed AOCS computer, the following parameters were measured to evaluate the performance: (i) reconfiguration time, (ii) state rollback and (iii) computational performance. The rest of this section describes the obtained results for each of these measurements. For the purpose of testing, faults were injected in the distributed AOCS computer using the fault injection mechanism presented in section 7.5. Two main scenarios were used, which were based on a TDMA cycle time of 30 ms and 300 ms, respectively

8.4.2.1 Reconfiguration Time

The reconfiguration time is measured for a different number of nodes failures. As shown in Table 8.6 the reconfiguration time, t_{Reconf}, is largely dependent upon the TDMA cycle time and its value is always less than or equal to the TDMA cycle time plus the fault detection time t_D.

Table 8.6: Reconfiguration Time Measurements

Failure Scenario	**AMFT Slot Time, t_{cs} (ms)**	**AMFT Network TDMA Cycle Time (ms)**	**Fault Detection Time, t_D (ms)**	**Fault Message Transmission Time, t_{FM} (ms)**	**Transmission Time, t_{TX} (ms)**	**Migration Time, t_{TM} (ms)**	**Reconfiguration Time (ms), $t_{Reconf} = t_D + t_{FM} + t_{TX} + t_{TM}$**	
							Measured Value (ms)	*Expected Value (ms)*
One node Fails	10	30	101	8 to 26	1	~1	111 to 129	≤ 140
Two nodes Fail			102	10 to 28	1	~1	114 to 132	
One node Fails	100	300	110	98 to 290	1	~1	210 to 402	≤ 430
Two nodes Fail			120	100 to 295	1	~1	222 to 417	

8.4.2.2 Task State Rollback

During normal operation, each task state of the AOCS is checkpointed. The task state checkpointing data is provided to each migrated task, when it restarts its execution on the target node. The rollback of the task state depends on the reconfiguration time

t_{Reconf} and task period T. The measured value of the rollback of each task state for the AOCS is stated in Table 8.7. It is evident from the results that the number of the task state rollbacks, $n_{rollback}$, is directly related to the TDMA cycle and the reconfiguration time. For a large value of TDMA cycle, the rollback increases linearly considering a constant task period *T*.

As observed above, the state data, Δ_{SD}, which is stored on the node following a node failure may be a few execution cycles old in case of the 300 ms network cycle time, so the task state would be "rolled back" to the previous state. This rollback only momentarily affects the AOCS output, which is acceptable for this type of applications and will be further analysed in section 8.5.1. However, the number of rollbacks, $n_{rollback}$, can be reduced by employing a high-speed communication protocols for the AMFT network.

Table 8.7: Rollback of Task State.

Task #	**Task**	**Task Period, T (ms)**	**State Data, Δ_{SD}**	**State Data Size $ts_{size}(bytes)$**	**No. of state Rollback $n_{rollback}$ of task state**	**No. of state Rollback $n_{rollback}$ of task state**
					TDMA Cycle Time= 30 (ms)	*TDMA Cycle Time= 300 (ms)*
1	Attitude Measurement & Anomaly Check	100	Sun, Magnetometer and rate gyro	18	1~2	2~5
2	Attitude Determination	100	Attitude Angles and angular rate	24	1~2	2~5
3	Attitude Control	100	Torque vector	12	1~2	2~5
4	Orbit estimation and Control	10000	Position, Velocity values and Torque vector	36	0~1	0~1
5	Telemetry	1000	AOCS Parameters	60	0~1	0~1
6	Telecommand	Aperiodic	Command	8	N/A	N/A

8.4.2.3 Computational Performance

To assess the computational performance of the proposed fault-tolerant distributed AOCS computer, it was compared with a centralized AOCS computer and an AOCS

computer with active replication in terms of Dhrystone Millions Instruction per Seconds (DMIPS). Both the FT distributed and the replication AOCS computers are comprised of three computing nodes. In all three cases, the computing nodes of the AOCS computers were implemented using the MPSoC design, presented in Chapter 7. For each AOCS computer, the total available DMIPS computing performance was obtained from Equation 8.1, where $DMIPs_{node}$ is a Dhrystone MIPs per MPSoC node and n is the number of nodes per computer. Dhrystone MIPs per second of 2380.95 was obtained by running the Dhrystone code on each individual computing node (implemented in the Xilinx Zynq FPGA on the ZedBoard).

The computational performance results for the three computer configurations are shown in Figure 8.5 in terms of (i) total available DMIPS, (ii) utilized DMIPS, (iii) overhead DMIPS due to fault-tolerance management and (iv) remaining available DMIPS (calculated by subtracting the total available DMIPS from utilized DMIPS). As it can be seen from Figure 8.5, compared to the centralized, the active replication AOCS and the distributed AOCS computers have higher total available DMIPS because they both are comprised of three computing nodes. The comparison of the three computers in terms of the utilized DMIPS shows that the computational demand of the distributed DMIPS is almost equal to the centralised computer DMIPS, while the active replication AOCS computer is less computationally efficient, which is due to the replication of the tasks on each node, causing a significant portion of the DMIPS to be spent on replicas execution and maintenance. The slightly higher utilised DMIPS in the proposed distributed AOCS computer, compared to the centralised case, is due to its fault management and task migration support. The active replication computer requires much higher overhead DMIPS too, as all the three computing nodes execute the same tasks and further computing resources are required for voting and consensus among the replicated tasks. Compared to the replication computer, the distributed computer requires considerably lower overhead DMIPS. The centralized computer has the lowest overhead DMIPS, which is not surprising, as it uses a single computing node running an internal FT scheme that does not require interaction among computing resources. The remaining DMIPS of the distributed AOCS computer are significantly higher than the two other options, providing additional computing power that can be utilized for enhancing its reliability or computational performance. Based

on these results, it can be concluded that the task-oriented fault-tolerant distributed approach is not only reliable but it is also computationally efficient.

$$DMIPS_{computer} = DMIPS_{node} * n \tag{8.1}$$

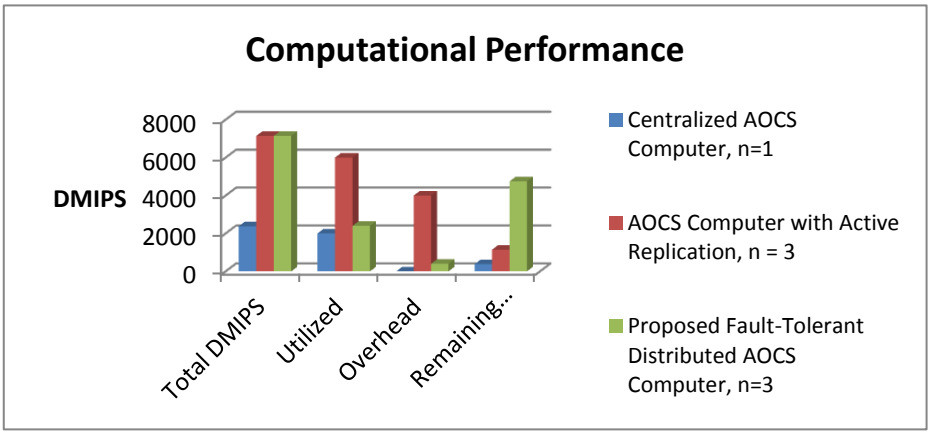


Figure 8.5: Comparison of Computational Performance

8.5 Analysis of Experimental Results

This section presents an analysis of the obtained experimental results in terms of computational integrity and fault coverage.

8.5.1 Computational Integrity

It was mentioned in section 3.6 that the computational integrity of the fault-tolerant computing system depends on the reconfiguration time, t_{Reconf} , and the number of the task state rollbacks, $n_{rollback}$. In fact, the computational integrity of the FTDC system increases with decreasing the values of both parameters. Computational integrity requirements are application specific. This section is dedicated to evaluating the computational integrity of the implemented FT distributed AOCS.

Reconfiguration Time: The reconfiguration time t_{Reconf} should be minimized and ideally should be lower than the minimum task period T to achieve high computational integrity. It is evident from the results given in Table 8.6 that when the cycle time was set to 30 ms, the reconfiguration time was measured to about 114 ~ 132 ms, and for

the large TDMA bus cycle of 300 ms, reconfiguration time was about 222 to 417 ms, which is higher than the acceptable limit of 100 ms (minimum task period). Although, this time value is deterministic, it will cause a task state to rollback, when the task resumes its function on other node.

To keep the reconfiguration time t_{Reconf} small, the sum of the network TDMA cycle time and fault detection time t_D should be less than the minimum task period T. The value of the TDMA cycle time can be reduced by employing a high speed communication network while the fault detection t_D time can also be reduced by the fault detection task period t_{FD_Period} as discussed in section 7.6.

Effect of State Rollback on Computational Integrity: It is evident from Table 8.7, that a maximum of 5 rollbacks can be observed during the migration process. To assess the state rollback impact on the AOCS, the associated effects were simulated in Simulink.

Figure 8.6 shows a Simulink model for the Attitude Determination and Control System (ADCS). The model is comprised of an attitude proportional derivative (PD) controller, spacecraft attitude dynamics block, external disturbances block and a set of sensor blocks. The connections of the each block to the other blocks are as follows:

- The input desired angles, ϕ^d, θ^d, Ψ^d, and angular rates ${\omega_x}^d, {\omega_y}^d, {\omega_z}^d$, are subtracted from the attitude determination angles ϕ, θ, Ψ, and angular rates $\omega_x, \omega_y, \omega_z$ by the vector subtraction blocks. The output of these blocks is the required difference in the desired and actual attitude of the spacecraft which needs to be corrected.
- The attitude difference in terms of angles $[\phi - \phi^d ; \theta - \theta^d ; \Psi - \Psi^d]^T$ and angular rates $[\omega_x - {\omega_x}^{\boldsymbol{d}} ; \omega_y - {\omega_y}^{\boldsymbol{d}} ; \omega_z - {\omega_z}^{\boldsymbol{d}}]^{\boldsymbol{T}}$ as obtained in previous step is passed to the proportional derivative (PD) controller block for attitude correction.
- The PD controller calculates the required torques $u = [T_x, T_y, T_z]$ for the attitude correction of a spacecraft. However, before applying to spacecraft, the external disturbances $[D_x, D_y, D_z]$ are added by the disturbance block.

- The value of the torque u is applied to the spacecraft, which represents the spacecraft attitude dynamics model. The output of the spacecraft dynamics block is fed back to the attitude sensor and determination blocks.

The following describes the insertion of state rollback in the ADCS Simulink model and its effects on the performance of the ADCS.

- In order to insert the state rollback into the Simulink ADCS model, the PD controller input was switched to some fixed attitude angles ϕ, θ, Ψ, and angular rate $\omega_x, \omega_y, \omega_z$ that are represented as angle error and angular rate error in the ADCS Simulink model. This momentary pause of the input angles and angular rates to error angles and angular rates values produces an effect similar to a node failure in a distributed AOCS Computer.
- The time of the pause depends on the number of the task state rollbacks. From Table 8.7, a maximum of 5 rollbacks were observed, however to be more realistic, six rollbacks were inserted into the PD controller input angles and angular rates, which corresponds to a pause of 1800 ms. The exact instance of the pause can be anywhere, but in this case a pause of 1800 ms as inserted at the simulation time of 50 sec, as shown in Figure 8.7.

- The corresponding output observed values of the PD controller are shown in Figure 8.8, where the change due to the state rollbacks is shown by a thin line and the actual attitude is represented by a thick line. It is evident from the results that the momentary rollback did not have a significant effect and recovered quickly.
- The simulation results show that six task state rollbacks is acceptable for this type of application. This result confirms that the experimentally obtained value of 5 state rollbacks is within the acceptable margin.

Based on the modelling outcomes above, we conclude that the distributed AOCS computer meets its computational integrity requirements.

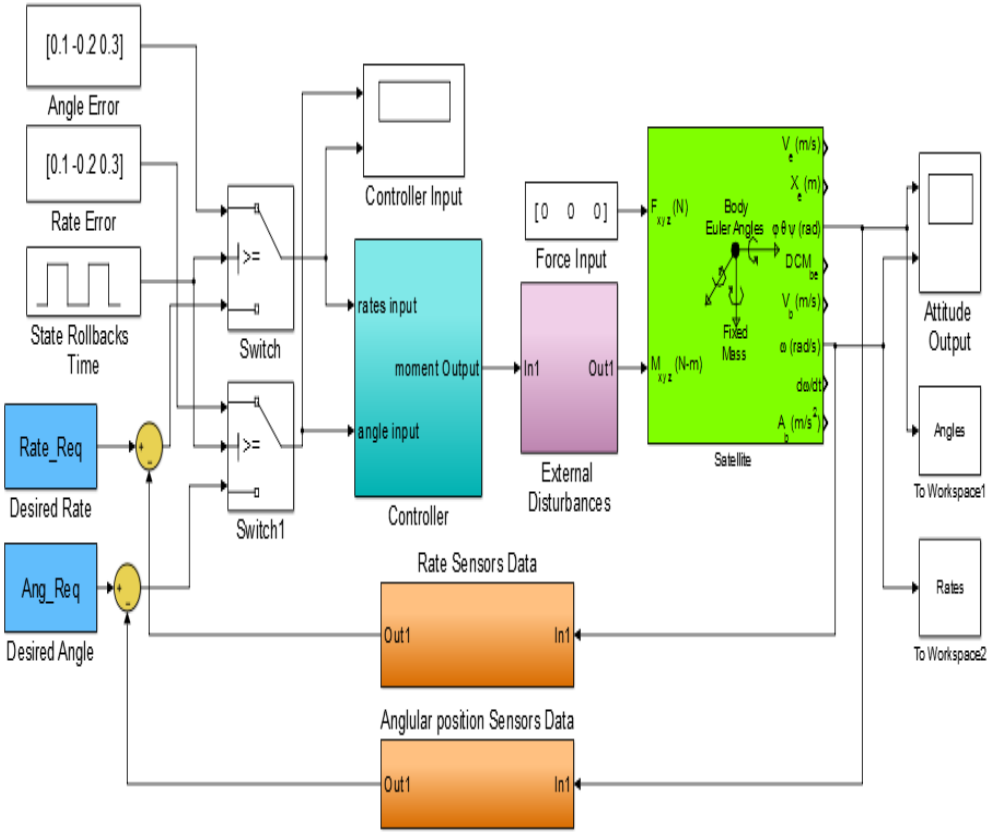


Figure 8.6: Simulink Model of ADCS

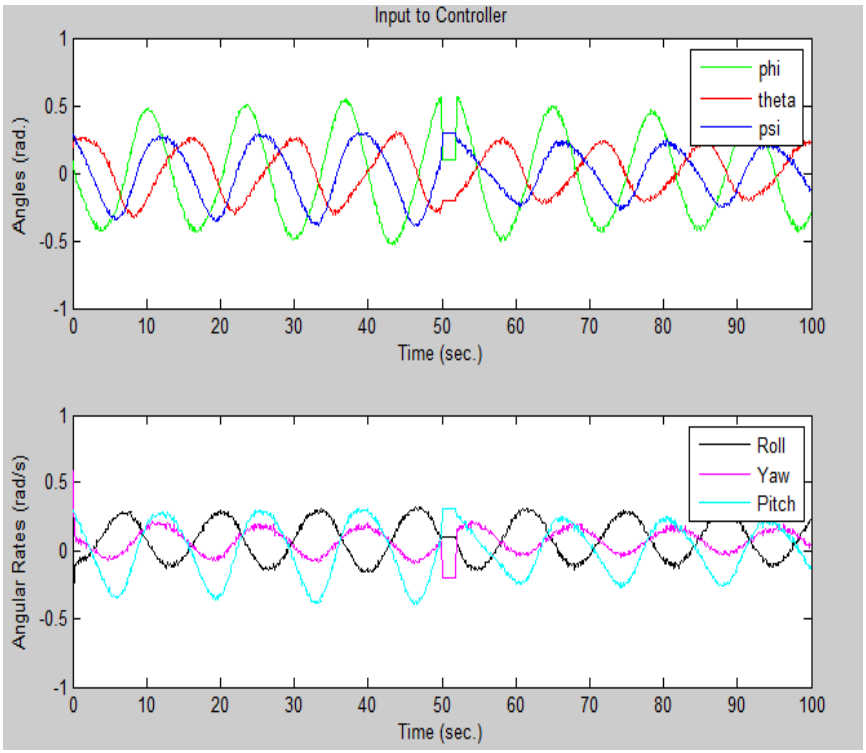


Figure 8.7: ADCS Controller Input with a State Rollback of 6 a) Angles b) Angular Rates

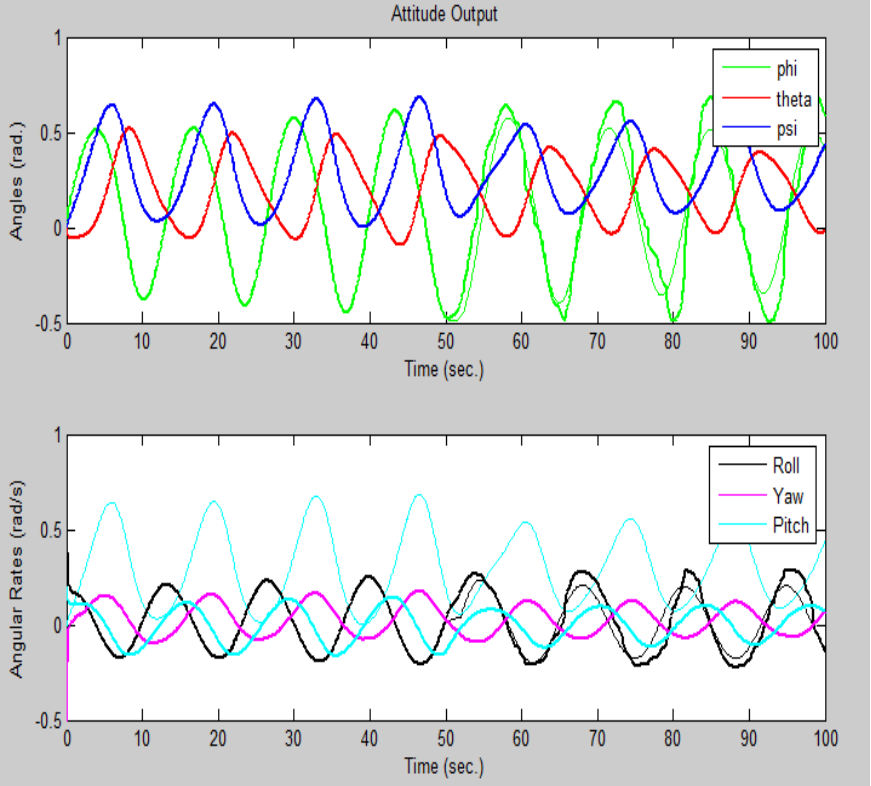


Figure 8.8: Satellite Attitude with a State Rollback of 6 a) Angles b) Angular Rates

8.5.2 Fault Coverage

As stated in section 3.6, a fault-tolerant system is assessed based on the fault coverage. As mentioned earlier in section 4.8 and section 5.3.1, both a software-based and a hardware based fault detection mechanisms were employed. The software-based fault detection covers mainly faults in the application software that may (i) arise due to a design error or (ii) propagate as a result of hardware faults. Hardware fault detection is handled by the AMFT block and uses (i) monitoring of the health status (temperature, current, voltage), (ii) WDT, and (iii) EDAC signals for errors in the main memory. In the following section, we only cover the SDC and the hardware faults.

From the experimental observation of the distributed AOCS computer, the fault coverage for the different methods is shown in Figure 8.9. Any abnormality in the measured values is reported as a fault. The hardware based methods are capable to detect all faults related to health signals and memory errors as shown in Figure 8.9.

With regards to software-based fault detection, only the proposed algorithms for detection of SDC errors were evaluated for fault coverage, as shown in Figure 7.15. The transient algorithm covers the temporary faults in the CPU registers, data memory, and peripheral registers. As the memory faults are covered by a hardware EDAC, therefore, during the implementation of the SDC algorithm, most of the memory was not covered. Due to this, the memory fault coverage value observed during the evaluation was only 20 %. Permanent SDC algorithm covers the faults related to stuck-at-bit faults and bridging faults in microarchitecture elements. Up to 90% of the permanent SDC errors due to struck-at-bit in the registers and ALU are detected, while, in case of bridging faults, the lowest detection capability of 70 % was observed for ALU bridging faults.

It is evident from the results that the proposed algorithms are capable to detect transient as well as permanent faults. On any of the above fault detection, the AMFT block starts the task migration process, enabling fault-tolerance at the architectural level.

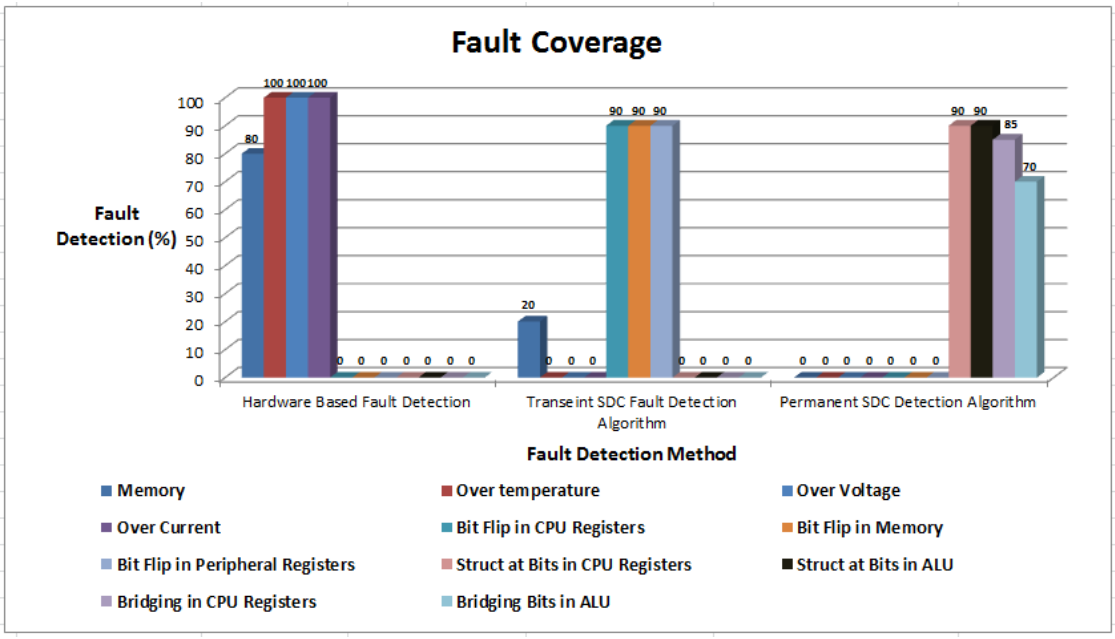


Figure 8.9: Fault Coverage of the FT Distributed AOCS Computer

8.6 Summary

In this chapter, a novel design of a fault-tolerant distributed AOCS computer was presented, which uses task migration to tolerate the failure of computing nodes. The functional design process is presented from the point of the distributed computing required by the AOCS application, which allows us to understand and observe the interaction between the various AOCS processes. The developed AOCS task set and its algorithms were presented.

The AOCS tasks were mapped on a three node FTDC architecture. The hardware and software design of the AOCS distributed computer was then accomplished. The hardware design of the proposed computer was realized as a three-node MPSoC based distributed system. The software part comprised of the AOCS application, system support tasks and fault management functions were implemented in hardware. The proposed distributed AOCS computer was tested to evaluate of proposed FTDC approach under a near-realistic scenario.

The observed results on the reconfiguration time, state rollback, and computational performance are promising. The small, deterministic value of the reconfiguration time showed that a failure could be immediately masked by migrating tasks to other healthy nodes. This reduces the system downtime, thus improving the overall availability of the system. The observations on the state rollback showed that a task could restart its execution with recent state data Δ_{SD} values minimizing the amount of computational loss, thus increases the computational integrity. The observations on the computational performance showed that a significant improvement is achieved by using task migration as compared to other approaches.

In the last section, the approach is further evaluated for computational integrity and fault coverage. This showed that the achieved computational integrity is sufficient for the proposed AOCS application. The results on the fault coverage showed that major faults are covered, which are required to detect a node failure at the architectural level.

In conclusion, the proposed approach to fault-tolerant distributed computing was demonstrated with regards to a satellite AOCS, demonstrating promising results, and future applications. However, the approach can easily be extended and made

applicable to intra-spacecraft and inter-spacecraft payload applications, as well as to other mission critical application domains.

Chapter 9

Conclusions and Future Work

This thesis addresses current requirements of the space industry for fault tolerant High Performance Embedded Computing.

9.1 Research Summary

Current and future space missions demand High Performance Embedded Computing that has to be highly reliable. We propose the use of a distributed computing architecture to address the high-performance demand, and determine a suitable fault management scheme to make the computing reliable. To address the proposed solution, we have investigated the existing literature from the point of view of existing architectures. However, after a detailed review, it was found that no single solution could meet the requirements. Although distributed computing has been employed in space applications, the fault tolerance capability relies on physical redundancy schemes, rendering them highly inefficient and costly. Therefore, a new approach that utilizes the best of distributed computing and a novel distributed fault management scheme is proposed.

The FTDC architecture comprises three main components, a node, a communication network and a fault management scheme. There are two types of nodes suggested: a distributed computing node and an input-output node. A high-speed communication network is recommended to interface all nodes. To determine

the faults in the nodes, a fault management scheme based on distributed coordination is proposed and employed. The fault management scheme requires a separate hardware in parallel with the processing unit within the distributed computing node. This enables a high reliability and availability measure of the scheme. The performance parameters that did not exist in the literature had to be developed to assess our or any other architecture in question.

The proposed fault management scheme is named AMFT, which has been designed from scratch, developed and implemented on hardware, and assessed through newly developed assessment methods. The performance of the AMFT is deemed fully satisfactory, as a standalone unit, as well as when operating in a distributed computing application, after performing a rigorous set of tests.

The proposed architecture improves the current state of the art in addressing a gap in the present knowledge and engineering practice as well providing a design that is practically realizable. This has been proven by the implementation of an MPSoC based distributed computing system that was found to perform to specification.

To further examine the performance, the FTDC architecture was mapped to a realistic space AOCS application, which showed promising results. The system met the rigorous objectives of the AOCS application performing timely task migration in the event of a fault. The effect of the critical issue of rollback of state on system performance was shown to be minimal and sets a baseline for future work.

9.2 Contributions to the State of the Art

The following major contributions to the state-of-the-art have been accomplished in this research:

- *Novel architecture for Fault-Tolerant Distributed Computing is proposed*. The first ever architecture for task oriented fault-tolerant distributed cooperative computing, which applies to intra-spacecraft and inter-spacecraft fault-tolerant distributed computing, has several unique features which make it different from the spacecraft conventional computing architectures. First, it divides a computing system into multiple groups, where each group has its dedicated TDMA based communication network, making the system operations more reliable and timely.

The segregation of computing and input-output nodes avoids the problems of isolation. Furthermore, separate networks for application tasks and fault management functions do not affect the performance of each other. In particular, the architecture is proposed for space applications, however, numerous other embedded applications can also benefit from it.

- *Novel Adaptive Middleware Design for Distributed Fault-Tolerance Management is proposed.* The adaptive middleware for fault-tolerance (AMFT) block, which provides the necessary functionality for fault management and seamless adaptability of a distributed system by tasks migration in case of a failure is the first work that has addresses fault management by migrating tasks in a distributed system. In addition, this is the first work that has adopted a novel approach of task oriented fault-tolerant distributed computing for on-board spacecraft computing systems.
- *Novel MPSoC based Design for Fault-Tolerant Distributed System is designed and implemented.* The MPSoC based approach to the implementation of fault-tolerant distributed computing system is the first ever MPSoC based design of fault-tolerant computing node that uses multiprocessor based design to integrate middleware and application functions. Contrary to traditional design of on-board computing node, this is more flexible, computational efficient while simultaneously requires less area and volume. Its design flexibility allows its use for intra or inter-spacecraft distributed computing applications by just modifying the communication network.
- *New Fault-Tolerant Distributed Attitude and Orbit Control System Computer for use on board satellites is designed.* The AOCS design presented in this thesis is the first ever distributed design of a satellite AOCS computer which is able to migrate tasks following a failure of a computing node. The new MPSoC based design was implemented and fully tested and evaluated.
- *Novel Fault Detection Algorithms are proposed and designed.* The two novel algorithms for detection of silent data corruption combined with symptom-based fault detection require a very small amount of detection time. Contrary to other detection algorithms, are most suited to distributed computing in terms of resource utilization.

- *New Reliability and Availability Models for fault-tolerant computing systems are proposed.* The reliability and availability models for the evaluation and comparison of the proposed approach are the first attempt of using mathematical models for fault-tolerant computing systems, which allowed comparing it against the conventional spacecraft architecture and schemes. The evaluation shows that proposed approach of fault-tolerant distributed computing is more efficient in terms of factors reflecting reliability, availability and high performance computing.
- A *Fault Injection Mechanism is proposed and implemented, which is particularly suitable to distributed computing.* The mechanism to inject faults into distributed computing system is different to other schemes in that it is particularly designed to test distributed computing systems. It is capable of injecting a fault in any of the computing nodes via a software interface on a host computer software. This eases the testing of a distributed computing system under the influence of faults.
- *MPSoC design for a CubeSat mission is designed and implemented*. The features of the MPSoC make it suitable for the computing system design of future very small satellites, e.g. satellites with a mass < 1 kg. For in-orbit demonstration of the proposed approach of fault-tolerant distributed computing, a printed circuit board (PCB) of MPSoC for a CubeSat mission was designed and implemented.

9.3 Future Work

The proposed architecture and the underlying fault management scheme, has shown promising results in the proof of concept as covered in this thesis. However, due to limited scope of this research there is room for future studies in the following directions.

- Extensive performance testing of the Middleware block in the light of scalability, by adding more nodes, in multiple groups. The various implementation of AMFT can be further studied and assessed for the performance improvement.
- Fault detection method can be further studied in light of false alarms.
- The communication network can be further researched, both physical aspects as well as the multiple access scheme employed.

- Resource sharing in the multicore processor scenario can be further researched to minimize access times.
- PCB designed be realized and deployed in future CubeSat based mission, for space qualification.
- Utilize FPGA on the fly reconfigurability to migrate hardware modules, to address the task dependency of special hardware.
- Employ an actual application for a spacecraft payload such as image compression or synthetic aperture radar (SAR), and assess the performance.
- The underlying hardware technology for the implementation of design needs further investigation in terms of radiation susceptibility.

Appendix A.

Definitions

Definition A.1: *Fault*

A fault is a hardware or software defect that can lead to the system entering an incorrect state. Faults are classified as transient, permanent and intermittent based on their duration.

Definition A.2: *Error*

A fault manifests itself as an error, such as a bit that is a zero instead of a one. An error is that part of the system state which is liable to lead to system failure.

Definition A.3: *Failure*

A failure is a state in which the system is restricted from performing its required functions.

Definition A.4: *Fault-Tolerant*

It is an ability of a computing system to continue its service in the event of failure. Failures can be a power, memory or processor failure.

Definition A.5: *Fault Avoidance*

An approach to protect a system so that happening of faults in a system can be avoided. Common fault avoidance methods in spacecraft are shielding, parts screening and rigorous testing.

Definition A.6: *Fault Detection*

A system cannot tolerate faults unless it is aware of it. Fault detection is a process, which enables a system to know its faults.

Definition A.7: *Fault Isolation*

Fault isolation is the property of a system that when something fails, the effect of the failure is limited in scope.

Definition A.8: *Error Recovery*

When a fault is detected, the processor must take action to recover from its effects. Recovery from errors is characterized as roll forward and roll back.

Definition A.9: *Fault Diagnosis*

The process of fault identification is called fault diagnosis.

Definition A.10: *Fail-Safe*

A fail-safe device is one that responds in a way that will cause no harm in case of a failure.

Definition A.11: *Fail-Stop*

In case of a fail-stop system, the system stops producing outputs when it fails.

Definition A.12: *Fail-Symmetric*

The fault results in the same erroneous value being sent to all other redundant units.

Definition A.13: *Fail-Asymmetric*

The fault results in different erroneous values being sent to other redundant units.

Definition A.14: *Real-Time and Non-Real-Time Systems*

Real-time systems include safety-critical systems in which correct behaviour depends upon meeting the real-time requirements of the system. In the Non-Real-Time system, no strict time limit is required for the computing operations.

Definition A.15: *Asynchronous Systems*

In *asynchronous systems,* there is no global clock, and no assumptions about process execution speeds and message delivery delays are made.

Definition A.16: *Synchronous Systems*

In synchronous systems, where computers share a common notion of time, the relative speeds of processes and communication latency are bounded.

Definition A.17: *Reliability*

The reliability of a system at time *t* is the probability that system is operating correctly from time zero until time *t*.

Definition A.18: *Availability*

The availability of a system at time *t* is the probability that the system is available for the time *t* duration.

Definition A.19: *Mean-time-to-failure (MTTF)*

It is the amount of the time that a system is available between outages or failures.

Definition A.20: *Mean-time-to-Repair (MTTR)*

It is the amount of time to repair a system and bring it back online.

Definition A.21: *Mean-time-between-failure (MTBF)*

It is the amount of time that elapses between one failure and the next failure. It is mathematically equal to the sum of the MTTF and MTTR.

Definition A.22: *Fault Monitoring*

Fault monitoring observes the behaviour of the system for checking errors and malfunctions in the software and hardware of computer system.

Definition A.23: *Task Migration*

Tasks that were running on a processor that subsequently failed are migrated to other healthy processor or processors are termed as task migration.

Appendix B.

Derivation of Reliability

B.1 Reliability of Series System

In the series system, components are connected in a series configuration. A failure of one of the system components fails the entire system. Conceptually, a series system is one that is as weak as its weakest link. A graphical description of a series system is shown in Figure B.1.

Figure B.1: Series System of *n* Components

In order to calculate the reliability of the system, usually block diagrams are used whereby each block having its reliability for a given mission *T*. The reliability of the series system is described by equation B.1, if each block reliability differs.

$$R_s = R_1 \times R_2 \times \ldots R_n \tag{B.1}$$

If components are identical, then the reliability is represented by equation B.2

$$R_s = [R_i]^n (\text{if all } i = 1, \ldots, n) \tag{B.2}$$

To obtain equation B.2, statistical knowledge for the derivation of equation B.2 is essential. In the following section, we will derive the equation B.2 using the statistical knowledge.

In a series system of "n" components, the following are two equivalent "events":

"System Success" = "Success of every individual component."

Therefore, the probability of the two equivalent events, that define total system reliability for mission time T (denoted $R(T)$) must be the same:

$$R(T) = P[sys_{success}] = P[\, comp1 \; and \; comp \; 2 \ldots and \; comp \; n] \quad \text{(B.3)}$$

$$= P[comp1_{suc}] \ldots P[comp \; n_{suc}] = \; R_1(T) \ldots R_1(T) \quad \text{(B.4)}$$

$$= e^{-\lambda T} \ldots e^{-\lambda T} = \; (e^{-\lambda T})^n \quad \text{(B.5)}$$

All system components are assumed identical and independent with the same failure rate $"\lambda"$. Hence, the entire system reliability $R(T)$ is equal to the product of all component reliability.

B.2 Reliability of Parallel System

In the parallel system configuration, as long as not all of the system components fail, the entire system works. As all components of a parallel system are connected in a parallel configuration, therefore total system reliability is higher than the reliability of any single system component. A graphical description of a parallel system of "*n*" components is shown in Figure B.2.

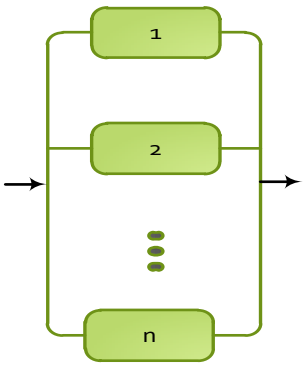


Figure B.2: Parallel System of *m* Components.

Reliability of a parallel system is derived as follows:

$$R_s = 1 - (1 - R_i) \tag{B.6}$$

$$= 1 - (1 - R_1) \times (1 - R_2) \times \ldots (1 - R_n) \tag{B.7}$$

If the component reliability differ, or

$$R_s = 1 - (1 - R_i) = 1 - [1 - R]^n \tag{B.8}$$

If all "*n*" components are identical: [$R_i = R$; $i = 1, ..., n$]

To derive equation B.8, a simple parallel system composed of $n = 2$ identical components are considered. The system can survive only if the first component, or the second component, or both components, survive for mission time, *T*. The same can be written in terms of statistical "events":

$$R(T) = P[sys_{success}\, T] = P[\, X_1 > T \; or \; X_2 > T \; or \; Both > T] \tag{B.9}$$

$$= P\,(X_1 > T) + P(X_2 > T) - P(\, X_1 > T \; or \; X_2 > T) \tag{B.10}$$

$$= R_1\,(T) + R_2\,(T) - R_1\,(T)\; x\; R_2\,(T) \tag{B.11}$$

$$= R_1\,(T)\,[1 - R_2\,(T)] + R_2\,(T) + (1 - 1) \tag{B.12}$$

$$= 1 + R_1\,(T)\,[1 - R_2\,(T)] - [1 - R_2\,(T)] \tag{B.13}$$

$$= 1 - [1 - R_1(T)\,]\,[1 - R_2\,(T)] \tag{B.14}$$

$$= 1 - [1 - P(X_1 > T)]\,[1 - P(X_2 > T)] \tag{B.15}$$

This same approach can be extended to an arbitrary number of "*n*" parallel components which can be identical or different.

B.3 Reliability of Satellite On-Board Computers

In this section, state-of-the-art OBCs and their reliability values are discussed. Firstly, a centralized OBC design will be presented. Following that, different OBC designs will be discussed that use redundancy for fault-tolerance purposes.

The reliability of the OBCs in this section is evaluated using the binomial distribution, referred to as the Bernoulli distribution, which is a simplified and suited to reliability applications [28]. It applies to a situation in which there are n independent trials, whereby an event can either occur (success) or not occur (failure). The probability of success on any one trial is *b*, and that of failure is $1 - b$. The number of successes is denoted by *r*. Thus, the probability of *r* successes in *n* trials with the probability of one success being *b* is given by Equation B.16.

$$B(r; n, b) = \binom{n}{r} b^r (1-b)^{n-r} \tag{B.16}$$

$for\ r = 0,1,2, \dots, n$

where:

$\binom{n}{r} = \frac{n!}{r!(n-r)!}$ ≡ number of combinations of *n* things taken *r* at a time.

B.3.1 Centralized OBC

In the centralized OBC, shown in Figure B.3, one physical internally redundant OBC unit is used. Although the OBC is internally redundant, physical damage or functional failure affecting the complete OBC unit can be catastrophic. Failure of the OBC can lead to either fail-safe mode or it can cause a complete satellite failure. The reliability of the centralized OBC, $R_{cent\_obc}(t)$, is measured by the success probability *b* of the OBC, which follows an exponential distribution. So $R_{cent\_obc}(t)$ can be represented by Equation B.17 where λ represents the failure rate of the centralized OBC, as follows:

$$R_{cent_{obc}}(t) = e^{-\lambda t} \tag{B.17}$$

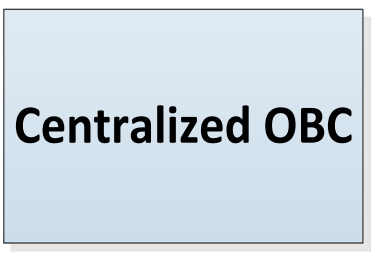


Figure B.3: Centralized OBC Reliability

B.3.2 Cold Standby Redundant OBC

The cold standby OBC, shown in Figure B.4, is the simplest method to provide redundancy against failures in an OBC design. In a cold standby redundancy, usually one identical OBC unit is placed in power off state along with the primary OBC. The redundant back-up OBC unit is powered up in case of failure of the primary OBC. As the primary and back-up OBC units are not synchronized, a considerable amount of time is required for the back-up unit to switch on and to reach a known state. In addition, a supervisory unit is required for the failure detection, isolation and power-up of the back-up OBC as shown in Figure B.5. The supervisory unit monitors the OBC parameters for detection of failures. If the parameter values are violated compared with the predefined limits; it performs switching of power and IOs from the primary to the redundant OBC unit.

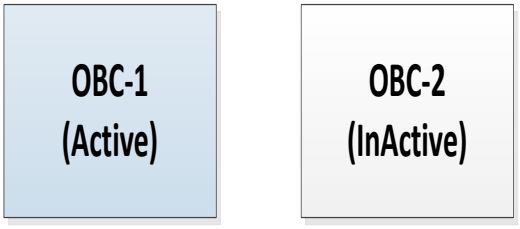


Figure B.4: Cold Standby OBC

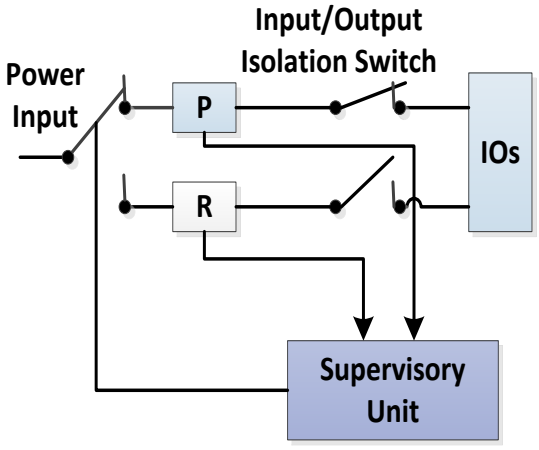


Figure B.5: Supervisory Unit

The reliability of the cold standby redundant OBC, $R_{csb\_obc}(t)$, is equal to the sum of the probabilities of 2 out of 2 and 1 out of 2 working OBCs as represented by Equation

B.18. After solving Equation B.18 using the exponential distribution for the probability of success *b*, we obtain the overall reliability, $R_{csb\_obc}(t)$, for the cold standby redundant OBC as given by Equation B.21.

$$R_{csb\_obc} = R_{sup} * \ [B(2;2,b) + B(1;2,b)] \tag{B.18}$$

$$R_{csb\_obc} = R_{sup} * \binom{2}{2} b^2(1-b)^0 + \binom{2}{1} b^1(1-b)^1 \tag{B.19}$$

$$R_{csb\_obc} = R_{sup} * (2b - b^2\ ) \tag{B.20}$$

$$R_{csb_{obc}}(t) = R_{sup} * \left(2e^{-\lambda t} - e^{-2\lambda t}\right) \tag{B.21}$$

where:

R_{csb_obc}	Reliability of cold standby OBC
R_{sup}	Reliability of Supervisor Unit
B(r;n,b)	Binomial expression showing '*r*' out of '*n*' are healthy [28]
b	Probability of Success
λ	Failure Rate

The reliability of the cold standby OBC, $R_{csb\_obc}(t)$, is higher than the reliability of the centralized OBC, $R_{cent\_obc}(t)$, assuming that the supervisory unit is a high reliability component too. This is because of the inherent availability of a redundant unit which can be switched on in case of a failure of the main OBC.

B.3.3 Warm Standby Redundant OBC

In case of warm standby redundancy as shown in Figure B.6, both OBCs are in a powered on-state. However, only the primary OBC is executing tasks while the back-up OBC unit is in an idle state. Similar to the cold standby redundancy, a supervisor is required for the detection, isolation and switch over to the back-up OBC unit in case of a failure. Optionally, the back-up unit can be used as a supervisor and can perform the fault tolerance management functions. There is a cost versus reliability trade-off of

using an external supervisor against an internally implemented supervisor on the back-up OBC. Usually, it is preferable to have a separate supervisory unit.

The downtime of the warm standby OBC is considerably less than that of the cold standby OBC. This is because the switching over to the redundant OBC requires very less time due to its on-state. So, in terms of availability the warm standby is comparatively a better option than the cold standby redundant OBC. However, the reliability of the warm standby OBC is similar and can be represented by Equation B.21 too.

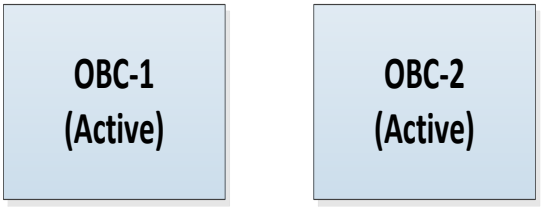


Figure B.6: Warm Standby OBC

B.3.4 N-Modular Redundant OBC

The N-modular redundant OBC design comprises more than one OBC units, all of them running in parallel, as depicted in Figure B.7. The choice of *N* is based on the required system reliability and usually a minimum three nodes are used because a two-node system can only detect a fault but it does not know which one of the two units is faulty. All units are synchronized, processing the same input information and generating the same output data. In addition, the final output, which is delivered to the target system, is derived as a result of a majority voting stage. A voter detects a fault based upon the majority vote of the module outputs [36]. In majority voting the number of healthy nodes should always be greater than the number of the faulty nodes in order for the voter to deliver the correct output.

A failure of the voter in an N-modular redundant OBC can be catastrophic, leading to the failure of the whole OBC. Therefore, the reliability of the voter unit is very important. A voter can be implemented in hardware or software. A hardware voter, as shown in Figure B.8, compares the processors' internal buses. Its placement is limited by a certain distance from the main processors because of the need for clock synchronization among the processors at a fine-grained level. On the other hand, a

software-based voter works at a message level, as shown in Figure B.9, whereby each of the processing units exchanges messages with the voter to generate an output. Software based voters are more relaxed in terms of synchronization among the processing units. In software based voting, messages are sent to the voter via network/bus from each processing unit. The voter then compares the messages and generates an output for the IOs. A software voter called 'master/slave' is implemented on the master processing unit as a software entity [124]. In a 'master/slave' voter configuration, all units send data to the master for the voted output.

The main drawbacks of the N-modular redundant method are strict clock synchronization, common mode failures, difficult isolation process, increased fan-in/fan-out, etc. In addition, it is difficult to resynchronize a node after it has recovered from a fault. Software voters can alleviate these problems, however, the final voted output has a higher latency due to the exchange of communication messages.

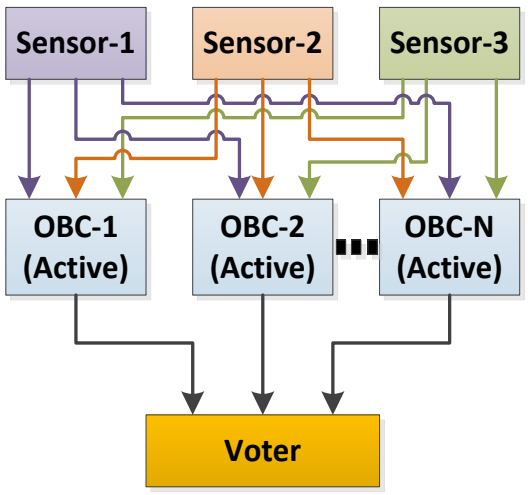


Figure B.7: N-Modular Redundant OBC

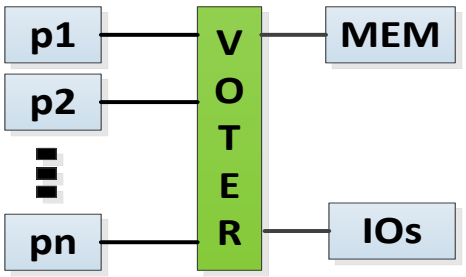


Figure B.8: Hardware Voter

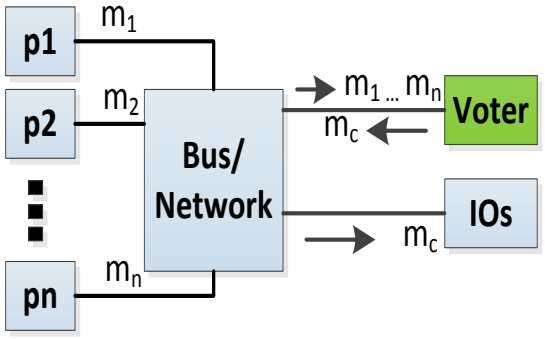


Figure B.9: Software Voter

We consider here the Triple Modular Redundant (TMR) OBC, as the most common case of an N-modular redundant OBC design. We assume that the voter is an external hardware entity. The overall reliability of the TMR OBC with a hardware voter, $R_{TMR\_HV(3\text{-}2)}$, is given by Equation B.24 as follows:

$$R_{TMR-HV(3-2)} = R_{voter} * [B(3;3,b) + B(2;3,b)] \tag{B.22}$$

$$R_{TMR-HV(3-2)} = R_{voter} * \binom{3}{3} b^3(1-b)^0 + \binom{3}{2} b^2(1-b)^1 \tag{B.23}$$

$$R_{TMR-HV(3-2)} = R_{voter} * (3b^2 - 2b^3) \tag{B.24}$$

where

R_{voter}: $Reliability\ of\ Hardware\ Voter$

Contrary to the hardware voter, if a software voter is embedded inside the redundant nodes in a master/slave configuration, then the system reliability with a software voter, $R_{TMR\_SV(3\text{-}2)}$, is represented by Equation B.25 below:

$$R_{TMR-SV(3-2)} = (3b^2 - 2b^3) \tag{B.25}$$

The success probability b is an exponential distribution, so the probability of the success b is substituted by $e^{-\lambda t}$ and the reliability of the TMR OBC can be represented by Equations B.26 and B.27, as follows:

$$R_{TMR-HV(3-2)}(t) = R_{voter} * \left(3e^{-2\lambda t} - 2e^{-3\lambda t}\right) \quad \text{(B.26)}$$

$$R_{TMR-SV(3-2)}(t) = \left(3e^{-2\lambda t} - 2\,e^{-3\lambda t}\right) \quad \text{(B.27)}$$

Generalized expressions for a maximum number of *n* nodes in a system with a hardware and a software voter are given by Equations B.28 and B.29 respectively, where *r* represents the healthy number of nodes and *k* is the summation index, as follows:

$$R_{TMR-HV}(t) = R_{voter} * \sum_{k=r}^{n} \binom{n}{k} e^{-k\lambda t}\,(1 - e^{-\lambda t})^{(n-k)} \quad \text{(B.28)}$$

$$R_{TMR-SV}(t) = \sum_{k=r}^{n} \binom{n}{k} e^{-k\lambda t}\,(1 - e^{-\lambda t})^{(n-k)} \quad \text{(B.29)}$$

B.3.5 1:N Redundant OBC

The 1:N redundant system comprises multiple computing units along with a standby unit. All the primary units have similar functions thus allowing the standby unit to back-up any of the primary units in case of a failure. The switch over to the back-up unit is decided by one of the spare units called a checker, which continuously steps through each of the working units. If the checker disagrees with one of the working unit, it is assumed to be faulty and is replaced by a spare unit.

Appendix C.

Implementation Details

C.1 Board Level Implementation

C.1.1 Resources

The resources required to implement the fault-tolerant distributed system depend on the number of computing nodes that are required, which will be determined by the requirements of the particular system. A greater number of nodes can be employed to provide a higher level of fault tolerance for environments in which multiple computing units may be expected to fail. A greater number of nodes may also be utilized to provide additional processing power. This must be balanced against the additional resources required, which is an especially important issue for applications such as spacecraft systems.

Electrical Power Consumption: Electrical Power is a scarce resource in embedded computing, particularly on board spacecraft. Therefore, it is essential for a distributed processing system to utilize electrical power as efficiently as possible. We measured the electrical power for the AMFT and processing unit using the National Instruments (NI) LabView 2011 and data Acquisition device [239]. As shown in Figure C.1, the computing load assigned to the individual units did not have a large impact on the unit's power consumption and the total power was almost the same. This is an important result that shows that migration of the tasks to other computing unit does not

have a large impact on the node electrical power consumption itself. In distributed computing system, the total power consumption increases linearly with the number of nodes. It may be possible to reduce power consumption through the use of dynamic frequency scaling. As shown in Figure C.2, by decreasing the operating frequency $1/4_{th}$ for each node, electrical power of the distributed system can be reduced to half value.

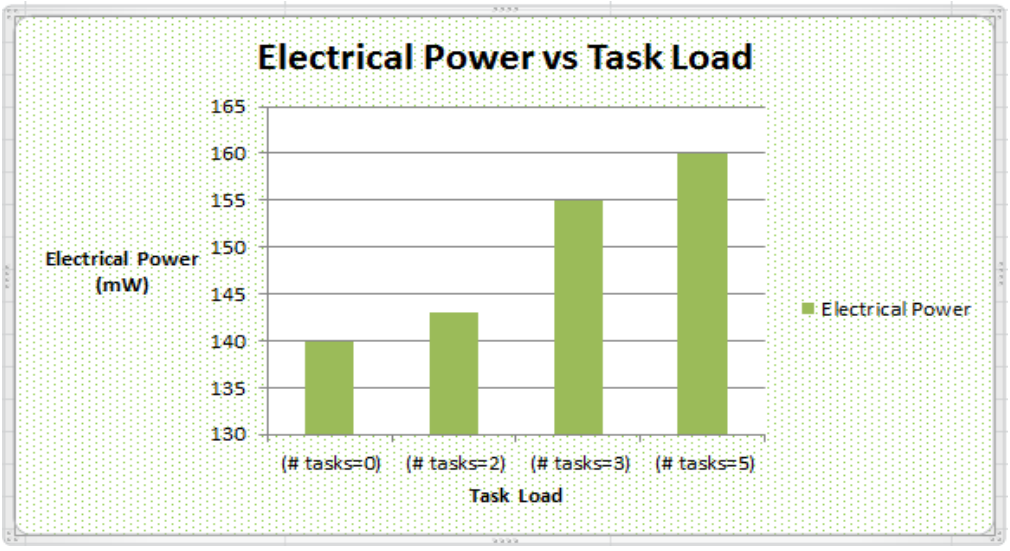


Figure C.1: Effect on Electrical Power with Task Load Variation.

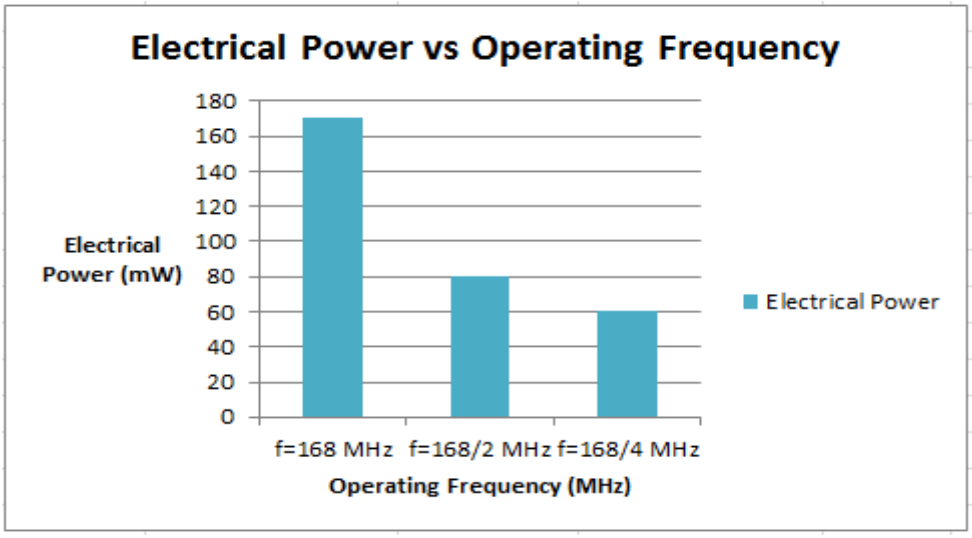


Figure C.2: Effect on Electrical Power with Frequency Variation.

C.2 MPSoC based Implementation

C.2.1 Electrical Circuit Diagram

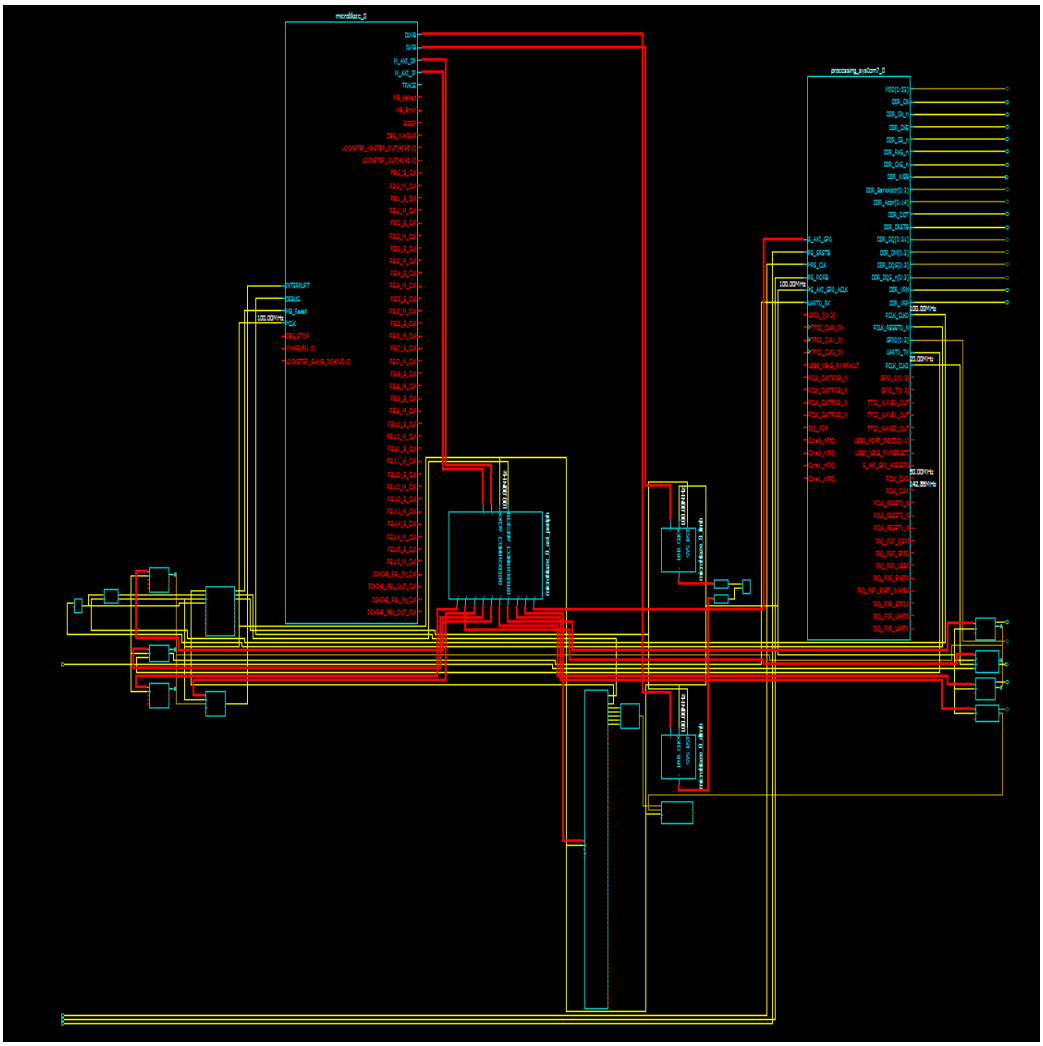


Figure C.3: Circuit Diagram of MPSoC Implementation.

C.2.2 Device Utilization

Slice Logic Utilization	*Used*	*Available*	*Utilization*
Number of Slice Registers	3,551	106,400	3%
Number used as Flip Flops	3,516		
Number used as Latches	0		
Number used as Latch-thrus	0		

Number used as AND/OR logics	35		
Number of Slice LUTs	4,628	53,200	8%
Number used as logic	4,234	53,200	7%
Number using O6 output only	3,263		
Number using O5 output only	135		
Number using O5 and O6	836		
Number used as ROM	0		
Number used as Memory	239	17,400	1%
Number used as Dual Port RAM	64		
Number using O6 output only	0		
Number using O5 output only	0		
Number using O5 and O6	64		
Number used as Single Port RAM	0		
Number used as Shift Register	175		
Number using O6 output only	174		
Number using O5 output only	1		
Number using O5 and O6	0		
Number used exclusively as route-thrus	155		
Number with same-slice register load	112		
Number with same-slice carry load	19		
Number with other load	24		
Number of occupied Slices	1,971	13,300	14%
Number of LUT Flip Flop pairs used	5,419		
Number with an unused Flip Flop	2,124	5,419	39%
Number with an unused LUT	791	5,419	14%
Number of fully used LUT-FF pairs	2,504	5,419	46%
Number of unique control sets	299		
Number of slice register sites lost to control set restrictions	1,117	106,400	1%
Number of bonded IOBs	20	200	10%
Number of LOCed IOBs	20	20	100%
Number of bonded IOPAD	130	130	100%
IOB Flip Flops	4		
Number of RAMB36E1/FIFO36E1s	66	140	47%
Number using RAMB36E1 only	66		
Number using FIFO36E1 only	0		

Number of RAMB18E1/FIFO18E1s	0	280	0%
Number of BUFG/BUFGCTRLs	5	32	15%
Number used as BUFGs	5		
Number used as BUFGCTRLs	0		
Number of IDELAYE2/IDELAYE2_FINEDELAYs	0	200	0%
Number of ILOGICE2/ILOGICE3/ISERDESE2s	0	200	0%
Number of ODELAYE2/ODELAYE2_FINEDELAYs	0		
Number of OLOGICE2/OLOGICE3/OSERDESE2s	8	200	4%
Number used as OLOGICE2s	8		
Number used as OLOGICE3s	0		
Number used as OSERDESE2s	0		
Number of PHASER_IN/PHASER_IN_PHYs	0	16	0%
Number of PHASER_OUT/PHASER_OUT_PHYs	0	16	0%
Number of BSCANs	1	4	25%
Number of BUFHCEs	0	72	0%
Number of BUFRs	0	16	0%
Number of CAPTUREs	0	1	0%
Number of DNA_PORTs	0	1	0%
Number of DSP48E1s	3	220	1%
Number of EFUSE_USRs	0	1	0%
Number of FRAME_ECCs	0	1	0%
Number of ICAPs	0	2	0%
Number of IDELAYCTRLs	0	4	0%
Number of IN_FIFOs	0	16	0%
Number of MMCME2_ADVs	1	4	25%
Number of OUT_FIFOs	0	16	0%
Number of PHASER_REFs	0	4	0%
Number of PHY_CONTROLs	0	4	0%
Number of PLLE2_ADVs	0	4	0%
Number of PS7s	1	1	100%
Number of STARTUPs	0	1	0%
Number of XADCs	0	1	0%
Average Fanout of Non-Clock Nets	4.6		

C.2.3 Permanent Fault Injection Design

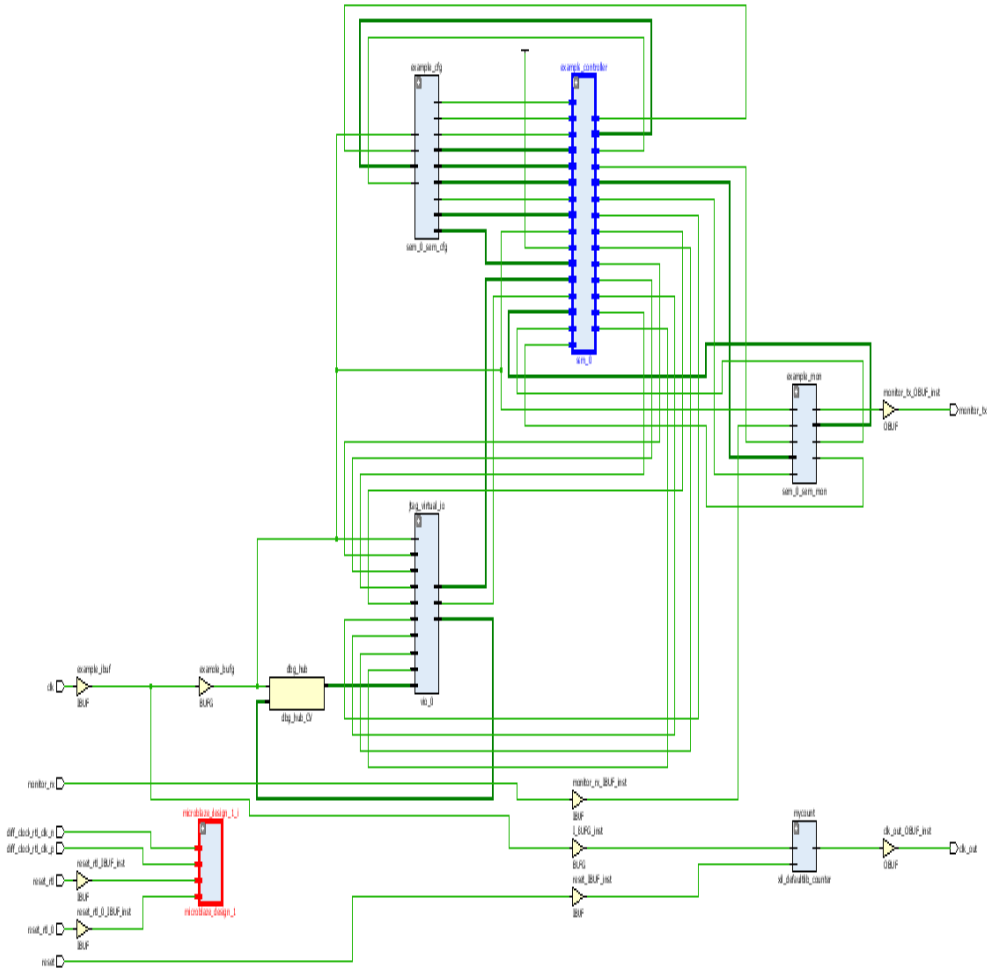


Figure C.4: Permanent Fault Injection Mechanism Implementation.

Appendix D.

Distributed Computing Node PCB Design Data

D.1 Printed Circuit Board Layout

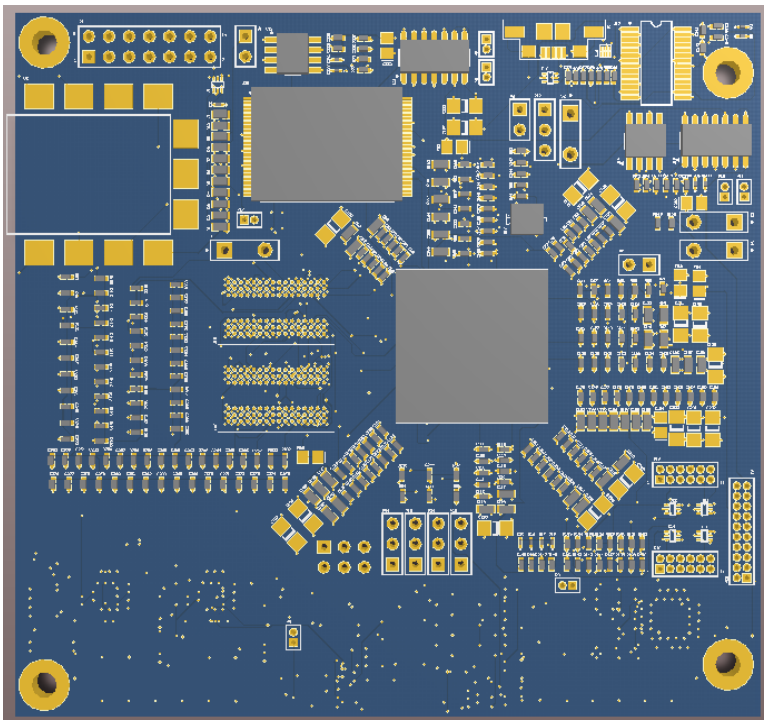


Figure D.1: Front View.

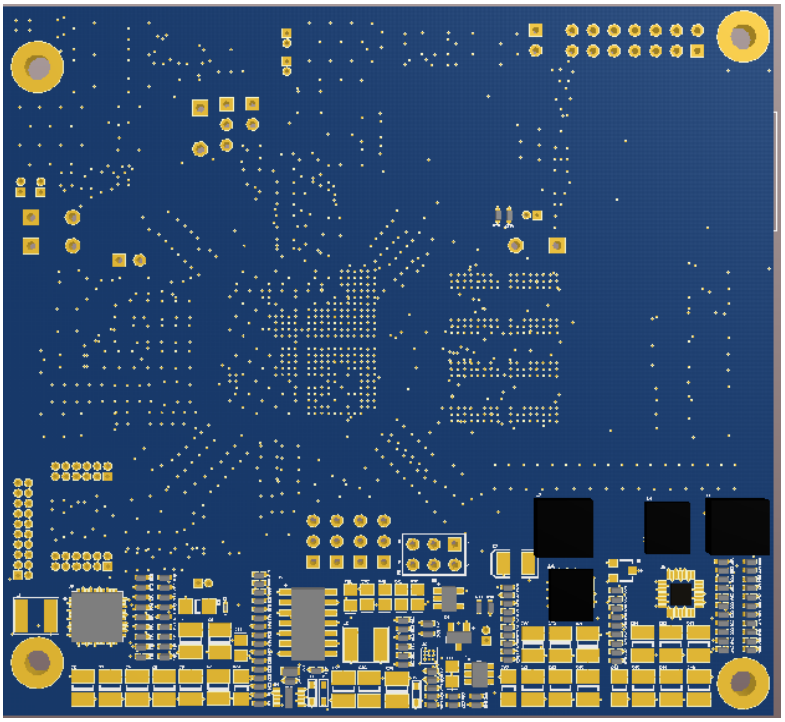


Figure D.2: Back View.

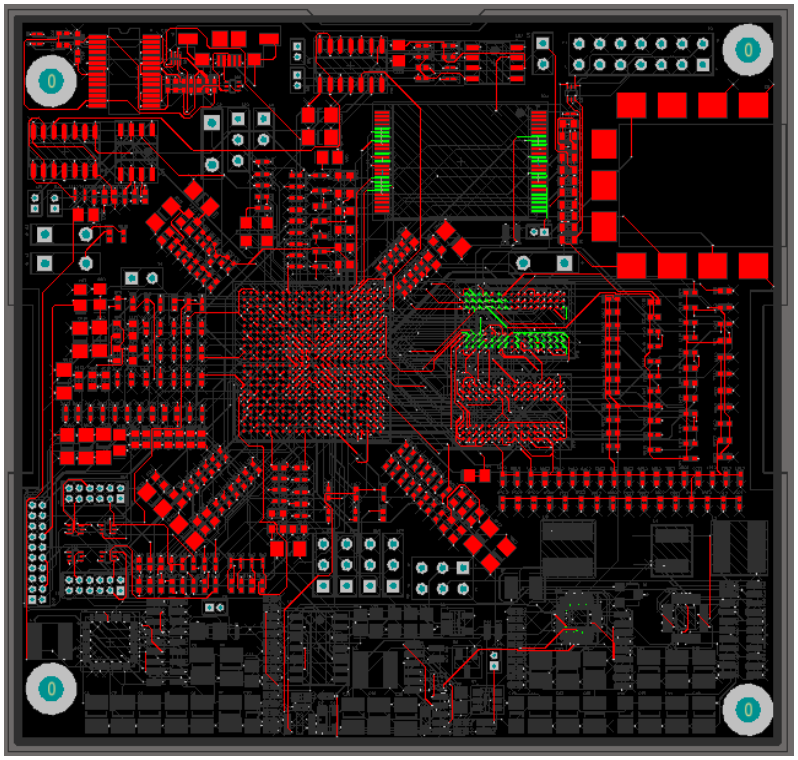


Figure D.3: Top Layer.

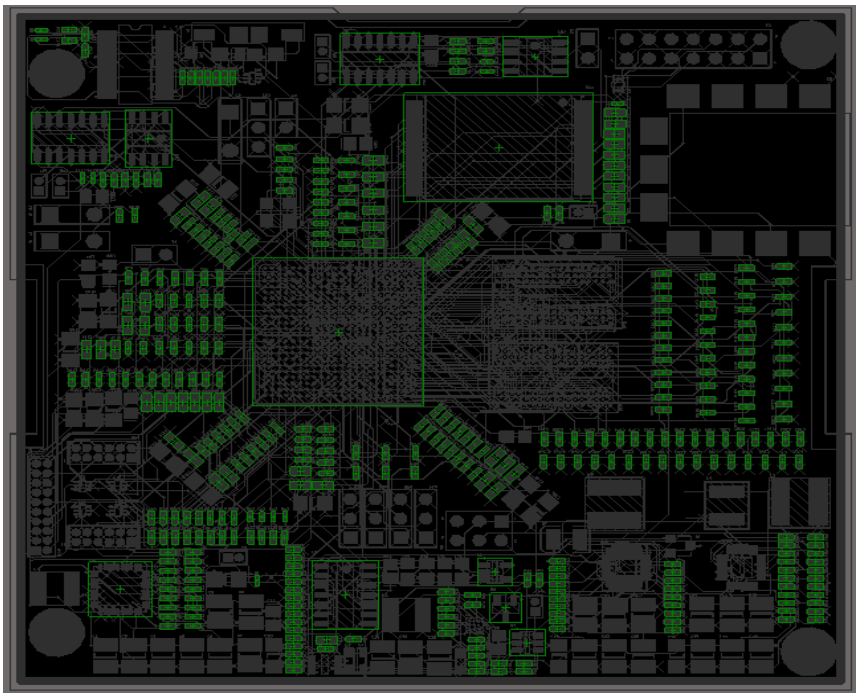


Figure D.4: Top Overlay.

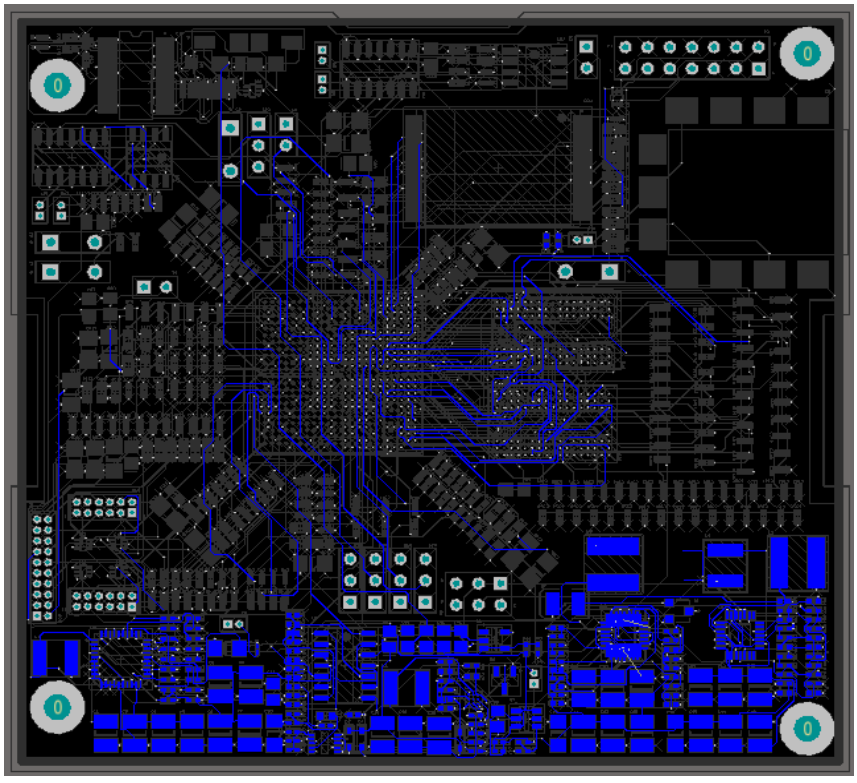


Figure D.5: Bottom View.

D.2 Bill of Materials

Bill of Materials

Bill of Materials For Project [PCB_Project1.PrjPCB] (No PCB Document Selected)

Source Data From: PCB_Project1.PrjPCB
Project: PCB_Project1.PrjPCB
Variant: None

Creation Date: 13/10/2014 10:58:13
Print Date: 13-Oct-14 10:58:16 AM

Footprint	Comment	LibRef	Designator	Description	Quantity
SSOP50P202	3DFN4G08VS16	3DFN4G08VS16	U20	4Gbit Flash Nand Organized as 512Mx8,bas	1
SOT23_N	BZX84-A3V0	BZX84-A3V0	D6	Voltage Regulator Diode	1
O28	CY7C64225-28P	CY7C64225-28P	U15	USB-to-UART Bridge Controller	1
MHDR2X10	Header 10X2	Header 10X2	P20	Header, 10-Pin, Dual row	1
HDR2X3	Header 3X2	Header 3X2	P5	Header, 3-Pin, Dual row	1
HDR2X7_CEN	Header 7X2H	Header 7X2H	P1	Header, 7-Pin, Dual row, Right Angle	1
SMT2	JTAG-SMT2	JTAG-SMT2	U2		1
3.5X2.8X1.9	LED3	LED3	D7	Typical BLUE SIC LED	1
SOIC127P600	LM339M	LM339M	U5	Low Power Low Offset Voltage Quad Compa	1
BGA3x3	MAX15053EWL+	MAX15053EWL+	U6	High Efficiency, 2A, Current-Mode Synchron	1
MAX1510	MAX1510ETB	MAX1510ETB	U4	Low Voltage DDR Voltage Regulators	1
SOT23-6N	MAX1983EUT+	MAX1983EUT+	U7	Low Voltage Low-Dropout Linear Regulators	1
SOT23-5N	MAX6037AAUK1	MAX6037AAUK1	U8	Low Power, Fixed and Adjustable Reference	1
QFN50P600X	MAX8686ETL+	MAX8686ETL+	U3	Single/Multiphase, Step-Down DC-DC Conv	1
SOT-23	NMOS-2	NMOS-2	Q1	N-Channel Power MOSFET	1
Osc33.33	OSC-33.333333	OSC-33.333333	U17		1
0603	Res3	Res3	F1	Resistor	1
1206	Res3	Res3	R21	Resistor	1
DQE-8	TXS0102DQE	TXS0102DQE	U16	2 Bit Bidirectional Voltage Level Translator F	1
USBMINIAB	USB Micro AB	USB Micro AB	J1		1
BGA484C80P	XC7Z020CLG48	XC7Z020CLG48	U1		1
SMD_IND		Inductor	L1, L2	IHLP-2020CZ-01_1.5 uH, Inductor	2
SMD_IND2		Inductor	L4, L6	NRS5020T1R0NMGJ, 1 uH, 3.6A,5020	2
SMD_IND1		Inductor	L3, L5	SRN6045-1R0Y, 1 uH, 4.2-8.5A, CDRH6D3	2
1608[0603]	Cap Semi	Cap Semi	C1, C23	Capacitor (Semiconductor SIM Model)	2
MHDR2X6	Header 6X2A	Header 6X2A	P18, P19	Header, 6-Pin, Dual row	2
Quad28	MAX15021ATI+	MAX15021ATI+	U9, U10	Dual, 4A/2A, 4 MHz Step-Down DC-DC Reg	2
SOIC127P600	MCP2561/2	MCP2561/2	U12, U14	High Speed CAN Transceiver	2
SDRAM_DDR	MT41K128M16H	MT41K128M16H	U18, U19	1.35 DDR3L SDRAM Memory	2
235E_L	TXS0104EDR	TXS0104E	U11, U13	4-Bit Bidirectional Voltage Level Translator	2
0805	0 ohm shunt	Inductor	FB2, FB3, FB4	Inductor	3
HDR1X2	Header 2	Header 2	P2, P6, P7	Header, 2-Pin	3
SOD-523F	RB751S40	1N5246B	D3, D4, D6	40V 30 mA Schottky Diode	3
0603		Cap Semi	C229, C231, C236, C238	Capacitor (Semiconductor SIM Model)	4
SPST-2	SW-PB	SW-PB	S1, S2, S3, S4	Switch	4
HDR1X3	Header 3	Header 3	P12, P13, P14, P15, P16	Header, 3-Pin	5
SOT-563	QZX563C6V8C	QZX563C6V8C	D1, D10, D11, D12, D13, D14	Zener Diode Pack	6
0603	LED2	LED2	D2, D8, D9, D15, D16, D17, D18	Typical RED, GREEN, YELLOW, AMBER G	7
MHDR1X2	MHDR1X2	MHDR1X2	P3, P4, P8, P9, P10, P11, P17	Header, 2-Pin	7
0805		Cap Semi, Induc	C22, C39, C40, C41, C42, C194, C225,	Capacitor (Semiconductor SIM Model), Indu	11
J1-0603	Res3	Res3	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,	Resistor	12
1206		Cap Semi	C20, C43, C60, C100, C101, C102, C120,	Capacitor (Semiconductor SIM Model)	19
1210		Cap Semi	C2, C3, C4, C5, C6, C7, C8, C9, C25,	Capacitor (Semiconductor SIM Model)	23
1608[0603]		Cap Semi	C97, C98, C99, C103, C104, C105, C117,	Capacitor (Semiconductor SIM Model)	36
0402	Res3	Res3	R11, R12, R13, R14, R15, R16, R17, R18,	Resistor	64
0402		Cap Semi, Res3	C10, C11, C12, C13, C14, C15, C16, C17,	Capacitor (Semiconductor SIM Model), Resi	281
					527

.

Appendix E.

Software

E.1 Application Software Top Level Design

```
int main(void) {

   LED_VAL = 0;
   //Disable cache on OCM
   Xil_SetTlbAttributes(
         0xFFFF0000, 0x14de2); // S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
   Xil_Out32(CPU1STARTADR,
         0x06000000);
   dmb();
   //waits until write has finished
   sev();

   prvSetGpioHardware();

   xTaskCreate(vLedFlashTask,
         (signed char *)"LED Flash",
         configMINIMAL_STACK_SIZE,
         NULL,
         configMAX_PRIORITIES - 4,
         (xTaskHandle*)NULL);
   /* Configure the hardware ready to run the test. */

   prvCreateQueues();
   prvCreateSemaphores();
   prvCreateFtdcTasks();

   /* Start the scheduler. */
   vTaskStartScheduler();

   /* If all is well, the scheduler will now be running, and the following line
    will never be reached.  If the following line does execute, then there was
    insufficient FreeRTOS heap memory available for the idle and/or timer tasks
    to be created.  See the memory management section on the FreeRTOS web site
    for more details. */

   for (;;)
      ;
}
```

```
/*-----------------------------------------------------------*/
static void prvCreateFtdcTasks(
      void) {
   xTaskCreate(
         vAmftReceiverTask,
         (signed char *)"AMFT_RECEIVER",
         configMINIMAL_STACK_SIZE,
         NULL,
         configMAX_PRIORITIES - 1,
         (xTaskHandle*)NULL);
   xTaskCreate(
         vAmftSenderTask,
         (signed char *)"AMFT_SENDER",
         configMINIMAL_STACK_SIZE,
         NULL,
         configMAX_PRIORITIES - 3,
         (xTaskHandle*)NULL);
   xTaskCreate(
         vMissionTaskManagerTask,
         (signed char *)"MISSION_TASK_MANAGER",
         configMINIMAL_STACK_SIZE,
         NULL,
         configMAX_PRIORITIES - 2,
         (xTaskHandle*)NULL);
}
/*-----------------------------------------------------------*/
/*-----------------------------------------------------------*/

void prvCreateQueues(void) {
   xTaskListQueue =
         xQueueCreate(10,
               sizeof(struct taskListData));
   if (xTaskListQueue
         == pdFALSE) {
      //xil_printf("ERROR: Unable to create queue 1\r\n ");
      // for kernel aware debugging purposes only
   } vQueueAddToRegistry(xTaskListQueue, "TaskListQueue");

   xStateUpdatedQueue =
         xQueueCreate(100,
               sizeof(uint8_t));
   if (xStateUpdatedQueue
         == 0) {
      //xil_printf("ERROR: Unable to create queue 2\r\n ");
      // for kernel aware debugging purposes only
   } vQueueAddToRegistry(xStateUpdatedQueue, "StateUpdatedQueue");
}

/*-----------------------------------------------------------*/
void prvCreateSemaphores(void) {
   vSemaphoreCreateBinary(
         amftDataReceivedSemaphore);
   int a;
   if (amftDataReceivedSemaphore
         == NULL ) {
      a = 1;
      // #ifdef print_status
      //   printf("ERROR: Unable to create semaphore 1     ");
      //#endif
   }

}

/*-----------------------------------------------------------*/
```

```
/*-----------------------------------------------------------*/
void prvCreateQueues(void) {
   xTaskListQueue =
         xQueueCreate(10,
               sizeof(struct taskListData));
   if (xTaskListQueue
         == pdFALSE) {
      //xil_printf("ERROR: Unable to create queue 1\r\n ");
      // for kernel aware debugging purposes only
   } vQueueAddToRegistry(xTaskListQueue, "TaskListQueue");

   xStateUpdatedQueue =
         xQueueCreate(100,
               sizeof(uint8_t));
   if (xStateUpdatedQueue
         == 0) {
      //xil_printf("ERROR: Unable to create queue 2\r\n ");
      // for kernel aware debugging purposes only
   } vQueueAddToRegistry(xStateUpdatedQueue, "StateUpdatedQueue");
}

/*-----------------------------------------------------------*/
void prvCreateSemaphores(void) {
   vSemaphoreCreateBinary(
         amftDataReceivedSemaphore);
   if (amftDataReceivedSemaphore
         == NULL ) {
      //   printf("ERROR: Unable to create semaphore 1     ");
      }

}

/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook(
      void) {
   taskDISABLE_INTERRUPTS();
   for (;;)
      ;
}
/*-----------------------------------------------------------*/



/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook(
      xTaskHandle pxTask,
         signed char *pcTaskName) {
   (void) pcTaskName;
   (void) pxTask;

   /* Run time stack overflow checking is performed if
    configCHECK_FOR_STACK_OVERFLOW is defined to 1 or 2.  This hook
    function is called if a stack overflow is detected. */taskDISABLE_INTERRUPTS();
   for (;;)
      ;
}
/*-----------------------------------------------------------*/
void vApplicationSetupHardware(
      void) {
   int Status;
   Status =
         vInitialiseSerialPort_0();
   if (Status != XST_SUCCESS) {
      //xil_printf("UART_INIT Failed \r\n");

   }
   /*Do Nothing */
}
```

```
/* Task for activation / de-activation */
void vMissionTaskManagerTask(
      void *pvParameters) {
   struct tasksRunningInfo tasksRunning;
   tasksRunning.taskId[0] = 0;
   tasksRunning.taskId[1] = 1;
   tasksRunning.taskId[2] = 2;
   tasksRunning.taskId[3] = 3;
   tasksRunning.taskId[4] = 4;
   int i;
   for (i = 0;
         i < MAX_MISSION_TASKS;
         i++) {
      tasksRunning.taskRunningNow[i] =
            0;
   }

   struct taskListData receivedTaskListData;
   uint8_t taskInTLM;

   for (;;) {
      xQueueReceive(
            xTaskListQueue,
            &receivedTaskListData,
            portMAX_DELAY);

      int allTasksIndex;
      for (allTasksIndex = 0;
            allTasksIndex
                  < MAX_MISSION_TASKS;
            allTasksIndex++) {
         taskInTLM = 0;
         int receivedTasksIndex;
         for (receivedTasksIndex =
               0;
               receivedTasksIndex
                     < receivedTaskListData.numberOfTasks;
               receivedTasksIndex++) {
            // check if task is in Task List Message
            if (receivedTaskListData.taskId[receivedTasksIndex]
                  == tasksRunning.taskId[allTasksIndex]) {
               taskInTLM = 1;
            }
         }

        }
        // task is in Task List Message but is not currently running
        if ((taskInTLM == 1)
              && (tasksRunning.taskRunningNow[allTasksIndex]
                    == 0)) {
           if (startTask(
                 tasksRunning.taskId[allTasksIndex])
                 == 1) {
              // task is now running
              tasksRunning.taskRunningNow[allTasksIndex] =
                    1;
           }
        }
```

```
         // task is not in Task List Message but is currently running
         else if ((taskInTLM
               == 0)
               && (tasksRunning.taskRunningNow[allTasksIndex]
                     == 1)) {
            if (stopTask(
                  tasksRunning.taskId[allTasksIndex])
                  == 1) {
               // task is now stopped
               tasksRunning.taskRunningNow[allTasksIndex] =
                     0;
            }
         }
      }
   }
}
/*-----------------------------------------------------------*/


uint8_t startTask(
      uint8_t taskId) {

   //unsigned char timing_flag=pdFALSE;
   switch (taskId) {
   case 0:
      if (xTaskCreate(vMissionTask0, (signed char *)"MISSION TASK 0",
            configMINIMAL_STACK_SIZE, NULL, configMAX_PRIORITIES - 4,
            &missionTask0Handle) != pdPASS) {

         return 0;
      }
      break;
   case 1:
      if (xTaskCreate(vMissionTask1, (signed char *)"MISSION TASK 1",
            configMINIMAL_STACK_SIZE, NULL, configMAX_PRIORITIES - 4,
            &missionTask1Handle) != pdPASS) {
         return 0;
      }
      break;
   case 2:
      if (xTaskCreate(vMissionTask2, (signed char *)"MISSION TASK 2",
            configMINIMAL_STACK_SIZE, NULL, configMAX_PRIORITIES - 4,
            &missionTask2Handle) != pdPASS) {
         return 0;
      }
      break;
   case 3:
      if (xTaskCreate(vMissionTask3, (signed char *)"MISSION TASK 3",
            configMINIMAL_STACK_SIZE, NULL, configMAX_PRIORITIES - 4,
            &missionTask3Handle) != pdPASS) {
         return 0;
      }
      break;
   case 4:
      if (xTaskCreate(vMissionTask4, (signed char *)"MISSION TASK 4",
            configMINIMAL_STACK_SIZE, NULL, configMAX_PRIORITIES - 4,
            &missionTask4Handle) != pdPASS) {
         return 0;
      }
      break;
   default:
      break;
   }


   }

   return 1;
}
/*-----------------------------------------------------------*/
```

```
uint8_t stopTask(
      uint8_t taskId) {
   switch (taskId) {
   case 0:
      vTaskDelete(
            missionTask0Handle);
      break;
   case 1:
      vTaskDelete(
            missionTask1Handle);
      break;
   case 2:
      vTaskDelete(
            missionTask2Handle);
      break;
   case 3:
      vTaskDelete(
            missionTask3Handle);
      break;
   case 4:
      vTaskDelete(
            missionTask4Handle);
      break;
   default:
      break;

   }
   return 1;
}
/*-----------------------------------------------------------*/

/*-----------------------------------------------------------*/
// Attitude Measurement Task
void vMissionTask0(
      void *pvParameters) {

   uint8_t taskId = 0;
   int i;
   uint32_t sumState;

   portTickType xLastWakeTime;
   const portTickType xDelay =
         1000
               / portTICK_RATE_MS;
   xLastWakeTime =
         xTaskGetTickCount();

   for (;;) {
      /* Delay for a task-specific period */

      vTaskDelayUntil(
            &xLastWakeTime,
            xDelay);
      vParTestToggleLED(LED1);
      //  state updated values
      sumState = 0;
      for (i = 0;
            i
                  < mtState[taskId].numStateBytes;
            i++) {
         (mtState[taskId].stateData[i])++;
         sumState +=
               mtState[taskId].stateData[i];
      }

      // send message to AMFT Sender task: state updated

      xQueueSendToBack(
            xStateUpdatedQueue,
            &taskId,
            portMAX_DELAY);

   }
}
/*-----------------------------------------------------------*/
```

```
void vAmftReceiverTask(
      void *pvParameters) {
   struct taskListData rcvdTaskListData;
   uint8_t messageBytesProcessed =
         0;
   uint32_t processedIndex = 0;
   uint32_t processedCounter =
         0;

   rcvdTaskListData.numberOfTasks =
         0;
   int i;
   for (i = 0;
         i < MAX_MISSION_TASKS;
         i++) {
      rcvdTaskListData.taskId[i] =
            0;
   }

   for (;;) {
      if (xSemaphoreTake(amftDataReceivedSemaphore, portMAX_DELAY)
            == pdTRUE) {
         while (processedCounter
               < RxCounter) {
            // first byte should contain the message ID, indicating the message type
            if (messageBytesProcessed
                  == 0)
            {
               processedIndex =
                     processedCounter
                     % RXBUFFERSIZE;
               currentMessageBuffer[0] =
                     RxBuffer[processedIndex];
               if (currentMessageBuffer[0]
                                       == 'S') {
                  currentMessageType =
                        MSG_TYPE_SUM;
                  messageBytesProcessed++;
                  processedCounter++;
               } else if (currentMessageBuffer[0]
                                          == 'T') {

                  currentMessageType =



                        MSG_TYPE_TLM;
                  messageBytesProcessed++;
                  processedCounter++;
               }

               else {
                  currentMessageType =
                        MSG_TYPE_UNKNOWN;
                  //lcdDisplayMessage("Unknown message received               ");
                  //lcdDisplayError("ERROR: Unknown message received        ");
                  currentMessageLengthBytes =
                        0;
                  messageBytesProcessed =
                        0;
                  processedCounter++;
               }
            } else if (messageBytesProcessed
                  == 1) // obtain message length from second message byte
            {
               processedIndex =
                     processedCounter
                     % RXBUFFERSIZE;
               currentMessageBuffer[1] =
                     RxBuffer[processedIndex];
               if (currentMessageType
                     == MSG_TYPE_TLM) {
                  currentMessageLengthBytes =
                        currentMessageBuffer[1]
                                          + 2;
```

```
                } else if (currentMessageType
                      == MSG_TYPE_SUM) {
                   currentMessageLengthBytes =
                         mtState[currentMessageBuffer[1]].numStateBytes
                         + 2;
                }

                messageBytesProcessed++;
                processedCounter++;
             }
             // if the entire message has been received in the buffer
             else if (messageBytesProcessed
                   < currentMessageLengthBytes) {
                //vParTestToggleLED(LED2);
                processedIndex =
                      processedCounter
                      % RXBUFFERSIZE;
                currentMessageBuffer[messageBytesProcessed] =
                      RxBuffer[processedIndex];
                messageBytesProcessed++;
                processedCounter++;
             }

             // entire message is available - process the message
             if ((currentMessageType
                   != MSG_TYPE_UNKNOWN)
                   && (messageBytesProcessed
                         == currentMessageLengthBytes)) {
                if (currentMessageType
                      == MSG_TYPE_SUM) {
                   // lcdDisplayMessage("State Update Message received          ");
                   taskId =
                         currentMessageBuffer[1];
                   storeStateData(
                         taskId,
                         currentMessageBuffer);
                   messageBytesProcessed =
                         0;
                   currentMessageLengthBytes =
                         0;
                   currentMessageType =
                         MSG_TYPE_UNKNOWN;

                   // complete Task List Message received - send to Mission Task Manager
                } else if (currentMessageType == MSG_TYPE_TLM)
                {
                   // lcdDisplayMessage("Task List Message received");
                   for (i = 0;i< MAX_MISSION_TASKS;i++) {
                      rcvdTaskListData.taskId[i] =
                            0;
                      //  rcvdTaskListData.stateData[i] = 0;
                   }
                   rcvdTaskListData.numberOfTasks =
                         currentMessageBuffer[1];
                   for (i = 2;i< currentMessageLengthBytes;i++) {
                      rcvdTaskListData.taskId[i- 2] =currentMessageBuffer[i];
                   }
                   messageBytesProcessed =0;
                   currentMessageLengthBytes =0;
                   currentMessageType = MSG_TYPE_UNKNOWN;
                   xQueueSendToBack(xTaskListQueue,(void*)&rcvdTaskListData,
                         portMAX_DELAY);
                }
                for (i = 0;i< MAX_STATE_SIZE_BYTES+ 2;i++) {
                   currentMessageBuffer[i] =0;
                }
             }
          }
       }
    }

}
/*-----------------------------------------------------------*/
```

```
/*-----------------------------------------------------------*/
void vAmftSenderTask(void *pvParameters)
{
   uint8_t taskId;
   int i;

   for (;;)
   {

      if(xQueueReceive(xStateUpdatedQueue, &taskId, portMAX_DELAY) == pdTRUE)
      {
         TxBuffer[0] = 'S';
         TxBuffer[1] = taskId;


         for (i = 0; i < mtState[taskId].numStateBytes; i++)
         {
            TxBuffer[2+i] = mtState[taskId].stateData[i];
         }
         NbrOfDataToTransfer = mtState[taskId].numStateBytes + 2;

          vTaskDelay(1);
          sendData_uart0( &TxBuffer[0], NbrOfDataToTransfer);

      }
   }
}
/*-----------------------------------------------------------*/
```

E.2 AMFT Software Top Level Design

```
int main(void) {
   /* Configure the hardware ready to run the test. */
   prvSetupHardware();
   microblaze_disable_dcache();

   //Display_Init();
   /* creates all the tasks and makes them ready.  */
   vInitialiseAMFTTasks();
   xTaskCreate(dummyTask,
         (signed char *)"Dummy Task",
         configMINIMAL_STACK_SIZE,
         NULL,
         (tskIDLE_PRIORITY + 1),
         (xTaskHandle*)NULL);
   /* starts the kernel scheduler      */
   xil_printf("scheduler");
   vTaskStartScheduler();

   /* Should never reach here! If you do then there was not enough heap
    available for the idle task to be created. */

   for (;;)
      ;

}


/*-----------------------------------------------------------*/
static void prvSetupHardware(
      void) {
   /* Setup GPIOs, UART and CAN Bus */

   prvSetGpioHardware();
   vInitialiseSerialPort_0();
   vInitialiseSerialPort_1();
   vInitialiseCAN_0();

}
/*-----------------------------------------------------------*/
```

```
void vInitialiseAMFTTasks(
      void) {
   /* Initialise the Obc Tasks*/
   //modify this function to change the tasks of the OBC
   // vInitialiseObcTasks();
   /* Initialise the task list structure  */
   vInitialiseTaskStateData();

   /* This queue is used to communicate between the amftcommtask and task manager
    the AmftCommTask through this queue tells the task manager which node to activate
    or deactivate the data field of the xStatusMessage structure is
    unsigned char id
    unsigned char state
    we just need space for one item on the queue since every slot at most there will
    be only one item */
   Queue_ForActivateDeactivate =
         xQueueCreate( 10, sizeof( xStatusMessage ));
   vQueueAddToRegistry(
         Queue_ForActivateDeactivate,
         (signed char*) "ActivateDeactivate"); // for kernel aware debugging purposes only

   /* This queue is used to communicate between the task manager and the
    * Processing Unit sender task When there is an updation, the task manager
    * calculates how many nodes are active, prepares the task list for its
    * own Processing Unit and then signals the Processing Unit sender task to
    * send the task list to the Processing Unit via UART0 There are a totally
    * of five tasks which can be assigned to a single OBC if the other two nodes
    *  are to fail Since the task list is variable, task manager inserts a
    *  End Of Frame after the task identifier which will be used
    by the OBC sender to identify the number of tasks
    .:   Max. there can be 7 Tasks + EOF, so a total of 8 items per slot                    */
   Queue_CurrentTaskList =
         xQueueCreate(18, sizeof(unsigned portCHAR));
   vQueueAddToRegistry(
         Queue_CurrentTaskList,
         (signed char*) "CurrentTaskList"); // for kernel aware debugging purposes only

   Queue_FDIR =
         xQueueCreate(12, sizeof(unsigned portCHAR));
   vQueueAddToRegistry(
         Queue_FDIR,
         (signed char*) "Queue_FDIR"); // for kernel aware debugging purposes only


   xTaskCreate(vFDIR,
         ( signed char * ) "FDIR",
         FDIRTaskSTACK_SIZE,
         (void *)NULL,
         FDIR_PRIORITY, NULL);
   xTaskCreate(vAmftCommsTask,
         ( signed char * ) "AMFTComm",
         AMFTCommTaskSTACK_SIZE,
         (void *)NULL,
         AMFTComm_PRIORITY,
         NULL);
   xTaskCreate(
         vTaskAllocationManager,
         ( signed char * ) "TaskManager",
         TaskManagerSTACK_SIZE,
         (void *)NULL,
         TaskManager_PRIORITY,
         NULL);

   /* create tasks that handle communication between the AMFTs */
   vInitialiseAmftCommServices();

   /* create tasks that handle communication between the AMFT and OBC   */
   vInitialiseObcCommServices();

   /*creates the semaphores and the queues used by the above two communication services   */
   vInitialiseCommUtilities();
}
```

```
/* Creates the AMFT Tasks {Amft Receive Task, Amft Sender Task}*/
void vInitialiseAmftCommServices(
      void) {
   xTaskCreate(
         vSendAmftMessage,
         ( signed char * ) "SendCanAmftTask",
         ObcSenderTaskSTACK_SIZE,
         NULL,
         ObcSender_PRIORITY,
         NULL);
   xTaskCreate(
         vReceiveCanMessage,
         ( signed char * ) "ReceiveCanTask",
         ObcReceiverTaskSTACK_SIZE,
         NULL,
         ObcReceiver_PRIORITY,
         NULL);
}

/* Creates the semaphores and queues used for communication*/
void vInitialiseCommUtilities(
      void) {
   vSemaphoreCreateBinary(
         Semaphore_Receive);
   xSemaphoreTake(
         Semaphore_Receive,
         0);
   vSemaphoreCreateBinary(
         Semaphore_SendTaskList);
   xSemaphoreTake(
         Semaphore_SendTaskList,
         0);



   /*these queues are used by the AmftCommTask for sending and receive the "codes" of the messages*/
   Queue_TxAmftMessageType =
         xQueueCreate( 100, sizeof( unsigned char ));
   vQueueAddToRegistry(
         Queue_TxAmftMessageType,
         (signed char *) "TxAmftMessageCode"); // for kernel aware debugging purposes only
   Queue_RxAmftMessageCode =
         xQueueCreate( 100, sizeof( struct SlotMessage ));
   vQueueAddToRegistry(
         Queue_RxAmftMessageCode,
         (signed char *) "RxAmftMessageCode"); // for kernel aware debugging purposes only
}

/* Initialises the task list structures*/
void vInitialiseTaskStateData(
      void) {
   unsigned char taskId;
   short int i;
   for (taskId = 0;
         taskId
               < TOTAL_MISSION_TASKS;
         ++taskId) {
      for (i = 0;
            i
                  < mtState[taskId].numStateBytes;
            i++) {
         mtState[taskId].stateData[i] =
               0;
      }
   }
}

/*                                                    */
```

```
portTASK_FUNCTION( vAmftCommsTask, pvParameters ) {
   short int
         timeforcommunicationslot,
         current_slot_node_id =
               0;
   unsigned char
         received_node_id,
         received_message_id;
   xStatusMessage message;
   xSlotMessage slot_message;
   portTickType StartTime;

   const unsigned char heartbeat_id =
         MSG_ID_HEARTBEAT;
   const unsigned char fault_id =
         MSG_ID_FAULT;
   const unsigned char sum_id =
         MSG_ID_SUM;

   if (xQueueReceive( Queue_RxAmftMessageCode, &slot_message,
         (COMMS_SLOT_DURATION * 3) ) == pdTRUE) {
      //received a message
      StartTime =
            xTaskGetTickCount();
      current_slot_node_id =
            slot_message.node_id;
   } else {
      //didn't receive a message
      StartTime =
            xTaskGetTickCount();
      current_slot_node_id =
            THIS_NODE_ID;
      xQueueSend(
            Queue_TxAmftMessageType,
            &heartbeat_id, 0);
   }
```

```
/* Either a heartbeat has just been received, or this node
   has just sent its heartbeat.
   Either way, this defines the start of a comms. slot. */

timeforcommunicationslot =
      COMMS_SLOT_DURATION;

vTaskDelayUntil(&StartTime,
      timeforcommunicationslot); // wait for the next slot
for (;;) {
   current_slot_node_id++;
   if (current_slot_node_id
         > NUM_NODES) {
      current_slot_node_id =
            1;
   }

   if (current_slot_node_id
         == THIS_NODE_ID) // this node's turn to transmit
   {
      /* If there is a fault then indicate to the Amft Sender task
       to send a fault. Mark the current node as inactive if it is active*/
      if (ucFaultStatus
            == FAULT_DETECTED) {
         xQueueSend(
               Queue_TxAmftMessageType,
               &fault_id,
               0);
         message.state =
               NODE_INACTIVE;
         message.id =
               THIS_NODE_ID;
         if (NodeTable[THIS_NODE_ID
               - 1][2]
               == NODE_ACTIVE ) {
            xQueueSend(
                  Queue_ForActivateDeactivate,
                  &message,
                  0);
         }
      }
```

```
    /* If there is no fault, indicate the Amft Sender Task to send the SUMs for local tasks
     The SUM message will be preceeded by a HBM to indicate the start of the node's time slot */
    else {
       xQueueSend(
             Queue_TxAmftMessageType,
             &sum_id, 0);
       message.state =
             NODE_ACTIVE;
       message.id =
             THIS_NODE_ID;
       if (NodeTable[THIS_NODE_ID
             - 1][2]
             == NODE_INACTIVE ) {
          xQueueSend(
                Queue_ForActivateDeactivate,
                &message,
                0);
       }
    }
 }
 /* Reaching here would mean that the time slot for the current node has not arrived
  Since the time slots for each nodes are fixed, we know which node is going to communicate in each time slot
  the task waits for each time slot untill it receives a message within it  */
 else {
    /* If a message is received, the Amft receiver task will load the "code" of the message onto this queue
     `As described at the top of this task, appropriate action will be taken              */
    if (xQueueReceive( Queue_RxAmftMessageCode, &slot_message, (timeforcommunicationslot - 10) )
          == pdTRUE) {
       received_node_id =
             slot_message.node_id;
       received_message_id =
             slot_message.message_id;

       if (received_node_id
             != current_slot_node_id) {
          // WARNING: HBM from unexpected node - resynchronising
          current_slot_node_id =
                received_node_id;
       }

       if (received_node_id
             < THIS_NODE_ID) // resynchronise timing
       {


           StartTime =
                 xTaskGetTickCount();
           current_slot_node_id =
                 received_node_id;
        }

        if ((received_message_id
              == heartbeat_id)
              && (NodeTable[received_node_id
                    - 1][2]
                    == NODE_INACTIVE )) {
           message.state =
                 NODE_ACTIVE;
           message.id =
                 received_node_id;
           xQueueSend(
                 Queue_ForActivateDeactivate,
                 &message,
                 0);
        } else if ((received_message_id
              == fault_id)
              && (NodeTable[received_node_id
                    - 1][2]
                    == NODE_ACTIVE )) {
           message.state =
                 NODE_INACTIVE;
           message.id =
                 received_node_id;
           xQueueSend(
                 Queue_ForActivateDeactivate,
                 &message,
                 0);
        }
     } else //no message during the node's slot
     {
        message.state =
              NODE_INACTIVE;
        message.id =
              current_slot_node_id;
        if (NodeTable[current_slot_node_id
              - 1][2]
              == NODE_ACTIVE ) {
```

```
            message.id =
                  current_slot_node_id;
            if (NodeTable[current_slot_node_id
                  - 1][2]
                  == NODE_ACTIVE ) {
               xQueueSend(
                     Queue_ForActivateDeactivate,
                     &message,
                     0);
            }
         }
      }

      /*Since there are many branching within the task, the TaskDelayUntil is used to keep it cyclic */
      vTaskDelayUntil(
            &StartTime,
            timeforcommunicationslot);
   }
}



void vReceiveCanMessage(void *pvParameters)
{
  xCanMessage message;
  xSlotMessage slot_message;

  for(;;)
  {
    if(xQueueReceive( Queue_RxCanMessage, &message, portMAX_DELAY))
    {

      if (currentStateBytesReceived < currentStateLength)   // more state data to be received
      {
        vStoreSum(message);
      }
      else
      {


        currentStateBytesReceived = 0;
        currentStateLength = 0;

        slot_message.node_id  = message.CAN_DATA1;
        //the higher bits will get truncated and only the 8 LSB bits will be assigned
        slot_message.message_id = message.CAN_DATA1 >> 8;
        switch(slot_message.message_id)
        {
        case MSG_ID_HEARTBEAT :
        case MSG_ID_FAULT     :   xQueueSend( Queue_RxAmftMessageCode, &slot_message, 0 );
        break;
        case MSG_ID_SUM    :   currentStateTaskId = message.CAN_DATA1 >> 16;
        currentStateLength = mtState[currentStateTaskId].numStateBytes;
        break;

        }
      }
    }
  }
}


portTASK_FUNCTION( vSendAmftMessage, pvParameters ) {
   unsigned char taskId;
   unsigned char message type =
         0;
   xCanMessage message;
   message.CAN_DATA1 = 0;
   message.CAN_DATA2 = 0;

   for (;;) {
      if (xQueueReceive(Queue_TxAmftMessageType, &message_type, portMAX_DELAY)
            == pdTRUE) {
         switch (message_type) {
         case MSG_ID_HEARTBEAT:
         case MSG_ID_FAULT:
            message.CAN_DATA1 =
                  THIS_NODE_ID
                        | (message_type
                              << 8);//fault message
```

```
            message.CAN_DATA2 =
                  0;
            xSendCanMessage(
                  message);
            break;

         case MSG_ID_SUM:
            message.CAN_DATA1 =
                  THIS_NODE_ID
                        | (MSG_ID_HEARTBEAT
                              << 8);//heart beat
            message.CAN_DATA2 =
                  0;
            xSendCanMessage(
                  message);
            vTaskDelay(1);
            // xQueueSend( Queue_TxCanMessage, &message, 0);
            for (taskId = 0;
                  taskId
                        < TOTAL_MISSION_TASKS;
                  ++taskId)//send sum
                  {
               if (currentTaskAllocations[taskId]
                     == THIS_NODE_ID) // if the task is running locally, send the SUM for the task
               {
                  sendSumViaCan(
                        taskId);
               }
            }
            break;
         }    //switch

      }    //if
   }    //for
}    //fn
```

```
portTASK_FUNCTION( vTaskAllocationManager, pvParameters )
{
   short int mask, i;
        unsigned char taskId;
   xStatusMessage UpdateMessage;
    //    const unsigned char EndOfFrame = MSG_ID_EOF;
        unsigned char numberOfLocalTasks;

   for(;;)
   {
      //message received for updating the task list
      if(xQueueReceive( Queue_ForActivateDeactivate, &(UpdateMessage), portMAX_DELAY ))
      {
           vParTestToggleLED(LED2);
        for(i = 0; i < 3; i++)              //activate node stored in operand
                        {
          if(NodeTable[i][0] == UpdateMessage.id)
                          {
                            NodeTable[i][2] = UpdateMessage.state;
                          }
                        }
        mask = 0;
        for(i = 0; i < 3; ++i)
           mask |= (NodeTable[i][2] << i);
        mask &= 0x0f;        //mask off except the first four

                        numberOfLocalTasks = 0;
           for(taskId = 0; taskId < TOTAL_MISSION_TASKS; ++taskId)
                        {
          if(taskAllocationTables[mask][taskId] == THIS_NODE_ID)
                          {
                            numberOfLocalTasks++;
                          }
                        }
                        for(taskId = 0; taskId < TOTAL_MISSION_TASKS; ++taskId)
                        {
                          currentTaskAllocations[taskId] = taskAllocationTables[mask][taskId];
                        }
                        xQueueSend(Queue_CurrentTaskList, &numberOfLocalTasks, 0);
                        for(taskId = 0; taskId < TOTAL_MISSION_TASKS; ++taskId)
                        {
                          if(currentTaskAllocations[taskId] == THIS_NODE_ID)
                          {

            xQueueSend( Queue_CurrentTaskList, &taskId, 0 );
                          }
                        }
        xSemaphoreGive(Semaphore_SendTaskList);
      }
   }
}
```

E.3 AOCS Telemetry List

Unit	TM Parameter	Qty	Minimum Bits
Magnetometer (MGM)	Magnetic Field	3	36
	Compensation/Bias	3	24
	Scale Factors	3	24
	Alignment Matrices	3	24
Sun Sensor (SS)	Pixel Value	4	48
	In-FOV flag - SS	4	4
	Value Offset-SS	4	24

	Exposure Time	4	12
	Scale Factor-SS	4	24
	SS Open Threshold	4	32
	Sun Vector - SS	4	56
Current Sensor	Current Value - CS	8	56
	Value Offset-CS	8	32
	Scale Factor-CS	8	32
Tempertaure Sensor	Temperature Value - TS	8	56
	Value Offset-TS	8	32
	Scale Factor-TS	8	32
Rate Gyro	Rate	4	64
	Value offset	4	64
	Scale Factor	4	64
	Gyro Speed	4	32
GPS	GPS Time	1	32
	GPS Date	1	16
	Position	3	63
	Velocity	3	54
	Valid Flag	1	1
Torquerods (MTR)	MTR Magnitude of Signal applied	3	30
	MTR Direction (+ve or -ve)	3	3
	MTR Mode	3	9
Thruster	Accumulated Firing Duration - THR	12	192
AOCS Computer (ACC)	Orbit Position	3	63
	Orbit Velocity	3	54
	Time	1	32
	Quaternion	4	80
	Bias Rate	3	48
	TC Download	-	-
	RAM Download	-	-
	Reference Bias	4	64
	Mode Status	1	4
	System Software Message	4	16
	Controller Output	3	48
	Sub-Mode Status	1	4
	Controller Gains/Paramters	18	576
	AOCS Mode Attitude Threshold Values	8	80
	Estimated Angular Rates	3	48
	Selection of Sinking Thrusters	1	4

	Thruster Firing Configuration	1	4
	Thrusters fire (with count and time-tagging)	36	576
	Sensor Calibration Parameters	12	96
	Sun Presnece	2	6
	Gyro Integral Angle	4	80
	AOCS Health State	48	48
	Memory of System Fault/Alarm State	1	8
	Faults and anomaly flags	40	40
Total		**335**	**3151**

Bibliography

[1] A. Helmerich, *et al.*, "Study of Worldwide Trends and R&D Programmes in Embedded Systems.," Fast GmbH, Munich Germany, 2005.

[2] A. Sehmi, "On distributed embedded systems," *J. ACSIJ*, vol. 2, Issue 1, No. 2, Jan. 2013.

[3] H. Kopetz, *et al.*, "Distributed fault-tolerant real-time systems: the Mars approach," *IEEE Micro*, vol. 9, pp. 25-40, 1989.

[4] A. S. Tanenbaum and M. v. Steen, "Fault Tolerance," in *Distributed systems : principles and paradigms*, 2nd ed Upper Saddle RIiver, N.J.: Pearson Prentice Hall, 2007.

[5] "Distributed systems," in *A Review of Ada Tasking*. vol. 262, A. Burns, *et al.*, Eds., ed: Springer Berlin Heidelberg, 1987, pp. 63-72.

[6] C. Georgiou and A. A. Shvartsman, *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*: Springer, 2008.

[7] C. Georgiou and A. A. Shvartsman, *Cooperative Task-Oriented Computing Algorithm and Complexity*, 1st ed.: Morgan & Claypool, 2011.

[8] H. J. Kramer. (2014). *Copernicus: Sentinel-2 — The Optical Imaging Mission for Land Services*. Available: https://directory.eoportal.org/web/eoportal/satellite-missions/c-missions/copernicus-sentinel-2

[9] G. Klančar, *et al.*, "Image-Based Attitude Control of a Remote Sensing Satellite," *J. Intell Robot Syst*, vol. 66, pp. 343-357, May 2012.

[10] J. D. Ruiz. (2013). *Overcoming the Embedded CPU Performance Wall*. Available: http://www.embedded.com/design/mcus-processors-and-socs/4405280/Overcoming-the-embedded-CPU-performance-wall-

[11] Y. Xie and B. Zhu, "Architecture design of spaceborne SAR imaging processing system," in *Proc. Int. Conf. of Signal Processing*, 2010, pp. 2283-2286.

[12] Gutierrez-Nava, *et al.*, "TOPMEX-9 DISTRIBUTED SAR MISSION EMPLOYING NANOSATELLITE CLUSTER," presented at the 63rd Int. Astronautical Congr., Naples, Italy., 2012.

[13] S. Buckreuss and M. Zink, "The missions TerraSAR-X and TanDEM-X: Status, challenges, future perspectives," in *General Assembly and Scientific Symp., XXXth URSI*, 2011, pp. 1-1.

[14] M. Zink and A. Moreira, "TanDEM-X mission: Overview, challenges and status," in *Proc. International Geoscience and Remote Sensing Symp.*, 2013, pp. 1885-1888.

[15] J. Kichun, *et al.*, "Development of Autonomous Car-Part I: Distributed System Architecture and Development Process," *IEEE Trans. Ind. Electron.*, vol. 61, pp. 7131-7140, 2014.

[16] C. Villalpando, *et al.*, "Reliable multicore processors for NASA space missions," in *Proc. Aerospace Conf.*, Big Sky, MT, USA, 2011, pp. 1-12.

[17] M. Hudaverdi and I. Baylakoglu, "Space environment and evaluation for RASAT," in *Proc. Recent Advances in Space Technologies, 5th Int. Conf.*, 2011, pp. 926-931.

[18] K. L. Bedingfield, *et al.*, "Spacecraft system failures and anomalies atrributed to the natural space environment," Marshall Space Flight Center, Alabama,USA, 1390, August 1996.

[19] DONALD J. KESSLER, *et al.*, "Limiting future collision to Spacecraft: An assessment of NASA's meteoroid and space debris programs," National Research Council of the National Academies, Washington DC., USA, 2011.

[20] U. Nations, "Technical report on space debris," New York, USA, ISBN: 92-1-100813-1, 1999.

[21] C. C. H. Stokes, *et al.*, "A detailed impact risk assessment of two low earth orbiting satellites," presented at the 63rd Int. Astronautical Congr., Naples, Italy, 2012.

[22] L. L. F. Lansing, A. Walton, G. Bothwell, K. Bhasin, and G. Prescott, "Needs for communications and onboard processing in the vision era," presented at the Int. Geosciences and Remote Sensing Symp., 2002.

[23] M. Graziano, "Overview of Distributed Missions," in *Distributed Space Missions for Earth System Monitoring*. vol. 31, M. D'Errico, Ed., ed: Springer New York, 2013, pp. 375-386.

[24] A. S. Tanenbaum and M. v. Steen, *Distributed systems : principles and paradigms*, 2nd ed. Upper Saddle RIiver, N.J.: Pearson Prentice Hall, 2007.

[25] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 1998 ed.: Kluwer Academic Publishers. , 1997.

[26] P. Thambidurai and P. You-keun, "Interactive consistency with multiple failure modes," in *Proc. Reliable Distributed Systems, Seventh Symp.*, Columbus, OH, 1988, pp. 93-100.

[27] R. W. Butler, "A primer on architectural level fault tolerance " Langley Research Center, Hampton, Virginia, 2008.

[28] M. L. Shooman, *Reliability of Computer Systems and Networks; Fault Tolerance, Analysis and Design*. New York: John Wiley & Sons, Inc., 2002.

[29] H. Kopetz, *et al.*, "Tolerating transient faults in MARS," in *Dig. of papers Fault-Tolerant Computing, 20th Int. Symp.*, Newcastle Upon Tyne, UK, 1990, pp. 466-473.

[30] C. Liming and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Proc. Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth Int. Symp.*, 1995, p. 113.

[31] J. Aidemark, *et al.*, "Experimental evaluation of time-redundant execution for a brake-by-wire application," in *Proc. Dependable Systems and Networks Conf.*, 2002, pp. 210-215.

[32] A. Avizienis, *et al.*, "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Trans. Comput.*, vol. C-20, pp. 1312-1321, 1971.

[33] A. Avizienis, "Design of fault-tolerant computers," in *Proc. Fall Joint Computer Conf.*, Anaheim, California, 1967, pp. 733-743.

[34] J. F. Wakerly, "Transient Failures in Triple Modular Redundancy Systems with Sequential Modules," *IEEE Trans. Comput.*, vol. C-24, pp. 570-573, 1975.

[35] S. D'Angelo, *et al.*, "Transient and permanent fault diagnosis for FPGA-based TMR systems," in *Proc. Defect and Fault Tolerance in VLSI Systems, Int. Symp.*, Albuquerque, NM 1999, pp. 330-338.

[36] R. E. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *J. IBM Research and Development*, vol. 6, pp. 200-209, 1962.

[37] A. L. Hopkins, Jr., *et al.*, "FTMP- A highly reliable fault-tolerant multiprocess for aircraft," *Proc. IEEE*, vol. 66, pp. 1221-1239, 1978.

[38] J. H. Wensley, *et al.*, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proc. IEEE*, vol. 66, pp. 1240-1255, 1978.

[39] S. Y. Yu and E. J. McCluskey, "On-line testing and recovery in TMR systems for real-time applications," in *Proc. of Int.Test Conf.*, 2001, pp. 240-249.

[40] J. R. Sklaroff, "Redundancy Management Technique for Space Shuttle Computers," *J. IBM Research and Development*, vol. 20, pp. 20-28, 1976.

[41] H. Pham, "Optimal design of hybrid fault-tolerant computer systems," *J. Mathematical and Computer Modelling*, vol. 16, pp. 29-33, 1992.

[42] W. Xinsheng and S. Hanxu, "Fault tolerance design on onboard computer using COTS components," in *Proc. Systems and Control in Aerospace and Astronautics, 1st Int. Symp.*, 2006, pp. 3 pp.-1224.

[43] P. K. Samudrala, *et al.*, "Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 51, pp. 2957-2969, 2004.

[44] B. Pratt, *et al.*, "Improving FPGA Design Robustness with Partial TMR," in *Proc. Reliability Physics, 44th Annu. Symp.*, 2006, pp. 226-232.

[45] F. L. Kastensmidt, *et al.*, "On the optimal design of triple modular redundancy logic for SRAM-based FPGAs," in *Proc. Design, Automation and Test in Europe*, 2005, pp. 1290-1295 Vol. 2.

[46] F. Lima, *et al.*, "Designing fault tolerant systems into SRAM-based FPGAs," in *Proc. Design Automation Conf.*, 2003, pp. 650-655.

[47] R. Guerraoui and A. Schiper, "Fault-Tolerance by replication in Distributed systems " in *Proc. Reliable Software Technologies, Ada-Europe*, 1996, pp. 38-57.

[48] A. Girault, *et al.*, "An Active Replication Scheme That Tolerates Failures in Distributed Embedded Real-Time Systems," in *Design Methods and Applications for Distributed Embedded Systems*. vol. 150, B. Kleinjohann, *et al.*, Eds., ed: Springer US, 2004, pp. 83-92.

[49] M. Pease, *et al.*, "Reaching Agreement in the Presence of Faults," *J. ACM*, vol. 27, pp. 228-234, 1980.

[50] L. Lamport, *et al.*, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382-401, 1982.

[51] U. Schmid, *et al.*, "Formally verified Byzantine agreement in presence of link faults," in *Proc. Distributed Computing Systems conf.*, 2002, pp. 608-616.

[52] S. Hin-Sing, *et al.*, "Byzantine agreement in the presence of mixed faults on processors and links," *IEEE Trans. Parallel and Distrib. Syste.*, vol. 9, pp. 335-345, 1998.

[53] F. B. Schneider and Z. Lidong, "Implementing trustworthy services using replicated state machines," *IEEE J. Security & Privacy* vol. 3, pp. 34-43, 2005.

[54] M. Chereque, *et al.*, "Active replication in Delta-4," in *Proc. Fault-Tolerant Computing Twenty-Second Int. Symp.*, 1992, pp. 28-37.

[55] C. Marchetti, *et al.*, "Fully distributed three-tier active software replication," *IEEE Trans. Parallel and Distrib. Syste.* , vol. 17, pp. 633-645, 2006.

[56] R. Baldoni, *et al.*, "Asynchronous active replication in three-tier distributed systems," in *Proc. Dependable Computing, Pacific Rim Int. Symp.*, 2002, pp. 19-26.

[57] R. Baldoni, *et al.*, "Active software replication through a three-tier approach," in *Proc. Reliable Distributed Systems, 21st IEEE Symp.*, 2002, pp. 109-118.

[58] D. Powell, "Distributed Fault-Tolerance," in *Delta-4: A Generic Architecture for Dependable Distributed Computing*. vol. 1, D. Powell, Ed., ed: Springer Berlin Heidelberg, 1991, pp. 89-124.

[59] R. Guerraoui and A. Schiper, "Fault-tolerance by replication in distributed systems," in *Reliable Software Technologies — Ada-Europe '96*. vol. 1088, A. Strohmeier, Ed., ed: Springer Berlin Heidelberg, 1996, pp. 38-57.

[60] X. Defago, *et al.*, "Semi-passive replication," in *Proc. Reliable Distributed Systems, Seventeenth IEEE Symp.*, 1998, pp. 43-50.

[61] N. Budhiraja and K. Marzullo, "Tradeoffs in implementing primary-backup protocols," in *Proc. Parallel and Distributed Processing, Seventh IEEE Symp.*, 1995, pp. 280-288.

[62] Z. Hengming and F. Jahanian, "A real-time primary-backup replication service," *IEEE Trans. Parallel and Distrib. Syste.*, vol. 10, pp. 533-548, 1999.

[63] A. M. Deplanche, *et al.*, "Implementing a semi-active replication strategy in CHORUS/ClassiX, a distributed real-time executive," in *Proc. Reliable Distributed Systems, 18th IEEE Symp.*, 1999, pp. 90-101.

[64] X. Défago and A. Schiper, "Semi-passive replication and Lazy Consensus," *J. Parallel Distrib. Comp.*, vol. 64, pp. 1380-1398, 2004.

[65] X. Défago and A. Schiper, "Specification of replication techniques, semi-passive replication, and lazy consensus. ," Japan Advanced Institute of Science and Technology, Ishikawa, Japan KS-RR-2002-001, 2002.

[66] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, pp. 225-267, 1996.

[67] M. Hecht, *et al.*, "A distributed fault tolerant architecture for nuclear reactor and other critical process control applications," in *Proc. Twenty-First Int. Fault-Tolerant Computing Symp.*, 1991, pp. 462-498.

[68] D. Nguyen and L. Dar-Biau, "Recovery blocks in real-time distributed systems," in *Proc. Reliability and Maintainability Annu. Symp.* , 1998, pp. 149-154.

[69] V. Izosimov, *et al.*, "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems," in *Proc. Design, Automation and Test in Europe*, 2005, pp. 864-869 Vol. 2.

[70] M. N. Lovellette, *et al.*, "Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed," in *Proc. Aerospace IEEE Conf.* , 2002, pp. 5-2109-5-2119 vol.5.

[71] P. Subramanyan, *et al.*, "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," in *Proc. Design, Automation & Test in Europe Conf.*, 2010, pp. 1572-1577.

[72] H. Kopetz, *et al.*, "Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System," in *Dependable Computing for Critical Applications*. vol. 4, A. Avižienis and J.-C. Laprie, Eds., ed: Springer Vienna, 1991, pp. 411-429.

[73] K. H. Kim and E. Shokri, "Minimal-delay decentralized maintenance of processor-group membership in TDMA-bus LAN systems," in *Proc. Distributed Computing Systems, 13th Int. Conf.*, 1993, pp. 410-419.

[74] K. H. Kim and C. Subbaraman, "Dynamic configuration management in reliable distributed real-time information systems," *IEEE Trans. Knowl. Data Eng.*, vol. 11, pp. 239-254, 1999.

[75] G. J. Nutt, "Tutorial: computer system monitors," *IEEE Computer*, vol. 8, pp. 51-61, 1975.

[76] J. J. P. Tsai, *et al.*, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 897-916, 1990.

[77] P. Calingaert, "System performance evaluation: survey and appraisal," *Commun. ACM*, vol. 10, pp. 12-18, 1967.

[78] J. Henry Lucas, "Performance Evaluation and Monitoring," *ACM Comput. Surv.*, vol. 3, pp. 79-91, 1971.

[79] J. P. Calvez and O. Pasquier, "Performance Monitoring and Assessment of Embedded HW/SW Systems," *J. Design Automation for Embedded Systems,* vol. 3, pp. 5-22, 1998/01/01 1998.

[80] M. El Shobaki and L. Lindh, "A hardware and software monitor for high-level system-on-chip verification," in *Proc. Quality Electronic Design, Int. Symp.*, 2001, pp. 56-61.

[81] H. Thane, "Monitoring, Testing and Debugging of distributed real-time systems," Kungliga Techkniska Hogkolan, Stockholm, 2000.

[82] D. A. Rennels, "Architectures for fault-tolerant spacecraft computers," *Proc. IEEE,* vol. 66, pp. 1255-1268, 1978.

[83] A. H. Bhagyashree, *et al.*, "A hierarchical fault detection and recovery in a computational grid using watchdog timers," in *Proc. Communication and Computational Intelligence Conf.*, 2010, pp. 467-471.

[84] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Trans. Comput.*, vol. 37, pp. 160-174, 1988.

[85] M. Namjoo and E. J. HcCluskey, "Watchdog Processors and Capability Checking," in *Proc. Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth Int. Symp.*, 1995, pp. 94-97.

[86] D. S. Rosenblum, "Correction to 'A Practical Approach to Programming with Assertions'," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 265-265, 1995.

[87] B. Meyer, *Object Oriented Software Construction*: Prentice-Hall, 1998.

[88] D. Bartetzko, *et al.*, "Jass — Java with Assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, pp. 103-117, 2001.

[89] M. Boul and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*: Springer Publishing Company, Incorporated, 2008.

[90] G. Leavens, *et al.*, "How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification," in *Formal Methods for Components and Objects*. vol. 2852, F. Boer, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2003, pp. 262-284.

[91] C. Jeffery, *et al.*, "A lightweight architecture for program execution monitoring," *SIGPLAN Not.*, vol. 33, pp. 67-74, 1998.

[92] B. Bruegge, *et al.*, "A framework for dynamic program analyzers," *SIGPLAN Not.*, vol. 28, pp. 65-82, 1993.

[93] G. Lyle, *et al.*, "An end-to-end approach for the automatic derivation of application-aware error detectors," in *Proc. Dependable Systems & Networks, Int. Conf.*, 2009, pp. 584-589.

[94] U. Schiffel, *et al.*, "Software-Implemented Hardware Error Detection: Costs and Gains," in *Proc. Dependability, Third Int. Conf.*, 2010, pp. 51-57.

[95] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Des. of Integr. Circuits and Syst.* , vol. 24, pp. 88-99, 2005.

[96] K. M. Zick, *et al.*, "Silent Data Corruption and Embedded Processing With NASA's SpaceCube," *IEEE Embedded Syst. Lett.*, vol. 4, pp. 33-36, 2012.

[97] M.-L. Li, *et al.*, "Understanding the propagation of hard errors to software and implications for resilient system design," *SIGARCH Comput. Archit. News*, vol. 36, pp. 265-276, 2008.

[98] M. Li, *et al.*, "SWAT: An error resilient system," in *Proc. SELSE4*, 2008, pp. 8-13.

[99] M.-l. Li, *et al.*, "Towards a Software-Hardware Co-Designed Resilient System," presented at the 3rd Workshop on Silicon Errors in Logic-System Effects, 2007.

[100] G. Yalcin, *et al.*, "SymptomTM: Symptom-Based Error Detection and Recovery Using Hardware Transactional Memory," in *Proc. Parallel Architectures and Compilation Techniques, Int. Conf.*, 2011, pp. 199-200.

[101] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *IEEE Trans. Dependable Secure Comput.*, vol. 3, pp. 188-201, 2006.

[102] R. A. Maxion and K. M. C. Tan, "Anomaly detection in embedded systems," *IEEE Trans. Comput.*, vol. 51, pp. 108-120, 2002.

[103] L. Fei, *et al.*, "Argus: Online Statistical Bug Detection," in *Fundamental Approaches to Software Engineering*. vol. 3922, L. Baresi and R. Heckel, Eds., ed: Springer Berlin Heidelberg, 2006, pp. 308-323.

[104] N. Nakka, *et al.*, "An Architectural Framework for Detecting Process Hangs/Crashes," in *Dependable Computing - EDCC 5*. vol. 3463, M. Cin, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2005, pp. 103-121.

[105] H. Chunping, *et al.*, "Performance analysis of high-speed MIL-STD-1553 bus system using DMT technology," in *Proc. Computer Science & Education, 8th Int. Conf.*, Colombo 2013, pp. 533-536.

[106] M. G. Hegarty, "High performance 1553: a feasibility study," in *Proc. Digital Avionics Systems Conf.*, 2004, pp. 7.D.4-7.1-9 Vol.2.

[107] G. D. Racca, *et al.*, "SMART-1 mission description and development status," *J. Planetary and Space Science*, vol. 50, pp. 1323-1337, 2002.

[108] Y. Kobayashi, *et al.*, "Nano-JASMINE: a 10-kilogram satellite for space astrometry," in *Proc. SPIE 6265, Space Telescopes and Instrumentation I: Optical, Infrared, and Millimeter*, 2006, pp. 626544-626544-10.

[109] A. M. Woodroffe and P. Madle, "Application and experience of CAN as a low cost OBDH bus system " presented at the MAPLD, Washington D.C. USA, 2004.

[110] M. Bertoluzzo, "Experimental Activities on TTCAN Protocol," in *Proc. Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications Conf.*, Sofia 2005, pp. 22-27.

[111] A. Emrich, "CAN application in avionics," OmniSys Instruments, Goteborg, Sweden, July 2001.

[112] I. Broster and A. Burns, "An analysable bus-guardian for event-triggered communication," in *Proc. Real-Time Systems, 24th IEEE Symp.*, 2003, pp. 410-419.

[113] E. Webb, "Ethernet for space flight applications," in *Proc. Aerospace IEEE Conf.*, 2002, pp. 4-1927-4-1934 vol.4.

[114] "TTEthernet – A Powerful Network Solution for All Purposes," TTTech Computertechnik AG, Andover, 2009.

[115] E. S. Agency, "SpaceWire - Links, nodes, routers and network," ed. Noordwijk, The Netherlands: ESA Publications Division, 2003.

[116] S. Parkes and A. Ferrer, "SpaceWire-RT," in *Proc. Int. SpaceWire Conf.*, Nara, 2008.

[117] S. Fowell, *et al.*, "The Adaptation and Implementation of SpaceWire-RT for the MARC Project," in *Proc. 3rd Int. SpaceWire Conf.*, St. Petersburg, 2010, pp. 397-401.

[118] M. Pignol, "COTS-based applications in space avionics," in *Proc. Design, Automation & Test in Europe Conf. and Exhibition*, 2010, pp. 1213-1219.

[119] A. T. Tai, *et al.*, "COTS-based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture," in *Proc. High-Assurance Systems Engineering, 4th IEEE Int. Symp.* , 1999, pp. 97-104.

[120] G. S. Aglietti, *et al.*, *Spacecraft Systems Engineering*, 4th ed.: John Wiley & Sons, Ltd, 2011.

[121] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations, An Introduction*: Springer-Verlag Berlin Heidelberg 2012.

[122] D. D. Stott, *et al.* (1996) The MSX Command and Data Handling System. *Johns Hopkins APL Technical Digest*. 143-152.

[123] D.-G. Alejandro, *et al.*, "A Comparison of GN&C Architectural Approaches for Robotic and Human-Rated Spacecraft," in *AIAA Guidance, Navigation and Control Confernce and Exhibit*, ed: American Institute of Aeronautics and Astronautics, 2007.

[124] D. A. Rennels, *et al.*, "A fault-tolerant embedded microcontroller testbed," in *Proc. Fault-Tolerant Systems Pacific Rim Int. Symp.*, 1997, pp. 7-14.

[125] X. Olive, "FDI(R) for satellites: How to deal with high availability and robustness in the space domain?," *Int. J. Appl. Math. Comput. Sci.*, vol. 22, pp. 99-107, 2012.

[126] W. Harkin, "Utilize FDIR Design Techniques to provide for Safe and Maintainable On-Orbit Systems," Johnson Space Center, Technique DFE-7, 1994.

[127] O. Emam, *et al.*, "A fault detection, isolation and recovery (FDIR) strategy based on a message exchange approach to implement autonomous FDIR management on the MARC system.," in *Proc. Data Systems In Aerospace*, Budapest, 2010, p. 33.

[128] B. Jackson, "A robust fault protection architecture for low-cost nanosatellites," in *Proc. IEEE Aerospace Conf.*, 2014, pp. 1-8.

[129] R. E. Kuehn, "Computer Redundancy: Design, Performance, and Future," *IEEE Trans. Reliab.*, vol. R-18, pp. 3-11, 1969.

[130] D. W. Caldwell and D. A. Rennels, "A minimalist hardware architecture for using commercial microcontrollers in space," in *Proc. AIAA/IEEE 16th Digital Avionics Systems Conf.*, 1997, pp. 5.2-26-33 vol.1.

[131] D. A. Rennels and R. Hwang, "Recovery in fault-tolerant distributed microcontrollers," in *Proc. Dependable Systems and Networks Int. Conf.*, 2001, pp. 475-480.

[132] D. W. Caldwell and D. A. Rennels, "Minimalist recovery techniques for single event effects in spaceborne microcontrollers," in *Proc. Dependable Computing for Critical Applications 7*, San Jose, CA, USA 1999, pp. 47-65.

[133] R. M. Keichafer, *et al.*, "The MAFT architecture for distributed fault tolerance," *IEEE Trans. Comput.*, vol. 37, pp. 398-404, 1988.

[134] H. Yashiro, *et al.*, "A high assurance on-line recovery technology for a space on-board computer," in *Proc. 5th Int. Autonomous Decentralized Systems Symp.*, 2001, pp. 47-56.

[135] R. Schlichting, *et al.*, "A Linguistic Approach to Failure Handling in Distributed Systems," in *Dependable Computing for Critical Applications*. vol. 4, A. Avižienis and J.-C. Laprie, Eds., ed: Springer Vienna, 1991, pp. 387-409.

[136] D. A. Rennels, "Fault tolerant computing: Issues, examples, and methodology," University of California, Los Angeles1987.

[137] J. H. Lala, *et al.*, "Advanced Information Processing System (AIPS)-based fault tolerant avionics architecture for launch vehicles," in *Proc. IEEE/AIAA/NASA 9th Digital Avionics Systems Conf.*, 1990, pp. 125-132.

[138] D. A. Rennels, "Reconfigurable Modular Computer Networks for Spacecraft On-Board Processing," *IEEE Computer*, vol. 11, pp. 49-59, 1978.

[139] S. N. Chau, *et al.*, "Design of a fault-tolerant COTS-based bus architecture," *IEEE Trans. Reliab.*, vol. 48, pp. 351-359, 1999.

[140] S. N. Chau, *et al.*, "Analysis of a multi-layer fault-tolerant COTS architecture for deep space missions," in *Proc. IEEE 3rd Application-Specific Systems and Software Engineering Technology Symp.*, 2000, pp. 70-76.

[141] C. Plummer and P. Planck, "Spacecraft harness reduction," in *Proc. Data Systems In Aerospace Conf.*, Dublin, Ireland, 2002.

[142] L. Jianhua, *et al.*, "Communication schemes for aerospace wireless sensors," in *Proc. IEEE/AIAA 27th Digital Avionics Systems Conf.*, 2008, pp. 5.D.4-1-5.D.4-9.

[143] Fink and P. W., "Wireless Network Communications Overview for Space Mission Operations," NASA CCDS 880.0-G-0.169, 2009.

[144] J. Sangeetha and S. Kumar, "A comparative study on WiFi and WiMAX networks," in *Proc. IEEE Int. Computational Intelligence and Computing Research Conf.*, 2010, pp. 1-5.

[145] J. Jansons and T. Dorins, "Analyzing IEEE 802.11n standard: outdoor performanace," in *Proc. 2nd Int. Digital Information Processing and Communications Conf.*, 2012, pp. 26-30.

[146] Z. Alliance, "ZigBee Specification," vol. 2012, ed. San Ramon, CA: ZigBee Alliance Inc., 2008, p. 604.

[147] Y. Morisawa, *et al.*, "A computing model for distributed processing systems and its application," in *Proc. Asia Pacific Software Engineering Conf.*, 1998, pp. 314-321.

[148] K. Sidibeh and T. Vladimirova, "IEEE 802.11 Optimisation Techniques for Inter-Satellite Links in LEO Networks," in *Proc. 8th Int. Advanced Communication Technology Conf.*, 2006, pp. 1177-1182.

[149] K. Sidibeh and T. Vladimirova, "Wireless Communication in LEO Satellite Formations," in *Proc. 8th Adaptive Hardware and Systems Conf.*, 2008, pp. 255-262.

[150] S. Li, *et al.*, "A modified 802.11 protocol applicated in space wireless local area network," in *Proc. Int. Computer Design and Applications Conf.*, 2010, pp. V2-585-V2-588.

[151] T. Vladimirova and K. Sidibeh, "WLAN for Earth Observation Satellite Formations in LEO," in *Proc. 8th Bio-inspired Learning and Intelligent Systems for Security Symp.*, 2008, pp. 119-124.

[152] P. P. Rodger Magness, "An Assesment of Wireless Proximity Networks for Space Application," in *Proc. ESA 9th Advanced Space Technologies for Robotics and Automation Workshop*, Noordwijk, The Netherlands, 2006.

[153] T. Vladimirova and M. Fayyaz, "Wireless Fault-Tolerant Distributed Architecture for Satellite Platform Computing," in *Convergence and Hybrid Information Technology*, ed: Springer, 2012, pp. 428-436.

[154] C. Ye and C. Li, "Using Bluetooth wireless technology in vehicles," in *Proc. IEEE Int. Vehicular Electronics and Safety Conf.*, 2005, pp. 344-347.

[155] L. Jianhua, *et al.*, "Feasibility study of IEEE 802.15.4 for aerospace wireless sensor networks," in *Proc. IEEE/AIAA 28th Digital Avionics Systems Conf.*, 2009, pp. 1.B.3-1-1.B.3-10.

[156] L. M. L. Oliveira, *et al.*, "A WSN solution for light aircraft pilot health monitoring," in *Proc. IEEE Wireless Communications and Networking Conf.*, 2012, pp. 119-124.

[157] J. Culbertson, *et al.*, "Application of wireless technology to CALVEIN launch vehicles," in *Proc. Fly by Wireless Workshop (FBW)*, 2010, pp. 12-13.

[158] Y. Tachwali and H. H. Refai, "System prototype for vehicle collision avoidance using wireless sensors embedded at intersections," *J. Franklin Institute,* vol. 346, pp. 488-499, 2009.

[159] P. N. Ravichandran, *et al.*, "Wireless Telecommand and Telemetry Systems for Satellite Communication Using ZigBee Network," in *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on*, 2009, pp. 274-278.

[160] W. Wilson and G. Atkinson, "Wireless Sensors for Space Applications," *J. Sensors & Tranducers,* vol. 13, pp. 1-9, 2011.

[161] S. Sathyamurthy, *et al.*, "Blue tooth intercommunication system [BTICS] for combat vehicle application," in *Proc. Int. ElectroMagnetic Interference and Compatibility Conf.*, 2006, pp. 148-150.

[162] B. Neil and A. Dawood, "Reconfigurable Computers in Space: Problems, Solutions and Future Directions," presented at the Military and Aerospace Applications of Programmable Logic Devices, Laurel, Maryland, 1999.

[163] M. M. Ibrahim, *et al.*, "Reconfigurable fault tolerant avionics system," in *Proc. IEEE Aerospace Conf.*, 2013, pp. 1-12.

[164] A. Jacobs, *et al.*, "Reconfigurable Fault Tolerance: A Comprehensive Framework for Reliable and Adaptive FPGA-Based Space Computing," *ACM Trans. Reconfigurable Technol. Syst.,* vol. 5, pp. 1-30, 2012.

[165] I. Herrera-Alzu and M. Lopez-Vallejo, "Design Techniques for Xilinx Virtex FPGA Configuration Memory Scrubbers," *Nuclear Science, IEEE Transactions on,* vol. 60, pp. 376-385, 2013.

[166] A. Shye, *et al.*, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Trans. Dependable Secure Comput.*, vol. 6, pp. 135-148, 2009.

[167] N. Aggarwal, *et al.*, "Configurable isolation: building high availability systems with commodity multi-core processors," *SIGARCH Comput. Archit. News*, vol. 35, pp. 470-481, 2007.

[168] C. LaFrieda, *et al.*, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor," in *Proc. IEEE/IFIP Int. Dependable Systems and Networks Conf.*, 2007, pp. 317-326.

[169] S. S. Mukherjee, *et al.*, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proc. 29th Annu Int. Computer Architecture Symp.*, 2002, pp. 99-110.

[170] K. P. Gostelow, "The design of a fault-tolerant, real-time, multi-core computer system," in *Proc. IEEE Aerospace Conf.*, 2011, pp. 1-8.

[171] A. Lofwenmark and S. Nadjm-Tehrani, "Challenges in Future Avionic Systems on Multi-Core Platforms," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, 2014, pp. 115-119.

[172] N. Aggarwal, *et al.*, "Isolation in commodity multicore processors," *Computer*, vol. 40, pp. 49-59, 2007.

[173] Y. A. Nanehkaran and S. B. B. Ahmadi, "The Challenges of Multi-Core Processor," *International Journal of Advancements in Research & Technology* vol. 2, June-2013

[174] J. Ramos, *et al.*, "High-performance, Dependable Multiprocessor," in *Proc. IEEE Aerospace Conf.*, 2006, p. 13 pp.

[175] J. Samson, *et al.*, "High Performance Dependable Multiprocessor II," in *IEEE Aerospace Conf.*, 2007, pp. 1-22.

[176] J. Samson, *et al.*, "Technology Validation: NMP ST8 Dependable Multiprocessor Project II," in *IEEE Aerospace Conf.*, 2007, pp. 1-18.

[177] J. R. Samson, Jr., "Implementation of a Dependable Multiprocessor CubeSat," in *IEEE Aerospace Conf.*, 2011, pp. 1-10.

[178] W. Gropp, *et al.*, *Beowulf Cluster Computing with Linux*, 2nd ed.: MIT Press, 2003.

[179] I. V. McLoughlin and T. R. Bretschneider, "Reliability through redundant parallelism for micro-satellite computing," *ACM Trans. Embed. Comput. Syst.*, vol. 9, pp. 1-25, 2010.

[180] D. S. Katz and P. L. Springer, "Development of a spaceborne embedded cluster," in *Proc. IEEE Int. Cluster Computing Conf.*, 2000, pp. 119-123.

[181] D. Ngo and M. Harris, "A reliable infrastructure based on COTS technology for affordable space application," in *IEEE Aerospace Conf.*, 2001, pp. 2435-2441 vol.5.

[182] C. Steiger, *et al.*, "GOCE end-of-mission operations report," ESA2014.

[183] L. D. Friedman, "Phobos-Grunt Failure Report Released," vol. 2015, ed, 2012.

[184] S. Fuchs and A. J. Wardrop, "Fault Tolerant Computer System," USA Patent, 2000.

[185] J. K. Kishore, *et al.*, "A real time fault tolerant microprocessor based On-Board Computer System for INSAT-2 spacecraft," in *Formal Techniques in Real-Time*

and Fault-Tolerant Systems. vol. 863, H. Langmaack, *et al.*, Eds., ed: Springer Berlin Heidelberg, 1994, pp. 476-487.

[186] M. Campola, *et al.*, "Single Event Effects (SEE) Testing of the Memtek Asynchronous Electronically Erasable Programmable Read-Only Memory (EEPROM) 4096x16 " MEI Technologies, Inc., 2007.

[187] D. Krawzsenek, *et al.*, "Single event effects and total ionizing dose results of a low voltage EEPROM," in *Proc. IEEE Radiation Effects Data Workshop*, 2000, pp. 64-67.

[188] J. L. Barth, *et al.*, "Single event effects on commercial SRAMs and power MOSFETs: final results of the CRUX flight experiment on APEX," in *Proc. IEEE Radiation Effects Data Workshop*, 1998, pp. 1-10.

[189] P. Milliken, *et al.*, "Single Event Effects of commercial and Hardened by design SRAM," in *Proc. 12th European Radiation and Its Effects on Components and Systems Conf.*, 2011, pp. 913-917.

[190] T. Noergaard, "Chapter 3 - Middleware and Standards in Embedded Systems," in *Demystifying Embedded Systems Middleware*, T. Noergaard, Ed., ed Burlington: Newnes, 2010, pp. 59-92.

[191] K. Birman and R. Cooper, "The ISIS project: real experience with a fault tolerant programming system," in *Proc. 4th ACM SIGOPS European workshop*, Bologna, Italy, 1990, pp. 1-5.

[192] O. Babaoglu, "Fault-tolerant computing based on Mach," *J. SIGOPS Oper. Syst. Rev.*, vol. 24, pp. 27-39, 1990.

[193] D. Powell, "Distributed fault tolerance: lessons from Delta-4," *IEEE Micro*, vol. 14, pp. 36-47, 1994.

[194] J. Balasubramanian, *et al.*, "Adaptive Failover for Real-Time Middleware with Passive Replication," in *Proc. IEEE 15th Real-Time and Embedded Technology and Applications Symp.*, 2009, pp. 118-127.

[195] M. Fayyaz, *et al.*, "Adaptive middleware design for satellite fault-tolerant distributed computing," in *Proc. IEEE NASA/ESA Adaptive Hardware and Systems (AHS) Conf.*, 2012, pp. 23-30.

[196] M. Fayyaz and T. Vladimirova, "Fault-Tolerant Distributed approach to satellite On-Board Computer design," in *Proc. IEEE Aerospace Conf.*, 2014, pp. 1-12.

[197] Tanya Vladimirova, *et al.*, "A report on Distributed Architectures" Contributing to D3.1 on WP3: System Level Solutions of FP7 ReVuS Project. ," 2012.

[198] S.M. Sadjadi, "A Survey of Adaptive Middleware," Michigan State University, East Lansing Michigan.,2003.

[199] W. River, "Product note on Wind River's VxWork," Wind River CA 2006.

[200] R. Mall, *Real-Time Systems: Theory and Practice*: Prentice Hall Press, 2009.

[201] "An Architectural Overview of QNX," in *Usenix Workshop on Micro-Kernels & Other Kernel Architectures*, Seattle, April-1992.

[202] (10th October-2015). *uClinux-dev Forum Website [Online]*. Available: http://mailman.uclinux.org/pipermail/uclinux-dev/2009-February/048040.html

[203] K. Andersson and R. Andersson, "A comparison between FreeRTOS and RTLinux in embedded real-time systems," 2005.

[204] D. Caban, "Efficiency and memory footprint of Xilkernel for the Microblaze soft processor," ed. Poland: EDN Networks, June 18, 2014.

[205] P. Felber and P. Narasimhan, "Experiences, strategies, and challenges in building fault-tolerant CORBA systems," *IEEE Trans. Comput.*, vol. 53, pp. 497-511, 2004.

[206] O. M. Group, "Fault Tolerant CORBA," in *The Common Object Request Broker: Architecture and Specification*, Sept. 2001 ed, 2001.

[207] P. Narasimhan, *et al.*, "MEAD: support for Real-Time Fault-Tolerant CORBA," *J. Concurrency Comput.: Pract. Exper.*, vol. 17, pp. 1527-1545, 2005.

[208] J. Balasubramanian, "FLARe: a Fault-tolerant Lightweight Adaptive Real-time middleware for distributed real-time and embedded systems," presented at the Middleware doctoral symp., Newport Beach, California, 2007.

[209] J. Balasubramanian, *et al.*, "Towards Middleware for Fault-Tolerance in Distributed Real-Time and Embedded Systems," in *Distributed Applications and Interoperable Systems*. vol. 5053, R. Meier and S. Terzis, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 72-85.

[210] H. Kopetz, "The Time-Triggered Architecture," in *Real-Time Systems*, ed: Springer US, 2011, pp. 325-339.

[211] M. L. Shooman, "N-MODULAR REDUNDANCY," in *Reliability of Computer Systems and Networks*, ed: John Wiley & Sons Inc., 2002, pp. 145-196.

[212] I. A. Zimmermann, "Software Tool for the Performability Evaluation with Stochastic and Colored Petri Nets," TU Berlin, Germany, 2013.

[213] S. Bernardi, *et al.*, "Petri Nets and Dependability," in *Lectures on Concurrency and Petri Nets*. vol. 3098, J. Desel, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2004, pp. 125-179.

[214] (2012, June). *STM-3240G-EVAL Evaluation Board (4th ed.)*. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00036746.pdf

[215] IAR-Systems, "IDE Project Management and Building Guide for Advanced RISC Machines Ltd's ARM Cores," 2011.

[216] S. LLC, "Saleae User Guide 1.1.15, Logic and Logic16 User's Guide.," 2012.

[217] A. Alonso, *et al.*, "Design of On-Board Software for an Experimental Satellite," presented at the Real-Time Conference, 2013.

[218] J. Garrido, *et al.*, "Analysis of WCET in an experimental satellite software development " presented at the 12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012), 2012.

[219] H. Topcuouglu, *et al.*, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260-274, 2002.

[220] Xilinx. (2014, DS190). *Zynq-7000 All Programmable SoC Overview*. Available: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[221] Xilinx. (2012, Jan.). *ChipScope Pro Software and Cores User guide*. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/chipscope_pro_sw_cores_ug029.pdf

[222] Avnet. (2012, November). *ZEDBAORD, Zynq Evaluation and Development Harware User's Guide*. Available: http://zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf

[223] R. Barry, "FreeRTOS, A Free RTOS for small real time embedded systems," FreeRTOS, 2005.

[224] Xilinx, "Power Analysis and Optimization (UG907)," 2014.

[225] Xilinx. (25th October). *XILINX ARTIX-7 FPGAS: A New Performance Standard for Power-Limited, Cost-Sensitive Markets*. Available: http://www.xilinx.com/support/documentation/product-briefs/artix7-product-brief.pdf

[226] A. E. Cooper and W. T. Chow, "Development of On-board Space Computer Systems," *J. IBM Research and Development,* vol. 20, pp. 5-19, 1976.

[227] Altium. (2008, December). *Tutorial: Getting Started with PCB Design* Available: http://www.altium.com/files/altiumdesigner/s08/learningguides/tu0117%20getting%20started%20with%20pcb%20design.pdf

[228] T. Grelier, *et al.*, "Formation flying radio frequency instrument: First flight results from the PRISMA mission," in *Proc. 5th ESA Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing*, 2010, pp. 1-8.

[229] G. Krieger, *et al.*, "Interferometric Synthetic Aperture Radar (SAR) Missions Employing Formation Flying," *Proc. IEEE,* vol. 98, pp. 816-843, 2010.

[230] P. J. Buist, *et al.*, "Functional model for spacecraft formation flying using non-dedicated GPS/Galileo receivers," in *Proc. 5th ESA Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing*, 2010, pp. 1-6.

[231] J. Davis, "Mathematical Modeling of Earth's Magnetic Field," Virginia Tech, Blacksbury2004.

[232] W. M. T. Flatley, A.Reth and F. Bauer,, "A B-Dot Acquisition Controller for the RADARSAT Spacecraft," presented at the NASA Conf. publication, 1997.

[233] M. K. F. Torben Graversen, Søren Vejlgaard Vedstesen, *Attitude Control system for AAU CubeSat*: Aalborg University. Department of Control Engineering, 2002.

[234] L. L. Show*, et al.*, "Spacecraft robust attitude tracking design: PID control approach," in *Proc. American Control Conf.*, 2002, pp. 1360-1365 vol.2.

[235] S. Beatty, "Comparison of PD and LQR Methods for Spacecraft Attitude Control Using Star Trackers," in *Proc. 6th World Automation Congr.*, 2006, pp. 1-6.

[236] M. O. Karslioglu, *et al.*, "A ground-based orbit determination for BILSAT," in *Proc. 2nd Int. Recent Advances in Space Technologies Conf.*, 2005, pp. 155-158.

[237] S. Sgubini and G. B. Palmerini, "Ground-based orbit determination for spacecraft formations," in *Proc. IEEE Aerospace Conf.*, 2010, pp. 1-7.

[238] N. Zhuo, "Onboard Orbit Determination using GPS Measurements for Low Earth Orbit Satellites," Cooperative Research Centre for Satellite Systems, Queensland University of Technology, 2004.

[239] N. Instruments, "M Series Multifunction DAQ for USB - 16-Bit, 250 kS/s, up to 80 Analog Inputs," 2008.