A FAULT-TOLERANT BUILDING BLOCK FOR TRANSPUTER NETWORKS FOR REAL-TIME PROCESSING

By

Yueying Fei

B.A. Sc., Northeast Heavy Machinery Institute, P.R.China, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL ENGINEERING

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

January, 1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) _____

Department of Electrical Engineering

The University of British Columbia
Vancouver, Canada

Date Jan. 27, 1993.

Abstract

Software Implementation of Multi-Processor Fault Tolerance for Real-Time processing is addressed in this thesis with the research focused on:

- Fault-Tolerant cells as building blocks that can survive concurrent transient physical faults and permanent failures in large parallel processing systems with potential for real-time processing.
- Efficient group communications for redundant data exchanges through multiple communication links that connect the group peers.
- Transparent fault-tolerance.
- On-Line Forward Fault-Repair using the **live execution image** from the non-faulty peer with a bounded delay.

By systematically connecting the redundant processing modules, the architecture offers regularity and recursiveness which can be used as building blocks for construction of fault-tolerant parallel machines.

The communication service protocols take advantage of redundant linkages to ensure reliable and efficient message deliveries among the fault-tolerant abstract transputer peer nodes through the concept of **activity observation**. The multiple redundant linkages provide a means for parallel communications. This is essential for redundant information exchanges in **fault-tolerance**. The **activity observation** concept further reduces the effort for reliable message delivery and simplifies the system design. As a result, messages are dynamically and optimally routed when link failure or processor failure occurs.

Through the group communication mechanism underlying the platform, application processes on each FTAT peer node are transparent to details that they are replicated, repaired upon fault detections, and reintegrated after fault repair. Based on a dynamic Triple Modular Redundancy scheme, each application process can survive up to two concurrent faults under the assumption that the probability of two faulty peer applications having the same fault is very small.

In a large interconnected network, the cost of fault-tolerance can be very expensive in terms of time and communication due to the cost of either synchronization or roll-back recovery. The use of redundant live execution images to repair the faulty module guarantees forward fault recoveries.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to express my sincere thanks to Dr. Mabo Robert Ito, my supervisor for his patient guidance and persistent financial help during the course of my thesis study.

I would also like to extend my thanks to Dr. Sam Chansom, Dr. Allan Wagner of Computer Science Department for their precious opinions and help in doing the research. Thanks also go to Mr. A. Zaafrani, Mr. Robert Lam, Mr. Jefferey Chow of Electrical Engineering Dept. for their help in editing up the thesis.

My deepest thanks to my wife for her help and encouragement, who had to face with the life alone much of the time and relieved me from most of the house work to allow me dedicate myself to my studies.

Chapter 1

Introduction

Computers have been widely applied in almost every aspect of our daily life due to the rapid development of the computer hardware technologies and enormous decrease in their costs. However, applications such as aircraft control, nuclear power station control, and spacecraft control etc. can not afford potential catastrophe in which human lives are at stake, caused by computer faults. Computer fault-tolerance has been an important research area for many years and is the subject of this thesis.

1.1 The Overview Of The Thesis

In this thesis research, we target a system that meets the following requirements:

- Transparent protection to the applications from limited permanent module failures, transient and intermittent physical faults based on multiple modular redundancy.
- A systematic structural architecture that easily fits into parallel machines.
- Transparency to applications of the underlying fault-tolerant architecture to facilitate parallel processing.
- Deterministic system behavior for real-time processing.

To meet the above requirements, we built an experimental system based on systematically connected redundant modules with multiple linkages in hardware and a structural

approach in software as building blocks. Such a building block is called Fault Tolerant Abstract Transputer (FTAT). Through the designed multi-link group communication mechanism, an application on top of the system gets transparent fault tolerance services that encapsulate acceptance testing, fault masking, possible fault-repairing, reintegrating the repaired module, and inter-FTAT replicated process message-passing. The system is also abstracted with a consistent view of a transputer to its applications through the mechanism, providing an easy model for programming fault tolerant parallel processing under the system.

The multi-link group communication is designed with the concept of **activity observation**, gaining much improved communication reliability and good performance in redundant data distribution. **Activity observation** is explained in detail in chapter 3. Generally speaking, it uses extra ACKs as the drivers for reliable data distribution and efficient group synchronization.

A dynamic Triple Modular Redundancy (TMR) scheme employed in the system makes it possible for an FTAT to be tolerant of two concurrent faults based on four transputers. The on-line forward fault-repair, described in detail in chapter 5, greatly extends the life of the system, repairing unlimited sequential transient physical faults without checkpointing and roll-back recovery. It guarantees forward fault recovery and eliminates the nondeterminism due to roll-back recovery that is common to most current fault-tolerant distributed systems. Our reliability analysis, in chapter 6, has shown a remarkable improvement in systems reliability by the use of forward fault-repair.

The interesting approach to process migration for volatile on-line fault-repair separates applications from their run-time environment at a minimal cost, making applications relocatable during run-time.

In this thesis, chapter 1 introduces the research by discussing the major design issues, reviewing the related work, and introducing our approaches and their rationals. Chapter

2 describes our system architecture. The design of the scatter and gather service for one-to-many and many-to-one pattern communication, which is the core of the multi-link group communication protocol, is discussed in chapter 3. The swap service and the FTAT services, together with the scatter and gather service constituting the multi-link group communication mechanism, are discussed in detail in chapter 4. Chapter 5 describes the fault-repair scheme in which the volatile run-time context is used to achieve forward fault-recovery. A brief reliability analysis is conducted on the multi-link group communication design and on forward fault-repair for its contribution to the system reliability in chapter 6. The multilink group communication design is formally represented in Predicate/Transition net, a variation of Petri Net, and is informally shown to be correct. Chapter 8 discusses the performance results of the two major components, the scatter and gather service as well as the on-line forward fault-repair, in term of time and amount of communication. These two components are the major contributions in this research. The thesis is concluded in chapter 9, which gives a summary of our conclusions.

1.2 Major System Design Issues

1.2.1 Fault Type

There are many fault types in computer systems such as permanent faults, transient faults, software design errors, human operation errors, etc. The errors which result from these faults may have a catastrophic effect on the system.

Hardware faults (physical faults) include short circuits between two leads, open circuited transistor junctions, alpha particle impact on dynamic MOSFET memory cells [7][10]. These errors most often appear as changes in the patterns of zeros and ones that represent the information. Other errors are time based; e.g., information fails to arrive or arrives at the wrong time [10]. Errors that are not detected and eliminated within the

system are likely to lead to the failure of the systems. In addition, faults due to temporary environmental conditions are hard to repair because the hardware is undamaged. It is this attribute of transient faults that makes them of interest [7].

It has been observed that more than 75% of faults occurring in a system are transient in nature [16], and it is known that more than half of all failures are intermittent [11]. Therefore, the system reliability greatly depends on its recoverability from these faults. Permanent or solid failures should be removed, with related parts, from the system.

For these reasons, transient physical faults are the major fault type to be dealt in this thesis.

1.2.2 Parallel Processing System Architecture

In recent years, the price of hardware has dropped to the point where non-self-checked computation redundancy, that can guarantee the computation integrity, can be introduced into commercial machines. Spatial redundancy [7][10] provides the potential for efficient fault detection and recovery if the redundancy is fully exploited. To be tolerant of concurrent faults, a system usually needs more processing modules than double of the number of concurrent faults to be tolerated. However, among independent modules, two concurrent faults on different modules are unlikely to be the same as suggested in [22].

The cost of fault tolerance in a large parallel processing system can be very expensive due to either synchronization or checkpointing and roll-back recovery. Since the probability of system failure increases rapidly with the number of processors [7], it is important to incorporate structural fault-tolerance in medium- to large-scale multiprocessor computers.

1.2.3 Multilink Group Communication and Real-Time Processing

Real-Time applications have strict requirements over the system's response to the service requests. Timeliness is the major characteristic for real-time systems. Asynchronous recovery does not offer any timeliness characteristic as a recovery may cause a nondeterministic delay because of a chain of roll-backs [8][18] in worst cases. Synchronous ones will have more overhead in synchronization and cooperation. The difficulty in efficient synchronization [17] means additional cost in terms of development effort and operation overhead. Efficient group communication technique should be developed for multiple modular redundancy fault tolerance.

It is shown in [5][21] that the more processors or computers are involved in cooperation to achieve a certain goal, the higher the overhead is. Redundant messages flowing among these replicas are essential for fault-masking but are desired to be transferred at minimal cost, especially for masking redundancy. As a result of a structural fault-tolerance approach, the memberships of a group at a level in the structure can be well defined. The communication among the members can also be simplified so as to minimize the synchronization overhead.

Multi-links for communication provides the potential for fault-tolerant communication. Efficient multilink group communication in support of an architecture of structural fault-tolerance in large parallel processing systems is a key issue to systems design.

1.2.4 Transparency of Fault Tolerance

In many fault-recovery systems, the designers of applications are required to explicitly program for recovery; e.g., by writing code to take checkpoints and to restore the internal states from the checkpoints after detection of fault, error or a failure, and by organizing computations as transactions and periodically committing them. Transparent recovery

[12] on the other hand, hides the details of checkpointing and fault occurrences in the underlying hardware from the programmer. The application development effort for fault-tolerance is thus greatly reduced.

Parallel programs are non-trivial to design and develop on a large parallel processing machine. It is even harder to program parallel processing with fault-tolerance requirements. Also, the costs in the software overhead of fault-tolerance and the development effort can be much higher than those in a single processor case. A structural development approach structures the system with different levels of abstractions in a modular way. Hence, the effort paid for fault-tolerance in development and operation of the overall system can be greatly reduced.

As system grows in sophistication, the advantages of transparent recovery increases. Transparent fault-tolerance service will be more significant for fault-tolerant parallel machines in the near future. In such systems, a simple programming model provided by the transparency to the architecture means great convenience for the developments of fault-tolerant parallel-processing.

1.2.5 Forward Fault Repair

As a support of the architecture transparency, faulty applications should be repaired and reintegrated into the system smoothly.

On-line fault repair on most concurrent systems is done through system reconfiguration and can run into problems when spares are exhausted. Checkpointing and roll-back recovery may cause a nondeterministic chain of roll-backs in a global system that requires deterministic behavior of a system and is not suitable for systems of time constraints.

On-line fault-repair means that both the availability and reliability of the system have to be maintained during run-time even when transient faults or failures occur.

When high degree of redundancy is used for fault-tolerance, the redundant run-time information should be exploited to the maximum to achieve an on-line forward fault-repair. Consequently no computation is wasted.

In this research project we have addressed the problems of:

- efficient multi-link group communication for redundant data distribution among the modules within a structurally built fault-tolerant cell.
- transparent fault-tolerance with dynamic Triple Modular Redundancy (TMR).
- on-line forward faulty process repair using the redundant live execution image.

Section 2 describe the previous related work. Section 3 introduces our approach to the systematically structured multiple-modular-redundancy fault-tolerance under real-time constraints, the transparency to the fault-tolerant architecture, and the on-line forward fault repair.

To aid in the discussion, we use the following terms consistently throughout the following chapters:

- Peer Module or Peer Node – a redundant module.
- Peer system – the system on each redundant module.
- System – the system that comprises the four redundant modules, called a Fault-Tolerant Abstract Transputer (FTAT), cooperating to provide a single transputer abstraction.
- The overall system – the target system built with FTATs to form a fault-tolerant multiprocessing system.

1.3 Previous Related Work Review

The previous work is reviewed under the separate topics to distinguish different areas addressed.

1.3.1 Fault-Tolerant Architecture

There are quite a few successful commercial fault-tolerant systems on the market today. Tandem's non-stop systems [26] package the CPU, memory, bus, and I/O controller as replaceable modules operated by message-based distributed operating system. I/O device controllers are dual-ported, accessible from either port. All the modules are connected by the dual-bus called "dynabus" with on-line repair features. The approach eliminates any single point vulnerability and is dynamic in its "primary and standby" architecture. However, it is difficult to build very large systems based on this architecture [26].

Stratus [26] employs a "pair and spare" architecture. The major functions on the system are replicated four times in independent and identical modules in a hierarchical way. At the first level, two modules form a paired function unit with each employing self-checking. Then the two function units work as "primary and standby" with on-line transparent repair. In the system, self-checking is employed throughout the system; but "pair and spare" is applied only to the CPUs and memory. It is based on a ring type of local area network, a message-based system. Nonetheless performance is degraded as the number of processing modules increases because of communication bottleneck.

Much other research has been done in the fault-tolerance area. An interesting approach in [14] suggested that a high abstraction module based on a petri-net can be used as an observer. This abstract model can preserve the specification redundancy through design diversities for software fault detection. Although not providing a full fault-coverage, it is useful for detection of software design errors.

Based on transputers, the DRAT system [28] is implemented in a ring connection for On-Line Transaction Processing (OLTP) applications. Each data processor has a recovery processor associated with it. The recovery processor is connected in a separate link route from the data base processing path to avoid link traffic bottleneck. Nevertheless, the separation of the recovery ring [28] from the data ring makes the system vulnerable to multiple link failures and node failures. In addition, the system is also difficult to be reconfigured to mask processing module failures.

The fault-tolerant processor described in [27] is built based on Triple Modular Redundancy (TMR). However, it is not designed to be used as building blocks for construction of large systems. It does not provide transparency to its underlying hardware architecture. The system can only mask faults by reconfiguration.

1.3.2 Group Communication Facilities

In order to support replicated distributed programs [19], a replicated procedure call mechanism is developed. The client group and server group, which are termed troupes, can exchange messages among the troupe members on the collective unit basis through the call. The members act as a logical unit and are brought in step by the call. The group communication mechanism deals with the design issues such as those in sending and receiving between different troupes. These systems are based on a LAN, which is vulnerable to the communication link failure.

1.3.3 Transparency in Fault-Tolerance Architecture

Transparent recovery in distributed systems is recognized to be of great convenience since a programmer does not have to explicitly "arrange" for taking checkpoints and recovering from the consistent state backup.

Compiler assisted checkpointing [15] is a technique to provide transparent checkpointing. The compiler generates sparse potential checkpoint code with adaptive checkpointing to reduce the size of the checkpoints. The volatile logging technique in [29] can hide the programmer from the fault-recoveries.

1.3.4 On-Line Forward Fault-Repair

On-line fault-repair is attractive because it provides continuous services to the application even in the face of failures, provided that the majority of the redundant modules still work correctly. Conventional fault-tolerant computer systems usually use a checkpointing and roll-back recovery scheme or use primary and hot-standby to replace the faulty modules.

In a system with a high degree of redundancy, it is desirable to exploit the redundancy to the maximum so as to provide forward fault-recoveries. The non-determinism due to roll-back and recovery can be eliminated if the non-faulty volatile states of the application can be used to repair the transient faults in the redundant modules. Process migration is an ideal technique for the on-line forward fault-repair.

Process migration has been used mainly for load balancing [23][24][25]. However, the technique has the potential to support fault-repair. Many systems that support process migration are based on shared virtual memory [23][24][25]. The memory paging mechanism supported by the hardware is very useful in reference rebinding problems [23]. On the other hand, the cost is very high for maintaining the shared virtual memory on distributed systems and accessing disk I/O, since every object on such systems is persistent and requires concurrency control to ensure its consistency under concurrent access [2].

Highly Availability Transputing Systems [1] provides dynamic fault recovery for MIMD parallel machines. However, it does not provide forward fault recovery to avoid waste of computation due to recoveries and retries.

1.4 Our Approaches In Meeting The Requirements And Their Rationals

We approach the major issues previously identified by the following:

- To facilitate fault tolerant parallel processing,
  - a systematically structural architecture to yield regularity and recursiveness.
  - transparent fault tolerance service to provide an easy programming model.
- To meet time constraints,
  - efficient multilink group communication for redundant data distribution to reduce communication overhead.
  - efficient group synchronization through cooperation among well defined processors of small sized group to reduce synchronization overhead.
  - forward fault recovery by the of the redundant live execution image to eliminate the nondeterminism due to roll-back recoveries.
- To maintain high reliability,
  - a dynamic TMR to tolerate concurrent faults.
  - forward fault repair to tolerate sequential transient physical faults.

We discuss our approaches to the design issues in separate topics for different areas in the same order as in section 1.3.

1.4.1 Multiple Modular Redundancy

As hardware cost decreases and microprocessing power soars (the recent DEC RISC microprocessor Alpha is rated 200MIPS), micro based parallel processing systems with

fault tolerance will become in demand. This is not only because they can be built from the off-the-shelf chips, but also they offer much more powerful capabilities when they are bound with parallel processing techniques while paying moderate cost in performance for fault-tolerance and guaranteeing reliability.

Since the probability of system failure increases very rapidly with the number of processors [7], it is important to incorporate structural fault-tolerance into medium- to large-scale computer [30]. A natural way of building a large system is to build smaller units first and then to construct bigger units from the smaller units in a hierarchical fashion.

Transputers are 32-bit RISC microprocessors with on-chip communication facilities as part of its architecture. They offer multiple links of high speed serial communication channels and can be built into parallel systems easily to provide nearly scalable processing power. This architecture makes transputers suitable for interconnection in hardware without high cost and provides the potential for fault-tolerance. DSP processors with multiple on-chip communication links and high processing power have been developed by Texas Instruments.

This may be an indication that multiple link processors will play a more and more important role in parallel processing as well as in fault-tolerance in the future. We choose the transputers because its architecture offers very good inter-connectivity for parallel machine construction, the message passing based communication and its availability in the department.

The experimental system platform is based on the transputers hosted on a Sun work-station. Four transputers form a Fault-Tolerant Abstract Transputer (FTAT). Each transputer has four on-chip serial communication links that are bidirectional with support from dedicated DMA controllers, at a data rate of 20Mbits/sec./link. An FTAT is based on four transputers which are fully connected in a systematic way, which is

described in detail in chapter 2. The modular group FTAT cooperates among its well defined members to minimize the overhead in fault tolerance and synchronization.

This abstract transputer FTAT can be used as a building block to build a large parallel processing system. It can also be used as a non-stop type of a computer system with the potential for real-time processing. In our system, the software distributed over the four redundant processor units hides all of the details of application replication, fault-masking, on-line forward fault-repair and reintegration after repair.

On message-based distributed systems, fault-recovery may involve roll-backs of a group of processes based on the checkpoints [8]. All the invocations to other processes, since the last consistent checkpoint, must be undone in order to eliminate the effect of a fault. This has a drawback that the recovery may take a nondeterministically long delay and wastes a lot of computation time on the overall global system. This approach is not suitable for real-time processing in a large parallel processing system.

Our approach is to try to confine the fault within the module on a process basis to avoid expensive global recoveries. Before each output from an FTAT, fault detection is conducted and only the correct output is sent. Each application process is a fault-tolerant object on the system. Inter-process communication is protected by error control in the protocols. A faulty process is repaired by using the run-time context of a non-faulty peer instead by undoing and then retrying of the last failed attempt. Hence, forward recovery is guaranteed. Our on-line fault-repair mechanism repairs the faulty module on a process basis instead of through pure reconfiguration.

1.4.2 Multilink Group Communication

A group communication mechanism is essential to enable the group members to communicate and reach agreement on the state of the applications running on top of the system despite random communication delays and failures. The mechanism should make the

behavior of the group indistinguishable from that of a non-redundant processing module [17] in order to implement the desired replication transparency. Multi-link group communication provides an essential means for redundant data distribution.

To achieve efficient multilink group communication, ACKs are broadcast to all the other peers in response to each data packet received, to create the transmission redundancy in the form of control packets and to spread the communication status of the receiver for efficient synchronization. The redundant ACKs creates more opportunities for the intended receiver to take active action. Packet loss can be detected promptly and can be possibly claimed back in parallel. The low cost in using ACKs as "hints" for optimal packet loss claiming outperforms sender-initiated dynamic packet routing in term of both time and communication.

1.4.3 Transparency

To provide a simple programming model, the service abstractions designed are similar to the **conversation** construct in [13] for a distributed recovery block spanning several processes. Applications need only to hand in the result or intermediate result for acceptance testing and determine its "pace" to use (call) the service primitives to achieve the degree of reliability in computation integrity and fault coverage [6].

The system provides Read and Write calls which performs the communication functions to read from and write to external FTATs. The write call encloses a set of symmetric operations among the four peer modules within an FTAT. It also provides a dummy call that only invokes the fault-tolerance functions implicitly without actually sending out the result to other FTATs. This allows computations to run on its own logical phases.

1.4.4 On-Line Forward Fault Repair

By incorporating process migration techniques into the system, a faulty application process can be repaired by using the volatile execution image from a non-faulty one. This guarantees a forward fault repair. The approach to fault repair in our system is novel and interesting. The availability of the redundant modules subject to sequential faults is unlimited, thus the life time of the system is greatly extended.

In process migration, one of the difficult problems is to cope with pointer references in the run-time context of a process. The problem is handled by the use of a pointer reference stack in conjunction with a set of resource management information tables maintained by the system. The architecture of a transputer with distributed memory provides only one linear address space without any memory paging mechanism support. The shared virtual memory solution is not practical if not impossible here.

The difficult reference rebinding problems in process migration are handled through the system's resource management information and run-time facility of the so called "pointer reference stack". The pointer creations are tracked by the system all the way down the current execution path. Our approach is to separate the applications from their run-time physical environment so that they can be moved around easily and effectively at minimal cost.

Chapter 2

System Architecture

This chapter discusses the fault-masking architecture and software architecture on our Fault-Tolerant Abstract Transputer (FTAT). By systematically connecting the redundant modules through the multilinks on each processor, the architecture designed is regular and recursive. The resultant architecture preserves the original transputer architecture. The FTATs are easily to be constructed into large parallel processing systems without creating a complex view of the system.

The system software implements an abstraction of a transputer while providing fault-tolerance services transparently. The overall system yields low cost for fault tolerance due to the avoidance of roll-back recovery and the small scale of synchronizations and cooperation among well defined group members.

2.1 General Assumptions

In order to focus on the issues identified, the system was designed under the following assumptions:

- The processing modules based are computers with private memory, 4 bidirectional communication links and a copy of the operating system.
- The application processes are statically created in the same order on each of the processing modules of an FTAT.

- Applications are allowed to use physical pointers residing only on the run-time stack.
- The FTAT system software is assumed to have omission failure semantics [6]. Arbitrary failures [3][17] are not considered in the system design.
- The application processes are cooperative as opposed to collaborative.

The system is built via an structural approach for both hardware and software. It provides a many-to-many inter-FTAT communication mechanism to carry out the redundant data distribution with the omission failure semantics [17]. A set of group communication primitives, called the scatter and gather services, are furnished as generic primitives for the reliable redundant data distribution and synchronization in the event of link failures and processor failures.

2.2 System Hardware Architecture

The hardware redundancy is based on four transputers so connected that from any individual transputer the scan of others is straight forward. Each transputer has its node number. Each board has four transputer nodes with their own private memory. The transputers are numbered from 0 to N sequentially in combination with a board number. Links 1 to 3 of a transputer are used as FTAT internal links leaving link 0 as a FTAT link. The relation between node numbers Peer#, board numbers FTAT#, and link numbers InternalLink# in the configuration of the system can be defined by the following formulas:

$$FTAT\# = \lfloor Peer\# \div 4 \rfloor;$$

$$InternalLink\# = (RemotePeer\# - LocalPeer\#) \bmod 4;$$

$RemotePeer\# = (InternalLink\# + LocalPeer\#) \bmod 4;$

$FTATLink\# = LocalPeer\# \bmod 4;$

As we can see, the architecture offers regularity and recursiveness, which makes it easy to build a parallel machine with the FTATs.

The hardware architecture is shown on the left side in Figure 2.1. On the right is shown the an example of the architecture applied in a grid connection.

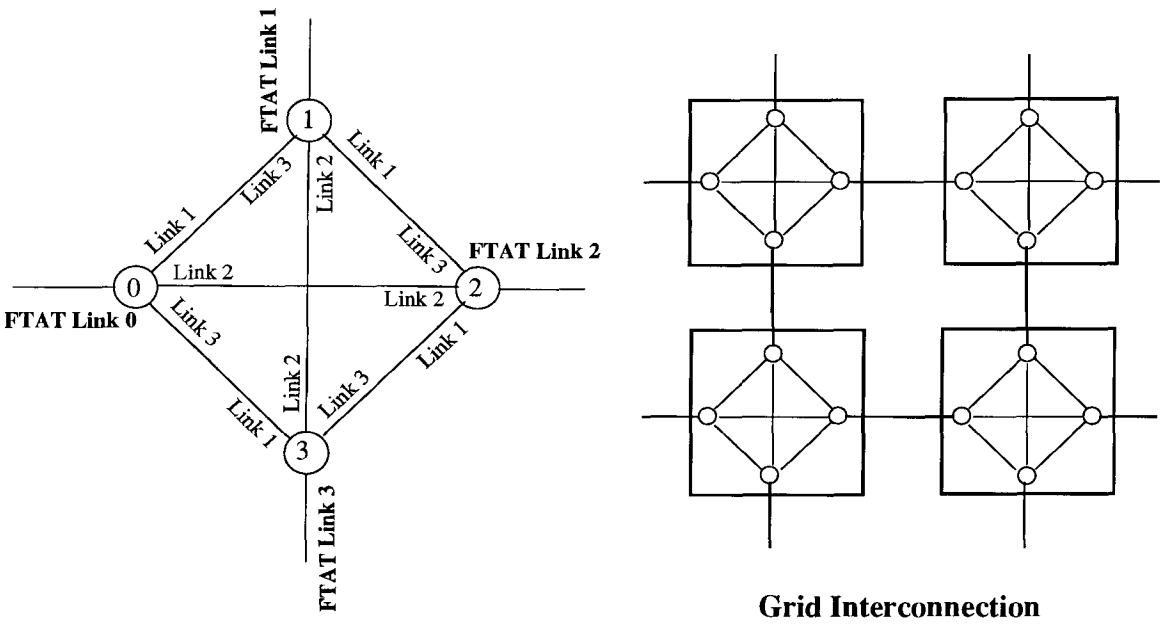


Figure 2.1: Hardware Architecture

2.3 System Software Architecture

Figure 2.2 shows the programming model as seen by systems programmers. The programming model is defined as following: Each transputer within an FTAT is a **peer module**. All four peer modules within one FTAT run an identical set of application

processes. We say that the set of applications are replicated over the peer modules. Each of these replicated processes is called a **replica process**. The four replica processes are collectively called a **replicated process** which is identified by a unique ID.

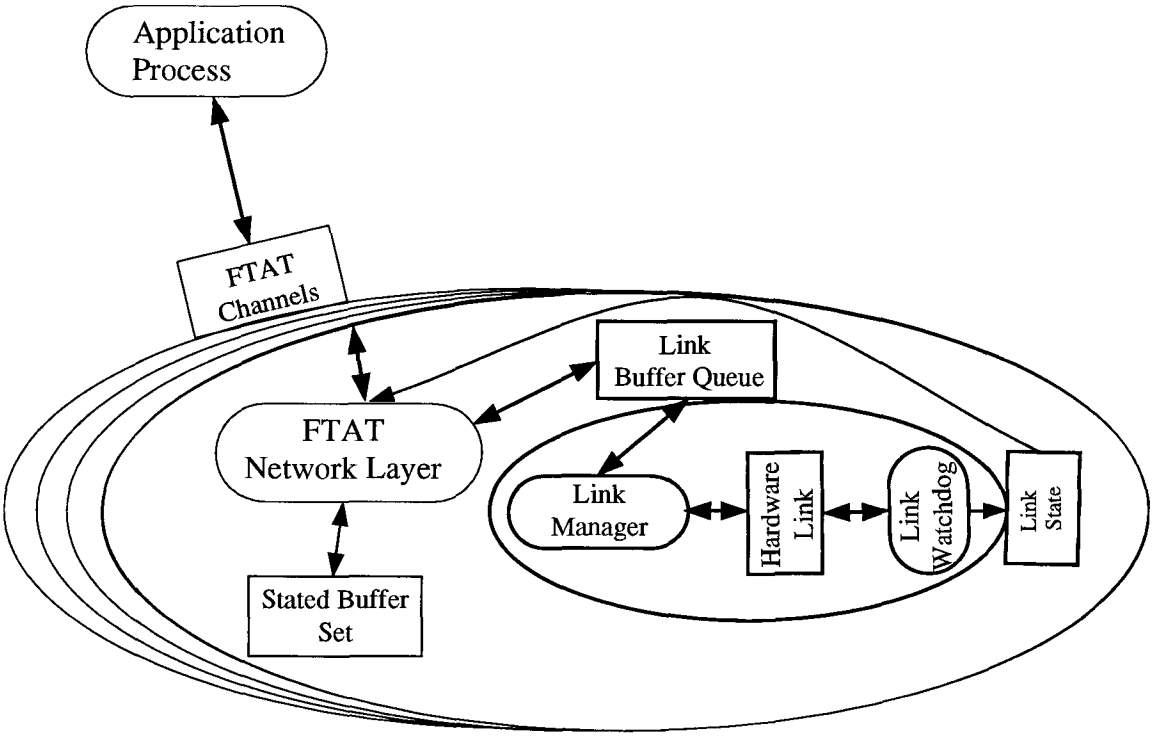


Figure 2.2: The system architecture of an FTAT

The system software implements the abstraction of an transputer with fault tolerance over the four peer modules. This provides to application programmers a simple programming model for developing fault tolerant parallel processing applications based on FTATs just like on individual transputers, while the system takes care of the fault tolerance requirements within the FTATs. On the right side of Figure 2.1 is an example of part of a mesh connected application system based on the FTATs as building blocks.

For each message packet, the sender is called the **originator** of the message packet, and the destination receiver the **receiver** of the packet. A peer module that has noticed

such a delivery is called the **observer** of the packet. These terms are used in the following chapters using the definitions given here.

Since a replicated process is a logical unit on the system, it must be identified uniquely so that an operation in group can be recognized by the system which provides the services. The ID of each replica process is formed by concatenating the peer module ID and process ID on the peer system. From the system assumption, the application processes are created on all peers of an FTAT in the exactly same order. So the same sequence number is be assigned to the same replica process. This results in much simplified design: the process ID can be used as the ID for the replicated process ID, since the replicated process identification is no longer a major issue.

The system is built on top of the hardware architecture defined above. The structural approach can greatly reduce the work to obtain overall system fault-tolerance when a multiprocessor or massively parallel system is built. The higher levels of a large system, constructed with the building blocks, need only to take care of the optimal routing of messages and the reliable delivery of packets between FTATs. Moreover, the system ensures that the received packet on an FTAT will not be lost due to peer module crash right after its reception. It is the receiving FTAT's responsibility to ensure the correct and optimal delivery of the message.

The four transputers in an FTAT work in dynamic TMR under the assumption that two modules having the same fault simultaneously is very small due to the autonomy of each module. Each peer system must get one and only one vote in the majority consensus to assert "non-faulty". The output from the FTAT masks any fault in the system automatically. Once a fault is detected, the peer system sends a request for repair to one of the non-faulty peers and the on-line fault-repair process is started.

Chapter 3

Highly Efficient Multiple-Link Group Communication

This chapter describes our contribution with respect to efficient redundant data distribution among group members through the multiple link connections. The concept of **activity observation** used in the protocol design provides receiver-initiated dynamic packet rerouting. This may lead to possible parallel claim of lost packets via optimal routes, low cost by using ACKs as rerouting drivers and better synchronization among the group peers.

3.1 Communication System Structure

The system has 3 layers in its implementation of the communication services. The physical layer provides a blocking call that requires an exact matching count of data transferred by both parties of the communication. Based on this layer the link layer communication primitives are developed. On top of the link layer communication services, the FTAT group communication network layer packages three logical level of services to provide the transparency to the underlying service details.

3.2 Link Channel Communication Layer

The link layer service provides a reliable communication packet channel. It preserves the packet sending order, employs flow and error control, and guarantees the correct reception of the packets carried. It works with a **watchdog** that monitors the link status through

the associated watchdog timer. This layer is not discussed in detail since it is based on well defined principles [9].

The packet address header at the link layer is quite simple since it is a point-to-point communication. One thing worth mentioning is the link status vector which is a bit-mapped link indicator packaged into an integer. During the packet exchanges, the link status vector provides to adjacent peers about the link status of third party peers as well as about themselves.

3.3 The FTAT Group Communication Service Layer

The FTAT network layer provides the service primitives that enclose several rounds of group communications, fault-detection, fault-masking, on-line fault-repair and reintegration after repair. In addition, they bring the replica processes over the peer processors into synchronization. The layer has three logical levels packaged in it for parallelism.

As mentioned earlier, the replica processes of an application must be uniquely identified individually and collectively. The FTAT service calls on behalf of the replica processes must also be uniquely identified. With these identifications, the group communication among the peers can be recognized as either scatters or gathers, and processed accordingly.

Each FTAT layer packet has an address header consisting of 3 parts: the originator (original sender on the message), the source (the last sender which could be a forwarder), and the receiver (the eventual receiver). As part of message identification, each packet has a logical ID which consists of the message ID, process ID, and FTAT No.

There are two types of packets flowing in an FTAT: message packets, and control packets. Control packets are those such as ACKs, NAKs, and SPECIAL; the latter is used for prevention of loop sending during dynamic packet rerouting when the destination

is not reachable.

To achieve reliable and efficient group communication through multiple link connections, a concept of **activity observation** is devised. The concept of **activity observation** is basically the use of extra ACKs from the recipients as the observations of scatter operations. Here a group send (scatter), or a group scatter of the related data packet(s) is defined as a **logical activity**. The use of ACKs from the group member recipients results in low cost and high reliability in dynamic packet routing in the event of link failures. The concept also enables the system to detect peer application failure and to result better timeliness in group synchronization.

3.4 Group Communication

A group communication mechanism is essential to enable the group members to communicate and reach agreement on the state of the applications running on top of the system despite random communication delays and failures. The mechanism should make the behavior of the group indistinguishable from that of a non-redundant processing module [17] in order to implement the desired replication transparency.

Much research has been done recently on group communication. The V system [20][23] exploited the use of one-to-many group communication. Replicated procedure call in [25] was developed for many-to-many pattern calls including one-to-many call from each group member and handling many-to-one call by each. The group communication services needed in this thesis should be able to handle both one-to-many and many-to-one calls through the multiple links. However, the information distribution is among the group members instead between different groups.

Multi-link group communication provides an essential means for redundant data distribution. A many-to-many communication call involves two subproblems: 1) a group

broadcast from one-to-many through all the internal links; this is termed here as "scatter" of the message, and 2) the handling of a many-to-one call from the point of view of a receiving group member; this termed as "gather". The peer processors in the FTAT form a group. In [19], the issues related to the group send and receive are identified as follows:

- Scatter should be a blocking call so that the return point is the synchronization point for the group.
- Gather must solve the following two problems:
    1. The receiver must be able to distinguish unrelated call messages from ones that are part of the same group call.
    2. When one call message of a group call arrives, the receiver must be able to determine how many other call messages to expect as part of the same group call.

Although the applications and services in our system are performed on the same set of machines instead of in the form of RPCs (remote procedure call), the same issues also apply. The multiple linkage among the group members makes the problem more complicated. A link failure in a local area network may result in a network partition. However in the multiple link case, a network can still be connected.

A complete swap exchange involves the following steps among all peer modules of a group through the internal links within an FTAT:

- Each group member scatters the message to all other peer modules.
- Each group member gathers the messages from all other peer members.

- Each group member scatters the ACKs to all others in response to each data packet received.
- Each group member gathers the ACKs from all others.

Although the third step does not seem necessary here, it is designed on purpose for the concept of **activity observation**.

To detect these failures, normal techniques are based on timers. First, link failure can be asserted when no communication is heard on a link for a period of certain time. Secondly, a processor failure exhibits failure of all links and vice versa.

Arbitrary failures may mean that the faulty peer system is malicious. To simplify the design and to concentrate on the issues identified in the introduction of this thesis, arbitrary failures in the system are not considered in our design. When this type of faults occurs, the protocol here may fail to work.

In this research, the focus is on the highly efficient multi-link parallel group communication, the transparent fault-tolerance services and the on-line forward fault-repair. We assume that the FTAT network layer have the omission failure semantics [17]. This assumption can be implemented by employing self-checking technique to the FTAT network layer to avoid arbitrary failures. The Petri-Net based Observer suggested in [14] can be one of the candidate techniques.

3.5 Activity Observation

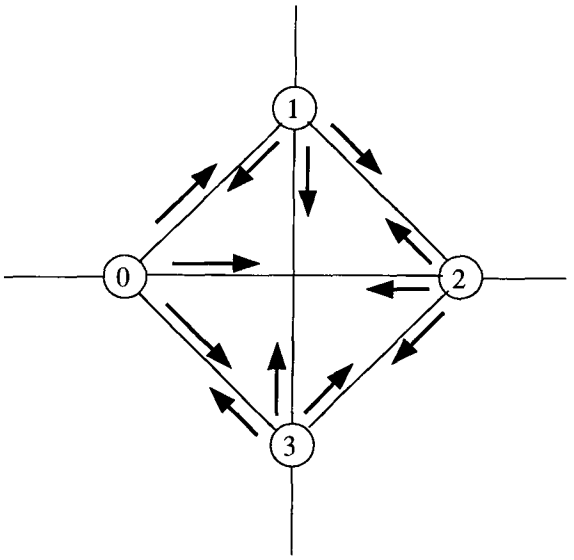
The concept of **activity observation** is designed to help the system to handle link failures, processor failure, and peer process failure in group communication. These failures may affect the membership of the related peer in group communication and hence the correct operation of the protocol.

The concept of **activity observation** works in the following way. When a peer initiates a new logical activity, it scatters the data packet to the group peers. The receivers scatter the ACKs in response to a data packet packet in order to quickly spread the observation of the activity. **Activity observation** serves two purposes:

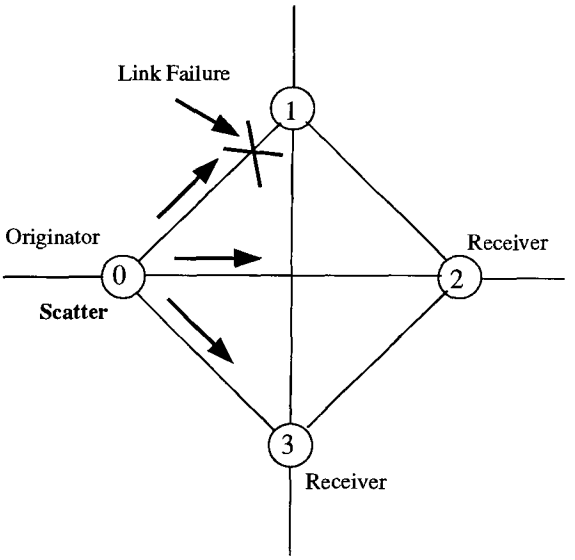
- To achieve efficient and reliable packet delivery in the event of link failures or processor failure.
- To help identify the application, whose participation is expected by the others of the group, from the list of applications running on the system.
- To detect the failure of a peer process whose attention is expected by the other peers in group communication.

Since a packet may go through a third, or even a fourth peer module, to reach the destination in the face of link failures, the receiver must be able to quickly identify the link through which to ACK the packet forwarded by the last sender. The ACK packet keeps the originator and the logical ID of the message as well as the last sender. The originator and the logical ID together identify the scattered message and the last sender tells where the ACK is to be returned to. It is very likely that the last sender remains the best route to ACK a claimed packet. This way, when an observed ACK is received the observer gets enough detail about the activity of the observed peer application to respond correctly.

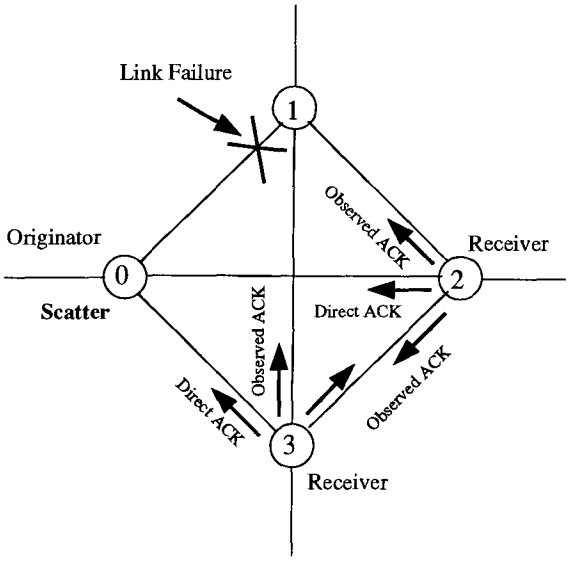
Figure 3.3 describes the relations among the peers during a scatter operation and explains the concept of **activity observation**. When a packet is scattered, each receiving peer, upon the correct reception, will **scatter** an ACK to the other peers including the originator (original sender) and the other two peers which act as packet observers. The extra two ACKs serve as the observation of the **logical activity**.



1. A complete logical swap in an IRPCWrite operation.



2. Scattering of a logical swap operation from node 0.



3. ACKing a scatter from node 0 and observation of ACKs.

Figure 3.3: The relations among the FTAT peer modules during a scatter operation.

Since each "scatter" is paired with "gathers" on the other sides, the pairs enforce the synchronization among all the peers. At the receiving side, the "gather" is completed when all the expected packets have arrived from all the other peers. On the sending side, when all the direct ACKs have arrived at the originator a scatter operation is completed.

If a peer module does not receive a packet due to link failure but received the related ACK from another peer, the receiving peer can now actively claim the missing packet from the known recipient instead of waiting for originator to reroute the packet. The logical ID and the originator ID, carried in the observed ACK packet header, can identify which of the data packets gathered is being claimed.

The use of the extra ACKs yields redundancy of a sending activity in the form of the control packets. The status about the other peers in the group communication can be known from these ACKs. In addition, the scatter and gather operations are all engaged with all the internal links in a "symmetrically complementary" pattern, i.e., all receivers listen to and the sender talks to all the working internal links. This form of balance helps functional clarity of the scatter and gather services in the system design and enhances the timeliness of the group synchronization.

It is necessary to keep the process executions on all the peers in close step for real-time processing. The synchronized execution can produce better timeliness, better throughput and deterministic behavior of the group communication. For real-time systems, timeliness is most important. In contrast, in an asynchronous system a fault can not be detected promptly and an latent fault may result in great waste of computation due to a possible chain of roll-back recovery.

As a result, the following advantages are obtained:

- low cost in transferring ACKs for packet rerouting.

- optimal routing of a claimed packet from a known recipient with an adjacent working link.
- in case of multiple link failures, dynamic rerouting of a packet may be done in parallel since it is a receiver-initiated operation.
- prompt detection of lost packets.
- effective timeliness in synchronization, which is essential for real-time processing.

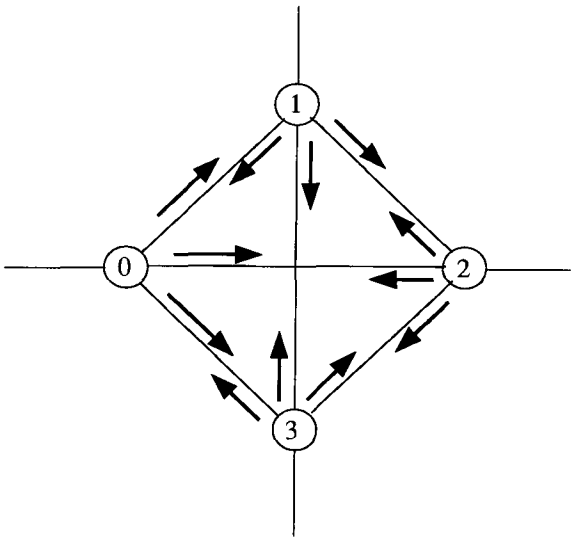
3.5.1 Operation When All Peers Are Available

As mentioned earlier, scatter is an operation that broadcasts a data packet to the rest of the group members as receivers.

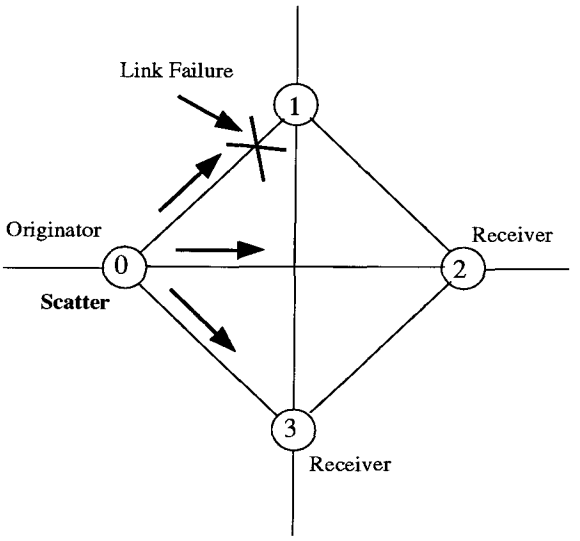
When a peer scatters a packet, the receivers scatter ACKs through all internal working links connecting the other 3 peers. The extra two ACKs would seem to be an extra overhead in the operation. However since they can be broadcast in parallel, only a small increase (caused by two extra ACKs) in time is seen. Moreover, the redundant ACKs help the distribution of data reliably and efficiently through the concept of **activity observation**.

At the receiver side, when all ACKs have arrived and the packet is received and ACKed, the end of the **gather** operation in response to a scatter is triggered. On the other end, receipt of all direct ACKs by the originator, will trigger the end of the scatter. In this process, each receiving peer will receive 1 message and two observed ACKs, and the originator will receive 3 direct ACKs in response to the scattered message.

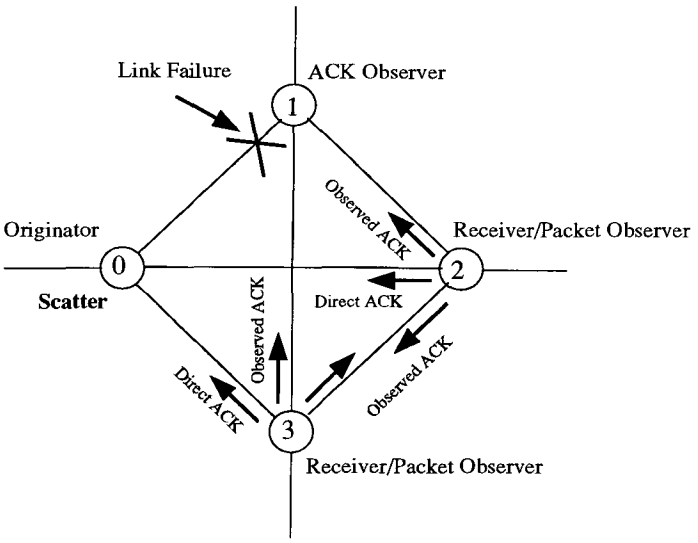
3.5.2 Operation When a Link Failure Occurs



1. A complete logical swap exchanges the same set of logical messages



2. Node 0 is the originator of the scattered message.



3. Node 2 and 3 are the receivers of the scattered message, but node 1 is only an observer.

Figure 3.4: Observing undergoing activities

When a link failure occurs in an FTAT, such as the case shown in Figure 3.4, the watchdog will detect it eventually after at most two probing periods by the link layer. After the scatter operation is started, the peer on the receiving end of the failed link may observe the ACK(s) from the adjacent receiving peer(s). When link failure is detected, the peer system is ready to use the Observed ACK to claim a lost packet from a known recipient of the packet. The above process is illustrated in Figure 3.5.

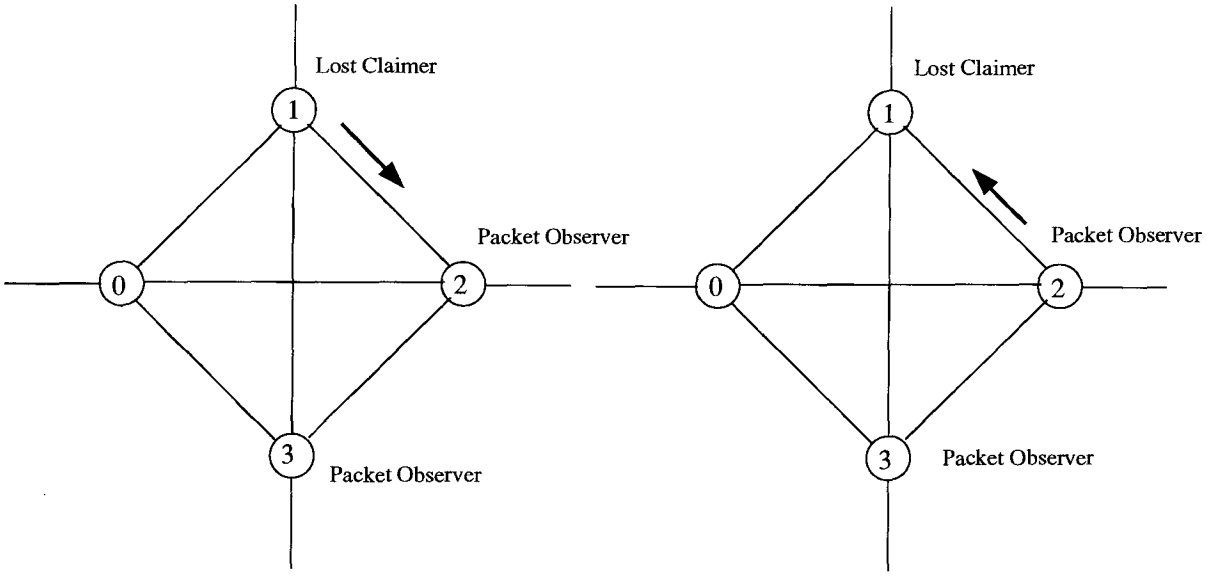
When the lost packet is eventually received, the direct ACK in response to the packet must be received by the originator since it confirms the correct reception of the packet concerned. Indirect ACKs are not important because they serve only as the observation of a reception for possible lost packet claim or identification of the application.

Only direct ACKs must be delivered to the destination even in case of link failures because of its importance to the originator. They represent memberships in the group communication. The direct ACK is handled in the following way. The claimer of a lost packet first returns the ACK to the forwarder since it is very likely that the latter still has a working link to reach the originator. If the forwarder becomes no longer reachable, the only link left is tried. Nonetheless, new link failure may occur and a known recipient may not be able to reach the originator. In section 4.7.1, the further link failure handling is discussed.

In the service mechanism, there is a filter process that forwards any packet to an appropriate link if it is not destined to the peer module where it has currently arrived. Within the address header of a packet, the source ID part of the header tracks the last sender as the packet gets delivered to peers along the route.

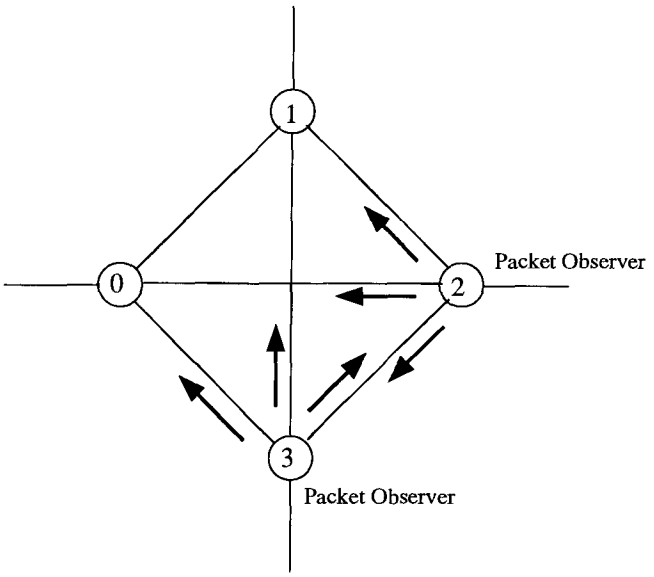
3.5.3 Operation When Multiple Link Failure Occurs

Multiple link failures may result in a network partition. In the system, an even network partition is not considered in our design for simplicity. Processor failure is assumed when



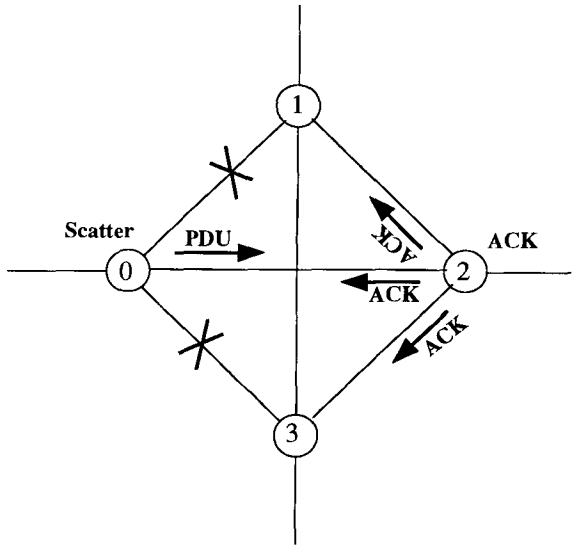
1. ACK Observer claims a lost packet from Observer

2. Observer forwards the packet in response to LostClaim request

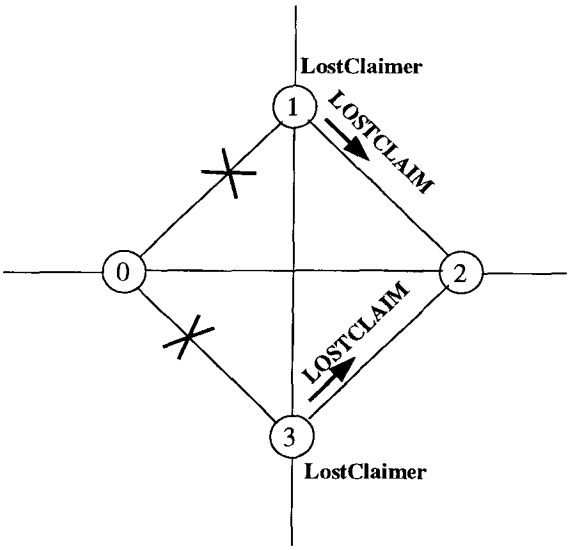


3. Observer waiting IPCWRITE operation initiated from inside

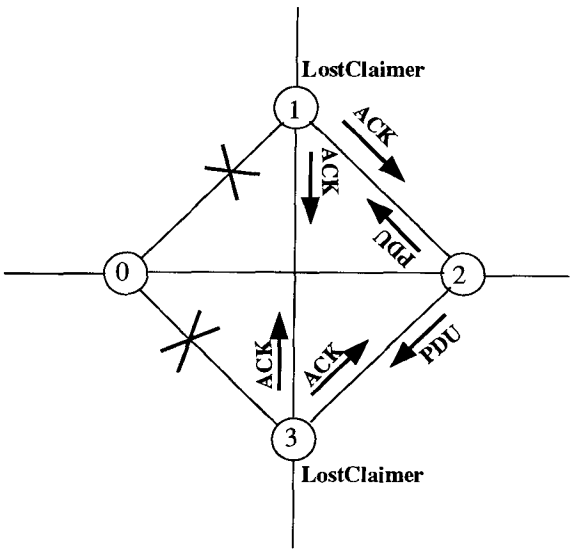
Figure 3.5: Lost packet claiming



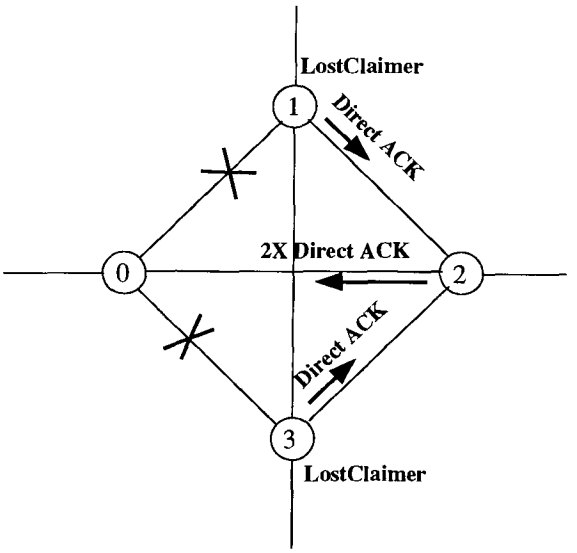
1. Node 0 scatters the PDU, Node 2 receives and ACKs it.



2. Node 1 and Node 3 observe and claim it.



3. Lost claimers ACK the PDU received.



4. Direct ACKs are returned to the originator.

Figure 3.6: Multiple link failure handling

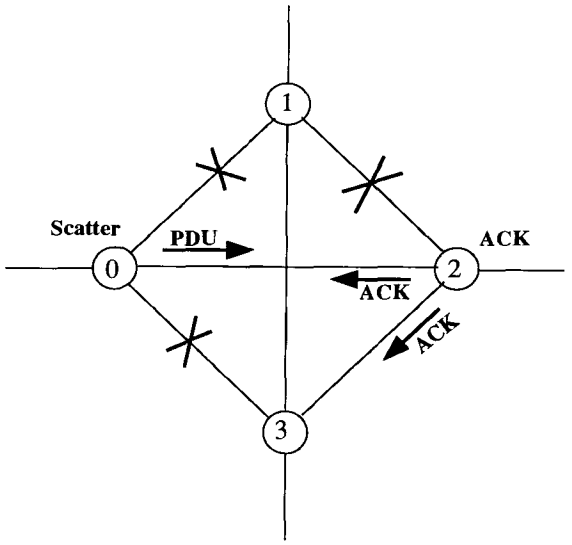
all its internal links appear to have failed because an all-link failure exhibits the same behavior as a processor failure does.

Consider first the simple case of a processor's two links failing as shown in Figure 3.6. After the peer scatters a data packet to the available working links, only one peer receives it. ACKs are then scattered by the recipient to all the working internal links in response to the packet. Now if the other two links to the recipient are working, the other two peers observe the reception through the ACKs.

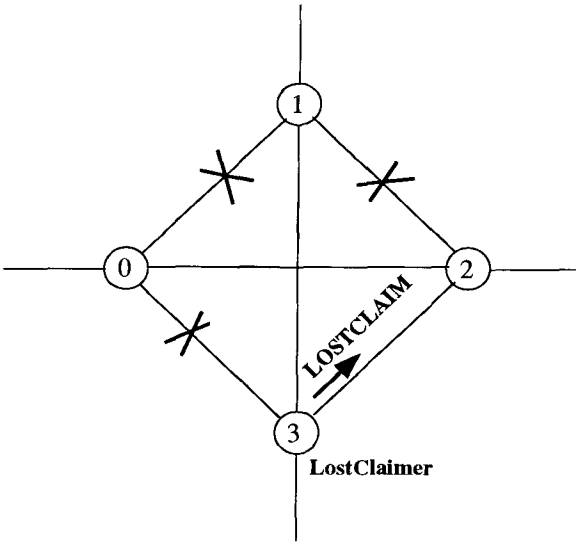
A simultaneous claims for the packet observed from both receiving peers are now possible. The recipient sends the data packet to the two peers in response to the reception of the Lost Claim packets. The ACKs returned to the originator (direct ACK) are also directed to the recipient as a known peer reachable from the originator. This shows that this method may result in possibly better performance (parallel vs sequential) and hence is much better than single sender-controlled rerouting.

When the ACKs are scattered from the lost claimer, the ACK destined to the originator is first sent to the peer that provided the claimed packet. It is then forwarded by the peer who should have a working link to reach the originator, provided that no further link failure occurred so far. If the link between the two claimers is working, the observed ACK from either should be received by the other. By now, both claimers should have gathered the set of packets, i.e., one message packet and two observed ACKs. The previous recipient should now also have gathered the packets in response to the scatter, and the originator should have gathered all the direct ACKs.

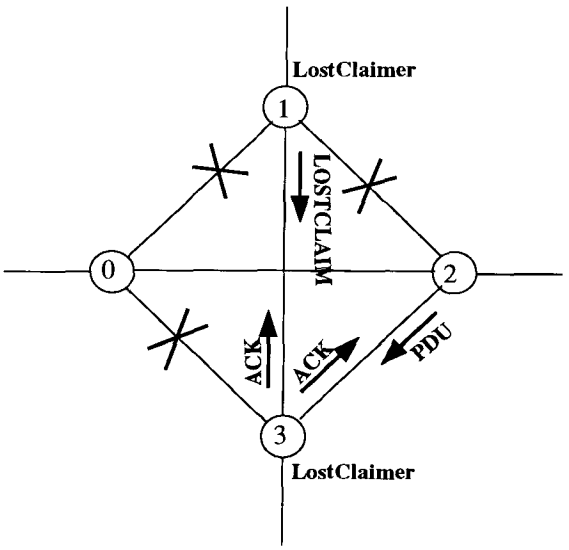
The second scenario in Figure 3.7 is the case of link failures which result in a single line connection in the network. This is the worst case for the performance of the FTAT protocol. This time the third peer will first observe the reception of the packet. A successful claim by the claimer from the second peer triggers the scatter of the claimer's ACKs to all the working internal links. The direct ACK is routed to the second peer.



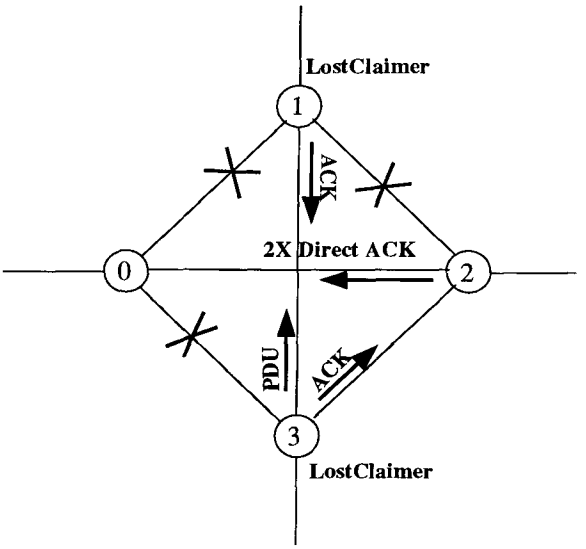
1. Node 1 scatters PDU and Node 2 receives and ACKs it.



2. Only Node 2 receives PDU and Node 3 observes and claims it.



3. Node 3 receives and ACKs the PDU, and Node 1 observes, and claims it.



4. Node 1 receives the PDU and ACKs the reception of the PDU

Figure 3.7: Multiple link failure handling 1

Now suppose the fourth peer has observed the reception by the third. The above process applies again. This time, since there is no link connection between the second and fourth peer modules, only the ACKs to the third peer and the originator are sent while the one to the second is dropped according to the protocol. Eventually the direct ACKs will reach the originator as long as the network remains connected.

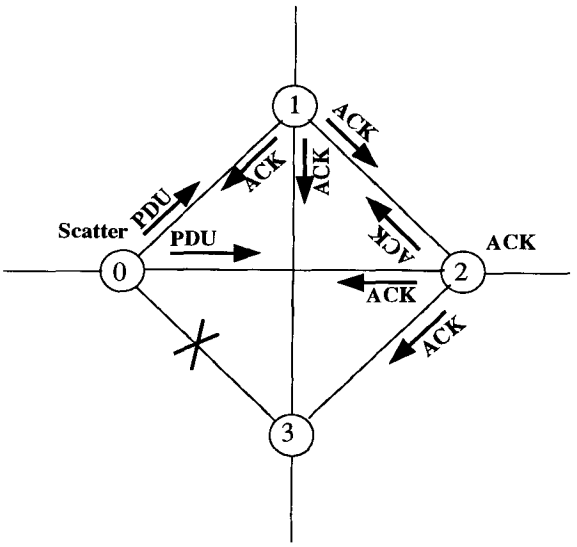
In Figure 3.8, the direct ACK is returned to a claimee but this time the link is not working any longer. The algorithm is designed so that if any other link is still working, then the ACK is forwarded to that link, otherwise the ACK will have to be returned to the link through which it came. The original ACK sender now will have to try the only link left to the peer to reach the originator.

To prevent a possible loop in sending between the claimee and the claimer given the link between them is still working, the ACK is turned into a **special** packet so that it can be recognized and dropped if it is to be sent again. Since there is only one chance left, there is no need to send the ACK if the last attempt has failed.

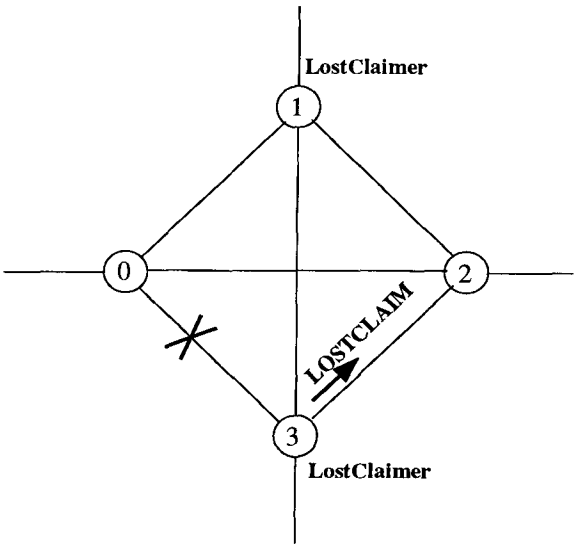
3.5.4 Operation When A Processor Failure Occurs

When a processor fails, we shall assume in this case that when a peer system has crashed nothing can be heard from any of its links. A peer system should always be ready for processor failures so that time is not wasted on idle waiting. Processor failures are always checked first before a decision is made to wait any further for the events from the corresponding peer module.

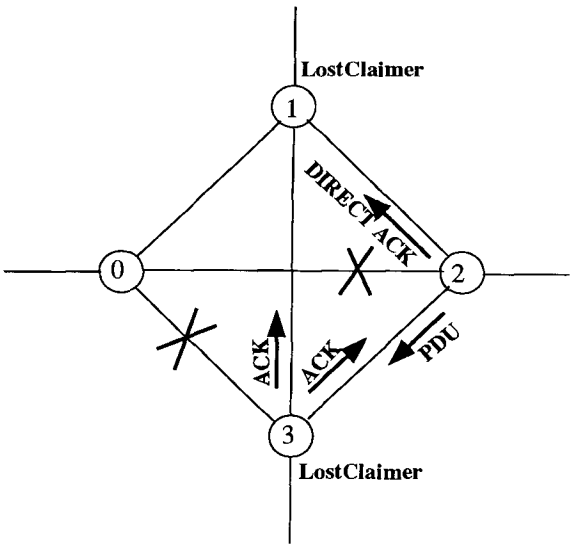
The knowledge about the status of a peer is also available from the other peers. The detection of a processor failure is essential in order to stop meaningless waiting or sending attempts. For this purpose, at the lowest layer, each time a packet (data packet, control packet, or probe packet) is exchanged, the current knowledge about link status of all peers is attached to the header of the packet. Each peer periodically updates the link



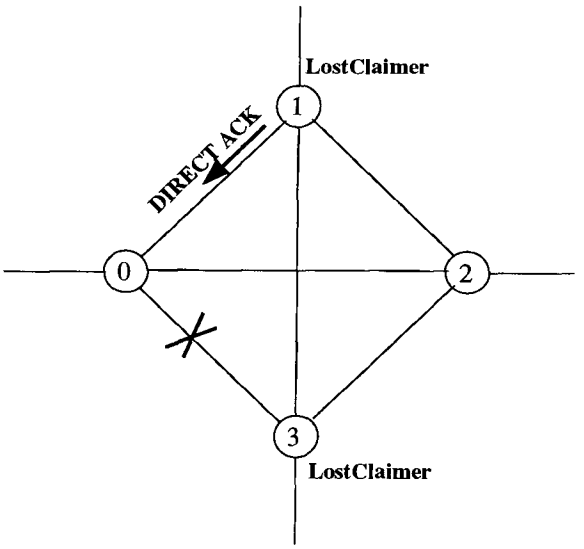
1. Node 0 scatters the PDU, and Node 1 and 2 receives and ACKs it.



2. Node 3 observes and claims it.



3. The link to reach node 0 from node 2 has failed. The Direct ACK is routed to node 1.



4. Now the Direct ACK can reach the originator.

Figure 3.8: Multiple link failure handling 2

status vectors as packets get exchanged with other peers. A failed peer processor will be excluded from the group membership in the protocol operation until it is repaired.

3.6 Timer Handling

In the detection of a packet loss due to link failure or processor failure, timers are usually involved in a message-passing multi-computer system. In [4], timers associated the group communication introduce difficulties in group synchronization. Packets may come in early, or late within the time out period because of the autonomy of each peer and the transputer communication protocols. If only one timer is used for group receiving, an early packet can trigger the timer, but a late packet from the other two links can also trigger the time out [4]. A second timer is proposed so that a two phase timeout can be used to allow early packets or late packets resulting from the looseness in synchronization.

Because ACKs are used extensively for **activity observation**, more timers would be needed for each of the packet sent on each link. This would increase more complexity and overhead in the synchronization management. In our system design, the use of the link status eliminates the timers associated with "scatter". However, this requires that the underlying communication service ensure the correct delivery of the packets once they are on route unless the link has failed. This is achieved by our link layer service.

To detect link failures and peer failures, a watchdog mechanism is built in the link layer. There is a watchdog monitoring the link status. With the link status and the concept of **activity observation**, the FTAT network layer handles link failure and processor failure in the communication.

A watchdog timer is associated with each link and gets reset when a correct packet is received from the corresponding link. When a pending link output queue is empty, the write worker will send a probe packet to each link on a periodic basis. Once a link failure

is detected, the associated failure flag is turned on.

Another set of timers is employed to detect peer processes' failures. Each application process has an associated timer. If a packet as an observation of a new FTAT service arrives at a peer module, an associated timer for the identified process is started so as to detect the replica process failure to participate in the group communication.

Chapter 4

FTAT Services with Transparent Fault-Tolerance

The fault-tolerance services provide transparency to the underlying architecture, data exchanges among the peer modules, and possible on-line forward fault-repairs. The system implements dynamic TMR with a spare (hot-standby). The resultant system can survive up to two concurrent faults and unlimited sequential transient faults.

Location transparency hides the hardware redundancy from programmers. Fault-transparency masks faults occurring in the underlying hardware. Replication transparency relieves the programmer from having to cope with the details of the cooperation and synchronization among the peer modules.

The FTAT service primitives are based the scatter and gather services. For efficiency and parallelism, the scatter and gather services are not implemented as service abstractions in order to reduce the overhead in the cooperation and increase the parallelism in the FTAT network layer.

The synchronization points of scatter and gather operations trigger the state transitions in operation of the FTAT protocols. Application processes are synchronized at the application service interfaces. These interfaces enclose a series of scatters and gathers operations, fault detection and fault-repair if any fault occurred. Special buffers with structured states are designed to facilitate these services.

4.1 Swap Operation Based on the Scatter&Gather Mechanism

Based on scatter and gather operations, the **swap** operation (all-to-all exchange) is defined. A **swap** operation involves symmetric scatter-and-gather by all peers in the FTAT and includes:

- scattering the data packet.
- receiving the related data packets from all other working peers.
- scattering ACKs in response to the reception of each of the related data packets.
- gatherings for all of the peers: one gather for the scatter and three gathers corresponding to scatters from the other peers.

The synchronization point is the point where data packets and ACK packets from all other peers have arrived. Due to the autonomy of each peer, which may be driven by different external events from the connected FTATs, each peer may be running different application process at a particular time, so some peer may be the first in the initiation of a logical activity. The **activity observation** can tell the other peer systems to bring up the related application process to participate. Eventually the replica processes will become active as result of the buffer allocations and round robin service fashion.

The swap service is formally defined in chapter 7 using Petri Nets. It is also shown to be correct.

4.2 FTAT Services

The network layer FTAT communication manager is based on the scatter and gather mechanism described in the previous chapter. The cooperation among the four peers

requires several rounds of information exchanges to ensure the correct operation of the protocols. The system services provide two primitives: IRPCWrite and IRPCRead.

For inter-FTAT packet routing, there is a routing table on each peer system that records the current network distances to the other FTATs in the overall system. This table is dynamically established as inter-FTAT packets are exchanged. It is not an major issue to be addressed and is not discussed in this thesis.

4.2.1 IRPCRead Service

For IRPCRead, the reading peer, on behalf of an FTAT with an adjacent FTAT, scatters the received packet to the other peers. It then waits for the ACKs to arrive from the other peers before ACKing the sending FTAT. This gives the sending FTAT an opportunity to reroute the packet in case the reading peer crashes right after the reception of the packet, thereby preventing packet loss. A receiving peer of an inter-FTAT packet is called "the initiator" of the IRPCRead operation. It does not ACK the sending FTAT until all the other peers of the receiving FTAT have correctly received the packet.

After the sending FTAT gets ACKed, all the peers of the receiving FTAT will start state exchanges and attempt, in turn, to deliver the packet based on the representative selection rule. The selection rule decides which next best peer to deliver a packet based on the current routing table. After the selected peer has tried to deliver the packet, a round of state exchange (swap state packets) is performed to update the current distributed states of the service until all attempts have failed or a delivery is successful. It is the receiving FTAT's responsibility to ensure the further delivery of the packet to the destination.

If a packet reaches the destination FTAT, the peers will try to deliver the packet to the receiving process. If it arrives at only an FTAT on its way to the destination, the peers will try one after another in turn by the representative selection rule.

4.2.2 IRPCWrite Service

IRPCWrite constitutes an output operation from an FTAT. The IRPCWrite service is similar to the IRPCRead except that the first step is a many-to-many complete data exchange among the peers of the outputting FTAT. A majority consensus is then performed on the data exchanged by all peers before the data is sent to another FTAT. The rest of operations are the same as those for IRPCRead.

An IRPCWrite packet becomes an IRPCRead at the next FTAT on its way to the destination. This is natural since the next FTAT is the reader of the IRPCWrite.

The FTAT services are based on the swap operations described in the previous section. Its formal definition of the protocol design is given and discussed in chapter 7.

4.3 Start of Service Operations

When an application process calls an FTAT service for either Inter-FTAT communication or fault testing, it gets blocked until the system finishes with the service and returns control. Before an application is rescheduled a resumption point is taken for later repair or control resumption. For details about the resumption point, please refer to section 5.2. There are two important lists in the system: the application channel list and the buffer set list. They are all system resources. The channels in the application channel list blocks the processes upon their calling for FTAT services. The buffer set list comprises of a list of grouped buffers for many-to-many pattern data exchange buffering.

Depending on the service called, the application waits on its corresponding associated channel. A buffer set must be acquired before the request can be processed. A buffer set is a set of buffers that can hold all data packets as well as ACKs from all the other peers for a group communication. The so called "structurally stated buffer sets (SS buffer) are designed for this purpose and is discussed in section 4.4. Only the service requests that

have acquired such buffer sets are processed by the system. These two stages are due to limited buffer resources and deadlock avoidance in the buffer allocations.

The peer system scans through the channel list looking for pending service requests. If such a request is found, a structurally stated buffer set (SS buffer set) for the service is assigned for it if any is available. The system then scans through the SS buffer sets to perform the actual service.

Once a service request is in processing, there are two types of swapping of packets: data packets and state packets. The first step of an FTAT service is the data distribution followed by a series of the state exchanges. To the scatter and gather service, the data packets and the state packets here are same.

4.4 Structurally Stated Buffer Set

Since there are limited system resources and the group communication services require a large amount of memory, only a limited number of service requests can be processed at any one time. Such a limitation must allow the avoidance of possible deadlocks due to resource allocations at the system level.

4.4.1 The Group Communication Buffer Consumption Requirements

Each transputer link is actually two separate channels, one for each direction. This avoids communication confusion, which results in failure of the link communication protocols. Concurrent in-bound and out-bound communications can be carried om the link. Each communication on an FTAT link represents a logical operation. To allow concurrent in-bound and out-bound communications with other FTATs, separate buffers are necessary.

To reach agreement on a group decision, synchronization among the peer applications must be enforced at some point. The observation of activities again helps to identify the

applications being served by other peer systems. Limiting the number of the requests being processed concurrently is a way to enforce synchronization of the group members on the services, provided that service fairness is ensured by the system.

When each service primitive is called, the calling process is blocked on the corresponding channel. Once a service call is being processed, all peers must be able to buffer all necessary information related to the service call for the purpose of fault-detection, masking and loss claiming. This data redundancy requires more than four times of memory than the single data packet does in an FTAT service.

If all service requests are processed concurrently, the limited memory resource on each peer module becomes exhausted. Therefore it is impossible to process all service calls at the same time. But what should be the minimum number of service requests that can be processed by the system concurrently? This question is resolved in the next section

4.4.2 The Deadlocks And Deadlock Avoidance

To avoid deadlocks, each service request to be processed must be able to acquire a set of buffers distributed over all the peer modules. This means that the initiation of a system service by a peer should be honored by all the other peers in order to proceed with the service request. Another interpretation is that all the peer systems should ultimately serve the same set of service requests in order to avoid cyclic waits among the peers.

A partial acquisition of a buffer set in the system means that different service requests are being processed by different peer systems. Cyclically dependent waits are consequently formed between parties, with different service requests being served. A deadlock could result because the peer has to wait forever for the attention of all the other group peers for the service while some of them are expecting another service from the first peer!

Due to the autonomy of each peer module, service requests from different applications

may be processed on different peer systems. To see how the independence of the peers may affect the system, imagine the following situation of an FTAT concerned in a mesh connected network. All the four adjacent FTATs are trying to pass something into the FTAT. Supposing that the FTAT peers all read at the same time, a SS buffer set is reserved for each of the abstract FTAT links on each of the four peers. If the system only has one SS buffer set, each peer allocates the SS buffer set for the FTAT it is responding to. A deadlock results from the above situation since no service initiative from a peer is honored by all the other peers within the FTAT.

To avoid deadlocks in buffer allocations, the solution is to ensure the same set of service requests are served by all the working peers. Since there are two concurrent channel communications (In-bound and Out-bound) on each link, there will have to be two separate distributed buffer sets.

As can be seen easily, there are at least four possible concurrent service requests being processed at a time for each type of service primitive in the system. Therefore the minimum number of buffers on each peer is four for each type of service primitive if only one service initiation is allowed on each peer. Through activity observation, the service requests being processed by the other peers can be passively started. All the peers eventually reach agreement on which service requests to process without much negotiation and any deadlock.

4.4.3 The Service Buffer Set Design

As mentioned before, different types of buffer sets are used for different services: namely, write buffers and read buffers. A Write buffer set require more memory than a read buffer does because it has to be able to hold packets exchanged with all the other peers.

In the system design, each peer system is allowed to initiate one service at a time for each service type. All observed services through the concept of **activity observation**

should also be processed during the same period. As a result, the processing of same set of service requests is always guaranteed. Hence, up to 4 services of each type can be dealt with by the system concurrently.

This buffer design helps the system to enforce synchronization among the FTAT peers. Because up to only four outstanding service requests can be processed at any time on the system, other processes are either blocked or in execution (ready state and eventually will be blocked). The absent processes, which have passively acquired a service of the system, will ultimately be brought up on each peer. Therefore, the peer systems are brought into synchronization.

In the design we assume that applications are responsible for avoiding deadlocks at its logical level.

Service fairness is also very important to ensure the timeliness of the system. As mentioned in section 4, there are two important lists in the system: the application channels, and the SS buffer sets. The requests from the applications should get fair services so that they can behave deterministically. The same is true for the structurally stated buffer sets. Consequently, both application channel scanning and buffer set scanning are processed in a round-robin fashion.

4.5 Structured States

As the services proceed, different events can occur. These events have impact on the later operations and have to be recorded as states. Because the three logical levels of the FTAT network layer are implemented as one, the states of the system are structured. A collection of similar events can contribute to the occurrence of some (aggregate) event. For each data packet sent on an internal link, 3 ACKs are expected for it. For each data packet received, two extra ACKs accompany it. The arrivals of the 3 direct ACKs trigger

the event ACKGathered.

The aggregate state should be efficiently triggered without constantly polling all individual states. For this reason, the states of the FTAT services are structured. For example, the end of a complete swap is indicated by the event that all data and ACK packets from all working peers have arrived.

The structured state transitions are triggered by the possible inter-FTAT events, changes of link status and processor status. Whenever a new event for some state has occurred, it is checked to see if the transition condition is met for its aggregate state. The system handles the inter-FTAT events similar to responding to interrupts.

4.6 Fault Masking

In the system, we assume that the network layer is implemented with omission failure semantics [17] so that the key issues identified in the introduction can be focused. When faults are detected in any of the peers, they will show up in the distributed status after the first round of the states exchange. The peers then attempt to deliver the packet in turn according to the Representative Selection Rule explained below. The correct result of the system is always present at any given time, provided that no more than two faults occur at the same time.

4.6.1 Selection of the Representative & State Swapping

Once the majority consensus is done in the FTAT, the peers will select a representative to deal with the next FTAT to deliver the output to the best possible route based on the dynamically established routing tables.

The distributed algorithm to select the representative peer is based on the states

exchanged among the FTAT peers after the data packet distribution. A state packet contains the result of the majority consensus for the application, and the distance to reach the destination FTAT from the peer. A peer is elected if its status is not marked "FAILED", its distance is the shortest and the module number the smallest.

4.6.2 State Swapping

After the majority consensus is done, the status of each of the replica processes is determined by the system. The first round of swap is performed with such information as the outcome of the majority consensus and the distance to the destination FTAT. Then the representative peer currently selected will try to deliver the packet to the destination on behalf of the FTAT, while all the others wait for the outcome of its attempt. The representative peer broadcasts the outcome together with the distance to the destination FTAT. This process is called **state swapping** because the data exchanged form the distributed states of the system.

After the attempt to deliver the packet, the status of the peer for the particular service is marked "FAILED" if the attempt fails due to link failure or processor failure. Such a peer will be excluded from the set of candidates eligible to serve as the representative later. The next one will then be selected. This process continues until an attempt is successful or all the peers have been used.

Here the state "FAILED" means the failure of a delivery attempt instead of the failure of a peer process.

4.7 Failure Handling

The FTAT protocols must be able to provide correct services even in the face of link failures and processor failures. If the failures are left undetected, the group members in

communication may be confused, and/or the group membership may not be consistent any longer among the peers. Consequently, the protocols will fail.

4.7.1 Link Failure

In the system, the transputer links are monitored by a Watchdog with associated watchdog timers. Each time a packet is received, the watchdog timer is reset. If the buffer queue is empty, the writer worker process will send "probe" packets periodically. If the link is dead, the timer will never get reset. Thus, eventually link failures will be detected by the watchdog timer.

Repairing a failed link involves resetting the link channel and rescheduling both processes. Normally a process blocked on a failed link never comes back. However, the blocked process has to be rescheduled in order to proceed. A simple link channel reset may result in the loss of the process since its break point is stored in the link channel port address. To solve this problem, we save a resumption point before calling the communication instruction. When the failure is detected, the channel can be reset and the blocked process is rescheduled by the system from the resumption point saved.

4.7.2 Processor Failure

Processor failure detection is not straight-forward since a conclusion can only be drawn based on the link status from the other working peers. During each packet exchange between the peers, the distributed link statuses are passed to the receiving peers. This link status vector is maintained by the system watchdog and updated by the peer system.

The entry for the local peer in the link status vector is maintained by the watchdog. Whenever a link failure is detected, the corresponding bit is set to indicate the failure. The other entries in the vector are updated by the peer system when such a vector is received from another peer. The entry for the provider peer of the vector can be directly

copied. However, if a link fails, no link status vector can be received from the link. When a link failure is detected, the peer on the other side becomes "suspicious". In this case, the entry has to be worked out based on view of the third party peers. Only when all the other peers shows that its link with the suspicious peer has failed, then the failure of the peer can be asserted.

To ensure that the protocol works properly, the updating of the link status about an unreachable peer is designed as below:

```
/*************** Link Status Vector Updating Algorithm ***************/
; Peer is the sender of the currently received link status vector.
; Local is the receiving peer.
; LinkVector is the local Link Status Vector.
; ReceivedVector is the received Link Status Vector.

FOR ( PeerScan = 0 TO 3 )
    IF ( PeerScan = Peer ) CONTINUE;
    IF ( PeerScan = Local ) CONTINUE;
    link = (PeerScan - Local) MOD 4;
    IF ( LinkVector[Local][link] = BAD )
       IF ( ReceivedVector[Peer][(PeerScan-Peer) MOD 4] = BAD )
   LinkVector[PeerScan][(Peer-PeerScan) MOD 4] = BAD;
       ELSE
   LinkVector[PeerScan] = ReceivedVector[PeerScan];
   AnotherPeer = Peer XOR PeerScan XOR Local;
   IF ( LinkVector[Local][(AnotherPeer-Local) MOD 4] = BAD )
```

```
        LinkVector[AnotherPeer][(PeerScan-AnotherPeer) MOD 4]
    = LinkVector[PeerScan][(AnotherPeer-PeerScan) MOD 4];
    ENDIF
        ENDELSE
     ENDIF
ENDFOR
```

Using the link status vector, the status of a peer can be determined be the algorithm given below:

```
/********** Node Failure Detection Algorithm *************/
; Peer specifies the "suspicious" peer
; Local is the receiving peer.
; LinkVector is the local Link Status Vector.
; NodeStatus is the Node Status Vector.

NodeVector[Peer] = BAD;
IF ( LinkVector[Local][(Peer-Local) MOD 4] = BAD )
   FOR ( PeerScan = 0 TO 3 )
        IF ( PeerScan = Peer ) CONTINUE;
        IF ( PeerScan = Local ) CONTINUE;
        link = (PeerScan - Local) MOD 4;
        IF ( LinkVector[PeerScan][link] = GOOD )
   IF ( LinkVector[PeerScan][(Peer-PeerScan) MOD 4] = GOOD )
NodeVector[Peer] = GOOD;
EXIT;
```

```
   ELSE
AnotherPeer = Peer XOR PeerScan XOR Local;
IF ( LinkVector[PeerScan][(AnotherPeer-PeerScan) MOD 4] = GOOD )
     IF ( LINKVector[AnotherPeer][(Peer-AnotherPeer) MOD 4] = GOOD )
 NodeVector[Peer] = GOOD;
 EXIT;
     ENDIF
ENDIF
   ENDELSE
       ENDIF
   ENDFOR
ELSE
   NodeVector[Peer] = GOOD;
ENDELSE
```

To see that these algorithms are necessary, let us examine the following scenario: the peer concerned is trying to determine the link status of a third party peer A. Suppose the link between the two has failed. Peer B is also not reachable from the peer concerned, but peer C is. The link status vector provided from peer C shows that the link between A and C is also dead, how the status of peer A can be worked out. The peer can not determine the status of peer A until all other working peers are consulted. If peer B sees peer A is alive, then peer A is alive since it is still connected to the system.

Chapter 5

On-Line Forward Fault-Repair

The on-line forward fault-repair in the system aims at exploiting the spatial redundancy to repair faulty application processes. Forward fault-recoveries for real-time systems are achieved by using the volatile data redundancy, without checkpointing and roll backs, to handle transient and intermittent physical faults. To accomplish such fault repair, some of the issues that are common to Process Migration [23][24][25] must be solved, as identified in the introduction. The challenge here is that the fault-repair has to be able to handle reference rebinding problems [23], without use of shared virtual memory. We have developed a scheme to separate the application processes from their run-time physical environment so that their contexts are movable for fault-repair.

For on-line fault-repair, both the availability and the reliability of the system are to be maintained. In this research, the repair aims at not only reconfiguring the redundant modules but also retaining the availability of the redundant modules, unless permanent failures have occurred to the majority modules.

The system recovers the faulty processes by using the redundant volatile data in the run-time environment, the resource management information during the run-time of the system and the Pointer Reference Stack (PRS). Some restrictions to the applications on the system are imposed in order to focus on the development of the fault repair mechanism and reduce the complexity of the processing:

- Processes are required to be loaded and created statically in the same order.

- No global sharing data are allowed among the processes.
- Global data variables for each application process have to be bound in a single data structure.
- Physical pointers, allowed in a process, are only those created on the run-time stack.
- Physical pointers are not allowed in the heap allocations, but can be replaced by a logical pointer implementation.
- Global physical pointers are prohibited in the programs.

However these restrictions are not necessary for volatile fault-repair. Any of these restrictions can be removed at more processing cost.

5.1 Issues in Fault Repair

For fault-repair from the system level, the semantics of the high level logical faults are difficult to know. As a result, it is hard to do an incremental fault recovery and full recovery has to be employed. In addition, the repaired process has to be reintergrated into the system without interfering with smooth running of the system.

Many fault-tolerant systems use check-pointing and roll-back schemes. As mentioned earlier, these types of systems pay a high cost in check-pointing by logging or journaling [7][8] through persistent storage. In addition, the system has to undo previous operations through roll backs until the effect of a fault is totally eliminated. Such recoveries may require a chain of global roll-backs when distributed group cooperations or collaborations are involved. However, the time previously spent for the operations being undone is wasted.

To preserve the timeliness of the system, we use the volatile execution image to repair the faulty process. This approach is especially useful for fixing transient faults which cause only damage to the volatile states of a processing module.

Using the run-time context of a process on one peer module to repair a faulty process on another is not simply a matter of copying the run-time context. A difficult problem that needs to be solved is pointer reference rebinding [23]. In the V system [23] and the Accent system [25], shared virtual memory is used to deal with the pointer handling.

A shared virtual memory implements disk-based (persistent) data objects. It is usually based on a memory page-addressing mechanism so that the essential context can be used immediately. A non-existent pointer reference can be "paged in" later from the shared virtual memory upon a page-fault interrupt. The data of the pointer reference is then copied to the local memory cache.

This seems to be a simple solution but it burdens the system in the following ways:

- Shared virtual memory itself suffers expensive overheads in maintaining the data consistency between memory cache and disk storage distributed over the machines [2].

- Since the run-time process contexts are also persistent data objects, copy-on-demand [23] may result in extra disk I/O access. In addition, the startup costs becomes observable for intermittent disk I/O access if the reference straddles several memory pages due to the nature of copy-on-demand.

- Shared virtual memory can cache a virtual memory page only on a page basis. This means more intermittent remote virtual memory page requests to be processed, which takes more time than does consecutive loading of the context.

Volatile fault-repair is difficult since the run-time pointer references are hard to cope with. When pointers are copied to another machine they become meaningless unless

both machine carry exact memory images. They may be pointing to different data in stead of those that are bound to the process.

5.2 The Processing of The Fault Repair

The fault repair is started by the peer system which has detected a fault in a peer application process. The peer sends a repair request to one of the non-faulty peers and the repair is started. While the repair is be performed, the other peers wait until the fault repair is completed.

The consistent volatile part of a replicated process can be provided by the masking redundancy FTAT and can be used to fix a faulty peer process. While the corresponding memory parts can be copied from one process to another, pointers have to be readjusted to point to the same logical data because of the different memory images on the different peers due to their autonomy. The problem is how the pointers can be handled correctly and efficiently.

By the assumed restrictions, each peer should have the same image for the code part as in the memories on all the peer modules. Since the global data is bound in a single structure and contiguously laid out in memory, the only things needed are the reference and the size of the structure. All the global data of a process can be given by a single reference, moved around and updated as one entity without having to know the sizes and semantics of particular data. The information about the global data is part of an entry in a process management table. Process management tables are maintained by the system for the conventional process management [33]. All memory heap allocations for each process are maintained in its associated resource allocation table. The resource allocation tables, common to operating systems, provides the sizes and addresses of heap allocations

A repairing table is prepared by the repairing peer when the repair is started. This table contains the information such as the address and the size of the run-time stack, information about the resource allocation table, the address and size of the global data for each process, and information about the Pointer Reference Stack (PRS), etc.

The table is then packaged and delivered to the faulty site, followed by the run-time stack, the global data for the process, the heap allocations and the PRS. The resource allocations bound to the faulty process are released and then reallocated to be certain of the same resource allocations for both the faulty process and the repairing process. The received copy overrides the corresponding items at the faulty site. The pointers used in the process are then relocated to complete the repair.

On a transputer, when a process is rescheduled, the microcoded scheduler of the transputer pushes all the registers onto the stack of the process and the stack pointer, work space pointer [34], yields the resumption point of the peer process. A resumption point of the repairing peer process is taken before the FTAT service is started as stated in section 4.3. Because the repaired process is consistent with the repairing process, this resumption point is used for the repaired peer process to resume control.

In essence, the correct handling of pointer references must ensure the following two conditions exist between the two parties involved, after the fault-repair:

- The corresponding referenced data in both parties should be consistent.
- The corresponding pointers should point to the same logical data.

As long as these conditions are met, the repaired process will be in a consistent states with the other peers, no matter what have happened to the pointers and their referenced data in the past. Therefore no "undo" or roll-back is necessary.

5.3 Handling Resource Rebinding Problems

In order to be able to move the process context around among the peers, the system has to keep track of all the resource allocations, the birth and death of the pointers on the stack, in addition to those for the regular system management.

From the assumed restrictions, use of global pointers are prohibited in applications for simplicity. Pointers in heap allocations are not allowed but can be replaced by the logical number implementations. The pointers on the run-time stack are dynamically established during the run-time of a program. Only pointers used in the current execution path appear on the run-time stack. Other pointers vanish as the stack pops after return from procedures.

The resource allocation information is maintained in the resource allocation tables by the system when a system resource is allocated to the application processes. There is one such table for each application process. Only memory heap resource is considered in our system.

Only those pointers currently used along the process execution path are relocated. Hence, the number of pointers to be processed is minimal. The PRS, the process tables, and the resource allocation table, together with the assumed restrictions, facilitate the separation of applications from the run-time physical environment.

From the fault repair process, the first thing to be relocated is the PRS of the faulty process, because the PRS holds a copy from another peer. It gives the list of the addresses of the pointers used in the process. The local PRS overwritten by the received PRS must correctly locate the pointers residing on the run-time stack. The addresses of the run-time stacks for both peer processes are available, so it is quite easy to relocate these pointers by the difference of their address values.

PRSs lists all the pointers that need to be relocated. These pointers may point to

different type of memories, including the global data area, the run-time stack, and the memory heap allocations. They should be processed accordingly. Global data references and references of data on the run-time stack are simple. The references of data in the heap allocations have to be processed corresponding to the referenced allocations.

Starting with scanning the received PRS, the referenced heap allocation is looked up in the received resource allocation table for a pointer given by the PRS. The identification of the allocation also locates the corresponding local resource allocation. The corresponding pointer from the local PRS can be then relocated by the information about the local resource allocation and the allocation in the received resource allocation table.

5.4 Pointer Reference Stack

The PRS is designed for tracking the pointer creations during the program execution so that the system can use the information in the PRS to handle a minimal number of pointers. The PRS records the pointer creations all the way along the process execution path. It contains information such as the pointer value, the type of resources pointed to.

Use of PRS can be automated through a compiler so that PRS operations can be done transparently. The introduced overhead is not noticeable to a program since it is only called twice (push and pop) per pointer per function call.

Chapter 6

Reliability Analysis

6.1 System Reliability

Reliability modeling for the system is derived with the following system parameters and module parameters:

System Parameters:

c – the probability that a fault in the system is detected.

r – the conditional probability that a fault is repaired after its detection in the system.

Module Parameters:

t – time

λ – fault rate during power-on.

The reliability analysis of the system is based on the assumptions given below:

- All modules are identical and have the same reliability.
- Fault distributions $R_m(t)$ for all modules are exponential and identical, i.e. $R_m = e^{-\lambda t}$.
- Fault detection and repair are done instantly.
- All modules are independent of each other except when a repair occurs.

- The time between two consecutive acceptance tests $\leq$ T, where T is some constant.
- All modules are symmetric with respect to each other since they are fully connected.
- c and r are constant.

To obtain the system reliability R_c^r, we first calculate the system failure probability F_c^r. The system failure probability has three components. The first part F_1 is caused by concurrent faults that occur during T. The second F_2 is the sequential fault effect that occurs over time t where $t \gg$ T. The last one, F_3, results from fault repairs, corresponding to the case when faults occur on two modules. One fault is detected while the other is not and the undetected fault is on the repairer. The successful repair then results in the propagation of the fault and ultimately the failure of the protocol.

The derivation of F_1

The faults that occur during the time between two consecutive acceptance tests, T, are defined as concurrent faults since they have to be dealt with concurrently. If more than two concurrent faults occur during the time period of T, the system crashes. F_1 is the probability that all four modules have faults or three out of the four modules have faults during T.

This is can be obtained from the probability that faults occurred on all the peers during a period T plus the probability that three faults occurred on any three of the four peers. The first term in Eq. 6.1 below determines the former while the latter is given by the second term.

$$F_1 = (1 - R_m)^4 + \begin{pmatrix} 4 \\ 3 \end{pmatrix} [1 - R_m]^3 R_m \quad \text{if } t \leq T \tag{6.1}$$

Substituting R_m with $e^{-\lambda t}$ in Eq. 6.1,

$$F_1 = (1 - e^{-\lambda t})^4 + \begin{pmatrix} 4 \\ 3 \end{pmatrix} [1 - e^{-\lambda t}]^3 e^{-\lambda t} \quad t \leq T \tag{6.2}$$

The derivation of F_2

The sequential faults are repaired by the on line repair mechanism. Transient faults can be tolerated and the faulty modules are made available again to the system after being repaired. The failure probability F_2 comes from the situation in which the system has suffered two sequential faults, but has been unable to detect or recover from these faults and a third fault occurs. In this case, the system can no longer perform majority consensus correctly based on the dynamic TMR scheme.

Since all modules are symmetric to each other, each of the sequential fault occurrence sequences has the same probability. There may be 4! such sequences, so F_2 is the sum of all of the sequences. The fault detection probability c and the probability of successful repair r can change the system failure probability substantially, as can be seen below.

$$F_2 = 4![(1-c)+c(1-r)]^2 \int_0^t \frac{d}{dt_1}(1-R_m) \int_{t1}^t \frac{d}{dt_2}(1-R_m) \int_{t2}^t \frac{d}{dt_3}(1-R_m) R_m dt_3 dt_2 dt_1 \quad t > T \tag{6.3}$$

where $(1 - c) + c(1 - r)$ is the probability that a fault is not detected, or is detected but not repaired on a peer module. After two such faults have occurred, a third module fault crashes the system immediately due to the dynamic TMR scheme.

Substituting R_m with $e^{-\lambda t}$ in Eq. 6.3,

$$F_2 = [(1 - c) + c(1 - r)]^2[1 - 6e^{-2\lambda t} + 8e^{-3\lambda t} - 3e^{-4\lambda t}] \quad \text{if } t > T \tag{6.4}$$

We can further simplify Eq. 6.4 and obtain the following:

$$F_2 = [(1 - c) + c(1 - r)]^2(1 - e^{-\lambda t})^3(1 + 3e^{-\lambda t}) \quad t > T \tag{6.5}$$

The derivation of F_3

F_3 is the probability that when faults occur on only two modules one of them is detected while the other is not and the latter appears to have successfully repaired the former using its run-time context. This condition results in the propagation of the fault on the latter and ultimately the failure of the protocol due to the dynamic TMR scheme. F_3 results from the cases when two and only two concurrent faults occur. It is obtained as given below under the assumption that the repairer is selected from the non-faulty modules with equal probability:

$$F_3 = 2 \times \frac{1}{3} \begin{pmatrix} 4 \\ 2 \end{pmatrix} (1-c)cr(1-Rm)^2 R_m^2 \quad t \leq T \tag{6.6}$$

The system failure probability is:

$$F_c^r = \begin{cases} F_1 + F_3 & \text{if } t \leq T \\ F_2 & \text{if } t > T \end{cases} \tag{6.7}$$

Thus the reliability is $R_c^r = 1 - F_c^r$. Here, Eq. 6.2 and Eq. 6.6 are probabilities over the time period T. Eq. 6.5 reflects the contribution to the reliability from the on-line forward fault repair. The higher values of the probability c and r, the lower the system failure probability.

Figure 6.9 shows that the reliability increases remarkably by the on-line forward fault repairing even when the fault-detection probability and the fault repair probability are lower than 0.5. The curve in "*" in the figure is the reliability of one module.

6.2 Network Connection Reliability

The probability that packet exchanges between a pair of nodes can still be conducted in the event of link failures in the network is defined as the connection reliability R_c.

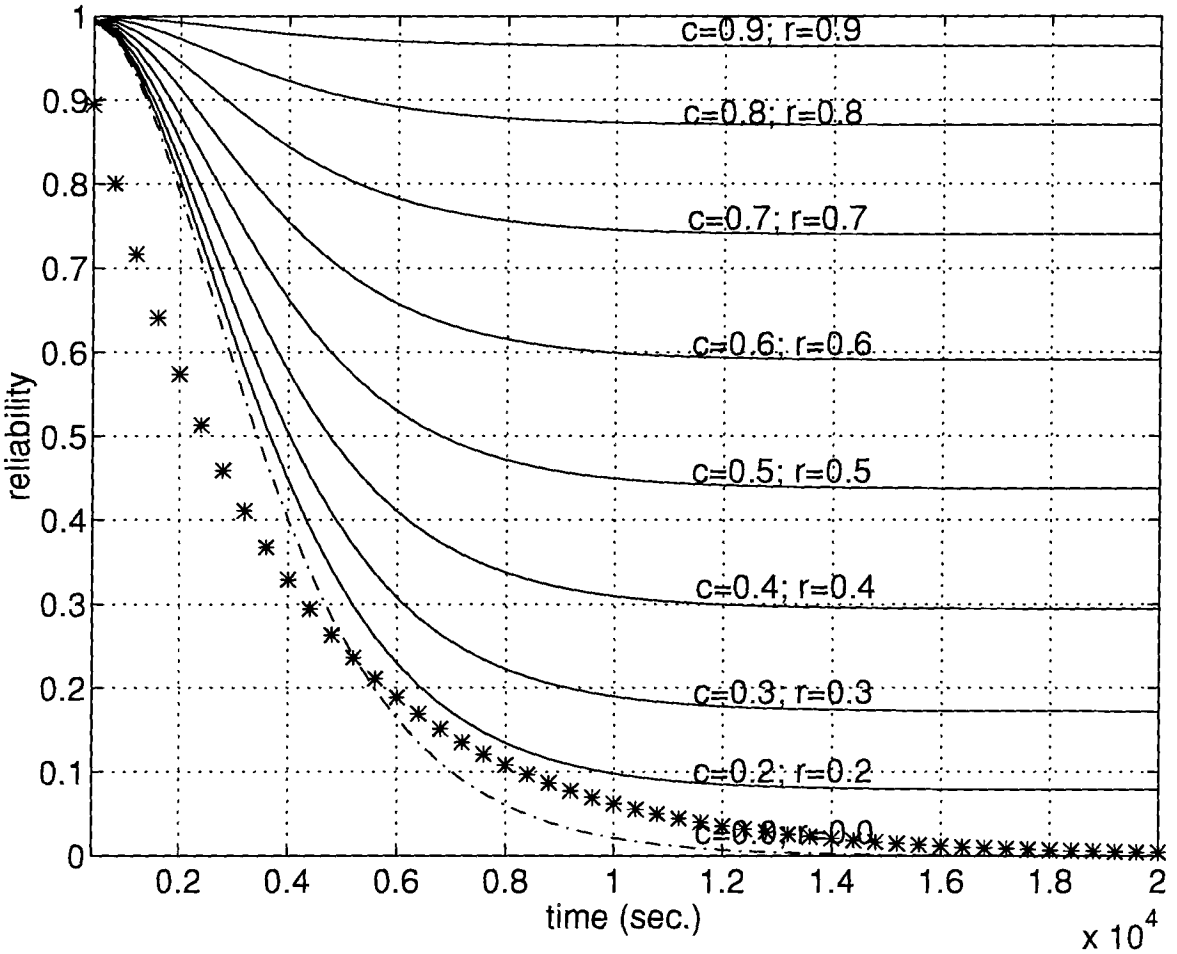


Figure 6.9: Reliability vs. fault detection prob. c and successful repair prob. r.

The connection reliability between any two peers of the network within an FTAT is enhanced by employing the concept of **activity observation**. A packet sent from a scatter is transformed into three redundant signals: one original packet plus two extra ACKs. This gives three chances for the receiver to know or receive the packet and take proper actions. As a result, the reliability of the connection, between any two peers in the system, is increased by the use of redundant ACKs.

Let R_l be the reliability of a transputer link, and R_c be the reliability of the connection between the two peer across a link. Assume all transputer links have the same link

reliability R_l. The probability of a connection failure is obtained and explained below:

$$F_c = (1 - R_l)^3 + 2(1 - R_l)^2 R_l[(1 - R_l)^2 + R_l(1 - R_l)^2] \tag{6.8}$$

Starting with the view from a node of a connection in the network, the first term is the probability that all links of the concerned node in the connection have failed. The second term is the probability that the other two nodes can not provide a route for the connection due to their link failures, assuming that these two nodes are symmetric and independent of each other as well as of the node pair of the connection.

Eq. 6.8 can be further simplified to the following:

$$F_c = (1 - R_l)^3(1 + 2(1 - R_l^2)R_l) \tag{6.9}$$

So the connection reliability R_c is:

$$R_c = 1 - (1 - R_l)^3(1 + 2(1 - R_l^2)R_l) \tag{6.10}$$

There are five possible routes in all possible link failures provided that the system is still connected. The extra ACKs, scattered in response to a data packet received, transform the transmission into four more opportunities to let the intended receiver take proper active actions.

Assuming that the link fault distribution is exponential, R_l is then $e^{-\lambda t}$. Figure 6.10 shows the contribution from the activity observation scheme based on the multiple link group communication. The dotted line is the link reliability R_l, and the solid line is the connection reliability R_c.

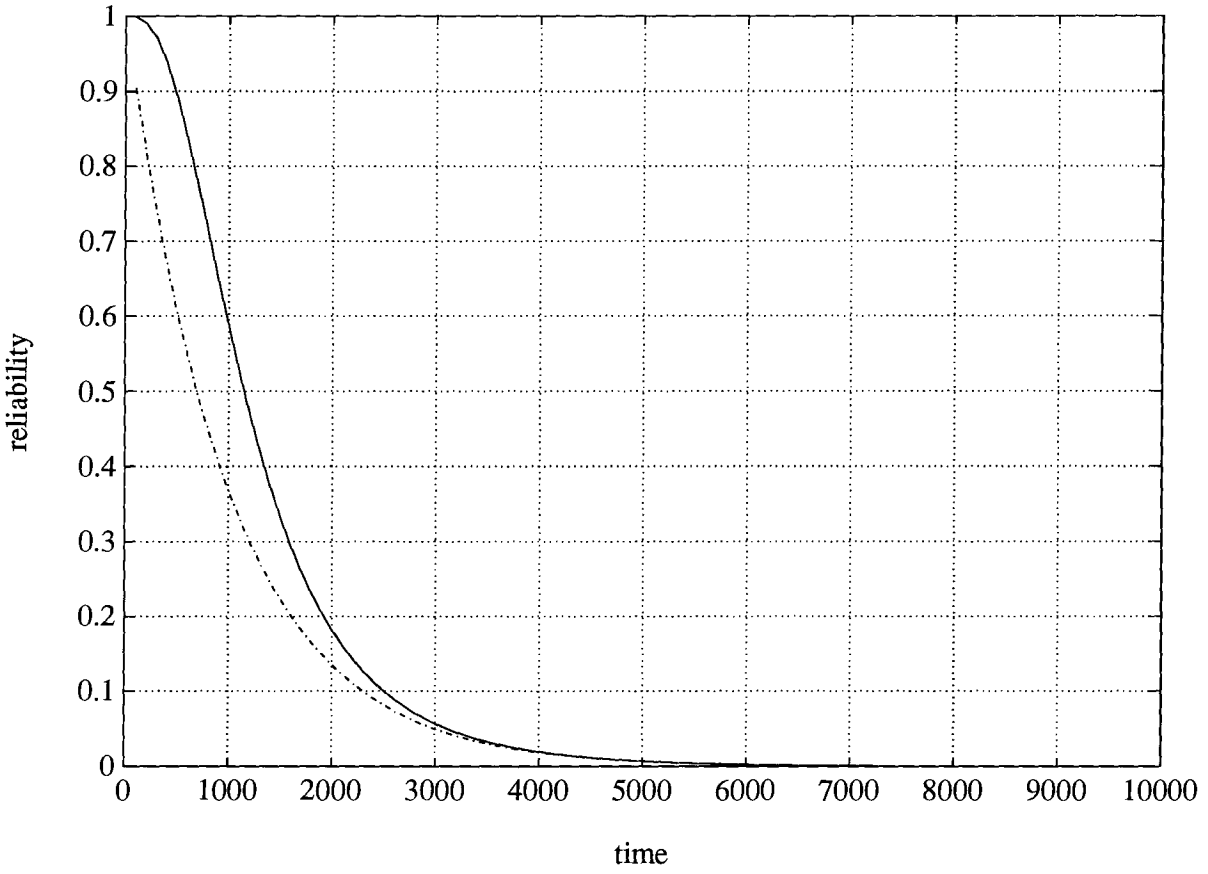


Figure 6.10: The connection reliability vs. the link reliability

Chapter 7

Petri Net Model of the FTAT Multilink Group Communication

In this chapter, the FTAT protocol is formally defined using Petri Nets. By playing with tokens in the net of the protocol modeled, one can derive all the behavior. Using certain properties of Petri Nets, the protocol design is informally shown to be bounded and live. The implementation of the design demonstrates that it is functional. Consequently, the design is shown to be correct.

Petri-Nets are a useful modeling tool for describing the dynamic behavior of concurrent systems. A variation called the Predicate/Transition PetriNet (Pr/T Net) [31] allows the formal treatment of individual tokens, their changing properties and relations. A Pr/T net can greatly reduces the model size and is very useful in replicated/redundant processing modeling, especially for systems with multiple redundant or duplicated modules.

Pr/T net is developed from normal Petri Net theory. It differs from the latter in that inscriptions are allowed to places (predicates), transitions, as well as arcs between places and transitions. The inscription can be a formal sum of tuples of individual tokens/variables. Upon transition, individual variables inscribed to the arc are substituted by the corresponding individual tokens that meet the condition inscribed of the transition, removed from the predicate, or deposited into the predicate.

The Pr/T net model of the system is structured into three levels of abstraction, i.e., the scatter and gather, the swap, and the FTAT services. Each of these levels serves the basis for the higher level, with the scatter and gather at the lowest in the model.

This structure is followed in the analysis of the system model. The scatter and gather service is the processing core of the multilink group communication for the interactions among the group peers. The swap protocol is based on the scatter and gather services, and provides service to the IRPCWrite and IRPCRead protocols. Our discussion starts with the scatter and gather protocol to hide the higher levels from the details of the peer interactions for easy analysis.

First the following notational convention is used in the Pr/T net inscriptions:

- $\Pi_{e_i}(< e_0, e1, e2, ...e_n)$ where $0 \leq i \leq n$ specifies all the elements with respect to all the values of e_i;
- $\Pi(S)$, where S is a set, specifies all the elements available in the set.
- S_s specifies the set of all the elements in S excluding s.
- $|X|$ is a function that gives the number of elements in X where X is a set.

The discussion of the system model assumes the following:

- the protocol is built on top of a reliable packet channel. The channel guarantees the correct delivery of packets and preserves the order, in which they are sent, without any duplicates.
- in order to concentrate on the protocols and simplify the analysis, the transfer of packets through bidirectional link channels are abstracted by a single virtual predicate (a place) to simplify the analysis.

To aid in presenting the model in different layers, the following notation is used:

- The predicates with a ground sign $\perp$ are from other nets in a different level of structure.

- A packet token is used with the format $< Sd, Rc, Or, Pk >$ where

  Sd – the sender.

  Rc – the receiver.

  Or – the originator of the relate data packet.

  Pk – the packet type which is the one of the following:

  pack – data packet.

  ACK – acknowledgement packet.

  claim – lost claim packet.

  special – an ACK packet being returned to its forwarder.

7.1 Modeling Scatter & Gather Operations

The following predicates are used in the protocol modeling:

- **Receivers**: the receivers of a scatter operation.
- **Sender**: the sender of a scatter operation.
- **PacketInQue**: the sender(s) has sent the packets to the specified receivers through the proper channels. This is a virtual predicate to hide the direct link connections. The following is assumed: once a packet (a token) is in the predicate, it will fire to the proper predicate corresponding to the given destination provided the corresponding link has not failed. When a linkage has broken (link failure), a token (a packet) in this predicate disappears if it is routed to the failed link.
- **ReceivedPackets**: the receiver has received the data packets from the sender.

- **ObservedACKs**: the receiver has received the ACKs from other receivers involved in the same scatter.
- **DirectACKs**: the sender has received the ACK(s) from receiver(s) of the scatter operation.
- **Gathered**: the receiver has received all the packets, from all the other peers related to a scatter operation (ACKs and the data packet).
- **ACKGathered**: the sender has received all the ACKs in response to its scatter operation.
- **LinkOK**: the links are known to be working properly on the processor.
- **LinkBad**: the link(s) are known to be dead on the processor.
- **NodeBad**: the processing module(s) is known to have crashed.

Figure 7.11, figure 7.12 and figure 7.13 together show the Pr/T net model of the scatter and gather protocol. The model depicts the behavior of the scatter and gather service in terms of the direct interactions among the redundant peer modules of an FTAT. It consists of three parts in a modular fashion. The main part, given in Figure 7.11, defines the behavior under no-failure conditions. The link failure handling part, shown in Figure 7.12, describes the behavior when link failures occur. The invariants part, defined in Figure 7.13, represents the invariant assertions about the system using the "dead" transitions [31]. "Dead" transitions are conceivable facts but impossible events of the model.

In the figure, set L is the set of working links; set R is the set of peers of an FTAT which is equal to P; and $s \in S = P$. For ease of representation, the set notation is used to show the individual tokens so as to fit in the diagram.

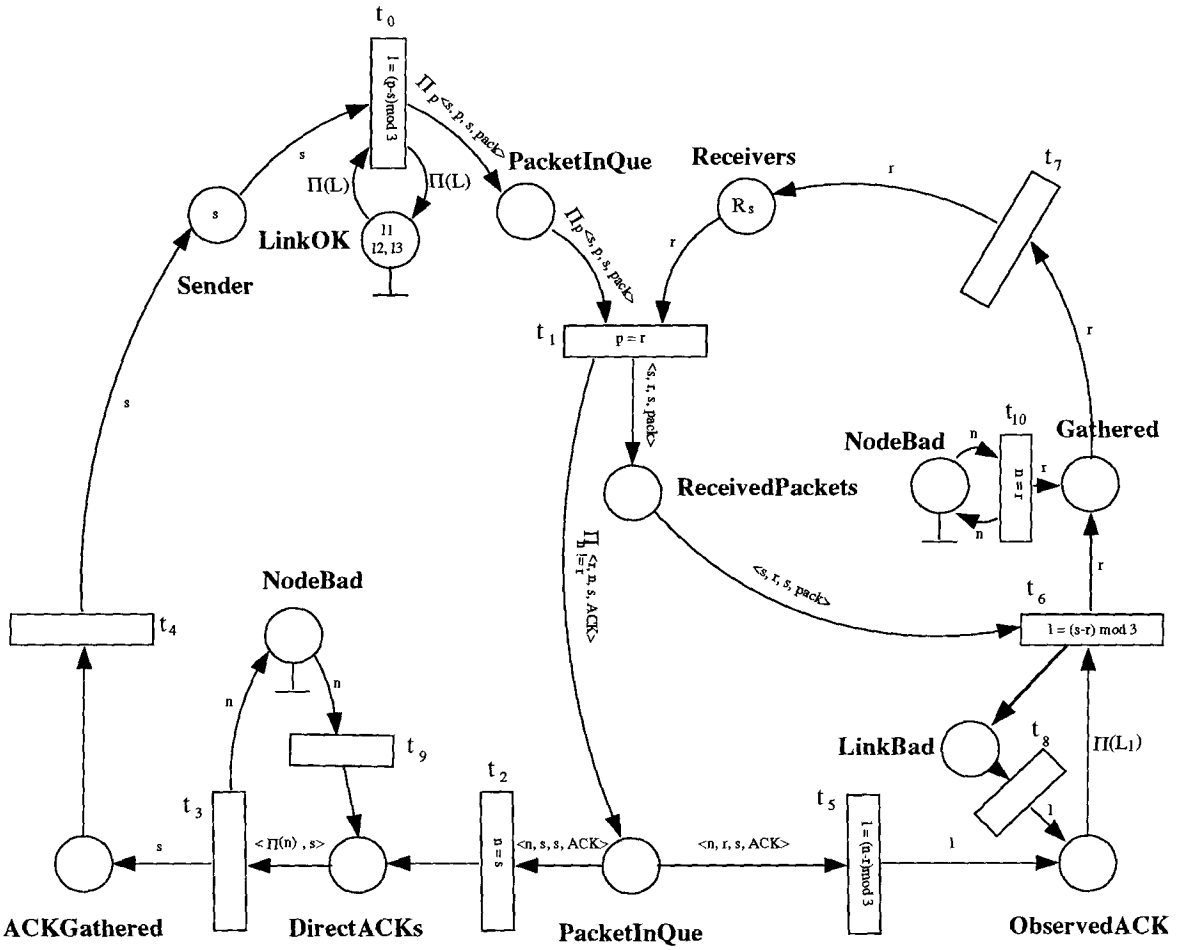


Figure 7.11: Scatter & Gather Pr/T net Model

7.1.1 Boundedness

Definition: A P/T-net N is called bounded iff M_N is finite and there exists $n \in \mathbf{N}$ such that for all $M \in [M_N >$ and all $s \in S_N, M(s) \leq n$, where M is a marking, $[M_N >$ the set of all derivable markings of the net.

First it is clear that there are finite number of predicates in the scatter and gather Pr/T-net model (SG net). Without considering the link failure handling part of the net, the net N_{SG} consists of single loops in one direction. **PacketInQue** is a virtual predicate. Therefore the number of markings that can be derived from this net is finite,

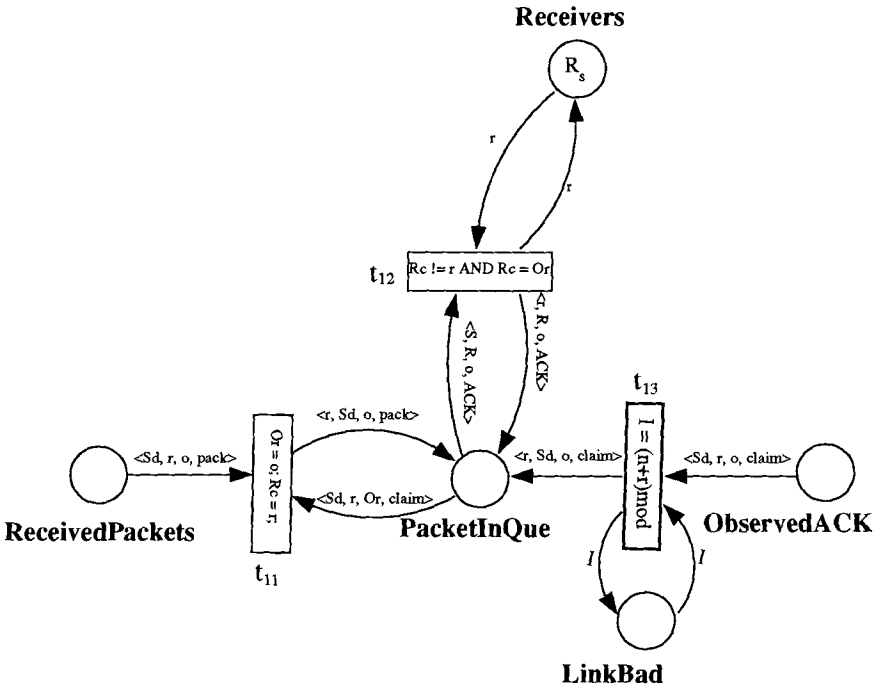


Figure 7.12: The link failure handling part of Scatter & Gather Pr/T net

i.e., $[M_{N_{SG}} >$ is finite. We then need to prove that any marking $M \in [M_{N_{SG}} >$ is finite. Assume at the initial marking M_0 we have:

$$M_0 = \text{M}(\textbf{Sender})+\text{M}(\textbf{Receivers})+\text{M}(\textbf{LinkOK})$$
$$= \text{s}+R_s+\text{L};$$

Then we follow the firings to the possible markings in N_{SG}:

$$M_1 = \text{M}(\textbf{Receivers})+\text{M}(\textbf{LinkOK})+\text{M}(\textbf{PacketInQue});$$
$$M_2 = \text{M}(\textbf{Receivers})+\text{M}(\textbf{LinkOK})+\text{M}(\textbf{PacketInQue})+\text{M}(\textbf{ReceivedPackets});$$
$$M_3 = \text{M}(\textbf{Receivers})+\text{M}(\textbf{LinkOK})+\text{M}(\textbf{PacketInQue})+$$
$$\text{M}(\textbf{ReceivedPackets})+\text{M}(\textbf{ObservedACKs})+\text{M}(\textbf{DirectACKs});$$
$$M_4 = \text{M}(\textbf{LinkOK})+\text{M}(\textbf{PacketInQue})+\text{M}(\textbf{ACKGathered})+$$
$$\text{M}(\textbf{ReceivedPackets})+\text{M}(\textbf{ObservedACKs})+\text{M}(\textbf{Gathered});$$
$$M_5 = \text{M}(\textbf{ACKGathered})+\text{M}(\textbf{Receivers})+\text{M}(\textbf{LinkOK});$$

At any time in the net,

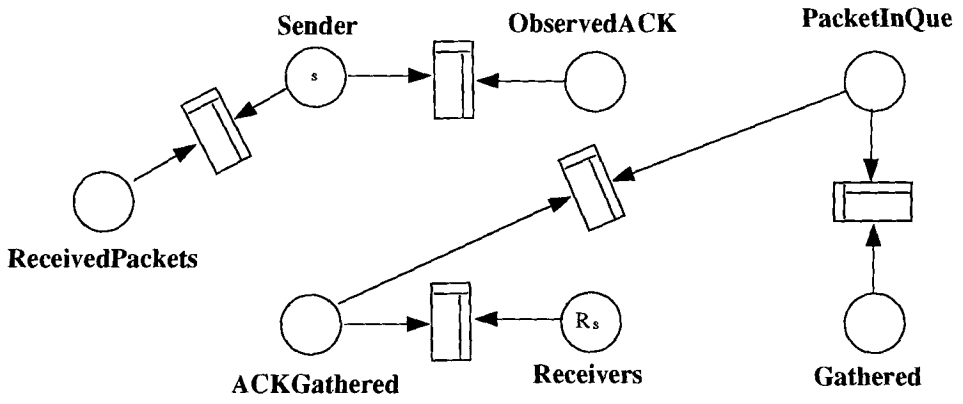


Figure 7.13: The invariant part of the Scatter & Gather Pr/T net model

M(**Sender**) $\leq |s|$;

M(**Receivers**) $\leq |R_s|$;

M(**LinkOK**) $\leq |L|$;

M(**PacketInQue**) $\leq |s| * |L| + |R_s|$;

M(**ObservedACKs**) $\leq |R_s| * |L| - |L|$;

M(**Gathered**) $\leq |R_s| - |L|$;

Now it is very easy to see that any marking $M \in [M_{N_{SG}} >$ is a finite number $n \in \mathbf{N}$, since any finite linear combination of finite numbers is still finite number. Hence the net N_{SG} is bounded.

7.1.2 Liveness

Definition: Under any marking $M \in [M_N >$, if there always exists an enabled transition, then the net is called *live*.

Deadlock can result when the set of transitions directed into a set of predicates belongs to, or equals the set of the transitions directed out from the same set of predicates, i.e., $\cdot S \subseteq S \cdot$ [32].

The possible deadlock can only result from the following two situations: 1) waiting for a direct ACK from a "dead" peer. 2) waiting an observed ACK on a channel of broken link.

Sender side

The sender gets blocked at **DirectACKs** predicate. Link failure handling can guarantee the delivery of the data packet as well as the ACKs, unless the receiver crashes. The **NodeBad** predicate can prevent the net from entering the "deadly" situation of waiting for the response from that failed processing module. Consequently, the sender is always live (deadlock free).

Receiver side

A receiver depends on the sender as well as the other receivers. **LinkBad** predicate eliminates the dependency on a receiver for an observed ACK as soon as the failure of the link is detected. The **Activity Observation** concept is reflected in the link failure handling part of the net model and guarantees the correct reception of the data packet provided that the receiver concerned is still connected to the FTAT network. The link status is monitored and maintained by the peer system periodically. Once a link failure is detected, the waiting on the channel of that link is terminated by depositing a token for that link into **ObservedACK**. Therefore, the net for the receiver has always at least one enabled transition $M \in [M_{N_{SG}} >$. The system is thus deadlock free.

7.1.3 Link Failure Handling

Since this part of the model, shown in Figure 7.12, is mainly responsible for correct transfer of packets between peer systems involved in a group communication, it simply

shows the rerouting of data packets and ACKs when link failures occur. When an observed ACK is received, a lost claim packet is sent at the transition t_{12} to request the data packet from the known recipient. After the data packet is received, the ACK is then passed to the recipient that provided the data packet.

Deadlock occurs when an ACK, being forwarded, loops back and forth around the working peers without a way to reach the originator. This condition is prevented by the detection of the **special** packet. A **special** packet can only be sent to a new peer module but never returned. If there is no way for the packet to be forwarded, it is dropped, from the assumption about the **PacketInQue**. This can only happen when the originator is no longer connected to the system. When such failure is detected, the scatter can proceed as the transition t_9 deposits the expected token in **DirectACK** in Figure 7.11.

When a direct ACK is received and not addressed to the current receiver, it is routed to the best available link. This is reflected by the condition inscribed to the transition $t_{12} : Rc \neq r$ AND $Rc = Or$.

The availability of the data packet for possible lost claim is ensured by the transition t_6. As long as the receiver system has working links with the other receivers within the FTAT it will wait for the observed ACKs from them. This yields the opportunity for determining the status of other peers or providing the data packet upon request.

On the other hand, if the link is broken, there is no need to wait since the purpose of an observed ACK is for knowledge of link connections in the system. Transition t_8 serves this purpose.

7.2 Modeling the Complete Swap Service

In [31][32], a special set of places, called S-invariant set, is defined. This set has the special property that the total joint token count in the net remains invariant during the

transition firings of the net. It can be used to show the boundedness of a net. One of its properties is, as a proofed theorem in [32], that if an net N is *covered* by S-invariants then the net is *bounded.*

A P/T net is said to be *covered* by S-invariants iff, for each place $s \in S_N$, there exists a positive S-invariant i of N with $i(s) > 0$. This is defined in [32].

The model here describes the behavior of the complete swap service based on the scatter and gather services. Figure 7.14 shows the Pr/T net model of the swap service protocol, N_{Swap}. In the model, the following predicates are used:

- **SEMPTY**: the buffer for swap is empty.
- **SACKOBSERVED**: the ACK(s) observed is/are in the buffer.
- **SPDULOSTCLAIM**: the buffer is expecting a lost packet from the claim.
- **SPDUOBSERVED**: a data packet has arrived in the buffer.
- **SACKSCATTERED**: ACKs have been scattered for the buffer.
- **SACKGATHERED**: the buffer has gathered packets from all the peers.

7.2.1 Boundedness

For a simple analysis, we can ignore the predicates from the scatter and gather net since they remain the same after the related transitions fire. The swap net does not change markings of the underlying net. A simplified net of the model is therefore shown in Figure 7.15.

We now construct the incidence matrix [31] and try to find an S-invariant. The invariant vector i can be acquired by solving the linear equation: $N'_{Swap} \times \overline{i} = \overline{\mathbf{0}}$.

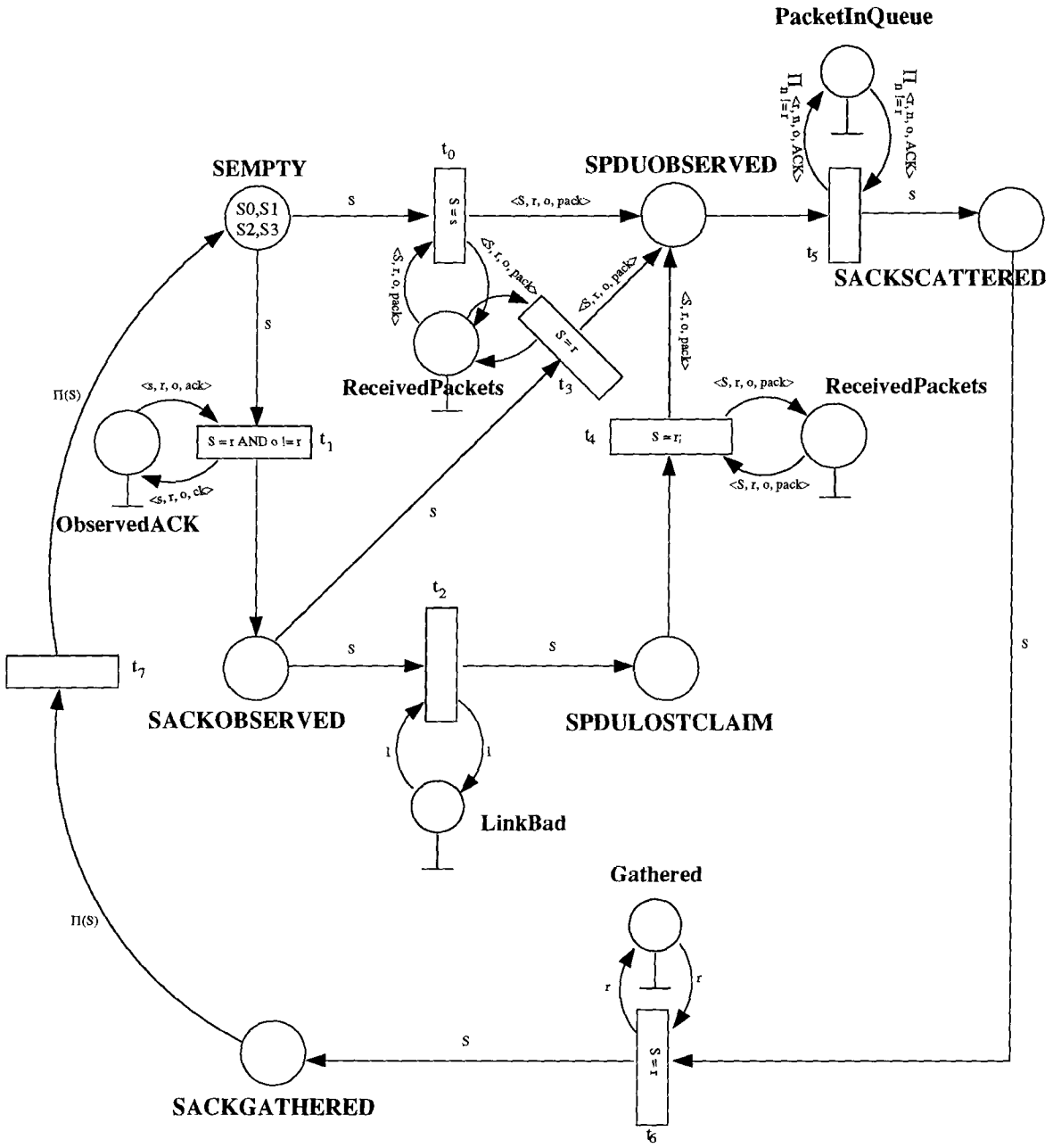


Figure 7.14: Swap operation Pr/T net model

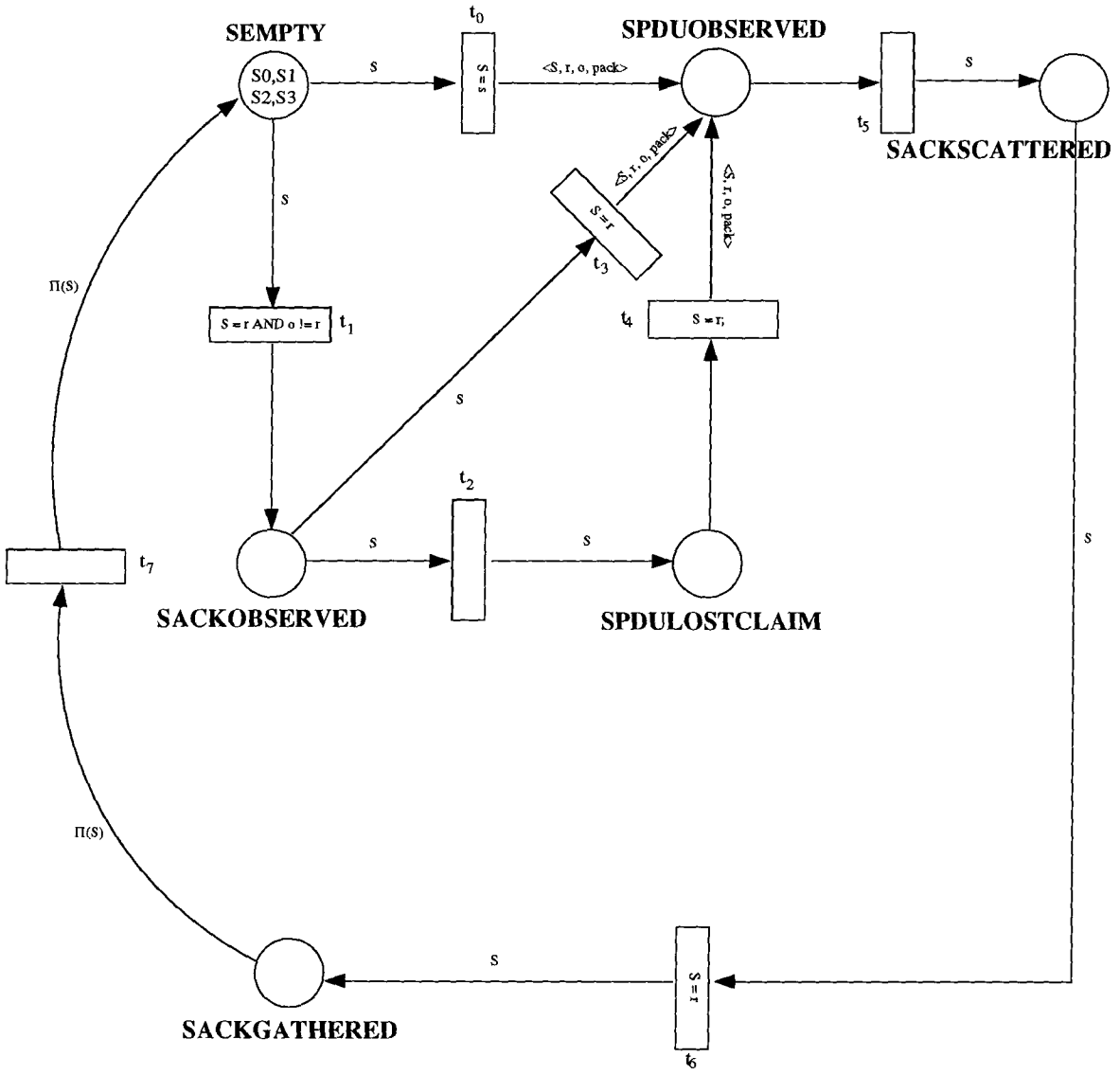


Figure 7.15: Swap operation Pr/T simplified net model

The incidence matrix and the invariant vector of the net N_{Swap} are shown in Table 7.1 where all the elements in the invariant vector are greater than zero. Using the definition and the property stated at the begining of section 7.2, which are given by definition (a) and theorem (d) of Section 6.2 in [32], we know that the net is **covered** by S-invariants and therefore draw the conclusion that it is bounded.

Predicate/Transition	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	Invariant
SEMPTY	-1	-1						4	1
SACKOBSERVED		1	-1	-1					1
SPDULOSTCLAIM			1		-1				1
SPDUOBSERVED	1			1	1	-1			1
SACKSCATTERED						1	-1		1
SACKGATHERED							1	-4	1

Table 7.1: Incidence Matrix and Invariants

7.2.2 Liveness

If we were to ignore the predicates used from the underlying net as in Section 7.2.1, under any marking in the net, there is always at least one transition enabled in N_{Swap} as can be seen directly from Figure 7.15. However, the firings are dependent on the underlying net, as the net is driven by the token movements of the underlying net. Therefore we need to prove that given this dependency, under any marking $M \in [M_{N_{Swap}}>$, there is always at least one transition enabled in N_{Swap}.

First we know that the underlying SG net is deadlock free as proved in Section 7.1.2 and 7.1.3 of this chapter. Secondly, the order that the predicates are marked with the related tokens from the underlying net is in the same partial order as the possible firing sequence in this net. This can be shown by ordering the token movements in partial order and comparing the orders from both nets. We start the ordering from the initial states

of the system, **SEMPTY** and **Sender/Receivers** in the direction of the transitions:

- For N_{Swap}, we have the partial order sequence: **SEMPTY, {SACKOBSERVED, SPDUOBSERVED, SPDULOSTCLAIM}, SACKSCATTERED, SACK-GATHERED**.
- For the SG net: **{Sender, Receivers}, PacketInQue, {ReceivedPackets, PacketInQue}, {ACKGather, ObservedACK}, Gathered**.

Note that the elements in brackets are equal in the partial ordering. In the sequence for the SG net, the **PacketInQue** is a virtual predicate and is ignored in our analysis. The unused predicates are eliminated from the sequence. Now we have: **{Sender, Receivers}, {ReceivedPackets, ObservedACK}, Gathered**.

We assume at the start of the swap service, the initial predicate marking is set to **SEMPTY, Sender, Receivers** for the Swap net and the SG net, respectively. Also we assume that the firings of the enabled transitions in N_{Swap} is always performed before those in the SG net. This assumption is made to simplify the analysis of the model and can easily be transformed into an implementation of the design.

Under any marking $M \in [M_{N_{Swap}} >$, there is at least one transition t is enabled as long as the attached predicate from the SG net has the related token. By the previous assumption, $(\cdot t)_{SwapNet}$ is always marked before $(\cdot t)_{SGNet}$. Since the partial ordering of the SG net reflects the firing sequence and the SG net is deadlock free, then $(\cdot t)_{SGNet}$ will be eventually marked, i.e., t is enabled. Thus, there exists at least one transition enabled under any marking $M \in [M_{N_{Swap}} >$ in the nets considered. Therefore, the Swap net is then live, deadlock free.

7.3 The FTAT Service Modeling

The FTAT services, IRPCRead and IRPCWrite, are based on the Swap mechanism described in the earlier section. They provide the services for inter-FTAT communications. Using a similar method used in Section 7.2, the model is now shown deadlock free. The model, N_{FTAT}, has two parts for the services modeled and is shown in Figure 7.16 and Figure 7.17.

In the following analysis, only the net model for IRPCWrite service is shown. The IRPCReac service model can be analyzed in a similar way.

7.3.1 Boundedness

N_{FTAT} has a limited number of predicates in a single directed transition loop. The net is basically a single loop type of transitions, except between one particular section of the loop. Between predicates **PDUGATHERED** and **STATEGATHERED** there exists an embedded loop which may loop n+1 times, where n is the number of the working peer modules. This results from the transition condition *All Stat = DONE or All Stat = FAILED*. This transition condition ensures that if all modules have tried and failed to deliver a packet, the service is aborted. Consequently, the number of possible markings is finite.

For each predicate in the net, the deposit and removal of a token is always balanced as can be seen from the model. Without considering the effects from N_{Swap}, the incidence matrix and its invariant vector of the net are shown in Table 7.2. Since all the invariants in the corresponding predicates are greater than zero, the net is said to be **covered** by S-invariants according to the definition (a) in Section 6.2 of [32]. By the theorem (d) of Section 6.2 of [32], the net N'_{FTAT} is bounded.

Now let us examine the effects stemming from N_{Swap} on the net N_{FTAT}. It is obvious

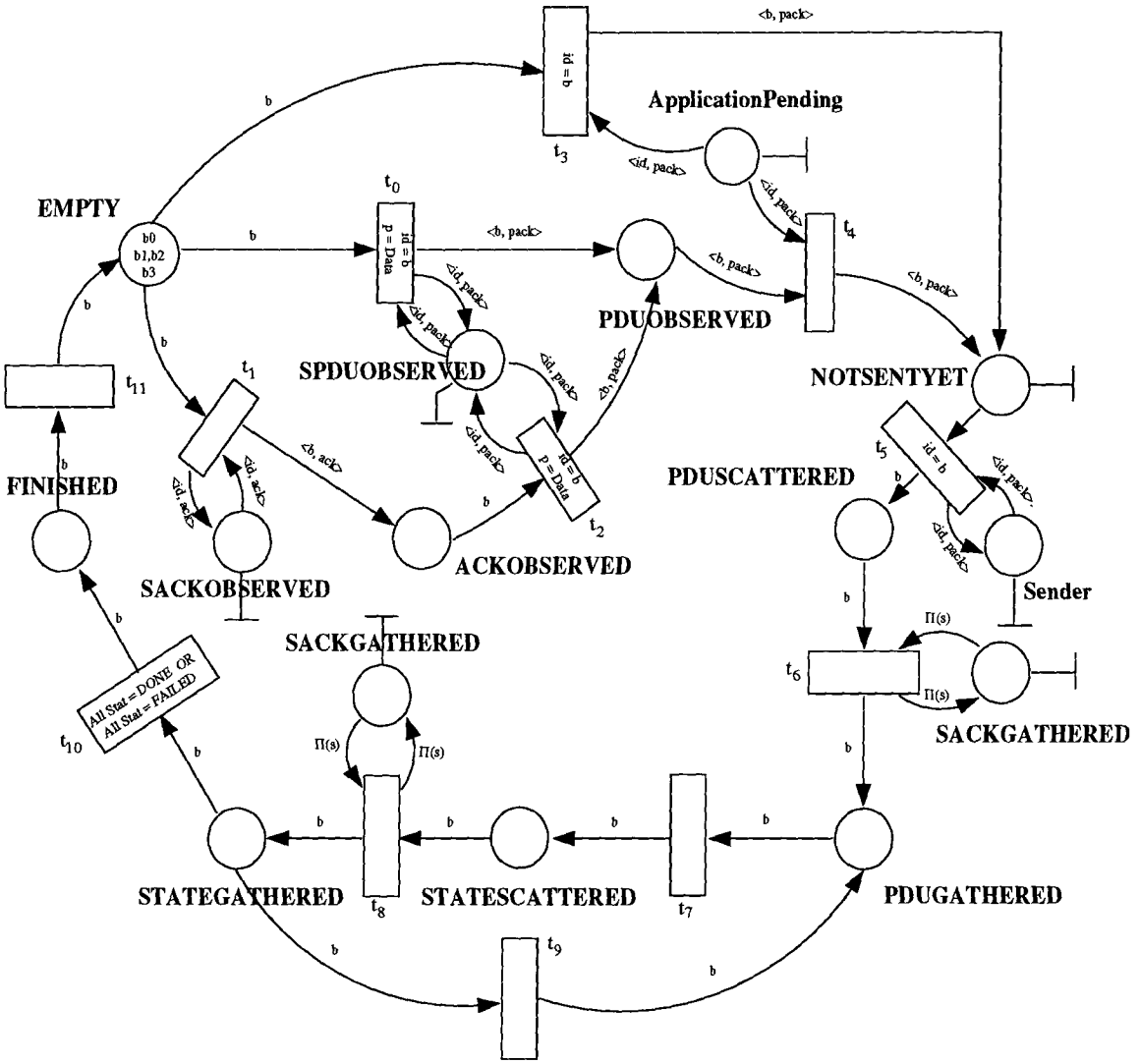


Figure 7.16: IRPCWrite Service Pr/T net model

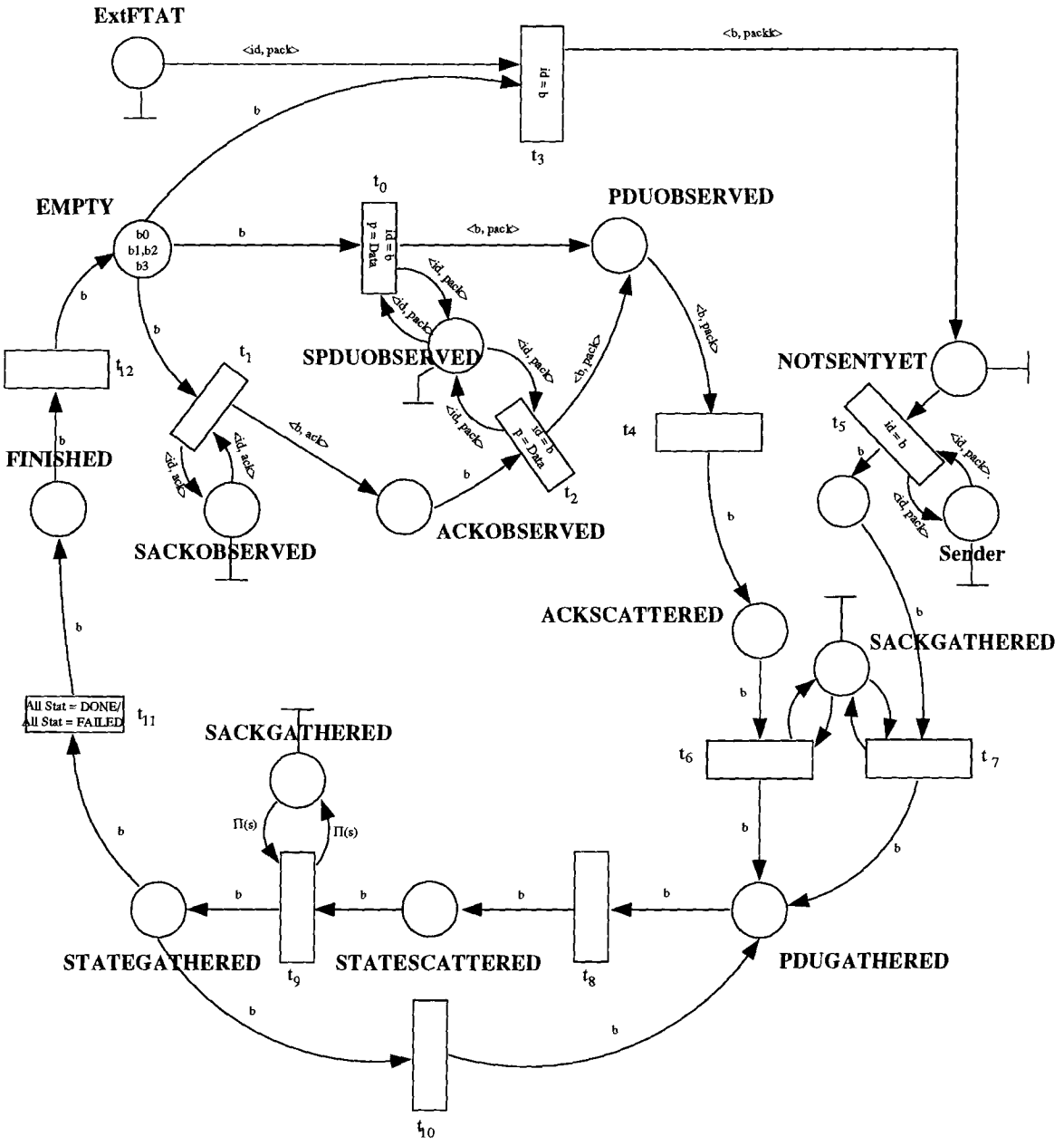


Figure 7.17: IRPCRead Service Pr/T net model

Predicate/Transition	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	Invariant
EMPTY	-1	-1		-1								1	1
PDUOBSERVED	1		1		-1								1
ACKOBSERVED		1	-1										1
NOTSENTYET				1	1	-1							1
PDUSCATTERED						1	-1						1
PDUGATHERED							1	-1		1			1
STATESCATTERED								1	-1				1
STATEGATHERED									1	-1	-1		1
ApplicationPending											1	-1	1

Table 7.2: FTAT Incidence Matrix and Invariants

that N_{Swap} does not change the total joint token count in the net N_{FTAT} and nor does N_{FTAT} in N_{Swap}. Net N_{FTAT} is thus bounded.

7.3.2 Liveness

For simplicity in the analysis, the swap net N_{Swap} is ignored first. Figure 7.18 shows the net model without N_{Swap}. In N_{FTAT}, each predicate has an arc to some transition t, and for any predicate with a marking greater than zero, the followed transition is enabled since each transition t has only one predicate in its $\cdot t$. As a result, N_{FTAT} would be a deadlock free net without N_{Swap} connected.

Now we examine N_{FTAT} with N_{Swap} connected. We still assume that the transitions enabled in N_{FTAT} are always fired before those in the underlying nets until the N_{FTAT} transitions are disabled due to the markings of the underlying nets. This assumption can be easily translated into an implementation and greatly helps the structured analysis by making the modules less dependent on others and without distorting the model from the design.

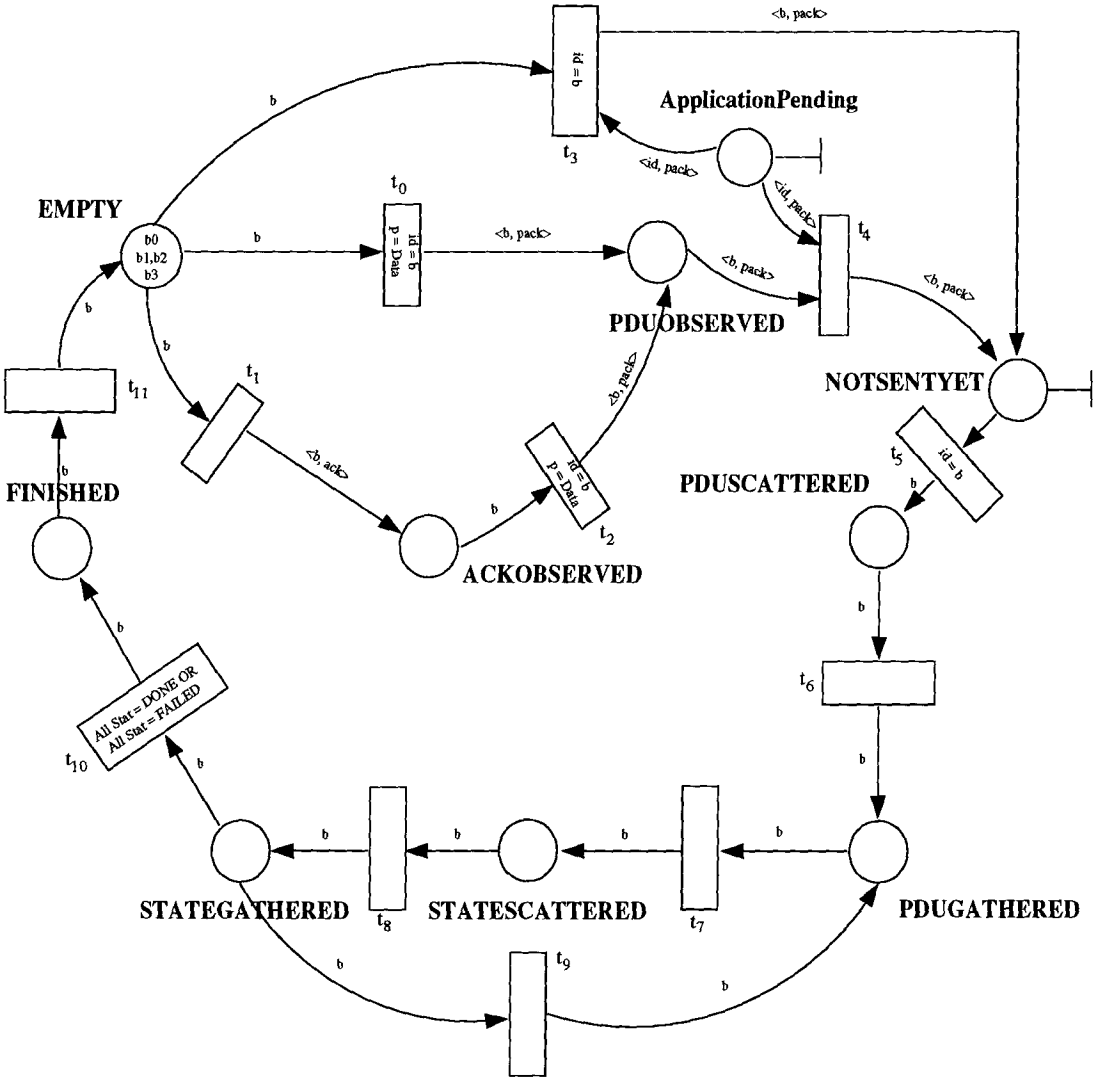


Figure 7.18: IRPCWrite Service Pr/T simplified net model

Firstly, the firing sequence is ordered from the initial states of the system in partial order. The result ordering is **EMPTY, {ACKOBSERVED, PDUOBSERVED, ApplicationPending}, NOTSENTYET, PDUSCATTERED, PDUGATHERED**, whereas in Section 7.2.2 the ordering was **SEMPTY, {SACKOBSERVED, SPDUOBSERVED, SPDULOSTCLAIM}, SACKSCATTERED, SACKGATHERED**. By the fact that N_{FTAT} is a deadlock free net without N_{Swap} connected, then for any marking $M \in [M_{N'_{FTAT}} >$ in N_{FTAT}, there exists some transition t that is to be enabled by some predicate in N_{Swap}. Thus we have $(\cdot t)_{FTATNet} > 0$ and $(\cdot t)_{SwapNet} = 0$.

Secondly, note that the firing sequence between t_0 to t_7 of N_{FTAT} is in the same partial ordering with that of N_{Swap}, i.e., for any t_n in the FTAT net N'_{FTAT}, $\cdot t_{n-1} \prec \cdot t_n$ and $\cdot t_n \prec \cdot t_{n+1}$ in both nets respectively; the transition loop, t_7 to t_9, is only dependent on **SACKGATHERED** from N_{Swap}.

Since N_{Swap} is deadlock free, as shown earlier in Section 7.2.2, eventually the $(\cdot t)_{SwapNet}$ becomes marked. That is, the transition t is enabled. Since t is some transition that would be enabled under any marking $M \in [M_{N'_{FTAT}} >$, we draw the conclusion that under any marking $[M_{N_{FTAT}} >$, there exists some enabled transition t in the FTAT model. The FTAT model has thus been shown to be live.

Chapter 8

Performance

In this chapter, we analyze the performance of the major components of the system in term of time and amount of communication. The multilink group communication and the on-line forward fault-repair are the major research issues addressed in this thesis. A brief analysis of the performance of the former is carried out quantitatively and the on-line forward fault-repair is analyzed qualitatively. The generic communication service measurements are listed last.

The performance analysis of the multi-link group communication demonstrates how the concept of **activity observation** improves the performance in the communication. The on-line forward fault repair is analyzed to see how the system performance is enhanced because of the repair and how the performance of the on-line forward repair itself is improved by the use of volatile data redundancy as opposed to the use of virtual shared memory.

The operations of the FTAT are mainly based on the scatter and gather mechanism built for group communication over multiple link connections. The bidirectional communication links are all supported by dedicated DMA controllers which allow concurrent broadcast on all the links. The distribution of redundant messages are parallelized and most of the operations are symmetric with respect to all the other peers within an FTAT.

In this chapter, a simple performance analysis of the system is conducted. For the multilink group communication, the analysis of a worst performance case and a best performance case are given and contrasted with what the performance would be without

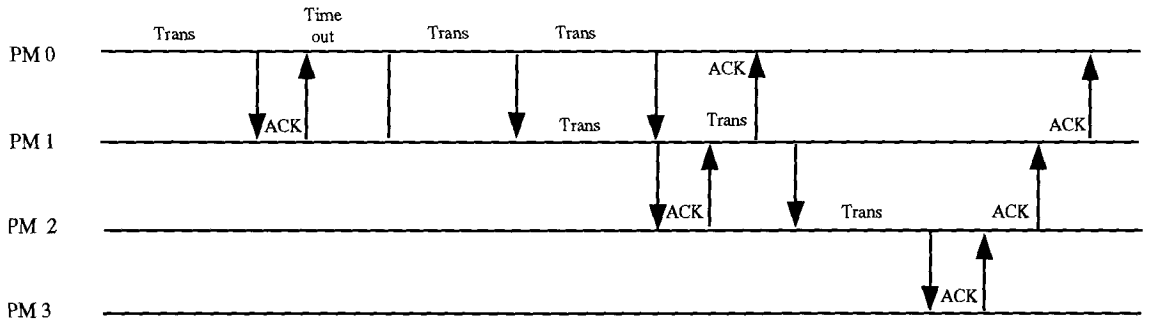


Figure 8.19: Scatter without **activity observation** in a single link connection

activity observation (A.O.).

The following definitions are used in the performance analysis:

- T_{trans} – the transmission time for a data packet.
- $T_{ACKtrans}$ – the transmission time for an ACK packet.
- T_{claim} – the transmission time for a lost claim packet.
- $T_{timeout}$ – the time period which if exceeded, results in a decision that a packet is lost.

8.1 Scatter and Gather

A scatter operation consists a broadcast on all the FTAT internal links from a peer and a reception of all the ACK packets from the other peers. The packet transmission on each link takes about the same amount of time since they are sent concurrently.

Assuming that T_{trans} is far greater than T_{claim}, $T_{ACKtrans}$, and $T_{reroute}$, then the major cost is mainly contributed by T_{trans}. In the following analysis, the packet processing time and the channel setup time are all assumed to be negligible.

- Single link failure:

  The time cost for a scatter without (A.O.) is:

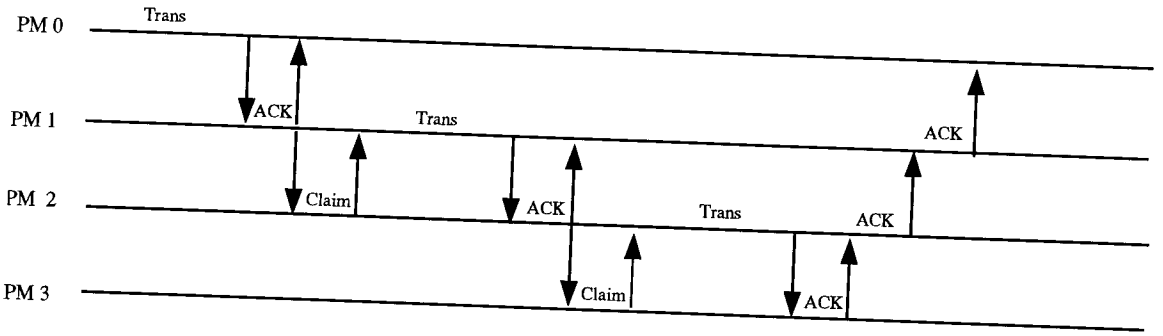


Figure 8.20: Scatter with **activity observation** and a single link connection

$$T = T_{trans} + T_{timeout} + [T_{trans} + T_{ACKtrans}] * 2 > 3T_{trans} \tag{8.11}$$

With A.O. it is:

$$T = T_{trans} + T_{claim} + T_{trans} + T_{ACKtrans} * 2 \approx 2T_{trans} \tag{8.12}$$

The overall time for the scatter operation with A.O. takes one less packet of data transmission than it would without A.O. at the cost of a lost claim packet. T_{claim} is the same as $T_{ACKtrans}$. An increase of the processing speed is about 30% in this case.

- Single line connection in the system resulting from Multiple link failures:

  From Figure 8.19, the time cost for a scatter without A.O. is:

$$T = T_{trans} + T_{timeout} + 4 * T_{trans} + 3 * T_{ACKtrans} \approx 5T_{trans} \tag{8.13}$$

From Figure 8.20 with A.O. it is:

$$T = T_{trans} + [T_{trans} + T_{ACKtrans} + T_{claim}] * 2 + 3 * T_{ACKtrans} \approx 3T_{trans} \tag{8.14}$$

Here, the scatter operation without A.O. would take 6 data packet transmissions in the network. Considering that some of the transmissions are parallel in time, the

effective time consumption is about 5 times of a data packet transmission, while that for the scatter with A.O. is only about 3. This is a nearly 50% decrease in communication traffic and a 40% increase in the processing speed.

- Parallel lost claiming from the recipient with multiple link failures in the network: From Figure 8.19, the time cost for a scatter without A.O. is:

$$T = T_{trans} + 4 * T_{trans} + 3 * T_{ACKtrans}] \approx 5T_{trans} \tag{8.15}$$

From Figure 8.21 with A.O. it is:

$$T = T_{trans} + T_{ACKtrans} + T_{claim} + T_{trans} + 3 * T_{ACKtrans} \approx 2T_{trans} \tag{8.16}$$

This is the most significant case with respect to performance. The scatter without A.O. would take 5 data packet transmissions while with A.O. it takes only 3 in actual transmission. The effective time consumed is 2 times of a packet transmission because some of the transmission are in parallel. This results in a nearly 40% decrease in communication consumption and a 60% increase in the processing speed.

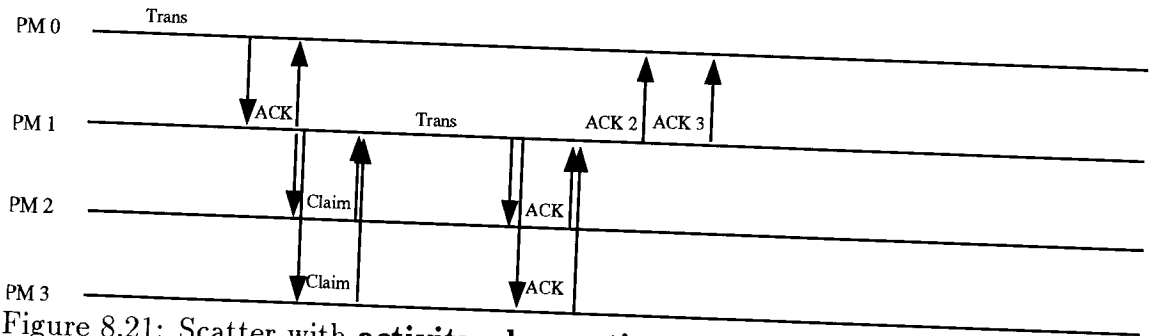


Figure 8.21: Scatter with **activity observation** and a parallel link connection

8.2 Swap

When no failures occur in the system, a complete swap of information takes slightly more time than a scatter operation since every peer in the group does the same operations symmetrically.

The extra costs are caused by the transmission of two extra ACKs over each link in each direction and the processing time for these extra ACK packets from all the other peers. If the average data packet is far longer than ACK packets and the processing time of ACK packets are negligible, then the cost is mainly due to the transmission of the data packet.

8.3 On-line Fault-Repair

On-line forward fault-repair involves selection of the repairer node, initiation of the repair request, packing of volatile data, delivering these data in packets, overwriting the context of the faulty process and relocating pointers in the context. Volatile data of a process includes global data, heap allocations, run-time stack, pointer reference stack, as well as the resource management table.

The repair service provides full context repair. Its performance is dependent on the size of context. The process of packing and delivering is much faster than disk accessing since memory access is faster than disk I/O, and the data communication (2.5MB/s) is faster than disk I/O. Compared with commonly used process migration schemes such as those used in the V system or the Sprite system[23][25], our approach has the following advantages:

- no checkpointing and roll-back is necessary for fault recovery.
- no disk I/O access is required.

- no shared memory consistency maintenance overhead is suffered.
- pointer adjustments is done in one batch.
- pointer references rebinding are performed on a resource allocation basis instead of a pointer basis.

8.4 Generic Communication Performance Measurements

The generic communication operations have been measured with and without link failures for the operations such as scatter & gather, swap and IRPCWrite. No significant difference has been observed between the first two. However the measured results are not the optimal performance for the protocol, realizing that the implementation can be further tuned for best results.

Table 8.3 shows the results of our implementation which can be further optimized. The processing overhead seems to be the major factor. However, our quantitative analysis shows an interesting potential performance gain for the multi-link group communication protocol. Our implementation is still subject to tuning to get the best performance.

Measured Time\ Service	Scatter & Gather	Swap	IRPCWrite
without link failure	20ms	27 ms	128ms
with link failure	21ms	30 ms	140ms

Table 8.3: Generic Operation Performance Measurements

Chapter 9

Conclusions

The FTAT architecture yields a systematic architectural approach to fault-tolerance systems. The regularity and recursiveness of the architecture makes it easy to use the architecture as building blocks to build large multiprocessing systems.

The multilink group communication mechanism provides redundant and parallel data distribution and group synchronization. The concept of **activity observation** employed in the design greatly improves the communication mechanism to provide efficient and reliable data distribution in the face of link failures and processor failures. The major gains can be summarized as follows:

- Packet rerouting is initiated by the receiver in response to the observation of a recipient of the data packet in a group communication. A lost packet is always "claimed" from a known recipient and the route is always the optimal.

- The low cost in transferring the ACKs, which used as "hints" for packet rerouting, greatly reduces the network operation overhead.

- Because of the receiver-initiated dynamic rerouting, parallel "claims" of a lost packet by multiple intended receivers are possible.

- The elimination of the many timers that otherwise would be needed for reliable redundant data distribution, reduces the overhead and complexity of the protocol implementation.

- The scatter and gather operation for the distribution of a data packet makes the sender and receivers symmetrically complementary, which yields better timeliness in group synchronization.

The reliability analysis of the multilink group communication of the system shows that the connection reliability in the group communication between any pair of the modules is increased. The concept of **activity observation** in the design of the protocol is shown to have the potential of considerable performance gain and high communication efficiency.

The transparency in fault-masking provides a simple programming model to the programmers for fault tolerant applications, especially for parallel programming applications. The dynamic TMR scheme used makes the system tolerant of two concurrent faults without using more than four processors. The underlying mechanism of the on-line fault-repair makes the system tolerant of unlimited sequential transient physical faults with smooth reintergration of the repaired module, unless permanent failures occur on majority modules. Our reliability analysis has shown that system reliability can be improved remarkably with on-line forward fault-repair.

The design of the multilink group communication protocol has been formally represented by Petri Nets in a structured modular way. The nets are shown informally to be bounded and deadlock free, and the implementation demonstrates that the design is functionally correct. Hence, the design is shown to be correct. The use of Predicate Transition Net which is a variation of Petri Net allows us to represent the model in simple diagrams and to give proofs in a simple way.

The on-line fault-repair eliminates faults on the process basis without roll-back of previous operations. Some of the similar issues in **process migration** are approached in a different way. By separating an application process from its run-time physical environment at a minimal cost, the live execution image can be used to repair the faulty

process to guarantee forward fault recovery. This is very attractive for real-time processing because of the following:

- No computation is wasted because the forward recovery occurs without any roll-back.
- Pointer reference rebinding is performed on a resource allocation basis instead of a pointer basis.
- Using volatile redundancy saves the cost of accessing persistent I/O devices.
- No shared virtual memory has to be used, thus concurrency control overhead is eliminated.

There is still much work that can be done in the future:

- The experimental system is subject to optimization for best performance.
- Investigation is needed on how system's reliability relies on the acceptance test pace under different module reliabilities. Since high fault coverage can be achieved at the cost of communication and comparisons, this may provide a guide line for the best performance while meeting the reliability requirements.
- The system itself is vulnerable to faults, thus highly efficient self-checking techniques should be exploited to cause the system to have crash-failure semantics.
- Efficient fault detection techniques of high coverage are needed for forward fault-repair to achieve high reliability.

Bibliography

[1] R. Beton, J. Kingdon and C. Upstil, "Highly Availability Transputing Systems", Proceedings of the World Transputer User Group Conference. April, 1991.

[2] G. Coulouris and J. Dollimore, "Distributed Systems Concepts and Design", Addison-Wesley Publishers Ltd. 1988.

[3] N. Brock, "Transputers and Fault-Tolerance", Proceedings of the World Transputer User Group Conference, April 1991.

[4] J. Standeven and M. Colley, "Hardware Voting of Transputers in Real-Time nMR Fault-Tolerant Systems", Dept. of Computer Science, Univ. of Essex. April, 1990.

[5] Q. Stout and B. Wager, "Intensive Hypercube Communication I: Prearranged Communication in Link-Bound Machines", The Univ. of Michigan Computing Research Laboratory, June 1987.

[6] D. Rennels, "Fault-Tolerance Computing – Concepts and Examples", IEEE Trans. on Computers, Vol. C-33, No.12, December 1984.

[7] D. Siewiorek, and R. Swarz, "The theory and practice of Reliable System Design", 1982 by Digital Press.

[8] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", IEEE Trans on Software Engineering, Vol. SE-13, No.1, January 1987.

[9] A. Tanenbaum, "Computer Networks", Prentice Hall, Inc., 1988.

[10] A. Avizienis, "The N Version Approach to Fault-Tolerant Software", IEEE Trans. on Software Engineering, Vol. SE-11, #12, December 1985.

[11] K. Tomita, S. Tashiro and T. Kitamura, "A Highly Reliable Computer System – Its Implementation and Result", Annu Int. Symp. on Fault-Tolerant Computing, 1980.

[12] R. Strom, D. Bacon, and S. Yemini. "Volatile Logging in n-Fault-Tolerant Distributed Systems", Annu Int. Symp. on Fault-Tolerant Computing, 1988.

[13] D. Russell and M. Tiedeman, "Multiprocess Recovery Using Conversations", Annu. Int. Symp on Fault Tolerant Computing, 1979.

[14] J. Ayache, P. Azema, and M. Diaz. "Observer: A Concept For On-Line Detection of Control Errors In Concurrent Systems", Annu. Int. Symp on Fault Tolerant Computing, 1979.

[15] C. Li, and W. Fuchs. "CATCH – Complier-Assisted Techniques for Checkpointing", 20th Annu Int. Symp on Fault-Tolerant Computing, June 1990.

[16] P. Velardi, and R. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System", IEEE Trans. on Computers, Jun. 1984.

[17] F. Christian, B. Dancey, and J. Dehn. "Understanding Fault-Tolerant Distributed Systems", Invited paper, 20th Annual International Symposium on Fault-Tolerant Computing, June, 1990.

[18] K. Kim and J. Yoon, "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Annu Int. Symp. on Fault-Tolerant Computing, 1988.

[19] E. Cooper, "Replicated Distributed Programs", Proceedings of 10th ACM Symposium on Operating System Principles. December, 1985.

[20] D. Cheriton and W. Zwaenepoel, "One-to-Many Interprocess Communication in the V-System", Report STAN-CS-84-1011, Department of Computer Science, Stanfard University, August 1984.

[21] H. Stone. "High Performance Computer Architecture", Addison Wesley publishing, 1987.

[22] Y. Chen, T. Chen, "Implementing Fault-Tolerance via Modular Redundancy with Comparison", IEEE Trans. On Reliability, Vol. 39, No.2, June, 1990

[23] M. Theimer, K. Lantz and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System", Proceedings of the 10th Symposium on Operating System Principles. December, 1985 .

[24] F. Douglis, "Process Migration in the Sprite Operating System", Electrical Engineering and Computer Sciences. Univ. of California Berkeley, Technical Report, Feb. 1987.

[25] E. Zayas, "Attacking the Process Migration Bottleneck", Computer Science, Canegie Mellon Univ. ACM SIGOP, 1987.

[26] O. Serlin, "Fault-Tolerant Systems in Commercial Applications", Computer, IEEE, August 1984.

[27] J. Ortiz, "Transputer Fault-Tolerant Processor", Proceedings of the Third Conference of the North American Transputer Users Group, 1990.

[28] R. Oates, J. Kerridge, "Adding Fault Tolerance to a Transputer-based Parallel Database Machine", Proc. of the World Transputer User Group Conference, 1991.

[29] R. Strom, D. Bacon, S. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems", Annu Int. Symp. on Fault-Tolerant Computing, 1988.

[30] S. Dutt, J. Hayes, "Some Practical Issues in the Design of Fault-Tolerance Multiprocessors", IEEE Trans. on Computers, Vol.41, No.5, May 1992.

[31] H. Genrich and K. Lautenbach, "System Modeling with High-Level Petri Nets", Theoretical Computer Science, 13, North-Holland Publishing Company, 1981.

[32] J. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Inc., 1981.

[33] D. Comer, "Operating System Design: The XINU Approach", Prentice Hall, Inc., 1984.

[34] INMOS, "The Transputer Databook", second edition, 1989.