FADI: A Fault-Tolerant Environment for Distributed Processing Systems

Taha Mohammed Osman

A Thesis submitted in partial fulfilment of the requirements of the Nottingham Trent University for the degree of Doctor of Philosophy.

March 1998

Acknowledgments

I am greatly indebted to my project supervisor, Prof. Andrzej Bargiela, for his valuable help and advice. Without his continuous support and able expertise this project could not have been completed.

I am grateful for my research colleagues for the constructive discussions and consultations about this work. I specially thank Mr. Evtim Peytchev for his useful suggestions concerning the implementation of the software model.

I would like to thank Mr. Chris Noble, our system engineer, for putting up with numerous requests regarding the setup of the distributed processing environment.

I would also like to thank my dearest friend, Dr. Sabbah Razzaq for his moral support. He has been a true comrade.

Finally, I would like to dedicate this work to my parents, Fatima and Osman, for their sincere belief in me and for their encouragement at all stages of life.

ABSTRACT

This thesis describes the research done on the development of a FAult-tolerant DIstributed Processing Environment (FADI). The main motivation for designing FADI is to create an efficient low-cost fault-tolerant environment, enabling reliable execution of concurrent user-application processes in presence of hardware faults that affect one or more of the distributed system nodes.

There are two aspects to any fault-tolerant system: *error detection* and *fault recovery*. In FADI, a user-transparent mechanism was developed for the detection of both permanent (power failures, network malfunction, etc.) and transient (temporary memory faults, radiation affects, etc.) processor node failures. The Detection mechanism in FADI also allows the incorporation of additional user-programmed error checks and, by dynamically measuring the network traffic, it identifies the error latency thus facilitating damage confinement and assessment.

The recovery of user-processes affected by faults is based on the principles of checkpointing and roll-back. The application processes record their execution states at regular time intervals (take a checkpoint), and upon the occurrence of a hardware failure, the failed process is restarted from the last recorded execution state (roll-back of the process).

In contrast to the redundancy-based fault-tolerant systems, the adopted approach does not require any extra hardware, nor does it demand the replication of application processes. Instead, it optimises the use of an existing distributed system to save the checkpoints, and to migrate the application processes from the faulty hardware to operating nodes. This reduces both the cost of building the recovery software and its overhead on the running time of the application-processes. The novel checkpointing mechanism developed in the course of this research is user-transparent and is computationally efficient since it does not require freezing the application-process while its checkpoint is being taken.

The checkpointing mechanism has been initially evaluated in the context of stand-alone applications and the experimental results have shown that it is very robust. Subsequent research extended the developed checkpointing protocol, so that it covers the possible inter-process communications taking place between the distributed application processes. This ensued the development of a novel algorithm that supports checkpointing and roll-back of message passing (interactive) processes in FADI. The algorithm introduces a novel technique to tolerate faults that might occur whilst inter-process messages are in transit. It also tolerates the duplication of inter-process messages and has a low failure-free overhead due to its policy of coordinated checkpointing and selective message logging.

The performance studies indicate that FADI exhibits low overhead on the execution time of the applications and it has confirmed its potential in the context of computation-intensive scientific programs and distributed telemetry and telecontrol industrial systems.

The work described in this Report is the Author's own, unless otherwise stated, and it is, as far as he is aware, original.

Table of Contents

List of Figures

List of Tables

CHAPTER 1 Introduction

This chapter introduces briefly the history of developing distributed computing systems, their applications, and why they are considered a powerful computing paradigm. Different technologies to achieve fault-tolerance in Distributed Computing systems are discussed and evaluated.

1.1 The Need for Distributed Computing Systems

There are many and varied definitions of distributed systems. In this research a distributed system is understood to be a system in which there are several autonomous but interacting processors and/or data stores at different geographic locations.

Two main stimuli have been behind the sharp rise in the utilisation of distributed systems since the late 1960s:

advances in computer technology. Until the spread of minicomputers in the early 1970's a commonly accepted rule was Grosch's law: *"The cost per machine instruction executed is inversely proportional to the square of the size of the machine"* [Enslow 78]. This economics of scale in computing led to the centralisation of computer resources [Martin 81], and all work became funnelled into centralised factory-like data processing shops.

With the growth in microelectronics and the introduction of VLSI in the early 1970s, people started questioning Grosch's law. The price-performance ratio was continuously changing in favour of multiple low-performance processors, rather than large single high-performance processors [Krame 87]. Moreover, the interconnection and communication costs have fallen dramatically in the past few years. Buses, packet-switched networks, LANs, and internet connectivity are now readily available and cost-effective.

need for distributed applications. User demand for distributed computing is manifested in two main areas:

1) Inherently distributed and potentially distributed applications: Inherently distributed systems are systems that must respond to, or manage, simultaneous activities in their external environment. Examples of such systems can be found in such diverse application

areas as monitoring and control of industrial processes, hospital patient monitoring, air traffic control, fly-by-wire and multimedia. Collectively these systems are sometimes referred to as real-time telemetry systems.

Another class of inherently distributed systems are database management and transaction processing systems such as flight reservation systems and electronic commerce, where the data is distributed on geographically remote locations by nature.

Potentially distributed applications are applications that might benefit from distributed implementation [Bacon 93]. These are usually number-crunching long-running programs that need to process large amount of data. Examples are FFT algorithms and computations of fluid dynamics [Fatoohi 94]. Distributed systems provide an ideal cost-effective platform for running the potentially concurrent components in parallel to speed-up their execution.

2) Resource sharing. A number of resources, such as computers, peripherals, special purpose processors, software, and databases are interconnected by a communication system in order to allow the sharing of the resources. These can be local networks within an office block or research establishment, or inter-continental client-servers over wide-area-networks (WANs).

1.2 Advantages of Distributed Systems

Low Cost. The advent of the low cost distributed system nodes provided the main impetus for distributed processing [Krame 87]. In the same time the cost of peripheral devices has not declined dramatically, which provided additional motivation for distributed computing systems that share expensive resources (laser printers, special-purpose processors, etc.).

Modularity. Distributed systems are constructed in a very modular fashion, where each component provides well-defined interfaces or services to the rest of the system. This enforced modularity leads to simpler system design, installation and maintenance, and allows application tasks to exploit the strength of different components of the distributed system.

Flexibility and Extensibility. The modular design of distributed systems has the added advantage of facilitating modification or extension of a system to adapt to a changing envi-

ronment without disrupting its operation. Therefore it is possible to start with a small configuration, and provide functional or performance upgrades by adding additional computers at low cost increments.

Reliability. Distributed systems introduce the possibility of independent failures - independent components can fail while the others continue running. Using Fault-tolerant techniques faults can be detected and recovered from by making use of the redundant resources.

1.3 Computer Organization for Distributed Systems

The term distributed processing is used to describe systems with multiple processors that cooperate to perform a computational task. However, the term has a broad meaning because the processors can be connected in many ways for various reasons. From the connectivity point of view, distributed systems can be generally divided into two categories:

1) Systems that have multiple processors connected by in-plant wiring (centralized systems). Examples of such systems are:

a. Multiprocessor systems which share a common memory (e.g intel_ i860, Sun Sparc20 MP, Cray Systems).

b. Massively parallel systems with thousands of processing elements, where each element has a dedicated memory module. (e.g transputer networks such as MEMSY and Parsytec).

These systems are built for *high-performance computing* and need specific class of applications (SPMD, MPMD, Multi-threaded programs) to fully exploit the parallelism of their multi-processor topology and the fast communication links between the processors.

2) Networked systems. They represent a group of autonomous computers (workstations, PCs) linked through a communication network. Each of the network computer systems has its private memory module, sometimes local stable storage, and local peripheral interfaces. This configuration makes such systems attractive to single users, in addition to their use in the distributed resources pool when inter-linked with a communication network.

Utilising networked systems for distributed computing has the following advantages:

a. The universality of networked systems made them the most popular distributed systems setup in various environments ranging from small office LANs to huge networks of heterogeneous computer systems. Networked systems are widely utilised in scientific, commercial, and industrial establishments. Sales of their computer systems (Unix-based workstations - HP-RISC, Sun Sparc, DEC Alpha, Power Mac) far exceed that of dedicated parallel and distributed systems of the first category, which made networked systems much more accessible.

b. Networked systems are interconnected in a loosely coupled structure that makes them highly scalable. They can grow in small increments to systems including many nodes, depending on the requirements of the targeted distributed applications.

c. The high availability of networked systems motivated research into development of high-level programming interfaces that enable the use of networked nodes as a large parallel computer. Examples of such interfaces are PVM [Geist 94], LINDA [Frings 97] and MPI [Sun 97].

However, networked systems concede in performance to centralised systems. The main reason for that is that networked systems share a single communication link -usually the relatively slow Ethernet (10-100 Mbps)-, while the centralised/massively parallel systems can make use of fast peer to peer and shared memory communication solutions.

Using a high-speed interconnection network like FDDI, ATM [Gaida 96] or the Fast Ethernet (with communication rates ranging from 100Mbps to a Gigabit per second) can significantly offset this disadvantage and, it is believed, it will ensure the future prominence of networked systems.

For the above reasons, it is important that *clusters of networked workstations,* which provide hardware environment for a large number of real-life applications, offer also a reliable distributed computing environment which avoids the loss of computations if one of the networked nodes fails. Pursuance of this objective is the subject of this research.

1.4 Fault-Tolerance in Distributed Processing Systems

1.4.1 Overview

With the advent of VLSI design, computer systems became more sophisticated, powerful and most importantly affordable. This led to an ever-increasing utilisation of computer systems in solving complex scientific and industrial problems, and consequently increased the demands on their reliability. The development of fault-tolerant techniques dates back to the work of Von Neumann in the mid 1950s [Birman 94]. The extent to which such techniques are actually used has of course varied as computer hardware technologies have changed, and has depended on the stringency of the reliability requirements that had to be satisfied.

1.4.2 Fault-Tolerance Principles

To make any fault-tolerant system one needs to consider: "What can go wrong with the system? What happens if it does go wrong and what can be done to prevent it? Answers to these three questions form the fundamental concepts of fault-tolerance:

(i) *Error Detection:* In order to tolerate a fault in a system, its effects must first be detected. While a fault cannot be directly detected by a system, the manifestation of the fault will generate errors somewhere in the system. Thus, the usual starting point for fault-tolerance techniques is the detection of an erroneous state. In a computer system errors might be caused by external influences (failures of power supply, radiation, etc.), by a design fault (temporary memory flips, stack overload, etc.) or indeed by a human fault whether it is the operator or the system user.

(ii) *Damage confinement and assessment:* When an error is detected, it is possible that because of the likely delay between the manifestation of a fault and its detection invalid information might have spread within the system, leading to other errors which have not yet been detected [Anderson 81]. Therefore it is important to asses the extent to which the system state had been damaged and possibly take appropriate measures to confine the damage.

(iii) *Error Recovery:* Following error detection and damage assessment, techniques for error recovery must be utilised. Unless the error is removed, the erroneous state may cause a failure of the system in the future.

1.4.3 Recovery in Distributed Systems

Due to the inter-dependency of their components, distributed systems are particularly vulnerable to failures. The failure of one component might induce the failure of the whole system. Therefore, it is necessary to act fast not only to recover the failed node, but to prevent the waste of processing accomplished on the whole distributed system when one of its nodes fail.

The problems of providing recovery in distributed systems are exacerbated by inter-process communication between the computing nodes. This complicates the recovery mechanism in two ways: 1) the inter-node propagation of the failure needs to be confined; 2) recovery of all interactive processes needs to be coordinated in order to recreate a *global consistent system state.* Hence, the fault-tolerant techniques have to take into consideration the state of the communication channels between the distributed system nodes at failure time, as well as the local state of the application processes running on each individual node.

One of the early approaches to providing fault-tolerance, which was widely studied by many researchers is that of process replication [Birman 94] [Appel 92]. With this technique, several copies of each process are executed concurrently on different computers so that the probability that all replicas would fail is acceptably small [Cooper 85]. Since the replicas execute the same code and make use of the same data, a distributed computation should proceed correctly as long as there exists one living replica of each process, i.e r-1 faults can be tolerated in an r-replicated system. Although these techniques incur a smaller degradation in performance when compared to checkpoints mechanisms, they are not overhead-free. The failure-free overhead is caused by using multicasting mechanisms to deliver every outgoing message to the *troupe* of replicated destination processes [Elnozahy 92].

However, the problem associated with the process-replication methods is the heavy cost of the redundant hardware for the execution of the replicas. Despite claims that such techniques are becoming more feasible with the increasing size of distributed systems and subsequently the amount of the available redundant hardware [Chiu 94], their use remains confined to large-budget, mission-critical systems with stringent reliability requirements (e.g air-traffic control systems [Dugan 94], nuclear reactor controllers, etc.).

An alternative approach that has drawn an increasing attention in distributed systems, is to provide fault-tolerance based on checkpointing/rollback mechanism. In contrast to process

replication mechanisms, this method does not require the duplication of the underlying hardware or the replication of the application processes. Instead, each process periodically records its current state and/or some history of the system in a stable storage, an action called *checkpointing*. When a failure occurs, processes return to the previous checkpoint (*rollback*) and resume their executions from this checkpoint. The overhead that this technique incurs is greater than that of process replication mechanisms because checkpoints are taken during failure-free operation, and rollback-recovery requires certain actions to be taken to ensure consistent recovery when processes crash [Deconinc 93]. Nevertheless, this overhead is getting smaller as the computers and communication networks become more efficient. Furthermore, new methods such as *checkpoint space reclamation* [Wang 92] and *copy-on-write message logging* [Elnozahy 94] are being devised all the time to reduce the checkpointing overhead. Such advances in checkpointing technology supported by the low-cost for providing it, motivated the adoption of the checkpointing/rollback strategy in a large european project FTMPS [Vounckx 93] concerning the development of fault-tolerant mechanisms for massively parallel systems.

CHAPTER 2 Background to Fault-Tolerant Environment Research

This chapter reviews state of the art reliable computing systems, and discusses the motivation behind developing the fault-tolerant environment, the class of application it targets and specifies the essential requirements for its implementation accordingly. Finally the chapter covers the underlying message passing interface used for interprocess communication in the FAult tolerant DIstributed environment (FADI).

2.1 General Background

Computational complexity of real life problems necessitates the use of distributed computing systems. The demand for distributed computing is growing with the sharp rise in the amount of applications that are inherently distributed, potentially distributed, or that simply requiring to share common computational resources.

Another factor in favour of distributed systems is the ever decreasing cost-performance ratio of distributed system nodes and the modularity with which they are constructed. This modular design facilitates the modification or extension of a system to adapt to a changing environment without disrupting its operation.

Loosely coupled distributed systems such as NoWs (Network of Workstations) consist of many nodes which, despite improvements in hardware reliability, can and do fail [Morin 97]. Consequently, for a system comprising many workstations the *Mean Time Between Failure* (MTBF) may be significantly reduced. Following the example given by Seligman [Seligman 94] on a workstation with a mean-time between failures of 16 days, a one day computation has a 94% chance of completing successfully, while on a cluster of ten machines, there is only a 54% (0.94^{10}) chance that a one day computation will complete before a failure occurs. Faults in the communication link (e.g network partitioning) further reduce the distributed system MTBF. This emphasises the importance of taking fault-tolerant measures not only to recover the failed computing node, but also to prevent the waste of processing accomplished on the whole distributed system when one of its nodes fails.

Three main components constitute fault-tolerant distributed computing environments:

1. An inter-process communication system that links the distributed computing resources. There exist many high-level interfaces that provide users with a parallel platform for running their applications without worrying about the complexity of the implementation of the underlying network protocols;
2. Error detection mechanism that covers permanent and transient faults that might affect the distributed applications and/or the underlying hardware.
3. A backup and recovery mechanism to safe-guard the executing application processes and the inter-process communication channels against detected faults.

Many existing systems such as PVM and P4 [Butler 94], and Amoeba [Tanenbaum 90] provide distributed computing services but have no provision for fault-tolerance. Other systems like Libckpt [Plank 95], Condor [Briker 91], Libft [Huang 93] support the reliable execution of independent stand-alone application processes, but do not support distributed processing. This research focuses on investigation of *reliable, distributed* computing systems.

‡ DOME (Distributed Object Oriented Migration Environment) [Seligman 94], provides a C++ library of data parallel objects and uses PVM for its process control and communication. DOME fault-tolerant model relies on checkpointing/rollback techniques, it periodically saves the current state of the distributed program to one or more checkpoint files and allows the program to be restarted from the most recent checkpoint after failure. Once restarted, the program should proceed normally from the position of the last checkpoint. Checkpointing/rollback is implemented at the application-level and require the application programmer to insert the calls to the checkpoint and restart mechanisms. It is also the responsibility of the programmer to ensure that all system variables are encapsulated in DOME objects for saving the application state. While DOME claims that this approach guarantees the environment portability to any system that supports PVM and C++, it sacrifices transparency to the user/programmer and restricts the utilisation of the environment to the application programmer. The error detection mechanism in DOME is restricted to transient faults that cause the premature exit of the application tasks and faults leading to node crashes are not covered.

‡ Fail-Safe PVM [León 93] is another package that uses PVM to facilitate network communications. This system employs transparent checkpointing/rollback techniques for backup and recovery of application processes. The Fail-Safe PVM reliability model can

recover from at most one failure at-a-time. In Fail-Safe PVM, all checkpoints must be taken at once. It forces a global synchronization before allowing a checkpoint to proceed. After such a barrier, the global state is guaranteed to be consistent and unchanging. The set of local checkpoints taken under these circumstances is a valid snapshot of the state of the session. This method incurs significant overhead on the running time of the application because all application processes are suspended until the complete checkpoint is taken. Another limitation of the Fail-Safe PVM reliability model is that it can recover from only one failure at-a-time. The failure model in Fail-Safe PVM assumes that faults cause complete failure of the node in which they occur. The error detection mechanism does not cater for transient faults.

‡ STAR [Sens 93] is a system managing fault-tolerant distributed applications in a network of workstations. STAR builds its own communication model for interprocess communication on UNIX-like operating systems using UDP, TCP/IP sockets. Information about the performance and spectrum of functionality of this model was not available. The recovery mechanism is based on checkpointing. The checkpointing process is user-transparent and relies on the logging of every inter-process communication message on the receiver's end to guarantee a global consistent system state. This technique is adapted for applications composed of processes exchanging small streams of data, but the logging overhead can be unacceptable for communication-intensive applications. STAR operates a logical structuring of hosts in a *logical ring of detection* for host crashes. Each host only checks its immediate successor in the ring. If the successor does not acknowledge, the fault is reported and the detection ring is re-configured. Transient hardware faults that might result only in the failure of the application processes are not covered.

‡ Paralex [Davoli 96] is a modern distributed system that makes extensive use of graphic editors to aid the parallel application programmer in defining, editing, and executing parallel scientific programs in a fault-tolerant environment. To achieve fault-tolerance, Paralex uses the ISIS *coordinator-cohort* toolkit to implement passive process replication. Each node (process) that requires fault tolerance is instantiated as a process group consisting of replicas for the node. One of the group members is called the coordinator in that it will actively compute. The other group members remain inactive other than receiving multicasts addressed to the group. Only the coordinator of the destination node will compute the data value while the cohorts simply buffer it in an input queue. When the coordinator completes computing, it multicasts the results to the process groups at the next level and signals

the cohorts so that they can discard the buffered messages. Upon the detection of a failure, one of the cohorts is nominated a coordinator and it resumes computing from the messages at the head of its input queues. ISIS relies on hardware-level multicast -whenever the hardware architecture permits- to reduce the overhead of message replication. The error detection methodology is part of the coordinator-cohort replication strategy. ISIS relies on an exchange of timed-out request-acknowledgment messages to detect the failure of the processes. The time-out interval is adaptively adjusted to the communication link load. Performance results have shown that fault-management incurs very small overhead on the execution time of the application. Replication-based techniques do not have the overhead of taking a snap-shot of the of the application execution image and its communication channels state and writing them to stable storage. However, the heavy cost of this method incurred by the need for redundant computing nodes to execute the replicas makes it unattractive for a large class of business and scientific applications which neither have large computing budgets nor very stringent real-time constraints.

The objective of this research is to develop a cost-effective, reliable, distributed computing environment that is transparent to the user/programmer of distributed applications. This environment should be structured to encompass all aspects of fault-tolerant distributed computing, it should: provide automatic support for distribution/remote execution of application processes; have a high fault-coverage mechanism for detecting hardware transient and permanent faults that is capable of dynamically adjusting the error latency to the distribution network load; use robust, low-overhead, checkpointing/rollback techniques for the recovery of interactive -message passing- application processes. These goals are translated into the requirement specifications for the fault-tolerant distributed environment (FADI) described in the next section.

2.2 Targeted Class of Applications

FADI is designed to support a class of applications that are computation-intensive but do not have stringent real-time constraints. These applications may require hours or days of computations to execute on dozens of networked workstations [Davoli 96]. While the required level of reliability for such applications necessitates the prevention of the loss of the results of the long-running computations, it does not justify the use of expensive hardware replication fault-tolerance techniques. Examples of this large and important class of applications are complex neural-networks training algorithms, structural analysis of

mechanical constructions [Chadna 96] or simulations of large scale engineering and environmental systems such as weather simulations [Chen 96], simulations of mass and heat transfer [Riberio 95], aerodynamic simulations [Meakin 90].

Computational fluid dynamics (CFD) provides a good example for this type of scientific computing. Fluid flows are modelled by a set of partial differential equations, the Navier-Stokes equations. Except for special cases no closed-form solutions exist to the Navier-Stokes equations, and it is interesting to note that this particular computational task was in fact one of the motivations by John von Neumann for the development of electronic computers.

Solving a particular CFD problem generally involves first discretizing the physical domain that the flow occurs in, such as the interior of turbine engine or the radiator system of a car. This discretization is straightforward for very simple geometries such as rectangles or circles, but is a difficult problem in CAD (Computer-Aided Design) for more complicated objects. Currently automatic "mesh generators" are simply not adequate, requiring extensive investment of time on the part of the scientist or engineer. This leads to problems in human-computer interfaces (HCI) and CASE tools, as well as fundamental problems in graph theory since the resulting discretization gives a mesh that is best dealt with as a graph.

On the discretized mesh the Navier-Stokes equations take the form of a large system of non-linear equations: the transition from the continuum to the discrete set of equations is a problem that combines both physics and numerical analysis: For example, it is important to ascertain that the laws of physics such as conservation of mass are not affected by the discretization, which in itself is necessary for numerical computations. This usually calls for the introduction of static variables, at each node in the mesh, which enable some compensation of the affect of discretization. Typically between 3 and 20 variables are associated with each node: the pressure, the three velocity components, density, temperature, etc. Furthermore, capturing physically important phenomena such as turbulence requires extremely fine meshes in parts of the physical domain. Currently meshes with 20,000 to 2, 000,000 nodes are common, leading to systems with up to 40,000,000 unknowns.

Such a system of non-linear equations is typically solved by a Newton-like method, which in turn requires solving a large, sparse system of equations in each step. Sparsity here means that the matrix of coefficients for the linear system consists mainly of zeros, with

only a few non-zero entries. With 4.0e7 unknowns, clearly one cannot store the matrix as a 2D array with 1.6e15 entries!

A practical approach to solving such engineering problems is to partition the problem domain into sub-problems and to solve them in a coordinated fashion on a distributed computing system.

The objective of this research was to develop a cost-effective and efficient distributed fault-tolerant system for a broad class of scientific and engineering applications (as exemplified above).

2.3 Requirements specification

Figure 2-1 represents the context model of the fault-tolerant distributed system at the highest level of abstraction. A more detailed illustration of the system DFD *(data-flow diagram)* is included in the appendix "A".

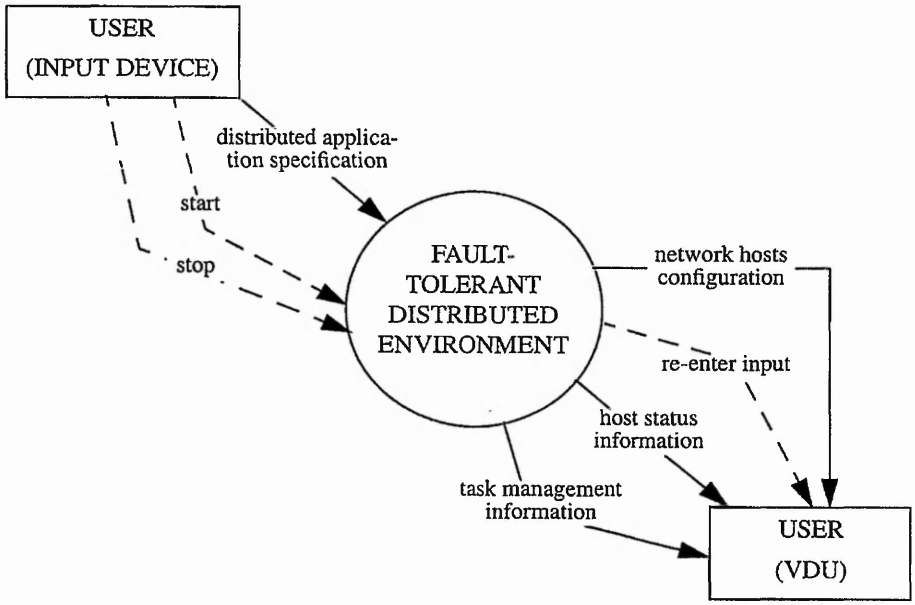


FIGURE 2-1 **System Context**

2.3.1 General Requirements

- In the Interest of portability the system is assumed to make use of the UNIX operating system environment. This assumption is consistent with the operating environment of a large class of scientific/engineering applications.

- The distributed system is assumed to operate with no additional hardware to support fault-tolerance (the fault-tolerance of the whole distributed system is defined by the most reliable node). The required reliability of the central host might be obtained by hardware duplication, but the detail of achieving it is not relevant to the discussion of the FADI itself.
- The system is required to be transparent to the application programmer, i.e no modifications to the original program code by the user/programmer should be necessary.

2.3.2 Requirements to the Implementation of the Software Model

- The system is required to automatically detect the hardware setup of the underlying distributed network and provide automatic means for the parallel execution of user-application tasks on the network hosts.
- A high-level interface is required to support the synchronisation and communication between distributed application processes.
- There will be an error detection mechanism capable of detecting hardware permanent and transient faults that might occur in the distributed nodes.
- A mechanism is required for recovering and migrating (if needed) of the application tasks running on the faulty hardware.
- Considering that the targeted class of applications is mainly computation-intensive, long-running tasks without stringent real-time constraints two decisions were made about the recovery mechanism:

  a) checkpointing and rollback techniques will be adopted as the backup and recovery technology.

  b) because of the large MTBF (Mean-Time Between Failures) of modern systems, greater emphasis will be put on efficient failure-free operation than on efficient recovery from failure.
- A user-friendly interface to input the distributed application specifications is required. The user should have the option to choose the host machines by default, by specifying a particular host, or a general hardware architecture.
- FADI should incorporate an on-line monitoring system that presents the current state of the distributed system hosts and the application tasks running on them.

2.4 FADI Development. A High Level View of the System Context

By analysing the requirements for FADI, the overall system design can be broken down to the modules displayed in Figure 2-2.

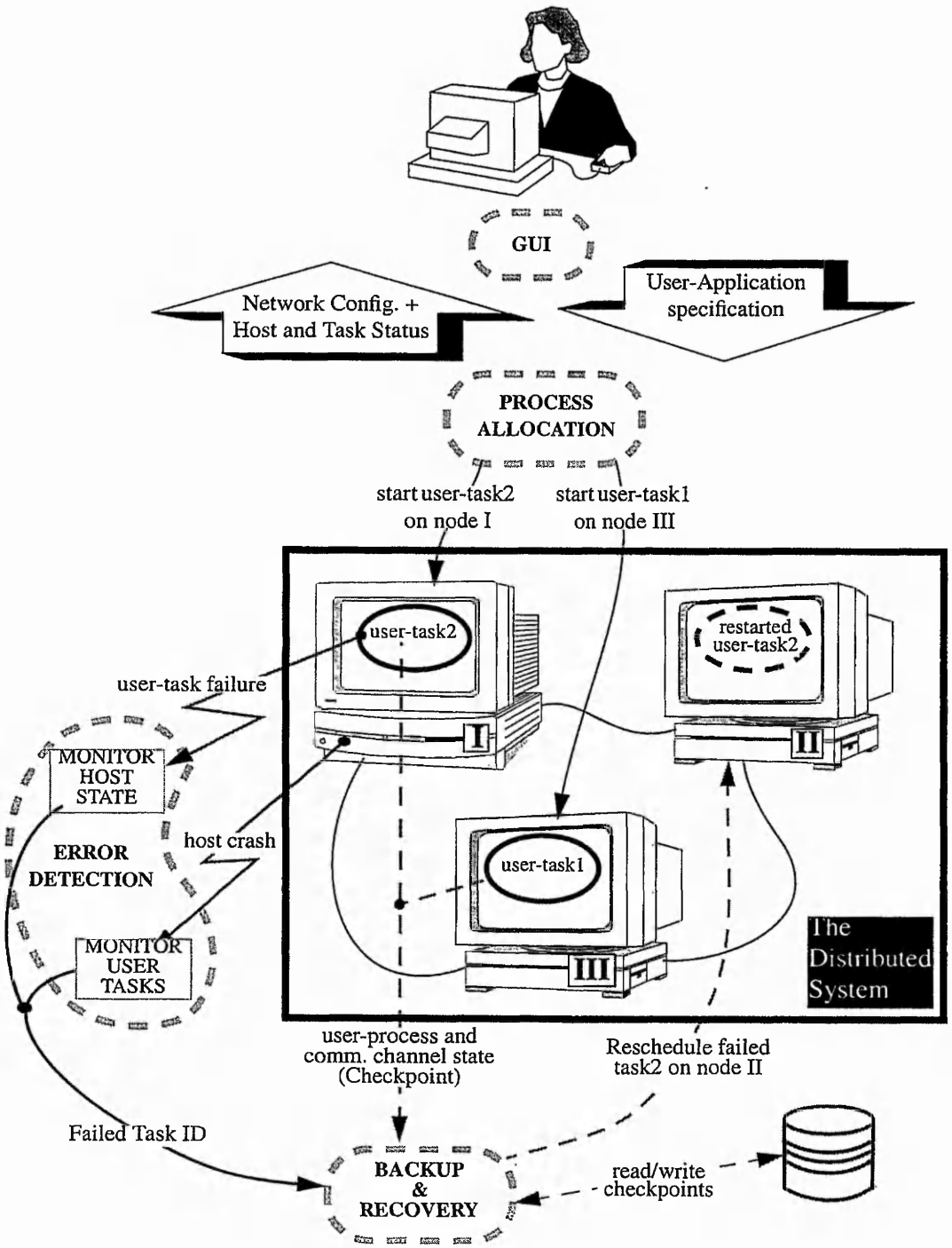


FIGURE 2-2 **FADI: Schematic Diagram**

FADI functionality can be represented by describing the task of each module and the data and control flows between the modules.

Process Allocation Module. Initially the module automatically detects the configuration of the distributed system network. Then it accepts the distributed application specifications entered by the user and validates them against the current distributed system setup. This is necessary in order to advise the user whether the available hardware resources are sufficient to run the distributed application reliably. Next it spawns the application tasks on the specified hosts.

Error detection module. This module is composed of two sub-modules:

The first monitors the state of the distributed nodes to detect any permanent hardware faults. If a host crashes it initiates the recovery of the user tasks running on the faulty node.

The second sub-module monitors the distributed user-application tasks. It analyses the exit status of the task to determine whether it exited normally or prematurely due to a transient hardware failure. If the latter occurs then measures are taken for the failed task recovery.

Application Backup and Recovery module. The most important module for FADI, is composed of two units:

The checkpointing (backup) sub-module is responsible for recording the state of the application processes and the communication channels between them.

The rollback (recovery) sub-module performs the re-configuration of the distributed system after a node crashes. This sub-module is also responsible for the recovery of the failed application tasks by retrieving their saved state from backup (and re-scheduling to operative hardware if necessary) to form a global consistent state of the distributed application.

User-Interface module. This module facilitates the user-friendly interaction between the user and FADI. It performs the following operations:

- entering the system specifications;
- extending the network by adding more nodes to the distributed system;
- on-line display of the distributed nodes status and the application tasks running on them.

The modular design of FADI supports the parallel execution of its control and monitoring tasks. In addition, the whole system is message driven which dramatically reduces the

response time to events occurring in the distributed environment. On the other hand this emphasises the importance of adopting a flexible and efficient message passing interface.

The interface should facilitate the communication between the control and monitoring processes of FADI as well as between the distributed application processes.

2.5 Inter-Process Communications

2.5.1 FADI Requirements for the Communication System

The message passing interface should provide a transparent mechanism for explicit message passing and process-synchronisation between the distributed application tasks. Figure 2-3 shows a simplified view of the envisaged interaction between the interface and the application programmes.This interface should relieve the programmer from having to deal with low-level networking details such as socket communication, IP addresses, network load, etc. The message passing interface should also be able to exchange messages between UNIX machines that have different data representations.

It is important that the communication interface preserves the order of messages sent from a single source for the purpose of analysing inter-process dependencies when recovering failed distributed application tasks.

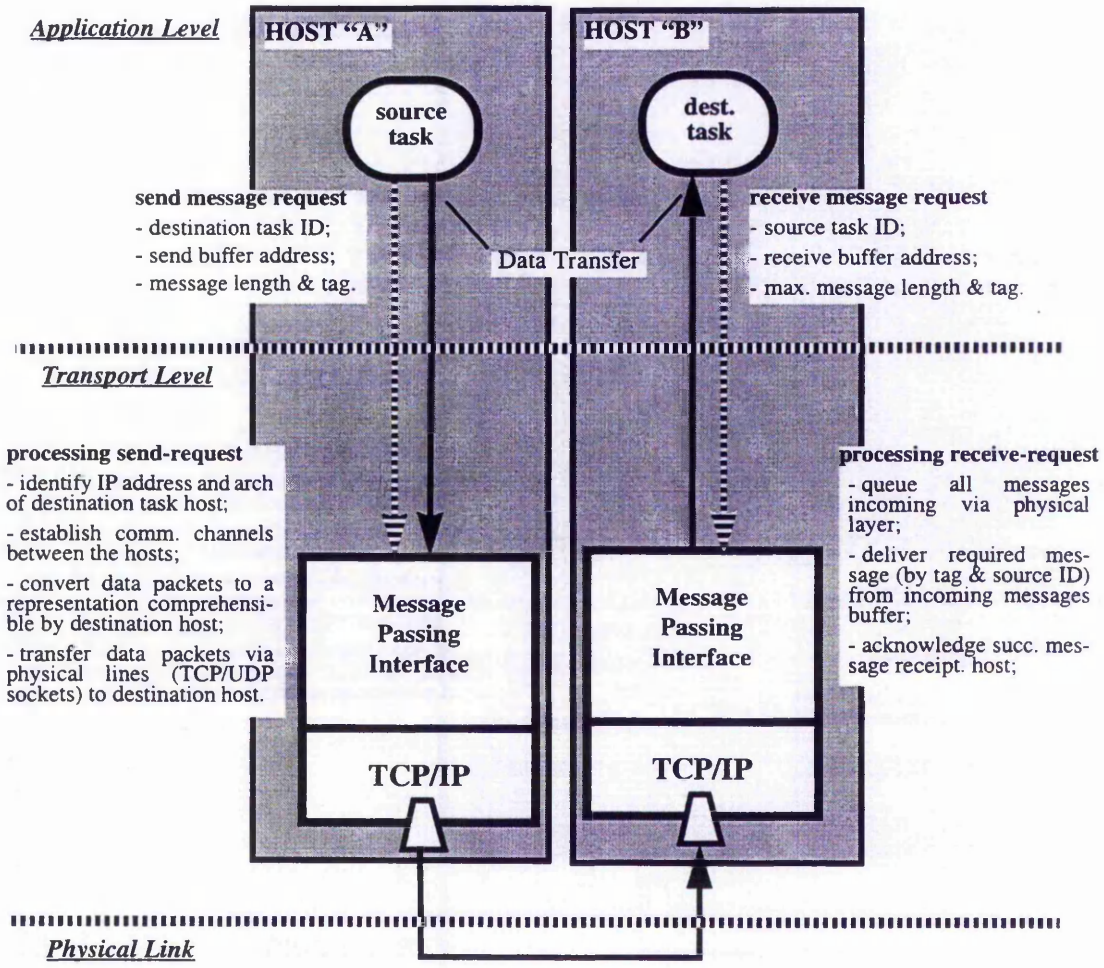


FIGURE 2-3 **The Communication Interface**

2.5.2 The Message Passing Interface

PVM (Parallel Virtual Machine) was adopted as the underlying communication medium for FADI. This message passing system has more users than any other parallel programming environment, and is a *de facto* standard for message passing environments [Mattson 94]. Many fault-tolerant systems rely on PVM for inter-process communication and process distribution and control (e.g [Stellner 95], DOME [Seligman 94], fail-Safe PVM [León 93]).

In addition to satisfying our essential requirements for message passing and process synchronization, the PVM library offers a rich set of programming tools that makes it

extremely attractive for any parallel programmer. Among these tools are: automatic spawning of user-tasks on the distributed system host, balancing of the computing load on the network nodes, dynamic process group operations, and many more.

PVM is a public shareware that can be downloaded using *anonymous ftp* and it delivers reasonable performance for heterogeneous distributed processing systems. In a study by I. Martin [Martin 95], investigating the performance of PVM compared to a two dimensional processor mesh architecture, it was found that although the processor-to-processor communication offered more efficient interprocess communications, it has been concluded that PVM is a suitable vehicle for building large scale distributed systems from the available workstations if the granularity of computations is coarse enough. With the availability of high speed communication networks (FDDI, Fast Ethernet, ATM), PVM can be an effective alternative to traditional supercomputers handling a significant class of large scale problems.

2.5.3 PVM: The Parallel Virtual Machine

The PVM software provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware. PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures [Geist 94].

The PVM computing model is simple yet very general, and accommodates a wide variety of application program structures. The programming interface is deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner. The user writes an application as a collection of cooperating tasks. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission.

Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

Owing to its ubiquitous nature (specifically, the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance scientific computing community.

The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output, and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism. In this method all the tasks are the same, but each one solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM can support well both functional and data parallelism.

Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case. An exemplary diagram of the PVM computing model is shown in Figure 2-4. and an architectural view of the PVM system, highlighting the heterogeneity of the computing platforms supported by PVM, is shown in Figure 2-5.

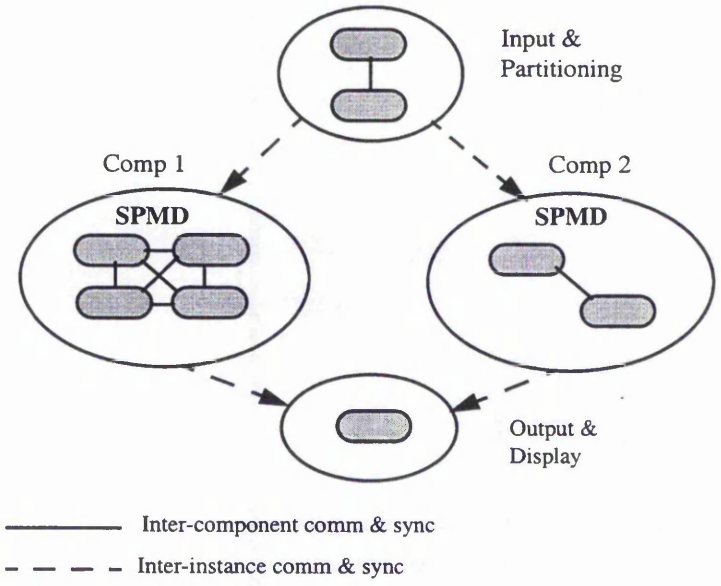


FIGURE 2-4 **PVM Computation Model**

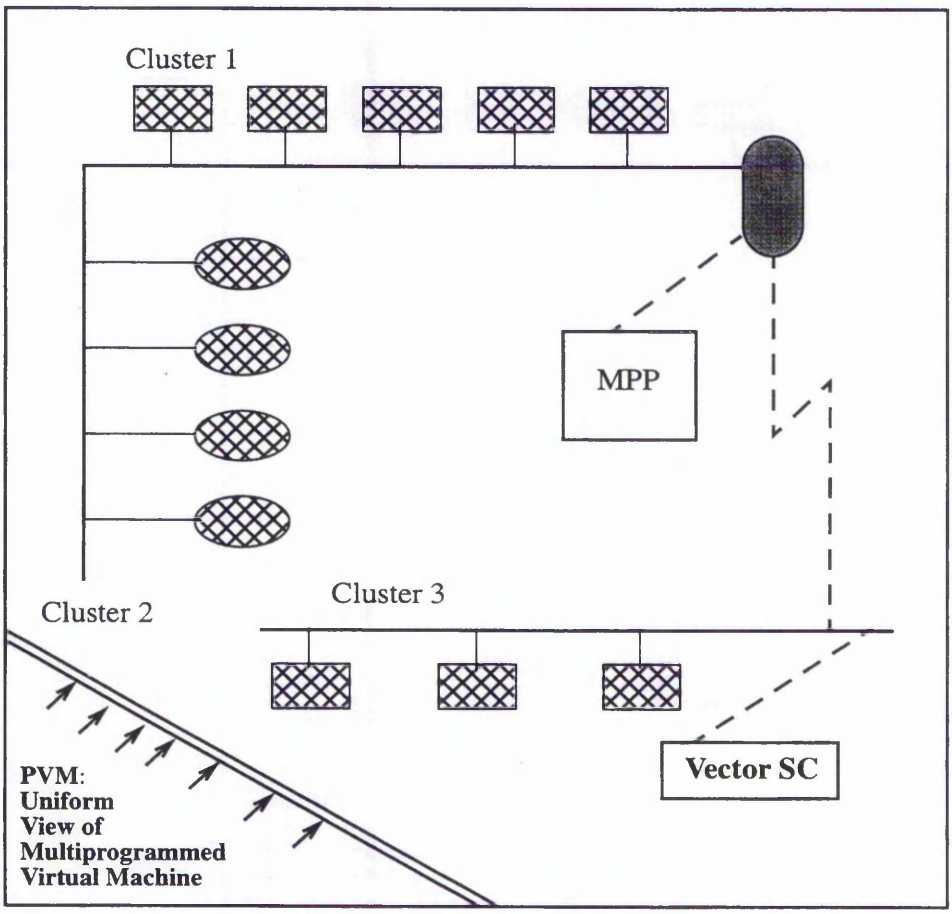


FIGURE 2-5 **PVM Architectural Overview**

2.6 Conclusions

This chapter surveyed several reliable computing systems and discussed the rationale for continuing research and improving the efficiency of fault-tolerant systems, which was the motivation behind developing FADI. It is argued that the development of a cost-effective, reliable, distributed computing environment that is transparent to the user/programmer of distributed applications is feasible. However, a research challenge is to develop new techniques for dealing with concurrent (communicating) applications and to assess their performance. FADI should provide automatic support for distribution of application processes and integrate an error detection facility for hardware permanent and transient faults with a low-overhead backup and recovery mechanism for the recovery of distributed application processes.

FADI targets scientific/engineering applications that are computation-intensive but do not have stringent real-time constraints. Consequently, it has been concluded that a checkpointing/rollback is the best-suited fault-tolerance technology for FADI. The checkpointing mechanism should have a small overhead during failure-free operation even at the expense of longer recovery time owing to the longer running time of the targeted applications and relatively large MTBF of modern computer systems.

PVM was chosen as the message passing interface for FADI. PVM provides a high level library interface for interprocess communication, process synchronization, and many useful tools for the parallel programmer as dynamic process-group operations. The Parallel Virtual Machine has a huge user-base support and research have certified that its parallel environment can be an efficient and economic alternative to some bespoke computer systems in handling some large scale problems.

CHAPTER 3 The Error Detection Mechanism (EDM)

This chapter presents the error detection mechanism for FADI. It gives an introduction to existing error detection techniques and subsequently defines the distributed system failure model. It explains the design and implementation of the EDM, and comments on the test results of the mechanism.

3.1 Faults in Computer Systems

There are a number of ways to classify faults in a computer system. From the error recovery viewpoint, it is useful to divide these faults into two groups:

External faults. These are faults caused by the interaction of the computer system with the outside world. Three types of failures are described below:

- Environmental faults: power supply, air conditioning, radiation, etc.;
- Design miscalculations of adequate safety margins built into the system to enable it to cope with any unusual load that may strain the processing or storage capacity of the system [Gibbons 76]. This is particularly relevant to distributed systems where resources such as disk space and RAM are shared by a number of processes or processors;
- Human operator mistakes can also affect the normal operation of a computer system. Some of the mistakes they can make are: overriding normal operation of the system by, for example, purging or locking files, aborting programs or responding incorrectly to console messages from the operating system or user programs.

Inherent design faults. These can be faults in any hardware component (CPU, memory, peripherals, storage media, etc.) caused by human or machine errors during the design or manufacturing stage of the production of the component. Inherent design faults can also be caused by mistakes made during the design or coding of the application program. Among the possible types are: Arithmetic errors (e.g division by zero, overflow); Data format errors (e.g. attempts to use character data as numeric); Violation of storage protec-

tion rules to disk files or process memory space; Attempts to execute privileged instructions.

This research is focused on the development of mechanisms for recovery from external faults only. The reason for that is that the inherent design faults are irrecoverable if there is no change to the physical hardware or software module.

However, detecting inherent design faults is useful during the development phase of the hardware or the software model of a computer system, and often *fault injection techniques* (the deliberate insertion of faults into an operational system to determine its response) are used for this purpose. FERRARI [Kanawati 95] is a fault injection tool that uses software methods to emulate hardware and software faults in order to validate the dependability properties of fault-tolerant computer systems and obtain statistics on parameters such as fault-coverage and detection latency. In the same way software-based error injection was used to evaluate the effectiveness of FADI error detection mechanism as discussed in section 3.6.

3.2 Error Detection Techniques

The starting point for all fault-tolerant strategies is the detection of an erroneous state, that is a state which, in the absence of any corrective actions, could lead to a failure of the system. Thus the success of any fault tolerant system will be critically dependent upon the effectiveness of the techniques for error detection.

From the user/programmer viewpoint, error detection measures can be classified into user-transparent and user-assisted mechanisms.

3.2.1 Transparent Error Detection

These checks are made by the error detection mechanism without the intervention of the application programmer, they include:

Replication Checks are one of the most powerful measures for detecting errors in a computer system, but are also among the most expensive in terms of resources required. The replicates must of course run on separate processing nodes. Errors are detected by comparing equivalent output messages from different replicates of a software components. The Delta-

4 Open Distributed Computing System [Powell 88] outlined some of the consistency conditions for such a comparison to be possible: first of all it is necessary that replicates on non-faulty nodes remain consistent so as to produce the same output messages. It is also necessary that there be some way of identifying equivalent messages, i.e messages that should be the same and can thus be compared. Each replicate must also receive the same messages in the same order from all the distributed system tasks. It is clear that the overhead of maintaining the consistency increases if the software replication checks have no information about the behavioural model of the application, which is a necessary precondition for maintaining the transparency of the EDM to all the distributed applications.

The advantage of this method of error detection is that the replication of software components and the associated error processing protocol serve not only for detecting the errors but also for the recovery from them. In order to tolerate "n" active faults, at least "2n+1" replicates are necessary for there to be a majority of replicates executing on non-faulty hosts.

A different approach to replication checks is adopted in [Cin 93]. In order to avoid the large overhead resulting from multiple modular redundancy for the whole system, they restrict the redundancy to the duplication of the most crucial hardware resources such as the CPU.

Duplication inside the computer nodes of the system is based on the *master-checker* (MC) mode. With this mode, both processors run fully clock-synchronously the same program and process the same data stream. Only one processor (the master) exchanges data with the outer world via the bus. The output bus drivers from the other processor (the checker) are disabled. During a data transfer the comparators on the checker pins compare the internal signals generated by the checker and those driven by the master. In the case of a mismatch an error is signalled.

The problem with this technique is that it requires modifications to the hardware architecture of the computer system, and therefore it is not applicable to off-the-shelf computer systems.

Bearing in mind the class of applications for which this research has set to develop the fault-tolerant environment, the replication technology was rejected as a recovery method for FADI on economic grounds.

Timing Checks. If the specification of the system includes timing constraints on the provision of service then a timing check can be provided in the system to determine whether the operation of the component meets these constraints. If the constraints are not met then the timing check can raise a "timeout" exception to indicate the failure of the component.

Such checks are implemented by an error detection algorithm implemented for the *Parsytec* [Altmann 95] massively parallel multiprocessor. Parsytec is built from units of 16 processing elements PEs (INMOS T805 transputers) which are interconnected by a two-dimensional grid. Timing checks are performed on each processor. With this algorithm there is no central testing node, results are obtained by self-tests sent by fault-free nodes within a predefined time-out limit to all neighbouring processors (*<I'm alive>* messages). On receiving a message, each neighbour compares the received self-test result with local reference values received previously from the same node. At the initial phase, all processors have an initial, *system-level diagnostic image* (i.e about the state of all the system processors not only the four neighbouring PEs). An error is detected if the *<I'm alive>* message does not arrive within the predefined time-out interval, or if its value does not match the local reference value (contains information about a faulty non-neighbouring processor). The algorithm also can segregate the diagnostic image into subgraphs if a set of faulty processors isolate a group of fault-free processors.

The overhead of the complexity of the algorithm incurred by the mesh connection of Parsytec PEs does not apply to FADI hardware module, where the topology of hardware platform is a group of autonomous processors connected by a common network.

System Error Detection Mechanism (EDM). Most operating systems have built-in error detection mechanisms that monitor address, data and control buses. The Sun SPARC system has detection mechanisms that include traps for illegal instructions, bus errors, segmentation faults, arithmetic exceptions, etc.

Illegal instruction faults can arise from attempts to execute an instruction with an unidentifiable opcode or inadequate operands. Bus errors result mostly from address line errors generated when a program seeks an illegal memory location (e.g non-aligned address value). One of the situations that can lead to a segmentation fault is writing to the read-only text segment. Improper mathematical operations (e.g division by zero or the square root of a negative number) raise arithmetic exceptions.

Application programs affected by these errors are usually terminated by the OS kernel with a signal number corresponding to the trapped error type.

A system dependability study in [Kanawati 95] established that 43% of the transient errors generated by fault-injection were detected by the OS built-in error detection and protection mechanisms. Hence, it is crucial that the error detection mechanism of FADI monitors the OS management of the application processes.

3.2.2 User-Assisted Detection

These techniques require the user/programmer to layout the strategy for detecting errors. These measures are based on knowledge of the internal design and construction of the application program and the computing environment in which it is executing. An example of this detection method is accounting checks. These checks are suitable for transaction-oriented applications with simple mathematical operations such as air line reservation systems, library records, and the control of hazardous materials. The simplest form of accounting checks is the cheksum. Whenever a large number of records is transmitted or received, a tally for both the total number of records and the sum over all records of a particular data field can be compared between source and destination.

Another type of programmer-induced error detection is reasonableness tests. These tests detect software failures by use of pre-computed ranges, expected sequences of program states, or other relationships that are expected to prevail.

An illustration of reasonableness checks is the determination of the airspeed in a flight control system. The speed of the aircraft must be within the structural capabilities of the air frame (e.g 140-1100 km/h). Thus if the true air speed is outside this range, then there is something wrong with either the sensor or the computer [Pradhan 86].

These error checks are of interest to the fault tolerant environment if they can diagnose a fault in the computer hardware as in the previous example. Then measures can be taken to migrate application programs to operative computing machines or switch execution to a redundant hot stand-by system.

3.3 Application, Environment and Failure Model

The distributed system consists of a number of nodes (processors) that can run concurrent user-tasks. Nodes communicate via a message passing interface over an asynchronous network. It is assumed that a central host will run the main error detection task. This central host must be fault-tolerant, i.e the probability of its failure is negligible. The required reliability of the central host might be obtained by hardware duplication. In the interest of portability and ease of use the error detection mechanism should be transparent to the hardware nodes and the application tasks, i.e no intervention from the user/programmer in the setup of the detection mechanism should be necessary [Storm 87].

The following assumptions are made about the detection environment and failure model:

- We assume that the processing nodes are fail-silent, i.e they only send correct messages, or nothing at all. However, the processes are not required to be fail-safe, i.e with a zero error latency.
- The detection mechanism should cover processor node crashes, as well as transient hardware failures (temporary memory flips, bus errors, etc.) that cause the failure of a single application task.
- The detection system should have a straight-forward interface to integrate user-assisted (application-specific) error checks into the detected errors dictionary.

3.4 Design of the EDM

Throughout the work, emphasis was put on the modularity of the design to ensure that the EDM can be smoothly integrated into a Fault-Tolerance System that will provide process recovery from the detected errors.

At the initialisation stage the EDM accepts validated system specifications (host & task specifications entered by the user are checked against the network configuration) from the user/programmer as shown in Figure 3-1. These specifications are then broadcasted to the *Host* and *User-Tasks* monitoring subtasks, they detect host crashes and recovery and user-tasks failure and recovery and subsequently update the *active host* and *active task* tables. These tables are shared with the fault-tolerance system *recovery mechanism (RM)* and are

used in the recovery process of the failed tasks. IDs of manually recovered hosts are also fed to the system for them to be subsequently monitored by the EDM.

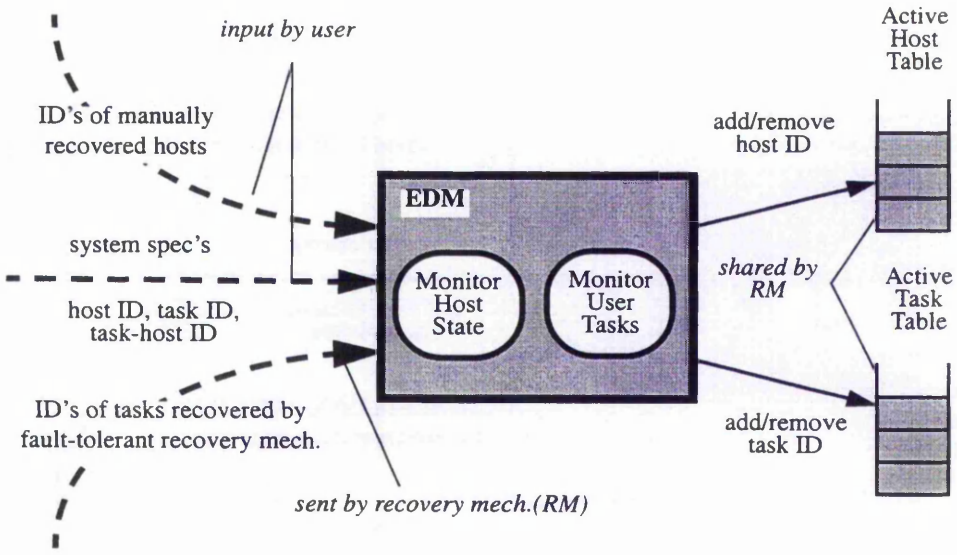


FIGURE 3-1 **The Error Detection Mechanism**

3.4.1 Detecting Host Failures

Detection of host failures is based on a central host monitoring task, running on the master host, which is responsible for the coordination of host crash detection in all the system nodes. This central host monitoring task (Figure 3-2) periodically sends acknowledgment requests to all the hosts in the system.

Each host must reply to the acknowledgment request within a predefined time interval "tack_timeout", otherwise it will be considered by the monitoring task as having "crashed". If the reply is received at a latter acknowledgment cycle (because of network delay) or if a message is sent by the user declaring that the host has been manually recovered, then the host is considered as "recovered", and it is reinstated to the system host pool.

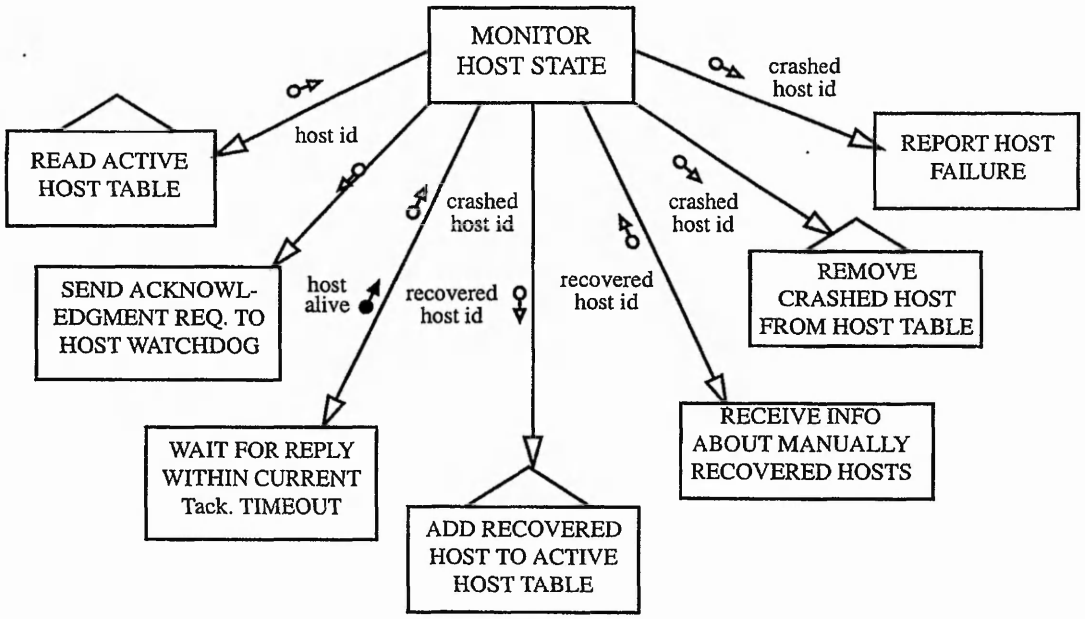


FIGURE 3-2 **The Host Monitoring Task**

The difficulty associated with this detection technique is the calculation of the acknowledgment timeout. Research in the Delta-4 system [Powell 88] asserted that the total message traffic between components of a dynamically evolving system (e.g, a multi-user shared communication inter-link as the Ethernet) cannot in practice be assumed to be deterministic. Thus it is impossible to know a *priori* how quickly acknowledgments return to the source. The total time-out required for a segment to travel to the destination and an acknowledgment to return to the source varies from one instant to another. Figure 3-3 which shows measurements of round-trip times across the network for 100 consecutive packets illustrates the problem.

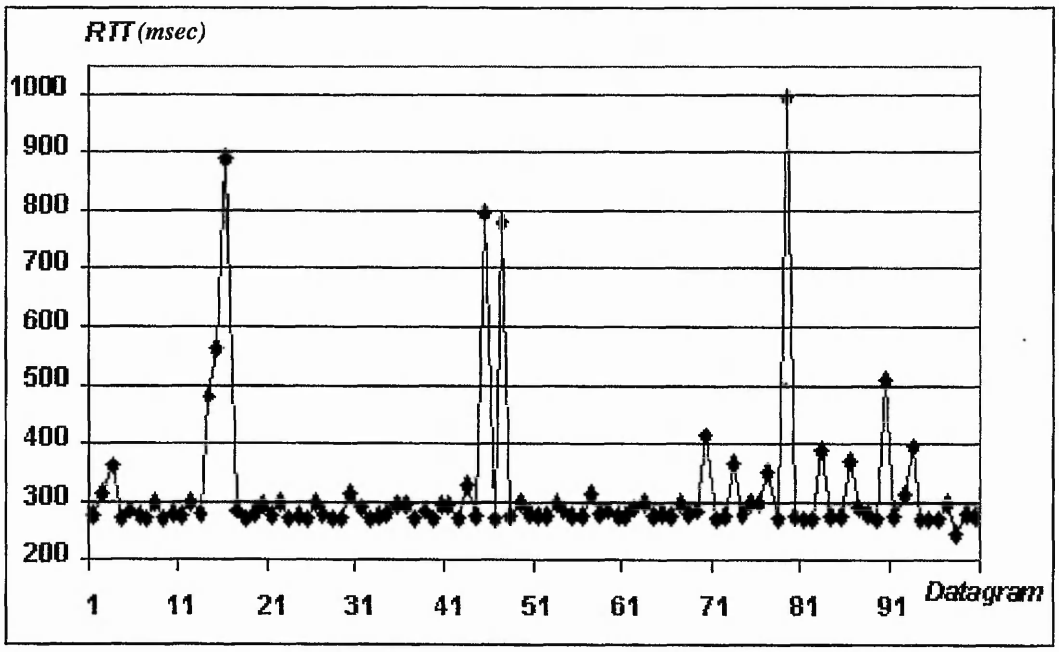


FIGURE 3-3 **Plot Of Network Round Trip Time (RTT)**

To accommodate the variation in the network delays, it is necessary to dynamically calculate the round-trip time *(RTT)* of inter-host messages with relation to the current network traffic, and update the acknowledgment timeout accordingly. A background task was designed to send dummy messages between pairs of machines in the distributed system. The difference in time between sending the dummy message and getting the acknowledgment is used to estimate the network round-trip time. Each new round-trip sample *(Srtt)* is filtered into a "*smoothed*" estimate according to the formula:

$$RTT_{i+1} = \alpha \times RTT_i + (1-\alpha) \times Srtt \qquad \text{(EQ 3-1)}$$

where RTT_i is the current estimate of round-trip time, RTT_{i+1} is the new computed value, and α is a constant between 0 and 1 that controls how rapidly the estimated *rtt* adapts to the change in network load.

The EDM's acknowledgment time-out *(ATO)* can then be computed from RTT_i. The formula is:

$$ATO_i = \beta \times RTT_i \qquad \text{(EQ 3-2)}$$

Where β is a constant, greater than *"1"*, chosen such that there is an acceptably small probability that the round-trip time for the packet exceed ATO_i.

The underlying implementation of the FADI distributed system interconnection (PVM message passing interface) uses the TCP/IP protocol to transfer datagrams between the hosts. The specification of this protocol [Postel 81] suggests values in the range of "0.8" to "0.9" for α and "1.5" to "2.0" for β.

Representative samples of the network round-trip time were taken during different intervals when the traffic of our academic network is expected to vary: Early in the morning, when most users are logging in at approximately the same time, creating extensive short overload on the network; in the early afternoon, when the number of users and the utilisation of the network is at its maximum; and late at night when the network usage is minimum. The measurement intervals were 4 hours each amounting to a total of 12 hours. The pattern of the network delay was similar to the *network RTT* measurements on Figure 3-3 and shows that the round-trip times are roughly Poisson distributed, but with brief periods of high delay. Using the standard way of calculating *RTT* and *ATO (equations 3-1 and 3-2)* with the TCP recommended values for α (average of 0.8, 0.85, 0.9) and β (2.0), the estimated *RTT* could not adapt swiftly enough, resulting in a an average of 37 transgressions or incorrect diagnoses of host failure over the 12 hours period. Considering the need of reconfiguration of the distributed system at the detection of the fault, which involves the migration of user-tasks to an *operative* host, this estimation algorithm has proven very costly.

An improvement to the algorithm was proposed by Mills [Mills 83]. He suggests a non-linear filter, that will allow the RTT_i to adapt more swiftly to sudden increases in network delay. The change amounts in using two values for α, one (α1) when $Srtt < RTT_i$ in equation 3-1, and the other (α2) when $Srtt >= RTT_i$, with α1 >α2, i.e:

$$\alpha = \begin{cases} \alpha_1 & : \; S_{rtt} < RTT_i \\ \alpha_2 & : \; S_{rtt} \geq RTT_i \end{cases} \qquad \textbf{(EQ 3-3)}$$

The affect is to make the estimation more responsive to upward-going trends in delay and less responsive to downward-going trends. For the purpose of estimating the acknowledgment time-out for FADI error detection mechanism, this method helps the RTT estimate to

follow sudden increases in network delay and smooth-out sharp drops, thus minimizing the transgressions of the calculated *ATO* over the *Srtt*.

Tables 3-1 and 3-2 present the accuracy of the *RTT* estimation and a number of incorrect diagnoses of the host failure (*ATO < Srtt)* for the above experiment respectively, where:

$\alpha 1 \in \{0.8, 0.91, 0.94, 0.95\}$ and $\alpha 2 \in \{0.67, 0.75, 0.83, 0.9\}$.

TABLE 3-1 **Accuracy of Round-Trip Time Estimation**

α2 \ α1	0.8	0.91	0.94	0.95
0.67	91.46%	89.87%	88.76%	87.80%
0.75	92.26%	90.78%	89.81%	88.97%
0.83	✖	91.84%	91.05%	90.36%
0.87	✖	92.43%	91.77%	91.18%
0.9	✖	92.81%	92.25%	91.73%

TABLE 3-2 **Num. of Incorrect Diagnoses of Host Failure**

α2 \ α1	0.8	0.91	0.94	0.95
0.67	6	2	1	1
0.75	6	3	1	1
0.83	✖	4	3	2
0.87	✖	4	3	2
0.9	✖	7	3	3

Unfortunately, what is good for reducing the latency of detecting the errors (more accurate estimates of *RTT*) is disastrous for diagnoses of host failures. If the RTT_i is very close to *Srtt,* this results in a large number of incorrect fault diagnoses, and subsequently unnecessary reconfiguration of the distributed system and migration of user-tasks (7 incorrect diagnoses for the highest measured accuracy "*92.81%*"). From the above tables I concluded that values $\alpha 1=0.94$ and $\alpha 2=0.75$ (with $\beta=2$) gave the best balance between the error latency and fault diagnoses.

This host crash detection technique makes effective use of fault-tolerant master-host and affords flexible mapping between the logical and physical connectivity of nodes.

An alternative technique used in STAR [Sens 93], operates a *logical ring of crash detection*, where each host only checks its immediate successor in the ring. Although this technique reduces the message traffic, it introduces a limitation associated with the dependence on network structure. The logical structuring of the crash detection ring must match the physical connectivity of the nodes in the network, which is not always possible, e.g the user/programmer might choose to omit certain computing nodes from the logical structure of the distributed system.

3.4.2 Detecting Application-Task Failures

Transient hardware failures can cause the failure of the application processes. Most of these errors are detected by mechanisms built in the system hardware and operating system kernel. These mechanisms include traps for illegal instructions, bus errors, segmentation faults, interrupts, arithmetic exceptions, etc. An elaborate study for the Sun SPARC system EDM can be found in [Young 74]. The errors that cannot be revealed by the system EDM, are classified as inherent design faults and imply the need to re-design the hardware and/or the OS kernel.

Upon the detection of an error the kernel EDM kills the affected process with a signal number that corresponds to the error type. Hence, the process monitoring user-tasks operates as follows: Initially, it receives the ID's of the active user-tasks, and waits indefinitely for a task to exit. When a user-task exits, the monitoring process analyses the task exit status to determine whether it exited normally or prematurely due to a failure. In both situa-

tions the task is removed from the active tasks table, but the in case of task failure, the fault is reported to the EDM as illustrated in Figure 3-4.

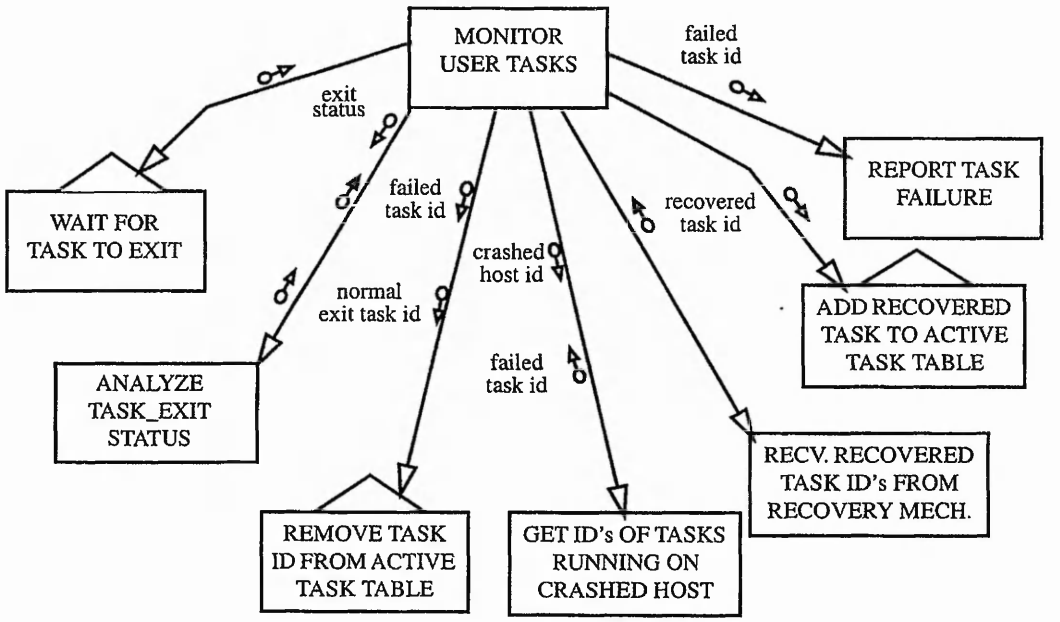


FIGURE 3-4 **User-Task Monitoring**

The monitoring process also receives notifications from the fault-tolerance recovery mechanism about the recovered tasks and includes their ID's in the active tasks table.

With regard to the user-assisted error detection, a special signal handler was dedicated to service the detection of such errors. All that the programmer has to do is to raise an interrupt with a predefined signal number and the detection mechanism will handle the error as if it was raised by the kernel EDM.

3.5 Implementation Details

3.5.1 Terms and Definitions

FADI error detection mechanism was implemented using the *Parallel Virtual Machine* (PVM) as the underlying message passing interface. PVM offers via a high-level interface most of the facilities required for distributed programming: process control, inter-task communication, and process synchronization in a heterogeneous environment. The main PVM routines used for the implementation of the FADI EDM system are listed below:

- **pvm_spawn()** - starts a new process on the specified host;
- **pvm_barrier()** - blocks the calling process until all processes in a group have called it;
- **pvm_send()** - sends a message to another process (including processes in remote hosts);
- **pvm_recv()** - receives a message (the calling process is blocked until msg is received);
- **pvm_nrecv()** - non-blocking receive;
- **pvm_trecv()** - receive with timeout;
- **pvm_notify()** - requests notifications of a certain event in the parallel machine.

The use of the PVM interface for process control and interprocess communication requires a minimum level of application code modifications. In addition to linking applications with the FADI EDM library, the application code must include two statements at the start and the end of the *main()* function: *pvm_mytid()* and *ed_pvm_exit()*. The first routine is necessary to enrol the task in the PVM system and the second is for the EDM to identify the task upon successful termination.

The following definitions are used in the functional description of the error detection mechanism:

- **active_hosts**: nodes that are currently considered as "operative" in the distributed system;
- **tack_timeout**: maximum acceptable interval to wait for host acknowledgment. This variable is dynamically updated in relation to the network load;
- **host_group**: dynamic group of processes that contains the *Host_Monitoring* task running on the master host and the *host_watchdog* tasks running on active_hosts;
- **hn**: number of active_hosts;
- **active_tasks**: user-tasks currently running in the distributed system;
- **received_message_type**: identification tag attached to the message;
- **Successful_Termination**: user-task was executed successfully;
- **Wait_Exit**: user-task exited;
- **Task_Recovered**: failed user-task recovered by the fault-tolerance recovery mechanism.

3.5.2 Functional Description

Functional description of the EDM tasks is shown in Figure 3-5, 3-6 and 3-7. Host crash detection is initialised by starting the *host_watchdogs* on the active hosts. The sole task of a host_watchdog is to send an acknowledgment message to the central *HostCrashDetection* task upon the receipt of an ack_request. pvm_barrier() is used to synchronize communication between the *HostCrashDetection* task and *HostWatchdog* tasks.

After each detection cycle, the EDM scans for recovered hosts. Firstly it checks if there are any pending acknowledgments from hosts considered as crashed at the previous detection cycle(s). The delay in receiving these messages might have been caused by network overload, but the host was considered failed regardless, because the delay exceeded *tack_timeout*. If that is the case, then contact with the recovered host is reinstated. Next the system checks if there is any notification sent by the user about manually recovered hosts, the recovered hosts are added to the active-hosts pool, and the watchdog task is restarted on them.

```
Initialize()
   pvm_spawn( host_watchdog on active_hosts )
   pvm_barrier( host_group, hn+1 )
   HostCrashDetection()
end Initialize

HostCrashDetection()
   repeat
     pvm_send(ack_requests to host_watchdogs in active_hosts )
     for each active_host do
       pvm_trecv( replies to ack_requests from host_watchdog /
                  in tack_timeout )
       if reply not received then
         remove host from active_host_table
         report the host failure
         set tack_timeout to zero
       endif
     end
     ScanForRecoveredHosts()
   until stopped
end HostCrashDetection

ScanForRecoveredHosts()
   pvm_nrecv( delayed reply from failed (considered crashed) host )
   if reply received then
     add host to active_host_table
   endif
   pvm_nrecv( notification about manual host recovery from user )
   if notification received then
     pvm_spawn( host_watchdog on recovered host )
     add host to active_host_table
   endif
end ScanForRecoveredHosts
```

FIGURE 3-5 **Central Host Monitoring Task**

```
HostWatchDog()
   pvm_barrier( host_group, n+1 )
   repeat
     pvm_recv( ack._request from detection task on mas-
ter host )
     pvm_send( reply to the ack_request )
   until stopped
end HostWatchDog
```

FIGURE 3-6 **Host Watchdog Task**

Monitoring the user-tasks starts by executing pvm_notify(). With *TaskExit* as an argument, this function causes all user-tasks registered in the distributed system to send notification

messages with message tag "Wait_Exit" to the calling task upon their exit. However, this notification does not manifest itself if the task exited abnormally or due to a fault. In order to overcome the last problem the pvm_exit() routine has been overloaded, and it automatically sends a message tagged with "Successful_Termination" declaring the successful execution of the user-task. Because PVM guarantees the order of messages delivered from one source, if a Wait_Exit message is received before Successful_Termination message from the same task, this means that a failure occurred and the task was abnormally aborted.

As can be noticed from Figure 3-7, the monitoring task is *message driven*, i.e managed by analysing messages sent either by the user-task (Wait_Exit and Successful_Termination) or the fault-tolerance recovery mechanism (Task_Recovered).

```
MonitorUserTasks()
   pvm_notify( about TaskExit of active_tasks with msg_type /
            Wait_Exit)
   repeat
      pvm_recv( any message from active_tasks )
      case received_message_type of :
         Successful_Termination:
            add task to normal_exits record
            break
         Wait_Exit:
            if task in normal_exits record then
               report successful task termination
            else
               report task failure
            endif
            remove task from active_task_table
            break
         Task_Recovered:
            add task to active_task_table
            pvm_notify( about TaskExit of recovered_task )
            break
      end case
   until all tasks exited successfully
end MonitorUserTasks
```

FIGURE 3-7 **Monitoring User-Tasks**

Taking into consideration the centralised error detection and the delays to the message passing interface that might be caused by the overload of the network, the error latency of detecting task failures can be calculated with 99% confidence by the formula:

$$Tlatency_i = E(Tedm) + 3 \times \sigma(Tedm) + RTT_i \qquad \text{(EQ 3-4)}$$

where:

E(Tedm) - average reaction time of the kernel error detection mechanism;

σ(Tedm) - standard deviation of the Tedm;

RTT_i - estimated round-trip time from the master host to active_hosts.

3.6 Testing The EDM System

In order to test the reaction of FADI error detection mechanism to permanent hardware faults, two processor node crashes were engineered:

1) powering down one of the distributed system nodes (SPARCstation *IPC*). In this case, the error latency was 0.63 seconds upon network load of: (Trtt = 76 msec).

2) disconnecting a processing node from the Ethernet. The error latency was 0.58 seconds under (Trtt = 52 msec). When the connection was restored, the EDM diagnosed that the crash was caused by network delay and reinstated the "recovered" host to the active host pool.

Three types of hardware transient failures were simulated by software-based error injection to test the efficiency of error detection for failed application tasks: a) Bus errors were simulated by attempting to access a non-aligned (even) memory address; b) Segmentation faults were generated by performing a write to a read-only memory location; c) An attempt to calculate the square root of a negative number was performed to raise an arithmetic exception. Table 3-3 gives a representative sample of results for "bus error".

Since the test experiments in Table 3-3 table were performed under approximately the same network load conditions, i.e implying the same Trtt, then according to formula 3-4, the error latency is determined by the reaction time of the OS kernel EDM (Tedm). Therefore, nodes running the same operating system have been found to have similar error latency.

TABLE 3-3 **Detecting Transient Hardware Faults**

Master Host & Arch. / OS	Remote Host & Arch. / OS	Latency (seconds)
Host1 SPARC multiprocessor SOLARIS 2.4	Host1	0.974
_"_	Host2 SPARCstation SOLARIS 2.4	0.802
_"_	Host3 SPARCstation SOLARIS 2.4	0.731
_"_	Host4 SPARCstation SUNOS 4.2	0.291
_"_	Host5 SPARCstation SUNOS 4.2	0.286
_"_	Host6 HP-APPOLO 400 HP-UX08	0.305
_"_	Host7 HP 340 HP-UX08	0.343

3.7 Conclusions

This chapter presented an efficient user-transparent error detection mechanism for distributed systems. The detection mechanism covers processor node crashes and hardware transient failures (e.g bus errors, segmentation faults, etc.). The EDM also enables integration of user-programmed error checks into the error detection mechanism.

We implemented a modular and efficient centralised error detection structure, where the main error detection modules (remote-host & user-process monitoring tasks) run on a failure-free master host. It is assumed that the processing nodes are fail-silent, and therefore, faults in the communication link (e.g messages delivered with erroneous content, or the sending of extra messages) are not considered. The detection mechanism does however, cater for a possibility of propagation of errors in the distributed computing system by allowing a non-zero error latency.

Since this research is motivated by developing a reliable support environment for *distributed* computing, message traffic on the inter-connection network might affect the error detection latency. Therefore, a mechanism was implemented that dynamically measures the round-trip time of the underlying network and updates the host acknowledgment time-out and error latency accordingly.

The detection mechanism was tested on a set of heterogeneous workstations connected by Ethernet and the results show that the system is viable for network distributed computing.

CHAPTER 4 Backup and Recovery of User-Application Processes

Chapter 4 begins with the explanation of the basic concepts of checkpointing and rollback recovery. It then reviews current trends in checkpointing technology and introduces FADI checkpointing mechanism and the measures taken to reduce its overhead on the run-time of the checkpointed application. The chapter is concluded by evaluating the performance of the checkpointing/rollback mechanism and summarizing the research results.

4.1 Saving the State of the Application Process: An Introduction

This research focuses on scientific and engineering applications that are computation-intensive but do not have stringent real-time constraints. While the required level of reliability for such applications necessitates the prevention of the loss of the results of the long-running computations, it does not justify the use of expensive hardware replication fault-tolerance techniques. Checkpointing and rollback provides support for fault-tolerance without requiring the duplication of the underlying hardware or the replication of the application processes.

4.1.1 The Checkpointing Process

To checkpoint an application, its entire state is periodically saved into stable storage. In the case of a fault, the system can be restored (rolled-back) to a previously valid checkpoint state. With checkpointing in place, loss of computation can be reduced to that which is performed between checkpoints.

The checkpoint contains the complete state of a process. It is a fairly complex task to determine the smallest necessary contents of the checkpoint. It includes *system* - as well as *application* parameters. For a UNIX process the system parameters include the state information about the process maintained by the kernel, i.e the state of the process registers, any special handling requested of various signals, and the status of open files and file descriptors. The application parameters mainly consist of the process address area (text, data, and stack segments). If the checkpoint is saved as a hot backup, i.e a freezed live

copy of the original process, then rolling-back will imply simply activating the checkpoint.

If the checkpoint is stored as a cold backup "to disk", then at the time of recovery, the process is loaded as it would be if it was started for the first time. Its address space is then allocated and copied from the recorded memory information.

4.1.2 Checkpoint Interval

The checkpointing frequency (and thus the length of the checkpointing interval) has a large impact both on the overhead during failure-free operation and on the work-progress that will be lost when rolling back. The optimal checkpointing interval is mainly dependent on the failure-rate of the system, on the time it takes to do the checkpointing (failure-free overhead) and the user/programmer consideration of the acceptable loss in computations due to a failure.

Several methods for adapting the length of the checkpoint interval (or to adjust the checkpointing frequency) can be given. The most important ones include:

- determination by the programmer *when* the checkpointing-routines are invoked. Chandy and Ramamoorthy [Chandy 72] originally proposed a graph-theoretic method by which the programmer could decide where to insert checkpoints. The program is decomposed by the programmer into a sequence of tasks between which the checkpoint can be inserted. It is assumed that the execution time, the checkpointing time and the recovery time for each of these tasks is known in advance. With this information their algorithm can determine the optimal places to insert checkpoints so that the checkpoint time and run time can be minimised. A similar approach was taken by Toueg and Babaoglu [Toueg 84]. However, as indicated by Chandy and Ramamoorthy, this approach requires a large effort on the part of the programmer to partition the program, to determine the execution time of program segments, and to select optimal checkpoints.
- invoking the checkpointing after k messages have been sent [Bhargava 88]. This approach is application-dependent and is beneficial only if the number of sent messages indicates that a useful computation cycle is executed and checkpointing is necessary to safe-guard it.

- invoking the checkpointing after *t* local clock ticks (when a pre-stable timer gets off). In [Young 74] a first order approximation to an optimal checkpoint-interval *(t clock ticks)* with respect to checkpointing duration and MTBF (mean time between failure) is given. In [Geist 88] an optimal checkpoint interval for transaction processing is given, as a function of system down-time. It maximises the probability of critical-task completion on a system with limited repairs.

This research aims to develop a *generic* environment for the reliable execution of distributed applications. Therefore, the checkpointing interval has to be determined in an application-independent manner, which favours the last approach. Additionally, in a distributed processing environment, the last approach alleviates the complexity of finding consistent recovery-lines of the distributed application tasks because all checkpoints can be taken approximately at the same time interval and hence reduces roll-back time of the application. In this work checkpointing is invoked at regular intervals that are determined by the user/programmer according to the reliability requirements of the application.

4.2 Review of Checkpointing Technologies

While there has been a great deal of research into checkpointing algorithms aiming at the derivation of consistent checkpoints, there is very little published work on checkpointing implementation and performance. From the transparency of implementation point of view, reported checkpointing techniques can be broadly classified into two categories:

4.2.1 Checkpointing Built into the Operating System

With these techniques, the checkpointing/restart protocols were either envisaged upon the development of the operating system, or part of the system kernel was re-designed and modified to accommodate the fault-tolerance facilities. KeyKOS and UNICOS are two examples of systems using such checkpointing technology:

‡ KeyKOS

The KeyKOS [Landau 92] is an object-oriented microkernel operating system. KeyKOS achieves persistence of objects by taking frequent system-wide checkpoints of the entire system state to disk. On restart from a failure, the entire system is restored to the state of

the last checkpoint, and processes resume execution as if there had been no interruption. The KeyKOS kernel was built around the idea of having reliable persisting objects, therefore, many of the system resources were built to support checkpointing, including the paging system, virtual memory and non-volatile storage swapping. For example, paging of virtual memory is integrated with the checkpointing mechanism, allowing KeyKOS to achieve high disk I/O performance while writing out the checkpoint. A drawback of the system is that all the processes running in the kernel domain are stopped until a snap shot of the entire system can be taken regardless of the behavioural model and reliability requirements of each individual application.

‡ UNICOS

UNICOS [Attig 93] operating system provides facilities for checkpointing of NQS Batch Jobs running on CRAY systems. It provides automatic checkpointing for all tasks running under its kernel, and also offers the user/programmer the possibility to change the time interval between two checkpoints and to switch to user-triggered checkpointing. The UNICOS spawns a checkpointing daemon that engages in a wait loop, waiting for a signal to arrive. The signal can be sent by the system alarm, by a user-defined timer, or directly from a user program (in case of user-triggered checkpointing). Upon receipt of the signal, checkpointing is performed for the corresponding NQS jobs which can be restarted from the saved checkpoint files. CRAY UNICOS report claims that the CPU time overhead induced by the additional checkpointing activity is negligible. For a 10 CPU hours, using the default checkpointing interval of 30 minutes and considering a normal production batch job (with average memory usage), the total CPU time spent for the checkpointing feature is about 240 milliseconds. In spite of the impressive performance, checkpointing in UNICOS is limited not only to the hardware platform of CRAY systems, but also to the NQS requests applications.

The conclusion is that despite the efficiency demonstrated by operating system built-in checkpointing/rollback mechanisms, they are not portable outside their environment, and are specialized in certain class of applications. Moreover, the extensibility and reusability of these mechanisms is virtually limited to their developers because of the necessity of a low-level knowledge of the OS kernel design and architecture to add/modify fault-tolerance library modules.

4.2.2 Checkpointing Built on the Top of the Operating System

These techniques use existing operating system facilities to save the execution state of the process, without the need for processing OS kernel libraries. Among these systems the following are reviewed:

‡ System kernel core dump

This checkpointing technique was developed as part of the Condor software package [Briker 91]. Two components of the UNIX process must be taken into consideration in order to save its state: the process address area (text, data, and stack segments), and the state information about the process maintained by the kernel. The state of the process registers, any special handling requested of various signals, and the status of open files and file descriptors.

The idea is to create a new checkpoint file from pieces of the previous checkpoint and a core image. The checkpoint itself is a unix executable file ("a.out") [Litzkow 92]. While core files are generally intended to aid in debugging a failed process, they also serve as a portable mechanism for saving the state of a process at a given point in time.

In order to save the process state a kernel core dump has to be produced by sending a termination "SIGQUIT" signal to itself.

The text for the new checkpoint (executable) is of course an exact copy of the text from the original. This core dump is then processed to copy the data area recorded in it to the initialised data area of the new executable file (Figure 4-1). The saved stack area is also copied into the new executable in a section which is not normally used by the UNIX process initialisation mechanism. At a later stage the old stack is restored and the program counter (PC) is set to resume execution where it stopped in the original process.

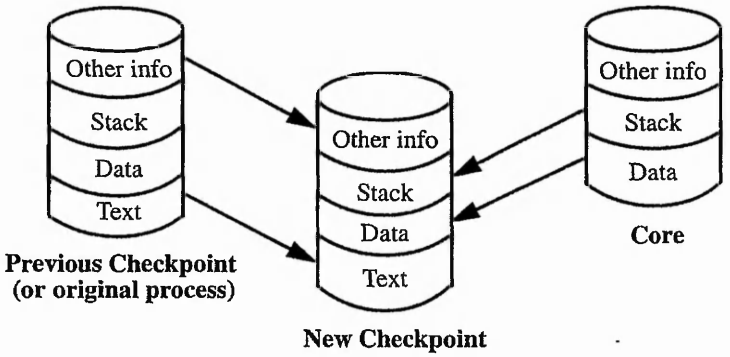


FIGURE 4-1 **Creating a New Checkpoint By Kernel Core Dumping**

A disadvantage of this method is the large size of disk space that the core dump occupies which implies significant network communication overhead. Moreover, the extensive manipulation of the data in the core to produce a core dump requires dealing with highly platform specific formats of the "core" and "a.out" files which restricts the system portability.

‡ A Checkpointing Mechanism for Mach3.0/UX

Russinovich and Segall in [Russinovich 95] present an application-transparent checkpointing mechanism for Mach 3.0/UX operating system. Their checkpointing implementation relies on a Mach 3.0 property that allows a process to use a pager that is *external* to the kernel to manage its memory. The checkpointing implementation uses this feature to add memory management to Mach 3.0/UX - the Mach implementation of UNIX 4.3BSD. Processes that run with the checkpointing policy use the Mach 3.0/UX pager to keep track of page modifications. After the process has been started, the pager keeps track of any modifications on page by page basis. At a checkpoint these modified pages are saved to stable storage.

At the time of recovery, a process is loaded as if it were started for the first time. Its address space is then allocated and copied from the checkpointed memory information.

The checkpointing mechanism also performs complex processing of the OS process control blocks, file descriptors, etc. to record the OS maintenance information of the process.

With this technique, monitoring and update of page modifications incurs an overhead on the failure-free operation of the application. The application processes are also suspended whilst taking a snapshot of the whole system state. This is reflected in the comparatively moderate performance of this checkpointing/rollback mechanism: for a typical workstation environment snapshots take less than *10* seconds and checkpoint commit duration up to *45* seconds (maximum measured checkpoint commit of FADI checkpointing mechanism was *5* seconds).

‡ Libckpt

Libckpt is a tool for transparent checkpointing on uniprocessors running UNIX [Plank 95]. It implements incremental and copy-on-write checkpointing, and introduces user-directed checkpointing facility that works under the assumption that the user/programmer has adequate knowledge of the functionality of the application.

In Libckpt, incremental checkpointing uses page protection hardware to identify the unchanged portion of the checkpoint, so that only the portion that was updated since the previous checkpoint is saved. This reduces the size of each checkpoint, and thus the overhead of checkpointing. However, incremental checkpointing can yield little or no reduction in the size of checkpoints if a large segment of the application program is modified between checkpoints.

Libckpt user-directed checkpointing is used to exclude static or temporary memory locations from the checkpoint, thus reducing its size. This is achieved by specifying points in the program where it is most advantageous for checkpointing to occur. Experimental results in libckpt have proven that user-directed checkpointing can yield large improvements in the performance of checkpointing. However, the utilisation of this method in a distributed environment - where the checkpointing of all the application processes has to be coordinated to guarantee consistent rollback- is very cumbersome. It requires extensive bookkeeping (checkpoints order in time, history of sent/received messages, etc.) to keep track of dependencies between processes that take their checkpoints asynchronously.

Libckpt saves the state of open files with the checkpoint but does not consider the handling information of the OS signals requested by the checkpointed process.

Although Libckpt offers more optimizations to the checkpointing mechanism, most of them are not suitable for distributed computing as described above.

‡ Bytestream Checkpointing

This is the checkpointing and process migration mechanism for *Condor* - a batch processing system for UNIX. Condor serves the purpose of executing long-running, computation-intensive jobs on workstations which would otherwise be idle. When Condor detects an activity on a workstation upon which it is currently running a job, it creates a checkpoint of the job before killing it. This checkpoint is written to disk and contains all the process state information necessary for Condor to restart the job exactly where it left off. Once a workstation becomes available, Condor transfers the checkpoint to the new workstation and restarts the job.

The Condor system has a different objective to FADI: It does not support inter-process communication and the checkpointing/rollback module is designed for process migration rather than for supporting fault-tolerance (e.g no integrated error-detection mechanism is available). However the idea of core checkpointing and job restoration is of relevance to FADI.

The basic idea of bytestream checkpointing is that a checkpointing process should write its state information (stack, data, OS and processor information) directly into a disk file, and a restarting process should read that information directly from a disk file byte by byte - hence the name "bytestream".

Unlike in core-dump checkpointing only the state information needed for process restoration has to be saved. This information is retrieved directly without the need for complex pre-processing of the OS platform-specific memory maps, process control blocks, or kernel-dump format. This increases the checkpointing efficiency, and enhances the recovery system portability. The Bytestream code is freely distributed (public shareware) with the Condor software package - university of Winscon-Madison ftp site: *ftp.cs.wisc.edu/condor,* and is supported by comprehensive documentation.

4.3 Bytestream Checkpointing: The Mechanism Functionality

The goal of checkpointing is to establish a recovery point in the execution of the program, and to save enough state to restore the program to this recovery point. For a UNIX process (Figure 4-2), this recovery point consists of the process address space which is generally divided into text, data, and stack areas, along with other state information about the process maintained by the kernel. This information includes the state of the process registers, any special handling requested for various signals, the status of open files and file descriptors. The following paragraphs briefly explain how every component of the process state is individually saved and restored.

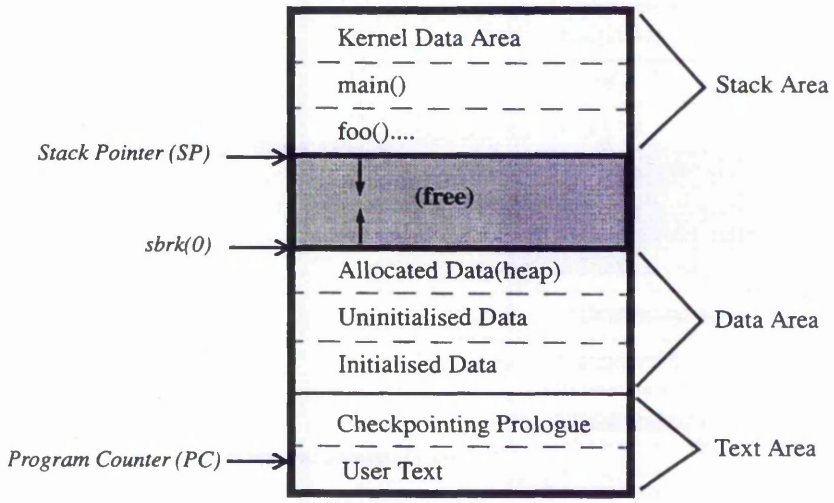


FIGURE 4-2 **Address Space of a UNIX Process**

‡ Text and Data Segments

Statically linked UNIX processes are created with their entire text loaded into virtual memory by the kernel, generally beginning at address *"0"*. Since the same executable is used for both the original invocation and when restarting a process, nothing has to be done to save and restore the text segment.

A process data generally begins at some page boundary above the text area, and is a contiguous area of memory. Once the process begins execution, the initialised data may be overwritten, therefore, information in the executable file cannot be used to save this area. Instead, in Condor Bytestream checkpointing, the entire data segment is written to the checkpoint file at checkpoint time, and is read back into the same address space at restart

time. To accomplish this, the start and end address of the data segment are needed. The starting address is constant location that is usually platform specific and can be found as a linker directive in the man pages. The ending address is effectively the top of the heap. This address can be obtained within UNIX via the sbrk() system call.

‡ Stack Segment

Preserving the stack requires saving and restoring the stack context (data structure containing the stack pointer amongst other stack related information), and the *actual data* that makes up the stack itself.

To save and restore the stack context, standard *"C"* functions setjmp() and longjmp() are used. setjmp() is called with a pointer to a system defined type called a JMP_BUF. setjmp() saves the current stack context into JMP_BUF and returns "0". If longjmp() is then called with a pointer to the JMP_BUF and some value other than "0", the stack context saved in JMP_BUF is restored and the execution returns back in the code to the point where the original setjmp() was made. This time the return value from setjmp() is the one specified in the longjmp() call, i.e something other than "0".

However, a limitation of setjmp()/longjmp() is that JMP_BUF does not contain the *actual data* contained in the stack space itself, only pointers into the stack space. To save the stack data, the stack's start and end points are required. The start of the stack is a well known static location which is defined as a constant on some platforms, or can be obtained from the man pages on others. The end of the stack, by definition, is pointed by the stack pointer. Thus, to determine the end of the stack, setjmp() is called to pull the stack pointer value out of the JMP_BUF.

Restoring the stack is more difficult. Unlike restoring the data area, the stack space is used (for local variables) while being replaced. Directly replacing a process stack space with the space saved in the checkpoint file will irreversibly corrupt the executable image. To avoid this, the stack pointer is moved into a safe buffer reserved in the process data area. Moving the stack pointer is accomplished with yet another call to setjmp(), manually manipulating the stack pointer in the JMP_BUF to point to our buffer in the data area, followed by a longjmp(). Then, the reserved space in the data area is used as a temporary stack to safely replace the process original stack area with the one previously saved in the checkpoint file.

So when moving the execution stack pointer back to its original place, we will effectively be using the restored stack.

‡ Open Files

Any files which are held open by a process at checkpoint time should be re-opened with the same "attributes" at restart time. The attributes of an open file include its file descriptor number, the mode in which it is opened (e.g. read, write, or read-write), the offset to which it is positioned, and whether or not it is a duplicate of another file descriptor. These attributes are recorded at the time the file descriptor is created via an open() or dup() system call. The offset at which each file descriptor is positioned is captured at checkpoint time by performing lseek() system call upon each file descriptor. All this information is kept in a table in the process address space. Upon restart, the checkpointing prologue walks through this table and re-opens and re-positions all of the files as they were at checkpoint time.

‡ Signals

In a UNIX process, signals may be blocked, ignored, take default action, or invoke a programmer defined signal handler. At checkpoint time a table is built, again in the process data segment, which records the handling status of each possible signal. The set of blocked signals is obtained from the sigpromask() system call, and the handling of each individual signal is obtained from the sigaction() system call. During restart, signal states are restored by stepping through this table. To handle pending signals (sent to a process while that process has the signal blocked), the checkpointing mechanism determines the set of pending signals with the sigispending() system call at checkpointing time call at checkpoint time. During restart, the checkpointing library first blocks each pending signal, then sends itself an instance of each pending signal. This ensures that if the user code later unblocks the signal, it will be delivered.

‡ CPU State

Saving the state of the process is potentially the most machine-dependent part of the checkpointing code. However, a characteristic of UNIX signalling mechanism is that the signal handler saves and restores all the relevant CPU states. In other words the signal handler can

interrupt the execution code, but when it returns, the interrupted code should continue without error. Hence, with the bytestream checkpointing mechanism a checkpoint is always invoked by means of sending the process a signal.

4.3.1 Limitations of Condor's Bytestream Checkpointing

The most important shortcoming of bytestream checkpointing is its inability to checkpoint and migrate one or more of a set of communicating processes. Processes which communicate with each other via signals, sockets, pipes, or those that are dynamically created via fork() or exec() are not dealt with in Condor's bytestream checkpointing. Another limitation is that the mechanism does not maintain the integrity of files if they are updated between checkpoints. At checkpoint time the offset location of the file is recorded and the file is rewound to that position upon restart, but modifications made to the file structure after taking the checkpoint are not undone.

4.4 Checkpointing and Rollback in FADI

The FADI checkpointing mechanism is intended to safe-guard the distributed application processes against faults that might occur in the underlying hardware. If all the distributed application processes are consistently checkpointed then, when the error detection mechanism detects a fault, the affected application processes can be rolled-back to the most recent global checkpoint and then restarted on the same host if the fault is temporary, or migrated to another host if the hardware fault is permanent.

Because FADI is intended to operate with NO additional hardware to support fault-tolerance there is a need for a degree of homogeneity of the distributed system, i.e at least one spare node per CPU architecture where the application processes can be rolled-back in case of permanent hardware failures.

The checkpointing/rollback mechanism should have access to a centralised file system (NFS type) that is identical to all the distributed system hosts. This is important for the relocatability of the application processes: if the process is recovered on another hardware, their checkpoints should still be accessible. This file system will form the stable storage for saving the checkpoints, therefore, it is necessary for it to survive the crash of the individual nodes of the distributed system.

FADI integrates Condor byte stream checkpointing model into the fault-tolerant environment checkpointing/rollback mechanism to aid in saving the execution state of the application process [Taha 97].

4.4.1 The Interface to the Checkpointing Mechanism

Our goal is to provide a fault-tolerant environment that is transparent to the user/programmer. However, unlike truly fault-tolerant operating systems such as KeyKOS and Sprite [Douglis 91], where process models are carefully defined and implemented to accommodate checkpointing and migration, creating a user-transparent checkpointing mechanism in a general purpose operating system like UNIX is possible only with some automated pre-processing of application source code.

The main pre-processing change made to the user code is the modification of the name of the initial procedure in user program from main() to user_main(). This enables FADI checkpointing prologue to gain control of the program as it starts, check the command line to verify if this is a normal run of the user program or a request for a rollback as a result of fault. The parameters passed to the checkpointing prologue are:

" <start/roll> <path> <checkpoint_interval> ", where:

start/roll - specifies if this the required execution is a normal run of the user program, or a request for a rollback to the last checkpoint to recover from a fault.

path - specifies the directory in which the checkpoint files are created. The default is the current directory.

checkpoint_interval - defines the interval between checkpoints. As argued in section 4.1.2, checkpointing the application processes at pre-determined intervals is particularly advantageous for distributed applications. It simplifies finding consistent recovery lines of the distributed application tasks as all checkpoints can be taken at the same checkpointing interval. FADI checkpointing prologue calls setitimer() that sets the local system interval timer to signal SIGALRM at the elapse of every *checkpoint_interval.* The signal handler for SIGALRM interrupts the execution of the target user code and initiates the checkpointing process.

4.4.2 Saving the User-Files State

Condor bytestream checkpointing code and most existing checkpointing algorithms support saving the state of user-files only to a limited extent. Condor confines user-files checkpointing/rollback to recording the file's attributes and offsets at checkpoint time. When rollback is initiated, the file is re-opened with the same attributes and its pointer is moved to the offset recorded at checkpoint time. Some algorithms go further and record the file size at checkpoint time, upon rollback, the user-file is truncated to the recorded size. This simple approach to save the file state has two shortcomings:

1) This scheme deals only with files that were active at the time of taking the checkpoint. Figure 4-3 gives an example for which the above solution will result in corrupting the data structure of the file. In Figure 4-3, the size of *int.dat* is not recorded in the checkpoint because it is not active at checkpoint(). As a result, *int.dat* is not truncated when a rollback occurs, and the character "4" will be incorrectly appended twice.

```
/* user file "int.dat" contains three integers: 1,2 and 3*/
checkpoint();
fp = fopen("int.dat", "a"); /* for append */
fprintf(fp, "%d", 4);
fcolse(fp);
/* failure occurs, roll back */
unlink("int.dat"); /* remove the file */
```

FIGURE 4-3 **Example of Incorrect Rollback of Files Opened for Append**

2) Modifications made to the contents of the file are not undone upon rollback. In Figure 4-4, the integer value held in the file is updated after checkpoint(). As a result, upon rollback the file will contain an incorrect initial integer value of "9" instead of "3".

```
/* user file "int.dat" holds initial integer value = "3" */
checkpoint();
fp = fopen("int.dat", "+r"); /* for update */
pos = ftell(fp) /* get current offset of "int.dat" */
fread(&d, sizeof(int), fp); /* d is = 3 */
d = d**2;
fseek(fp, pos, SEEK_SET); /* rewind file to prepare for update */
fwrite(&d, sizeof(int), 1, fp);
fcolse(fp);
/* failure occurs, roll back */
```

FIGURE 4-4 **Example of Incorrect Rollback of Files Opened for Update**

These incorrect rollbacks can be avoided only by direct intervention of the programmer, otherwise they can often lead to unpredictably corrupt files. Requiring the users to understand and deal with such limitations on file rollback contradicts the requirements of transparency and ease of use.

FADI maintains the integrity of user-files across the checkpointing/rollback process by tracking appends and updates to user-files. UNIX system calls and *"C"* library functions that write to user-files are augmented by in-house functions that first perform bookkeeping operations before calling the requested system call or *"C"* library function. The developed algorithm to save the user-files state is depicted in Figure 4-5.

upon opening a file
- record file name and attributes in open-files table;
- if the file was opened for append, then:
  if file name is not in appended-files table, then add it to the table
  and record its size into stable storage;
- call the OS open routine with the given attributes.

upon writing to a file open for update
- if a shadow copy of the file does not already exist, then create one;
  otherwise, if file name is not in updated-files table, add it to the table;
- call the requested write routine (write, fwrite, printf, etc.).

upon writing to a file open for append
- if file name is not in appended-files table, add it to the table;
- call the requested write routine (write, fwrite, printf, etc.).

upon closing a file
- remove file name from open-files table;
- call OS close routine.

upon checkpointing
- make shadow copy of all files in updated-files table, then clear the table;
- update file size of all files in appended-files table, then clear the table.

upon rollback
- replace files opened for update with their shadow copies;
- truncate files opened for append to their recorded sizes;
- re-open all files active at last checkpoint with the same attributes.

FIGURE 4-5 **FADI Algorithm for Saving and Restoring the State of User-Files**

4.4.3 Looking Ahead: Checkpointing and Message Passing

PVM is adopted as the message passing interface for FADI. It provides FADI with the tools to facilitate the communication between FADI control and monitoring tasks on one hand and between the distributed application processes on the other.

In order to enable FADI central error detection and task recovery tasks to interact with the checkpointed code, and allow the distributed application programmer to use PVM powerful communication facilities, the following measures had to be taken:

- at the start of the program execution the checkpointing prologue calls pvm_mytid() to enrol in PVM, allowing PVM to open communication channels between the current processes and others running under the PVM daemon on local and remote hosts;
- Investigation into PVM message passing interface has shown that taking a checkpoint when PVM routines are packing or unpacking data messages can corrupt the message content. The checkpointing mechanism can not checkpoint and recover whilst PVM is performing complex manipulation of TCP/IP sockets operations and data format conversions between different platforms. Therefore, in order to maintain the integrity of transformed data, checkpointing has to be temporarily blocked until the packing/unpacking process is completed. This is achieved by augmenting PVM routines responsible for packing/unpacking interprocess messages.

  When the user requests packing or unpacking of a message a special routine is called that firstly executes sigblock() to mask out the checkpointing signal SIGALRM, then the PVM routine responsible for packing/unpacking the message is called. After the completion of the packing/unpacking process, the masking routine executes sigsetmask() to restore the old signal mask and release the blocked checkpointing signal.
- upon the termination of the user program, the checkpointing prologue calls pvm_exit() to exit PVM and sends a message to the FADI task monitor process, declaring the successful termination of the user task.

4.4.4 Reducing the Checkpointing Overhead

For the targeted class of long-running scientific/engineering applications, reducing the failure free overhead is of essential importance. Checkpointing contributes to this overhead by suspending the execution of the application program while the checkpoint is taken and written out to disk. This overhead builds up with the increase in the checkpointed image size - larger process image incurs more "slow" I/O operations to disk.

In an attempt to minimise the checkpointing overhead, this research has introduced a novel technique of *non-blocking checkpointing*. A copy is made of the program's data space and use an asynchronous thread of control that performs the checkpointing routines, i.e reads the process state and records it to disk, while the user process continues the execution of the program code.

The UNIX fork() system call provides the mechanism needed to implement non-blocking checkpointing. The checkpointing prologue calls fork() that creates a child process (the checkpointing thread) with a fixed snapshot of the parent process (the checkpointed user-program) and a separate thread of control.

4.5 A Complete Checkpointing and Rollback Cycle

Figure 4-6 gives a high-level picture of FADI checkpointing and rollback mechanism. When the user-task is called with a set of parameters, the checkpointing prologue takes over the program execution. If the parameters indicate a normal run of the user program, FADI sets the local system timer to invoke the checkpointing of the user task at regular intervals specified by the user in the parameters, it then calls the user-main() routine. At the elapse of every checkpointing interval, the system timer signals SIGALRM to invoke the checkpointing routine. At this stage, a copy (thread) of the user process is spawned to perform the checkpointing routines while the user process continues the execution of the application code.

Checkpointing starts by recording information about the stack context, signal state, and open files into data structures built in the process data area. Then it writes the data and stack segments into the checkpoint file. Next it creates shadow copies of files that were open for update during the last checkpointing interval, and makes a record of the sizes of

files that were open for append. Now that the checkpoint is taken, and the forked thread is terminated with a user signal.

If a hardware fault was detected and a rollback of the user program is requested, then the checkpointing prologue calls the restore routine. It first replaces user files that were opened for update with their shadow copies and truncates files that were opened for append to the size recorded at the most recent checkpoint. These operations are performed only if the contents of the file were modified since the last checkpoint. Then the data segment is replaced with the one stored in the checkpoint file. Now the restore routine has the list of open files, signal handlers, etc. in its own data space, and restores those parts of the program state. Next it switches the stack to a temporary location that was current at the time of taking the checkpoint and returns to the user code at the same instruction that was interrupted when the last checkpoint was invoked. Finally, because checkpointing was performed by a signal handler, UNIX OS restores all CPU registers to their state before checkpointing took place.

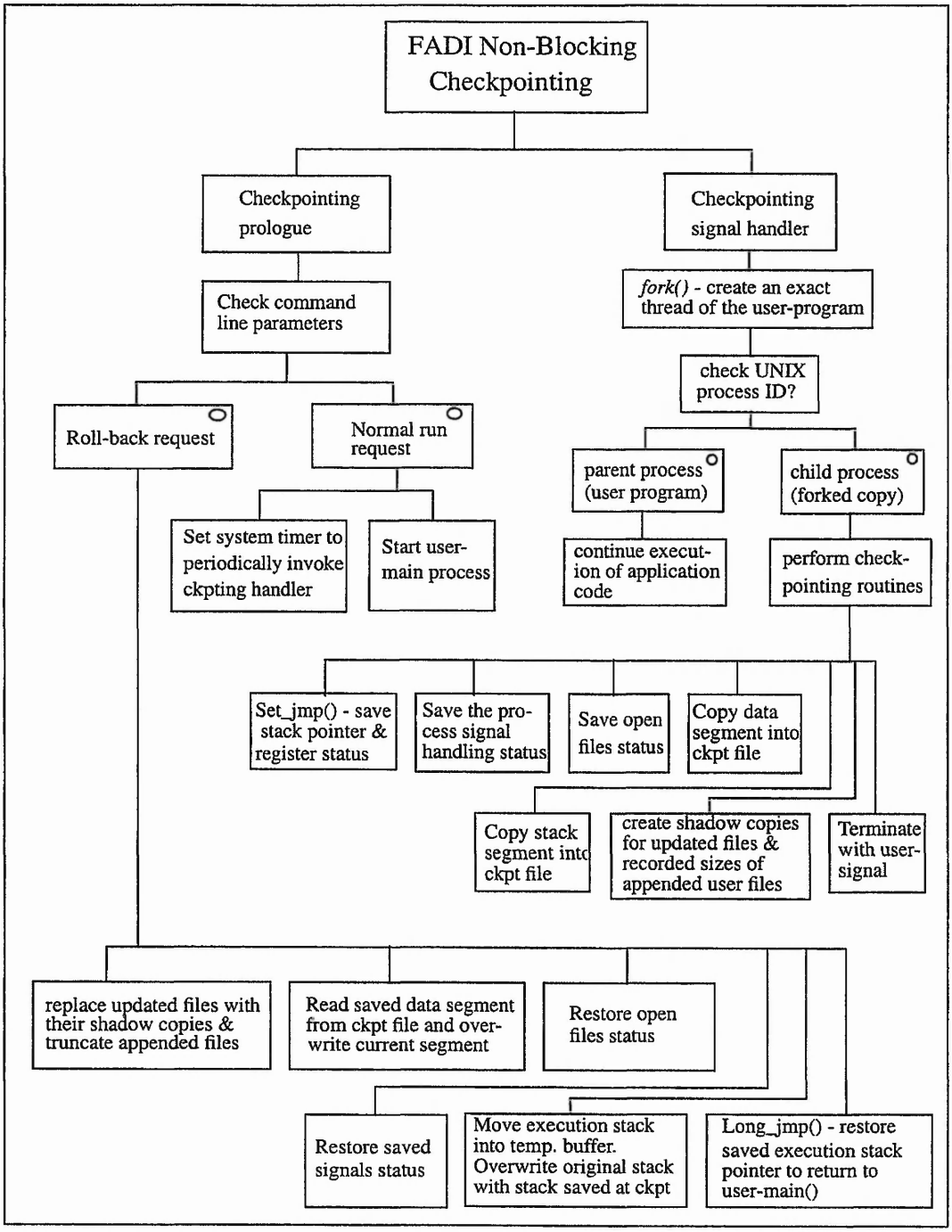


FIGURE 4-6 **Checkpointing & Rollback in FADI**

4.6 Integration of Checkpointing/Rollback Recovery into FADI

This section explains how the core checkpointing and rollback protocol -explained in section 4.4- is incorporated into the fault-tolerant environment to support the reliable execution of the application processes. The overall structural design of FADI is illustrated in Figure 4-7. A data dictionary interpreting the flow of data between FADI tasks is presented in Table 4-1.

At the start of execution, the process allocation module automatically identifies the configuration of the network where FADI is running, and passes the initial *active hosts table* to the user and the rest of FADI processes. The user can select to run the application processes on a specific host, on any of a group of hosts with a similar hardware architecture or on any default host. Upon receipt of the application tasks specifications from the user-interface, the *PROCESS ALLOCATION* process uses pvm_spawn() to automatically distribute the user tasks on the specified hosts. PVM performs a load balancing routine to identify the least loaded host if there is a choice between a number of them. If all the application processes are not spawned, the distributed system is declared to have insufficient resources to run the application and FADI is halted, otherwise, the *spawned tasks specifications* is broadcasted to the rest of FADI processes.

When the *CHECKPOINTING COORDINATOR* receives a list of spawned tasks on the distributed system hosts, it triggers their checkpointing at the elapse of every checkpointing interval. The checkpointing protocol computes and stores the checkpoints in the background using *non-blocking checkpointing* while the main application code continues normal execution.

Upon host crash the *MONITOR HOST STATE* process informs the *RECOVER FAILED TASKS* process of the *crashed host_ID* so that it won't try to restart (rollback) failed tasks on it. The *RECOVER FAILED TASKS* process is also promptly notified about *recovered hosts (failed host_ID)* to avoid migrating the failed task into another host if the crash was caused by a network delay and the host was reinstated into the active hosts pool.

The *failed host_ID* is also sent to the *MONITOR USER TASKS* process to determine the IDs of the user-tasks that were running on the faulty host before the crash. These IDs are then sent to the *RECOVER FAILED TASKS* process so that it can initiate their recovery.

The *MONITOR USER TASKS* process detects user-tasks that have exited prematurely due to a transient hardware failure and similarly sends the *failed task id* to the *RECOVER FAILED TASKS* process. The *failed task IDs* are also passed to the *CHECKPOINTING COORDINATOR* to suspend their checkpointing.

The *RECOVER FAILED TASKS* process retrieves the most recent saved checkpoint of the failed task from stable storage. If the failure was caused by a transient hardware failure, then it attempts to restart the failed task from the checkpoint file on the same host where it was running before the occurrence of the fault. Otherwise it migrates the checkpoint to an active host with similar OS architecture and attempts to restart (rollback) the failed process there.

Task_IDs of successfully recovered user tasks are broadcasted by the *RECOVER FAILED TASKS* process so that the *MONITOR USER TASKS* process can resume their monitoring and the *CHECKPOINTING COORDINATOR* can resume their checkpointing. It's crucial that the *MONITOR USER TASKS* gets the ID's of the recovered tasks because they have been restarted as new executables with different task IDs.

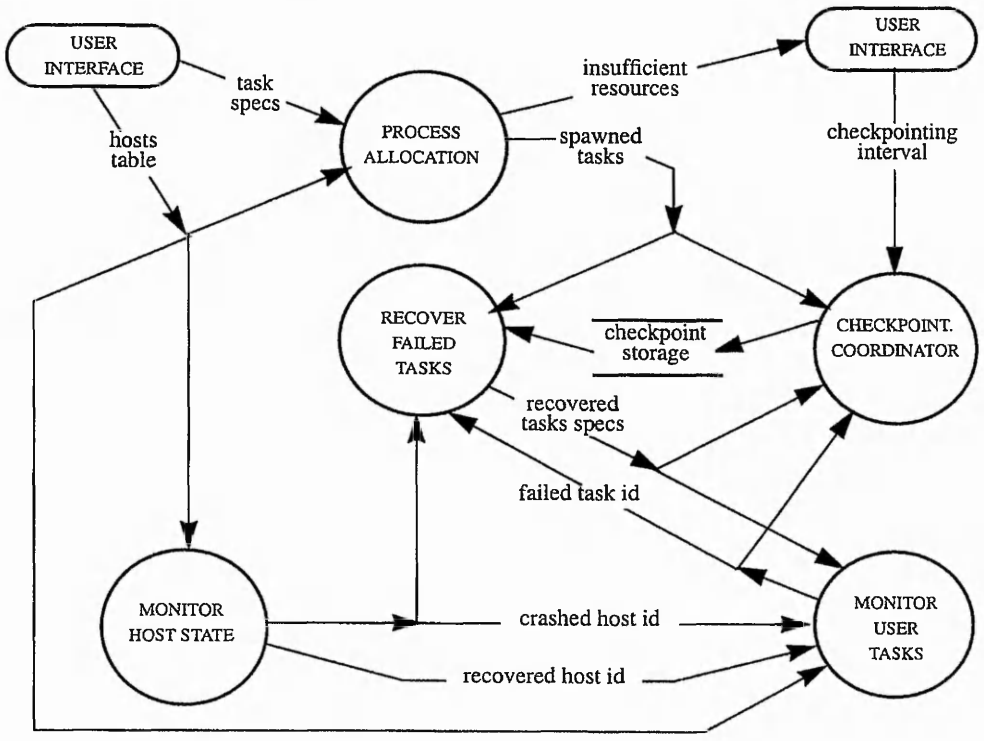


FIGURE 4-7 **FADI General Structural Design**

TABLE 4-1 **Data Dictionary**

ITEM	USAGE
crashed host id	\ hosts that failed to fulfil the acknowledgment request at the current moment in time \
failed task id	\ id of tasks that exited with erroneous status \
hosts table	\ list of available hosts on the net (host id + arch) \
insufficient resources	\ inadequate processing power to run all user tasks \
recovered host id	\ id of hosts that recovered either manually or from network delays \
recovered tasks specs	\ specifications of processes that were rolled back from the last checkpoint \
spawned tasks specs	\ task-id + task-host id + task checkpointing interval \
checkpointing interval	\ time interval between two consecutive checkpoints \
task specifications	\ task name + task-host name + checkpointing interval \

It is worth mentioning that the distributed nature (concurrent execution) of FADI fault-management processes and the fact that they are constructed in an event-processing fashion - that is message driven, supports acquiring more accurate approximations of the error detection latency and aids in the prompt update of the active hosts table to prevent attempts of roll-back on a faulty host and to avoid migrating the failed task into another host if the crash was caused by a network delay and the host was reinstated into the active hosts pool.

4.7 The Checkpointing Mechanism Performance

4.7.1 Overview

All fault-tolerance procedures will inevitably result in a degradation in the system performance. As a general guideline, it has been suggested in the literature that the overhead incurred by making the application tolerant to faults should not exceed *"10%"* of the application run-time [Plank 94]. The most significant overhead introduced by fault-tolerance measures is the overhead of taking and storing the checkpoint. The *non-blocking* checkpointing technique reduces this overhead by interleaving the execution of the original task and the checkpointing task. Because of this concurrent execution our system affords a degree of immunity to the variation of efficiency of taking and storing the checkpoint, as long as the current checkpointing process terminates before the next one is initiated.

The checkpoint size can increase the checkpointing overhead and occupy valuable disk space. A technique of *Incremental checkpointing* [Plank 95] has been suggested for reducing the checkpoint size and consequently the checkpointing overhead. The idea is that when a checkpoint is taken, only the portion of the checkpoint that has changed since the previous checkpoint needs to be saved. The unchanged portion can be restored from previous checkpoints.

However, experimental results by *J. S. Plank and K. Li* in *libckpt* [Plank 95] have shown that incremental checkpointing does not necessarily suit all types of applications. If large amount of the program's address space is modified between checkpoints, this can lead to little or no reduction in the checkpoint size. This conclusion is corroborated by other researchers (e.g *Pierre Sens* [Sens 93]). Consequently, the incremental checkpointing was not adopted for FADI.

Moreover, considering that it is necessary that the inter-checkpoint interval is larger than the expected error latency, an error occurring in the "n^{th}" checkpoint interval will be at most detected in the "$n + 1^{st}$" interval (Figure 4-8). Hence, there is no need to save more than two consecutive checkpoints at any moment in time to rollback safely a particular process. This devalues the gain from incremental checkpointing since there is only one possible increment on the checkpoint size.

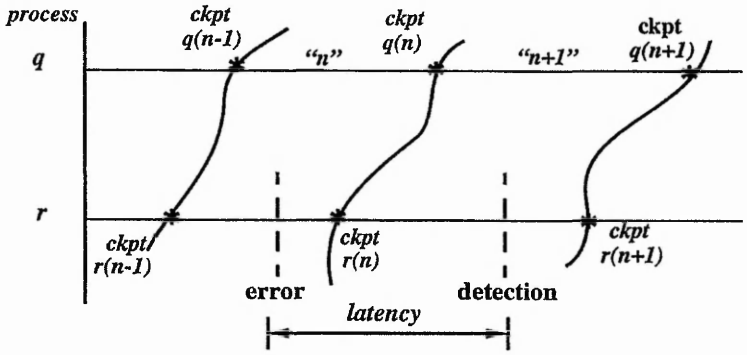


FIGURE 4-8 **Error Detection Latency**

4.7.2 Experimental Results

The checkpointing and roll-back mechanism was tested by disrupting the normal execution of the user-application process using all possible methods: powering down the host machine, disconnecting the host machine from the ethernet, killing the process explicitly from the command line "kill -9" and killing the process from the PVM console using pvm_kill(). Every time the user-application process was successfully rolled back to the most recent checkpoint and continued execution on the same host or on a host with similar OS architecture.

In order to evaluate the computational overhead of the checkpointing protocol, two numerically intensive applications have been benchmarked.

The first application involved the multiplication of two 256x256 matrices which are read from disk file via NFS, and the product matrix is written to an output file. The second application uses the principles of *Simulated Annealing* [Lee 96] to work-out an optimal decomposition of network nodes for distributed processing. The initial network node-configuration is read from disk and the optimization results are printed to the screen. The executables were built using GNU C++ compiler "*g++*" and were run on a SPARC*station IPC* running SunOS 4.1.3.

The average results for checkpointing the *matrix multiplication* and *simulated annealing* applications for a 20sec checkpointing interval are presented in Table 4-2, and the full experiment results are illustrated in figures 4-9 to 4-12.

TABLE 4-2 **General Checkpointing Results**

User Application	Normal run-time (sec)	Sequential Ckpting (sec)	non-blocking Ckpting (sec)	Ckpting interval (sec)
Matrix Multiplication	317.93	326.7	322.04	20
Simulated Annealing	804.53	808.58	806.00	20

As expected the checkpointing overhead for *matrix multiplication* is higher than that for *simulated annealing* application, the reason is the large image size caused by maintaining three 256x256 static variables in the processes address space.

Figures 4-9 and 4-10 show the effect of varying the checkpointing interval from 10 to 80sec on the execution time of the matrix multiplication and simulated annealing applications respectively. It is clear that the execution time for checkpointed applications (both sequential and non-blocking) increases as the checkpointing interval is reduced.

This ascertains the importance of balance between increasing the required level of reliability (taking checkpoints more frequently, i.e shorter checkpointing intervals) and the subsequent degradation in performance. The task of our software is to determine the minimum possible checkpointing interval, then it is up to the user to fine-tune the reliability-performance balance.

The percentage by which checkpointing is slowing the normal run-time of the applications is illustrated in Figures 4-11 and 4-12. We can notice that non-blocking checkpointing significantly reduces the application checkpointing overhead, specifically upon short inter-checkpointing intervals (up to *30%* reduction in overhead over sequential checkpointing).

The results in Figures 4-11 and 4-12 also show that FADI checkpointing mechanism performance is well below the recommended limit of *"10%"* overhead even for sequential checkpointing. The maximum recorded overhead with the minimal possible checkpointing interval (10 seconds) for Matrix Multiplication using sequential checkpointing (worst possible scenario) was *0.8%*.

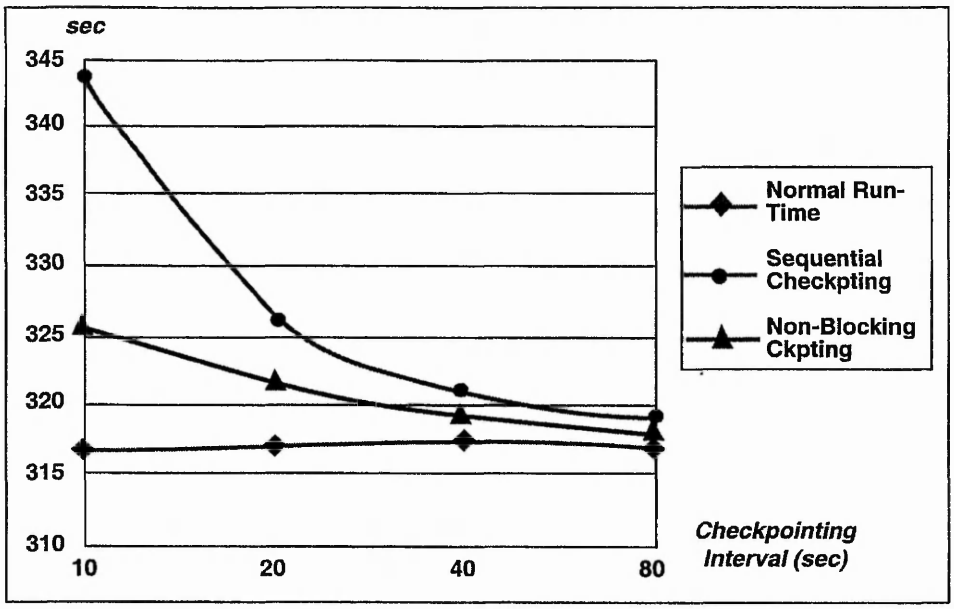


FIGURE 4-9 **Application run-time (*matrix multiplication*)**

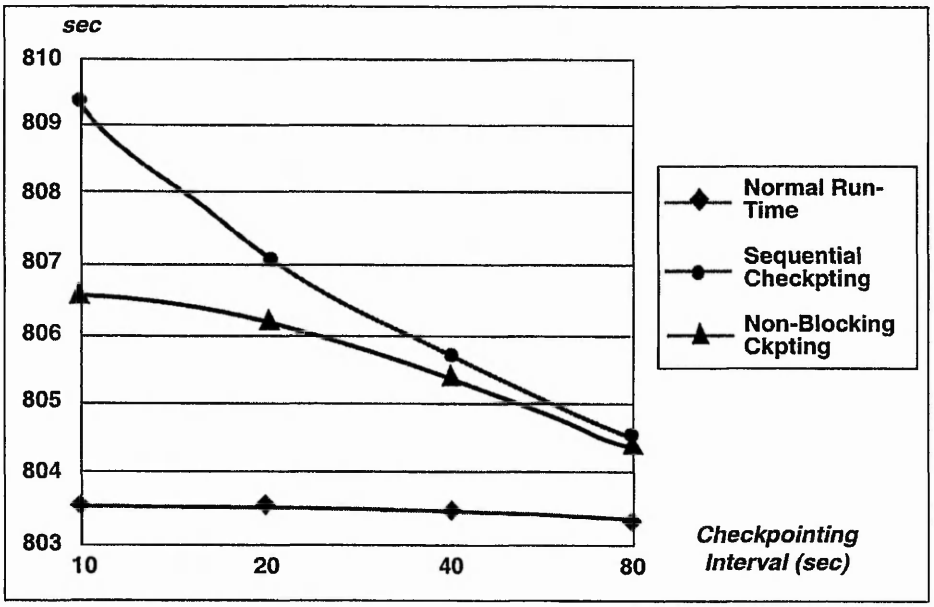


FIGURE 4-10 **Application run-time (*simulated annealing*)**

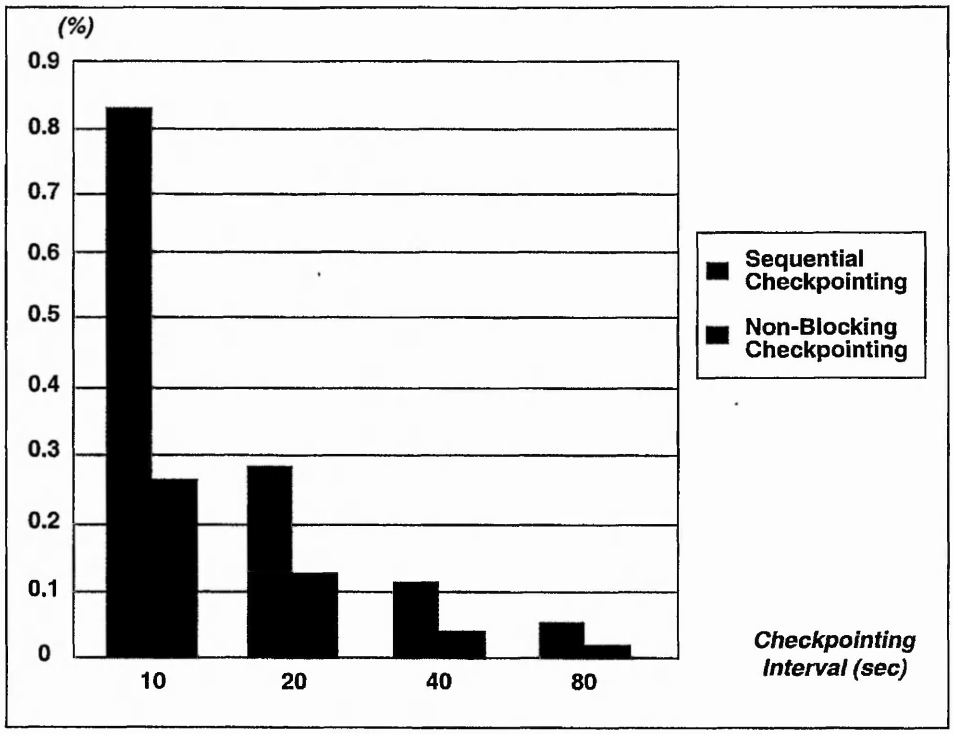


FIGURE 4-11 **Checkpointing Overhead of the Matrix Multiplication Program**

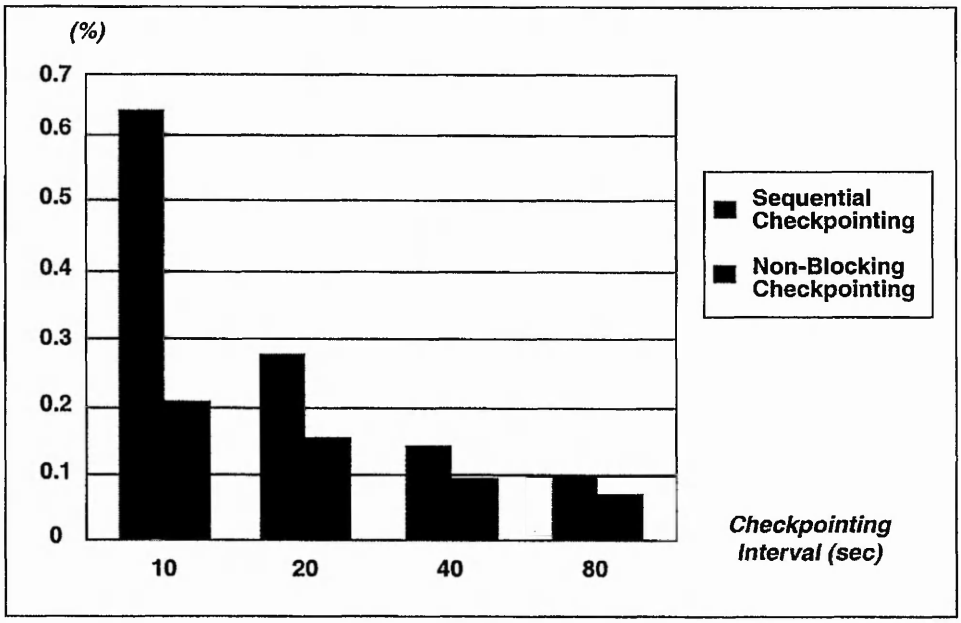


FIGURE 4-12 **Checkpointing Overhead of the Simulated Annealing Program**

4.8 Constraints of FADI Checkpointing Protocol

Due to the incompatibility of the UNIX systems (discrepancies in the a.out format, management of the process's address space by the MMU, etc.), rolling-back (restarting) the process from the previous checkpoint on another computer system is limited to hosts with similar Operating System architecture to the failed one.

Another limitation of the checkpointing mechanism is that user-application programmes have to be statically linked with FADI checkpointing library. Despite the fact that minor modifications have to made to be the user code, the user program must still be called as a function from the checkpointing prologue to allow for initialisation and control of the checkpointing and restart processes of the user-program. As a result, FADI checkpointing library is limited to users that have access to the source code but does not work for users of third party software.

4.9 Conclusions

FADI utilises Condor bytestream checkpointing model to save the execution state of the application processes. To guarantee the integrity of exchanged inter-process messages, a technique was developed to block checkpointing initiation during the packing/unpacking of message datagrams. Another original contribution to the checkpointing/rollback technique is the development of mechanisms for performing the rollback of user-files. It uses a combination of copy-shadowing and file size bookkeeping to undo modifications to user-files upon rollbacks. This module guarantees the reliable recovery of user-files that were open in *read-only* as well as in *append* and *update* modes.

Performance measurements showed that the new *non-blocking checkpointing* technique significantly reduces the checkpointing overhead. With this method, an exact copy(thread) of the checkpointed program is forked, this thread performs all the checkpointing routines without suspending the execution of the application code. Experimental results also demonstrated that the performance of the developed checkpointing protocol compares favourably with results published of similar work in [Sens 93] and [Plank 94].

The checkpointing protocol was combined with the error detection mechanism and an automatic process allocation module to provide an integrated environment for transparent relia-

ble execution of distributed application programs. The integrated modules execute concurrently on a fault-tolerant host and cooperate in a message driven system to enhance FADI's response to events occurring in the distributed system.

CHAPTER 5 Reliable Distributed Computing for Message Passing Systems

Chapter 5 generalises the FADI checkpointing and rollback technique to interactive (message passing) application processes. It reviews most of the conventional checkpointing and rollback methods developed by other researchers, evaluates their advantages and shortcomings, then it introduces a novel algorithm for checkpointing distributed interactive applications that is based on a coordinated checkpointing and selective message logging technology.

5.1 Introduction

Chapter 4 discussed the details of the design and implementation of FADI checkpointing/ rollback mechanism in the context of stand-alone applications and the experimental results have shown that the developed mechanism is very robust. Now, the developed checkpointing protocol has to be extended to cover the possible inter-process communications taking place between the distributed application processes. The next section discusses the problems incurred by checkpointing/rollback of interactive processes.

5.1.1 Live-lock Problem

If the rollback of the processes is not synchronized, *live-lock* can occur [Koo 87]. This means that one single failure produces an infinite number of rollbacks, which prevents the system from making progress [Deconinc 93]. This is illustrated in Figure 5-1: processes *'a'* and *'b'* send messages *m1* and *n1* respectively. The indices *'1'* denote the incarnation number (the current run/re-run) of the process. If a failure occurs in process *'a'* before message *n1* is received, then it rolls back to its last checkpoint and resumes execution, i.e sends *m2* and receives *n1* that was under way. To undo the receiving of message *m1* (of which process *'a'* has no notice), process *'b'* rolls back and resumes its operation, i.e it sends *n2*. Again the state is not consistent any more because process *'a'* has no notion of the receipt of *n1*, while process *'b'* has no notion of sending that message, so process *'a'*

should rollback a second time. Then process *'b'* rolls back to restore consistency to undo the receiving of *m2* which in turn forces *'a'* to roll back, etc. So, due to a single failure, the system will rollback endlessly to its previous checkpoint.

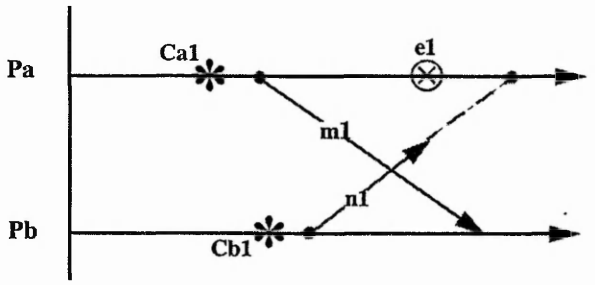


FIGURE 5-1 **Live-Lock**

This live-lock problem can be avoided when messages with other incarnation numbers are discarded [Silva 92] or if a two-phase commit-protocol assures that processes rollback at the same time [Koo 87].

5.1.2 The Domino Effect

Figure 5-2 illustrates a process execution sketch that can lead to the domino effect.

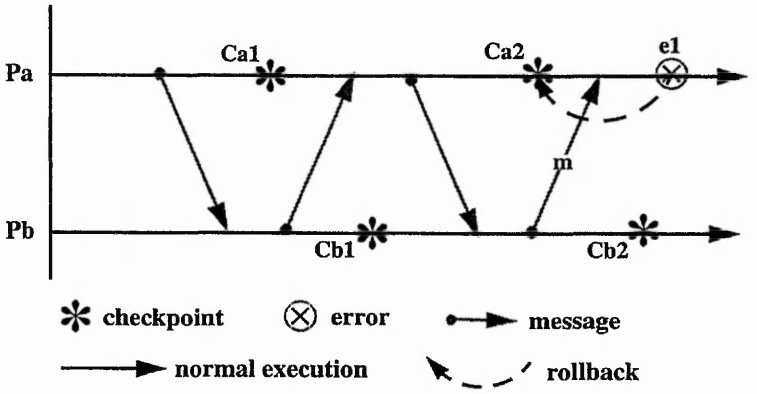


FIGURE 5-2 **Rolling-Back Interactive Processes**

Process *'a'* and *'b'* are part of the same concurrent application whose checkpoints are taken user-transparently (checkpoint #1: *Ca1, Cb1*; checkpoint #2: *Ca2, Cb2*). Looking at the

message *m* sent from *'b'* to *'a'*, then if process *'a'* fails at *e1*, it can be rolled-back to the last checkpoint *Ca2* and restart the execution from there.

However, by rolling back process *'a'* to *Ca2* the receipt of message *m* will be undone, and *'a'* will be waiting for a message from *'b'* that will never arrive, so *'b'* should also rollback to *Cb1*. Similarly, *'b'* would expect a message from *'a'*, so *'a'* should rollback, etc. Finally, both *'a'* and *'b'* roll back to their initial state. The so called *domino-effect* [Lee 90] can be avoided by using the following techniques:

- The specification of a consistent recovery line (Semi-automatic checkpointing);
- Storing inter-process communication into stable storage (message logging);
- Freezing of all the application processes until a snapshot of the entire system is taken (coordinated or consistent checkpointing);
- A combination of the two previous techniques (hybrid methods).

The above techniques differ with respect to their computation/communication overhead. Their characteristic features and relative strength will be evaluated later in this chapter.

5.1.3 The Checkpointing and Rollback Overhead

The checkpointing overhead can be quantified in terms of time, storage or communication and it can manifest itself during the *normal failure-free* operation or during the *rollback recovery*.

5.1.3.1 Time Overhead

The most important include:

- *Computation time lost during checkpointing:* while the checkpoint is taken, a fraction (depending on the checkpointing frequency) of computation time is lost for processing and storing the program execution state. Some of the operations performed during checkpointing are: retrieval of the program signal handling status, CPU registers, open files and communication sockets state, records of interprocess messages in transit, etc., and writing this information together with the contents of the data and stack segments into stable storage (the slowest component of the checkpointing cycle);

- *Computation time lost during rollback:* a set of consistent checkpoints has to be restored, and all computations performed by the application since the last global checkpoint should be re-executed. There is a trade-off to be made between time lost during checkpointing and during rollback; the checkpointing frequency depends on the desired fault-tolerance characteristics. For FADI application domain of long-running scientific/ engineering applications, emphasis is put on minimising the performance reduction (overhead) in the failure-free case, when errors are considered as an exception rather than a rule;
- *Kernel overhead:* a part of the CPU time is used for checkpoint/rollback related topics (sending/delivering bookkeeping, control messages, message logging, checkpointing timers monitoring, etc.). This is a persistent overhead over the life-time of the program (not only during checkpointing time) and therefore it has to be minimal.
- *Time overhead when other processes take their checkpoint or rollback:* for the *blocking* algorithms, other processes participating in a checkpoint or a rollback session should suspend normal operation until checkpointing or rollback of all partners is complete [Bauch 92].

5.1.3.2 Storage Overhead

Storage overhead is in local volatile memory (in RAM) as well as secondary storage (on disc). After a fault has occurred, access to the stored checkpoints must be guaranteed. This overhead includes:

- *local memory usage:* 1) For storing the checkpoints. If the checkpoint is stored in the volatile memory, the availability of this memory for active processes is significantly reduced. 2) A part of local memory is needed for storage of the checkpoint and the recovery management processes. Typically message-logs, databases and other bookkeeping information are stored in the node.
- *disc usage:* 1) For storing the checkpoints. At least one complete (permanent) checkpoint should be stored. Some schemes require also a place for the tentative checkpoint(s) on stable storage. 2) A part of the bookkeeping info should be kept in stable storage; this includes data about the stored checkpoints, the running processes and their interactions, message logs, etc.

5.1.3.3 Communication Overhead

Communication overhead has two components, one due to the communication between the distributed computing nodes and the other due to I/O operations associated with storage and retrieval of checkpoints, i.e the reduction in the disc bandwidth, useful for the application.

- *Load on data-network by increased number or size of the messages:* extra control-information must be sent, possibly attached to normal communication. This includes information associated with the checkpointing and rollback (send/receive sequence numbers [Storm 87], incarnation numbers [Elnozahy 92], or crash counters [Silva 92], etc.) and messages to obtain special communication protocols (acknowledgments, etc.). Especially for communication-intensive applications, this can produce a large overhead.
- *Load on I/O-network caused by exchanging checkpoints on disks:* in the worst case the entire system state should be checkpointed and restored at once. Some algorithms use hardware support, e.g segmentation, to store only those parts of the process state that changed since the last checkpoint *(incremental checkpointing).*

5.2 Conventional Checkpointing and Rollback Methods

5.2.1 Semi-Automatic Techniques

The application-programmer is directly involved in the backward error recovery, either in organizing the application process in a chain of recovery block constructs [Kim 83], or by calling the checkpointing routine, specifying to which recovery line the checkpoint should belong, and possibly its application-dependent contents.

This method's failure-free overhead is minimal because only when the checkpoint is taken (on demand of the programmer) the system suspends its operation for saving the checkpoint and doing some bookkeeping. In-between the checkpointing intervals there is no extra overhead. The user can also reduce the size of the checkpoint by specifying the contents of the checkpoint and indicating the memory-ranges and system parameters to be included (memory exclusion by user-directed checkpointing in Libckpt [Plank 95]).

However, Despite of its advantages, this method is only rarely used because of its reliance on an in-depth knowledge of the application source code by the user.

5.2.2 Message Logging Techniques

Different processes are logged independently of each other (one process at a time). All inter-process messages are recorded in a message log. After a failure is detected, the previous checkpoint is restored and the logged messages are replayed (in the same order) to bring the failed processes back to a consistent system state.

In *pessimistic* schemes the processes are suspended after each message until it is logged [Borg 83]. *Optimistic* schemes continue their operation during the log of messages (asynchronous), but need extra bookkeeping (e.g dependency tracking) to know which computation depends on which message and which messages have been logged [Sista 89].

Both schemes require only one process to take the checkpoint at a time, this lowers the load on the communication bandwidth especially if all the nodes share a single stable storage system. Pessimistic message logging draws a considerable failure-free overhead because normal operation is suspended until each message unit is logged.

Although optimistic logging avoids blocking processes by logging messages asynchronously, the overhead of logging every message is still significant. Optimistic logging also requires applications to be deterministic and adds significant time overhead associated with dependency tracking and communication overhead, especially if there is extensive inter-process interaction.

5.2.3 Coordinated (Consistent) Checkpointing Techniques

In these techniques the domino effect is avoided by checkpointing all (interacting) processes together; hence, these checkpoints form a consistent recovery line.

With *Global Checkpointing* the whole application is frozen to be able to take a snapshot of the entire system state. In [Bauch 92] a *two-phase commit-protocol,* where by tentative checkpoints are taken while permanent (previous) checkpoints are kept in memory. If all the new checkpoints of the application processes are successfully saved, a *commit* message is broadcasted. After acknowledging the commitment the application processes delete the old checkpoints and resume normal computation. This is a rather expensive method because of the time overhead caused by suspending all the application processes until the complete checkpoint is taken.

Process level Checkpointing was introduced to improve the performance of *global checkpointing* by allowing only interacting processes to checkpoint together rather than the application as a whole. These interacting processes are the set of processes that have been communicating since last checkpoint [Koo 87]. The cost is the extra bookkeeping needed to construct these interacting sets.

This technique does not require the application to be deterministic. The major advantages of coordinated checkpointing schemes are: the ease of finding a recovery-line (because consistent recovery lines are checkpointed as a whole), and the small overhead during failure-free operations (because no logging is necessary). The cost is time and communication overhead to store complete recovery-lines at once and hence the blocking of applications. This overhead can increase significantly with higher checkpointing frequencies. Another disadvantage is the load on the communication bandwidth resulting from saving the checkpoints of all of the process at the same time.

5.2.4 Hybrid Techniques

These techniques are based on coordinated checkpointing, but avoid the freezing of the application (hence, a non-blocking checkpoint) mostly by using the *marker rule* [Chandy 85] by logging "some" messages (e.g those crossing the recovery line).

The *marker rule* technique sends markers through the communication channels to determine their state. This technique assumes that the channels are error-free, preserve the ordering and guarantee message delivery in a *finite* time. Then a global consistent state can be obtained by using a *marker sending rule* (a process sends a marker through each of its send-channels immediately after taking a checkpoint) and a *marker receiving rule* (when a marker is received before a checkpoint is taken, this checkpoint is immediately taken and an empty channel state is included; if the marker is received after the checkpoint is taken, then the channel state contains all the messages between the time that the checkpoint was taken and the time that the marker was received). This ensures that a consistent global state will be taken.

Hence, these hybrid techniques merge the advantage of the message logging techniques to take checkpoints for a single process at a time (without blocking the application), with the benefit of coordinated checkpointing techniques to save complete recovery lines as a whole resulting in small failure-free overhead [Deconinc 93].

These conclusions are corroborated by Elnozahy in [Elnozahy 94] who found that the cost of writing the message logs and managing the recovery line outweigh the cost of coordinating the checkpoints during failure-free operation and the cost of contention on the stable storage server during the global checkpoint.

This research adopted the hybrid methods for developing FADI reliable distributed computing algorithm in favour of pure message logging and consistent checkpointing techniques because of the following:

- The algorithm has inherently low failure free overhead. No messages have to be systematically logged and no heavy bookkeeping (dependency tracking) as in optimistic logging schemes is necessary. This of course is at the expense of rolling-back all(interacting) processes and a possibly longer rollback that is tolerable because of the lengthy execution time of the FADI application domain.
- Strong coupling between the concurrent processes is typical of the majority of large scale scientific/engineering distributed applications, where the computation load is distributed between the application processes and strict synchronisation is required to communicate the results of various stages of the computation). Hence, checkpointing all the applications at once is an advantage because of process coupling. With independent checkpointing, the checkpoints of different processes cause slow down of the entire application. Instead, with coordinated checkpointing, all processes take a checkpoint at essentially the same time, causing only a single slow-down of the application [Zwaenepoel 92].
- The time-consuming task of finding consistent recovery lines is avoided - in coordinated checkpointing each sequence of checkpoints (a global checkpoint) by definition forms a consistent recovery line. This contributes to reducing the rollback time of the application.
- Coordinated checkpointing eliminates cyclic rollbacks (domino effect) by guaranteeing that processes do not need to roll back beyond their last checkpoint.

5.3 A Novel Reliable Algorithm for Checkpointing & Rollback of Distributed Applications

5.3.1 Design Considerations and Assumptions

The basic model is that all processes cooperate to create a global consistent recovery line, beyond which rollback is unnecessary. A centralised process running on a *fail-safe* node will coordinate the initiation and validation of checkpointing in all the application processes to form a consistent recovery line. This *coordinating task* will also be responsible for maintaining the information logged to stable storage necessary for the selective rollback of interacting processes (logged messages, communication trees, etc.).

Considering that the targeted applications are mainly computation-intensive, long-running applications, it is important to minimize the fault-tolerance overhead during normal (failure-free) operation in order to maximize the throughput of the system. This might be at the cost of a longer roll-back. The longer rollback can be admissible owing to the extended execution time of the application and the relatively small MTBF of today's Computer systems.

The checkpointing/rollback algorithm must provide an adequate level of user-application transparency. The fault-tolerance procedures and prologues should be integrated into the distributed application tasks without the involvement of the user. The following assumptions about the checkpointed environment were made:

- Fail-Stop processors are assumed, i.e processors generate correct results or no results at all. The system is assumed to stop immediately when a fault occurs and no false messages are sent. However, non-zero detection latency is tolerated [Taha 95].
- FIFO communication channels are assumed. i.e messages sent to the same destination are guaranteed to be delivered in the right order. These channels are also reliable, i.e they do not corrupt or lose messages. However, the duplication of messages is tolerated.
- A message received by a node might not be delivered to the destination process until it is requested, i.e messages might have already been *received* by the message passing daemon, but not yet *consumed (requested)* by the destination task. The algorithm should be able to recover from faults that occur while inter-process messages are in transit.

- Applications are not required to be deterministic in the sense that the output of the process is not only a function of its state and the input, but also is a function of time and other external factors (as is the case of real-time telemetry systems).

5.3.2 The Algorithm's Recovery Strategy: Coordinated Checkpointing with Selective Message logging

The adopted *hybrid* checkpointing methodology is based on the coordinated checkpointing strategy which implies taking checkpoints of the whole application at approximately the same time interval. Coordinated checkpointing is improved by the logging of messages that can invalidate the recovery line consistency, thus avoiding freezing of the application processes while the global checkpoint is being taken. So, two issues need to be addressed to develop a reliable distributed computing algorithm based on the hybrid techniques:

1) Checkpointing Coordination: processes should not start taking another checkpoint until the previous global checkpoint is already complete. It is important to allow the algorithm to investigate the consistency of the recovery line that is formed from the previous group of checkpoints (global checkpoint).

Since we assume a central *Coordinator* task responsible for initiating and controlling checkpointing process, probably the best way to guarantee the correct *checkpointing coordination* is to manage the exchanges of checkpoint requests-acknowledgments between the *Coordinator* task and the application processes. The *coordinator* task will not send another checkpointing request to any of the application tasks until all of them have acknowledged taking the previous one.

2) Recovery-Line Consistency: a consistent recovery line is a group of checkpoints, of all the application processes, beyond which rollback is unnecessary, i.e in the event of failure all the processes can safely rollback to checkpoints belonging to the last recovery line. The main problem here is the consideration of the state of the communication channels whilst checkpointing is in progress, i.e messages that cross the recovery line. This problem is illustrated by considering two execution scenarios and attempting to propose the appropriate solutions.

The following notations will be used throughout the rest of this section:

Pα : Process with index α;

Pset : Set of all the application's distributed processes;

C αi : the ith checkpoint of process α;

Ri : the recovery line containing the ith checkpoint of all processes;

GSti : the ith global state interval between recovery line i, i+1.

Scenario 1 Messages sent in one state interval and received in a subsequent one:

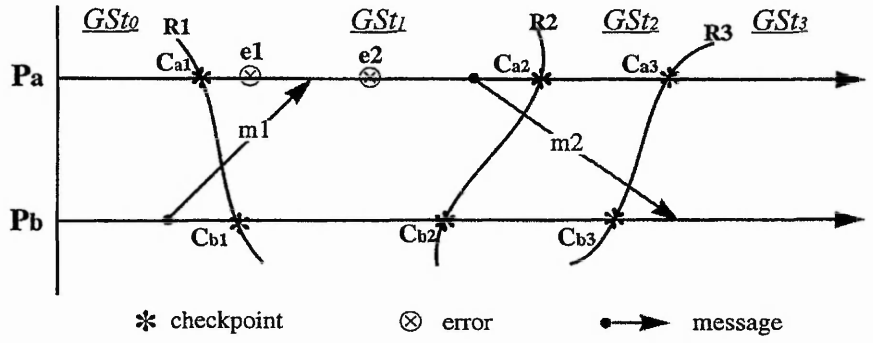


FIGURE 5-3 **Recovery-Line Consistency (a)**

In Figure 5-3, if an error occurs in *GSt1 (e1 or e2)*, then processes *Pa, Pb* will roll back to the recovery line *R1*. Hence, the receipt of *m1* by *Pa* will be undone, but *m1* will not be re-send by *Pb* after the rollback.

The most commonly adopted approach to solve this problem was suggested by Silva in [Silva 92] and Elnozahy in [Elnozahy 94]. The messages are appended with the sender's state interval *sSt* which is compared with the receiver's local (current) state interval *lSt* upon message receipt. If *lSt* > *sSt*, this indicates that the message was sent from a previous state interval across the last recovery line so it is logged. Upon rollback, these messages are replayed to maintain consistency.

However, these algorithms do not distinguish between the *delivery* and the *consumption* of the message. Messages *delivered* to destination tasks (on remote or local hosts) - whether using a low-level transfer protocol like TCP/IP [Comer 93] or a high-end message passing interface like PVM [Geist 94] - are *buffered* by the transfer protocol at the receiver's end until they are *requested* by the destination task *(consumed)*.

This *request* for consumption can be *pending* even before the message is delivered, can occur *shortly* after the message is delivered, or can be *delayed* for a considerable time while the destination task is performing other computations.

What follows is that the above approach compromises two aspects of the algorithm's reliability:

- It applies only to errors occurring after message consumption (*e2* in Figure 5-3), but does not take into account errors occurring after the message was delivered but before its consumption (*e1*), in which case the message (*m1*) will not be logged because the destination task (*Pa*) has not received it yet.
- It does not consider messages that might cross more than one recovery line (*m2*). All sent messages are assumed to be delivered at latest in the next state interval.

A principal prerequisite for recovery line consistency is the *logging of all messages that cross it from left to right.* In order to avoid the shortcomings of the solution suggested above by Silva, there are two realistic options to maintain the consistency of the recovery line:

1) Indiscriminate sender-based message logging;

2) Identification of messages that crossed the recovery line, waiting for their receipt, logging them and only then confirming that the recovery line is consistent.

The first option is quite expensive in terms of failure-free overhead, as explained in the previous section, so this research proposes a solution based on the second option.

Scenario 2 Messages sent in one state interval but received in a previous one:

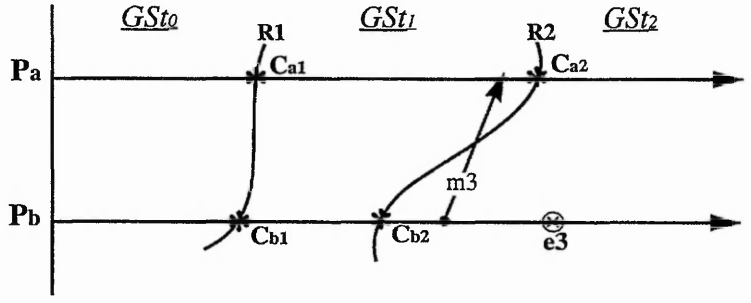


FIGURE 5-4 **Recovery-Line Consistency (b)**

Figure 5-4 illustrates the case when the error occurs in *GSt2 (e.g at e3)*, then process *Pb* will roll back to the recovery line *R2 and re-run*. The sending of *m3* by *Pb* will be repeated, but will not be matched by a corresponding receive request from *Pa*. Possible solutions to this problem include:

1) Assigning a *send sequence number "SSNα"* for every destination task "*dtaskα*", incrementing this number every time a message is sent to that destination and appending it to the sent message. The destination task compares the received *SSN* with that appended to the last message received from the same source. If it is less or equal, then the message is identified as redundant and discarded.

The disadvantage of this approach is that it requires the application to be deterministic since it assumes that an exact copy of *m3* will be sent after *Pb's* rollback. However, this approach can be used to identify duplicate messages (sent within the same state interval) that are caused by network malfunction.

2) Freezing of all processes until each have taken a local checkpoint. This approach implies a significant time and communication overhead during the failure-free operation of the application that can be computationally expensive for long-running algorithms.

3) Identifying messages sent from a subsequent state interval using state interval labels [Silva 92]. If *(lSt < sSt),* the receiving process takes a checkpoint before consuming the message to preserve the algorithms consistency.

The third solution does not require determinism because the sending and receiving of message *m3* will be accomplished within the same state interval *"GSt2"* and both operations (send and receive) will be redone (if needed) upon rollback. It also does not incur the overhead of freezing all the application processes as required in the second solution. Hence, we adopt the third solution to the consistency problem highlighted in scenario 2.

5.3.3 Data Structure of the Algorithm

5.3.3.1 Data Structure Maintained by the Application Processes

LSt : local state interval;

SSNα : send sequence number of latest message sent to process(α);

RSNβ : receive sequence number of latest message received from process(β);

LRmsg : message crossing the recovery line from right to left.

5.3.3.2 Data Structure Maintained by the Coordinator Task:

GSt : global state interval;

Pset : set of all applications of the distributed process;

C(l,m) : true/false communication flag indicating if process(l) has interacted with process(m) since the last global checkpoint;

UBmsg : record of unbalanced messages (sent but not yet received) since the last global checkpoint.

5.3.3.3 Control and Data Messages

Application message

Sender ID	Destination ID	SSN	LSt	Message Body

Checkpointing Request

GSt

Received Messages Request

—

Checkpointing Acknowledgment

Ackn. ID	SM(**ns**)	RM(**nr**)

where: **ns** - number of messages sent by acknowledging process since the last LSt;

nr - number of messages received by acknowledging process since the last LSt;

SM - array of: [Destination ID(i), SSN(i)], i = 1 .. ns;

RM - array of: [Sender ID(i), RSN(i)], i = 1 .. nr.

Received Messages Acknowledgment

Ackn. ID	RM(**nr**)

5.3.4 Functional Description

5.3.4.1 Sending and Receiving Messages

Upon *message send*, the message is augmented with three pieces of information: send sequence number of the destination process *SSNα*, the *sender ID*, and the sender current local state interval *LSt*. After the message is successfully delivered, *SSNα* is incremented. Every sent message is broadcasted to the Coordinator task. The Coordinator task keeps a list of communication flags *C* for all pairs *(l,m)* of application processes. This flag is set if any of the pair *(l,m)* sends a message to the other. The flags are cleared at the start of each new global state interval.

```
procedure send_msg()
begin
   append [SSN(dest_id) + sender_id + dest_id + sSt] to message body;
   broadcast message to Coordinator task;
   SSN(dest_id) ++;
end
```

Upon *message receipt,* the *SSN* included in the received message is compared with the sequence number of the latest message received from the same process (stored locally in *RSN*). If the previous sequence number (*RSN*) is greater or equal to the newly received *SSN*, this means that the message is a duplicate of a previously received one and it is discarded. Next the receiver process compares the received *sSt* with the current local *lSt*, if the local state interval is greater, then the message was sent from a previous state interval crossing the recovery line *LRmsg* and it should be logged, if it is smaller, a local checkpoint of the receiver state is taken to preserve the algorithm consistency.

```
procedure receive_msg()
begin
  unpack [ sender_id + dest_id + sSt & message body];
  if (SSN <= RSN(sender_id)) then
        discard received message;
  else
        RSN(sender_id) = SSN;
  endif
  if (sSt < lSt) then
        log msg to LRmsg in stable storage;
  else if (sSt > lSt) then
        take a local checkpoint;
      endif
  endif
end
```

5.3.4.2 The Checkpointing Protocol

At regular intervals the Coordinator task sends a checkpointing request *(ckpt_req)* to the application processes. The *ckpt_req* is sent together with the global state interval *GSt*. Each process takes a local checkpoint and sends an acknowledgment message *ckpt_ack* which contains local arrays of received messages *(RM)* and sent messages *(SM)*. Each *RM* array holds for every message originator two items of information: the "*sender_id*" and the receive sequence number of the last received message "*RSN*". Similarly to the *RM* array, the *SM* array contains: "*dest_id*" and the send sequence number "*SSN*" of last message sent to that destination.

After the Coordinator task receives *ckpt_ack* from the application process, it checks if all the sent messages were received by the correspondent destination tasks. If that is the case, then the recovery line is declared consistent and the global state interval is incremented.

Otherwise, information about unbalanced (unreceived) messages is added to the inconsistency (unbalanced messages) list "*UBmsg*". In this case, no new checkpoints are requested, and the Coordinator task inquires the processes on the receiving end of the unbalanced messages to check if the messages have been received *(recv_req)*. When all messages in *UBmsg* are received, the last recovery-line is declared consistent, *Gst* is incremented, and the normal checkpointing operation is resumed.

```
procedure checkpoint()
begin
  at periodic intervals do
  begin
    send ckpt_req[GSt] to all processes in Pset;
    await* for ckpt_ack[SM[sn], RM[rn]] from every p(i) in Pset;
    for each p(i) in Pset
      for each SM(k) in SM[sn]
        if not exists RM(j=1..rn) that ((RM(j).RSN == SM(k).SSN) &&
              (RM(j).sender_id == SM(k).dest.id)) then
          add SM(k) to UBmsg;
        endif
      end
    end
    if UBmsg is not empty then
      at periodic intervals repeat
        send recv_req to all SM(k).dest_id in UBmsg;
        await* for recv_ack [RM[rn]];
        for each SM(k) in UBmsg
          if exists RM(j=1..rn) that ((RM(j).RSN == SM(k).SSN) &&
              (RM(j).sender_id == SM(k).dest.id)) then
            remove SM(k) from UBmsg;
          endif
        end
      Until (UBmsg is empty)
    endif
    Declare Recovery-line(GSt) consistent;
    free logged messages from GSt-1;
    Gst++;
    clear communication flags C(l,m);
  end
end
```

(*) while the Coordinator is waiting for the acknowledgment messages to arrive, it concurrently listens to the error-detection task [Taha 95] for messages about process failure. If one of the requested processes fail, the Coordinator re-sends the request to it after its rollback.

5.3.4.3 The Rollback Protocol

When a process fails, the Coordinator task looks up the communication table C(l,m) and only the process that interacted with the failed process since the last global checkpoint are rolled-back. Messages addressed to the rolled back processes are replayed from the log.

```
procedure rollback(fail_processID)
begin
   look up processes in C(fail_processID, *);
   for all * processes
        rollback to Gst;
        unset C(fail_processID, *);
        replay all messages to * from LRmsg;
   end
   rollback fail_process;
   replay messages from LRmsg to fail_process;
end
```

5.3.5 A Distributed Checkpointing Scenario

Figure 5-5 illustrates a complex checkpointing scenario that demonstrates all the features of the proposed distributed checkpointing algorithm. The initial global checkpoint "*C0*" is taken before any communication or computation takes place, thus it is guaranteed that the recovery-line *R0* is consistent and if an error occurs at *"e1"* processes will rollback to their *"Cα0"* checkpoint.

After receiving *ckpt_ack* for checkpoint "*C1*" from *Pa,b,c* the Coordinator task adds *m1* and *m2* to the *UBmsg* list. Here if an error occurs at "e2", the application processes can not rollback to *"Cα1"* since the recovery-line *R1* is not consistent because the messages that crossed it from left to right are not logged yet.

During the next interval the Coordinator sends a *recv_req* to *Pb* and *Pc (Rqb1, Rqc1)*. By now *m1* is consumed and consequently logged, so it is removed from *UBmsg*. Subsequently the *recv_req* is sent to *Pc (Rqc2)*, which acknowledges it with a confirmation that *m2* has been received (consumed) and the recovery-line *R1 {Ca1, Cb1, Cc1}* is declared *consistent.* So, in the case of failure (e.g at *"e4"*), the application processes can now rollback to this recovery line. Following that the Coordinator initiates the next global checkpoint "*C2*".

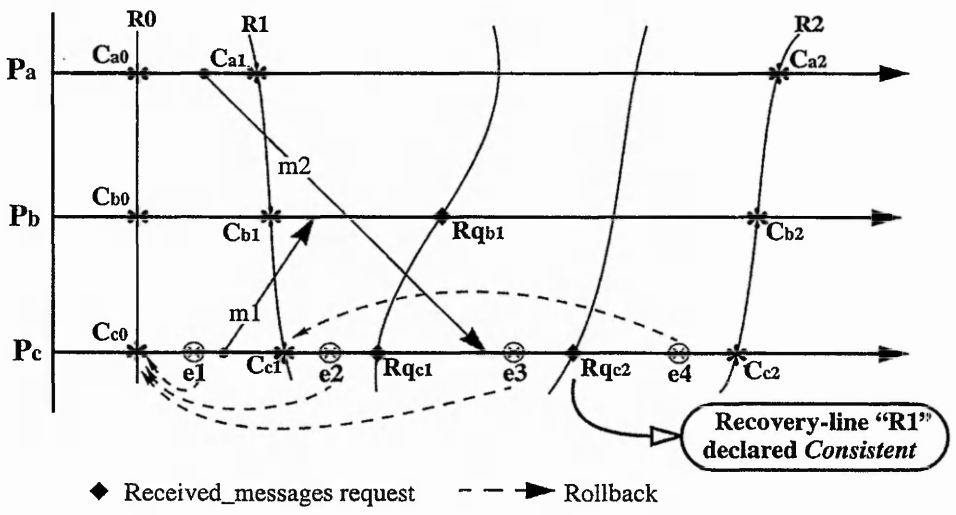


FIGURE 5-5 **A Checkpointing Scenario**

5.3.6 Proof of Correctness

In order to prove the correctness of the protocol, we need to show that the checkpointing protocol will eventually terminate forming a consistent recovery line (the set of checkpoints in stable storage is consistent). The protocol must be able to recover the application distributed processes to a consistent state after host crashes.

‡ **lemma 1:** *at any instance of the applications execution there will always be a consistent recovery line.*

proof: at the start of the applications execution, all the application processes take a checkpoint before they start executing their main code. These checkpoints of course form a consistent recovery-line because no computation or communication have been carried out yet. Taking into consideration that the Coordinator task will not discard the previous consistent recovery-line until the current is confirmed consistent, we prove the lemma.

‡ **lemma 2:** *The checkpointing process (protocol) eventually terminates.*

proof: for the checkpointing protocol to terminate successfully two conditions have to be met:

1) All the ckpt_req and recv_req messages must be delivered to the Pset of application processes, where they will be processed and the acknowledgment information (ckpt_ack, recv_ack) must be returned to the Coordinator process.

The Coordinator process is fail-safe and the communication channels are assumed reliable. Therefore, the request-acknowledgment cycle can deadlock only if the application processes fail before they receive the ckpt_req or recv_req* from the Coordinator. In this case the Coordinator task will be informed about the failure and it will re-send the request to the rolled-back process. This avoids the dead-lock trap.

2) The undelivered messages in UBmsg will be eventually received.

The Coordinator task will not terminate the current checkpointing until all the UBmsg list is empty. For every message m sent a communication flag C(l,m) corresponding to the sender/receiver is set. If any of them fails before the message is received, then they will both be rolled-back to the previous recovery-line - that always exists from lemma "1" - and the message will be re-sent and ultimately received.

(*) the ckpt_req and recv_req messages are interrupt signals, they suspend the application process until the requests are served. This means that they cannot be blocked and reduces the possibility of their loss.

‡ **lemma 3:** *at the end of the checkpointing protocol, a consistent recovery-line can be obtained.*

Definition: A recovery line is consistent if all the application processes have taken a local checkpoint and messages crossing the recovery line from left to right are logged.

Proof: from lemma "2" the Coordinator task will not terminate the checkpointing protocol until:

a) it receives ckpt_ack from the Pset processes, which implies that all the application processes have taken a checkpoint;

b) all unbalanced messages are eventually received. Since our algorithm identifies messages sent from a previous state interval (LRmsg) and logs them on receipt (see 5.2.1), then it is guaranteed that all messages crossing the recovery line will be logged.

Paragraphs "a)" and "b)" comply with the above definition of a consistent recovery line.

‡ **lemma 4:** *Only one global checkpoint need to be saved to stable storage.*

proof: from lemma "3" the last committed global checkpoint is consistent. Hence, failed processes do not have to rollback beyond checkpoints from the last recovery line. Messages that were sent from previous intervals are replayed from LRmsg log. Therefore the *domino effect* is avoided and multiple rollbacks are not possible.

‡ **Theorem 1:** *All the application distributed processes can be recovered to a consistent state after host failures. In addition, only processes that interacted since the last checkpoint are rolled back.*

proof: from lemma "2" & "3", there will always be a consistent recovery line in a stable storage, and messages crossing the recovery line are logged and replayed to/from stable storage in the same order via the assumed FIFO channels.

5.4 Algorithm Implementation

5.4.1 Introduction

Throughout the implementation of the algorithm emphasis was put on maintaining the transparency of the user application to the distributed checkpointing protocol. CUMULVS [Kohl 96] is a distributed processing environment similar to FADI, whose applications also use PVM as a message passing substrate. It supports interactive visualization and remote steering of distributed applications, and provides fault tolerance to applications running in heterogeneous distributed environments. Although its developers claim that it requires minimal modification of the user application to specify the nature and decomposition of the data fields, it nevertheless requires a profound knowledge of the user-application-programming model to be able to describe the decomposition of the program data fields and optimum points to insert the user-directed checkpoints.

Our goal is to develop a general-purpose transparent fault-tolerance model that will free the application-programmer from involvement in the application error recovery, and more importantly, permit the use of conventional PVM applications that were not customised to run in a particular distributed environment.

To achieve the required transparency, every application-program is automatically assembled with a *checkpointing prologue* with the help of a specially designed pre-processor without requiring any intervention from the user/programmer side. This prologue performs all the backup and recovery operations on behalf of the application-program, thus minimizing the modifications needed to integrate the application code with FADI environment. As illustrated in Figure 5-6 the checkpointing prologue serves the checkpointing and recovery of the application code with three procedures:

1. Initialisation of the checkpointing/rollback protocols (e.g setting checkpointing signal handlers) before starting/re-starting the user program.
2. Handling of the checkpointing-request signals sent by the checkpointing coordinator.
3. Performing recovery-related operations for augmented PVM and UNIX functions before calling the original ones (e.g saving the name and attributes of a file before calling UNIX *open()* to actually open the file).

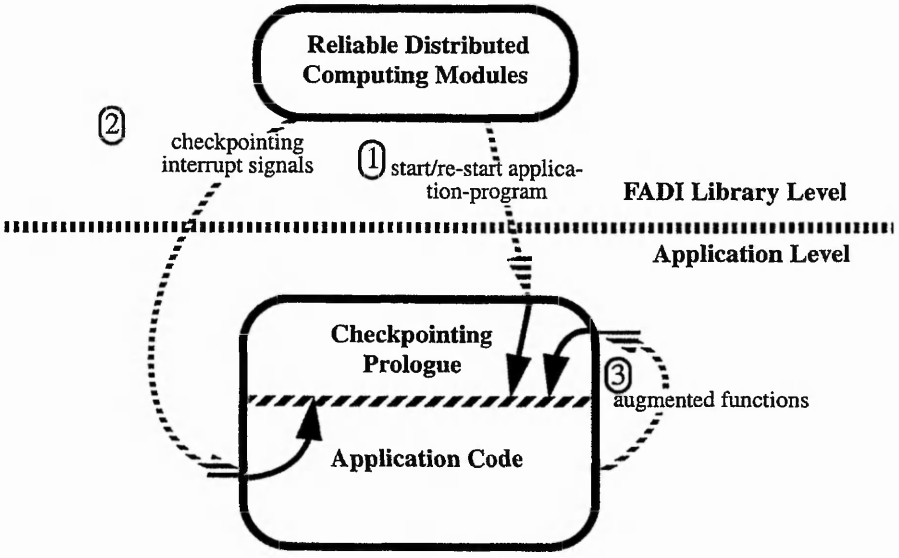


FIGURE 5-6 **Interfacing Application Programs with FADI Libraries**

5.4.2 Communication between Application Processes

The Parallel Virtual Machine PVM is deployed to facilitate the communication between the control and monitoring processes of FADI on one hand and between the distributed application processes on the other. Hence, FADI applications should use PVM as a message passing substrate if they want to exchange interprocess messages - although this is not a prerequisite if the application consists of stand-alone programs.

FADI distributed applications can utilise almost all the PVM functionality including inter-process communication, process synchronization and many other useful parallel programming tools such as dynamic process-group operations and multi-threaded debugging. The only exception is multi-cast (message broadcast operations). Investigation into PVM implementation of message broadcast operations *(pvm_mcast()* and *pvm_bcast())* revealed that they do not wait for the message to be received by all the tasks within the broadcast group (asynchronous message delivery). Therefore, if one of the destination tasks fails, PVM multi-cast will not be able to identify it and consequently measures can not be taken by the reliable distributed computing protocol to re-send the message after task recovery.

To achieve the reliability of the communication interface, PVM functions were augmented to perform recovery-related tasks. A header file is prepended to the application code that re-defines PVM functions targeted for augmentation. When the application code calls *pvm_recv()* for instance, the function is re-defined as *ftpvm_recv()* which is executed by the *checkpointing prologue*. *ftpvm_recv()* performs the necessary recovery related preprocessing before returning to the application code to execute *pvm_recv()*. In this manner, the following adjustments were made to the PVM library:

1. Checkpointing has to be blocked during the process of delivery (send buffer initialisation - data packing - message sending) and the receipt (receiving message - unpacking data) of a PVM message. The reason is that send and receive buffers are maintained by the host PVM daemon and are freed (deleted) after a message is successfully sent or received. So, if a checkpoint is taken during packing/unpacking of a message, then when a rollback is made to this checkpoint, the restarted task will fail to access these buffers because the PVM daemon has removed its internal pointers to them. Therefore, PVM functions that initiate the send or receive processes *(pvm_initsend(), pvm_mkbuf(), pvm_recv(), pvm_nrecv(), etc.)* are augmented to initially mask-out the checkpointing signal. PVM functions that terminate the send or receive processes *(pvm_send(), pvm_psend(), pvm_upck(), etc.)* are augmented to reset the checkpointing signals.
2. The checkpointing prologue needs to take over execution before any data is packed into the send message in order to prepend bookkeeping information first such as the message send sequence number, the local state interval of the process, the sender id, etc. Simi-

larly PVM message receive functions are preprocessed to unpack this bookkeeping information first and make decisions about logging the data packets carried in the message.

5.4.3 Integrating the Reliable Distributed Communication Protocol into FADI

The Data-flow diagram in Figure 5-7 illustrates how FADI main modules are interfaced with the application processes via their checkpointing prologues to realize the fault-tolerant communication protocol. A data dictionary interpreting the flow of data between the processes is presented in Table 5-1. See appendices "A" and "B" for complete listing of Data-flow and Data-structured Design of FADI.

Three modules collaborate to perform the reliable distributed communication protocol. The *checkpointing coordinator* and *recovery process* are integral parts of the FADI software daemon. They control checkpointing and rollback of the application processes by exchanging control and data messages with the *application-process prologue.*

5.4.3.1 The Application Process Prologue

When FADI spawns the application tasks on one of the distributed system hosts upon task start or restart (in case of failure), the application prologue takes over the process execution. It first logs the application process into PVM and disables direct task-to-task message routing. In PVM, it is difficult to track the set of open communication sockets if direct routing is enabled, because routes are created as messages are either sent or received. Therefore communication between application tasks is performed past PVM daemons *pvmd* that are resident in the application host kernel. Task-to-pvmd communication channels are stable and their state can be restored upon rollback. This implementation incurs overhead on message delivery time.

Next the prologue initiates the checkpointing procedures: it initialises the open file table to save/restore the state of files held open by the application at checkpointing time, then installs functions to handle the *checkpointing coordinator* checkpointing signals and calls the application-program main procedure.

At checkpointing time the application process prologue receives the global state interval index from the *checkpointing coordinator* and acknowledges it by sending sent & received

messages record. A copy of the application-program is forked to save the process image without blocking the execution of the application code. The saved image is considered as a tentative checkpoint and is not saved into permanent storage until the next checkpoint is requested - which confirms that the last recovery line (global checkpoint) was consistent.

As explained above the prologue also performs recovery-related tasks on behalf of the application process: It masks out checkpointing signals during checkpointing and analyses bookkeeping information prepended to received messages to decide whether to log the message body into a stable storage. The prologue also detects unsuccessful send operations due to destination task failure, in which case it requests the new ID of the recovered destination task from the *recovery process* and re-sends the message.

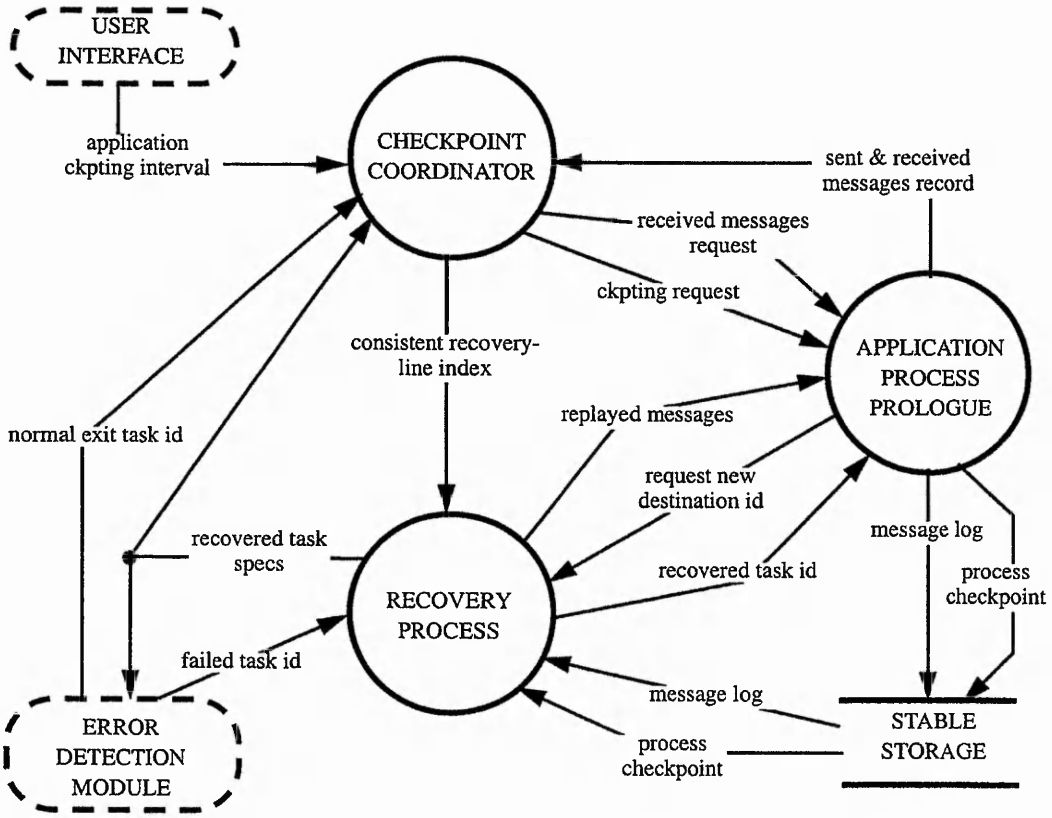


FIGURE 5-7 **Reliable Distributed Computing Protocol**

TABLE 5-1 **Data Dictionary**

ITEM	USAGE
checkpointing interval	\ time interval between two consecutive checkpoints \
normal exit task id	\ successfully terminated application task \
failed task id	\ id of tasks that exited with erroneous status \
recovered task specs	\ specifications of processes that were rolled back from the last checkpoint \
sent & received messages record	\ arrays comprising indices of last sent or received message by an application process to the others since last checkpoint \
received messages request	\ an interrupting request sent to send received messages array to verify if messages crossing the recovery line are received by the destination \
ckpting request	\ an interrupting request to take a local checkpoint and send sent & received messages records \
consistent recovery-line index	\ index of latest recovery-line verified by the checkpointed coordinator of holding consistent checkpoints \
replayed messages	\ messages replayed to recovered tasks after their rollback \
request new destination id	\ requesting ID of recovered destination task upon send failure \
recovered task id	\ responding to sender request for new destination ID \
message log	\ messages crossing the recovery line from left to right saved into permanent storage \
process checkpoint	\ saved image and kernel state information of an application process \

5.4.3.2 The Checkpointing Coordinator

Communication in FADI is built strictly on a message passing system, which allows the application tasks and FADI software modules to be distributed on several nodes. Hence, all FADI main modules reside on a fault-tolerant central host, from where they control and coordinate the reliable execution of all the distributed application-programs. This centralised structure alleviates the need for replicating the *checkpointing coordinator* and *recovery process* for every network host and eliminates the overhead of their backup/recovery if the network node fails.

The *checkpointing coordinator* sets a system timer to send checkpointing signals(requests) to the application tasks at the elapse of every application checkpointing interval. Software flags are used to send either a checkpointing request or a received-messages request if the recovery line is not consistent. The received sent & received messages records from all application tasks are balanced to verify the recovery line consistency. If the recovery line is

consistent, then its index is sent to the *recovery process* and the log is freed from messages belonging to the previous recovery line. Otherwise, the checkpointing flag is set to request received messages records at the next checkpointing interval. Upon the receipt of a recovered task specs from the *recovery process*, the ID of the failed task is updated and a checkpointing request is re-sent to the recovered task if its failed predecessor was in debt of checkpointing acknowledgments. When an application task exits normally, the *checkpointing coordinator* is informed to remove the task ID from the checkpointed task list.

5.4.3.3 The Recovery Process

Upon the receipt of a failed task ID from the *Error Detection Mechanism (EDM)*, the recovery task rolls back the failed task to the last consistent recovery line. The new ID is sent to the *checkpointing coordinator* and the *EDM* to resume its backup and monitoring. The index of the consistent recovery line is promptly sent by the *checkpointing coordinator* and updated locally. Logged messages belonging to that recovery line are replayed. The replayed messages are stripped of the bookkeeping information because that information needs not to be analysed since the messages were sent from a previous checkpointing interval, therefore should not be balanced and they already reside in the message log. To allow the *application process prologue* identify the replayed messages they are prepended with a *negative* state interval number.

5.4.4 Limitations Incurred by the Algorithm Implementation

Adopting off-the-shelf software product will always constrain the software module under development, and there will necessarily be points of conflict that would not arise had all the software been tailor-developed to suit a particular application. Using the PVM message passing interface for inter-process communication resulted in the following limitations to FADI's applications programming model:

- Multi-cast operations are not supported because PVM message broadcast operations are asynchronous and thus unreliable. Therefore, a failure of sending to a destination task cannot be singled out within the broadcast group of destination tasks.
- Direct task-to-task message routing is not allowed because its use would make it virtually impossible to save/restore the state of the communication channels as argued above.

- Programmers should avoid identifying incoming messages by the originator ID for two reasons: 1) the sender ID changes after task restart, of which the destination has no knowledge; 2) after task restart, logged messages will be replayed by the *recovery process,* not the original task. It is recommended to label messages with unique tags for message identification.

However, the advantages of using PVM message passing interface certainly outweigh the constraints resulting from its integration into FADI. PVM provides a high level library interface for interprocess communication, process synchronization, and many useful tools for the parallel programmer as dynamic process-group operations. Furthermore, The Parallel Virtual Machine has a huge user-base that consequently reflects on the number of potential FADI applications.

5.5 Conclusions

This research resulted in a novel checkpointing and rollback recovery technique for distributed computing systems. It is based on a hybrid technique that combines consistent checkpointing with its low failure-free overhead, with logging of messages that cross the recovery line (to avoid blocking the application process during the checkpointing protocol).

In contrast to other published fault-tolerant techniques, FADI is tolerant to errors occurring whilst messages are in transit, i.e messages are delivered to the destination (queued at message passing daemon or transport protocol thread), but not yet requested (consumed) by the receiving task. Another important feature of FADI is that it tolerates duplication of messages by the communication channels and requires only one global checkpoint to be recorded in a stable storage.

The correctness of the algorithm has been theoretically proven, and its integration into the distributed processing environment has been described.

CHAPTER 6 Evaluation of The Fault-Tolerant System

This chapter is composed of two sections: the first describes performance studies of FADI using a synthetic application. The purpose of this study is to benchmark FADI's operation and analyse the overheads associated with the reliable execution of application processes on distributed computing resources. The second section examines the application of FADI to a real-life complex distributed decision-support system.

6.1 Benchmarking FADI Using a Synthetic Application

Chapter '4' studied the performance of FADI checkpointing mechanism in the context of stand-alone *"non-interactive"* applications to verify its validity as the nucleus backup and recovery protocol for the fault-tolerant system. The algorithm presented in the previous chapter expanded on the non-blocking checkpointing mechanism to cover the possible inter-process communications taking place between the distributed application processes [Taha(2) 97]. Hence it is necessary to re-evaluate the performance of FADI in relation to providing reliable distributed computing for message-passing *"interactive"* applications.

This chapter does not document unit and integration tests (white-box testing) of FADI. Although these tests are necessary - *and have been carried out* - to verify the behaviour of the software modules and the interfaces facilitating communication between them [Pressman 92], they do not contribute to the scientific substance of this thesis. Here emphasis is given to validation and system tests that should permit to determine how the execution of distributed applications is affected by the overheads of managing their tolerance to hardware faults and establish the optimum conditions to minimize this overhead.

6.1.1 Hardware Setup

The performance tests were carried out on a network of three workstations connected by a 10 Mbit/sec Ethernet. The main host running the Fault-management tasks is a SPARC_20 server running SOLARIS 5.5.1 (60 MHz clock rate, 32 Mbytes of main memory, 5 Gbyte SCSI disk drives). The applications were distributed between two diskless SPARCstation *IPC*s running SunOS 4.1.3 (40 MHz clock rate, 8 and 16 Mbyte of main memory).

The communication network is part of the general departmental LAN, and is affected by the network traffic of other users, unlike dedicated networks used in performance studies by other researchers such as in [Elnozahy 94]. This is bound to affect the overhead of the fault-tolerant system, albeit on a small scale.

6.1.2 Application Programs

Some related work have implemented solutions to engineering problems (FFTs, matrix multiplication, Gaussian elimination, etc.) to benchmark the performance of their fault-tolerance techniques for distributed systems [Janakiraman 94] [Sens 95]. We opted for a synthetic application because it allows us to fine-tune the application variables (heap size, message rate, communication-computation ratio) to test various aspects of the system performance.

This study evaluates FADI's performance for managing fault-tolerance for message-passing applications. Therefore, a test application was designed which is communication-intensive and with small computation cycles, so that the measured overhead will mainly be due to interprocess communication. Four tasks were distributed amongst two SPARCstation *IPCs*. The first generates random numeric data, passes it to the second, where it is hashed by a simple arithmetic operation, and the same is repeated by the third task. The fourth task holds the hash keys for the second and third tasks. It decodes the received data package and sends it back to the first task where it is checked against the original values to verify that the data integrity was maintained throughout the pipeline. The application code was written in "C" and PVM was used to facilitate interprocess communication.

6.1.3 Evaluation

6.1.3.1 Performance Metric Requirements

As shown in Figure 6-1 application programs execute in parallel with the FADI fault- management modules. These modules run on the main *"server"* host and can affect the execution of the application code only upon start/restart of the application programs and during processing the checkpointing cycles using *interrupt* control messages. The application prologue takes over the execution of the application program - *the shaded area in Figure 6-1* - to handle the interrupt and perform checkpointing initialisation or recovery related operations.

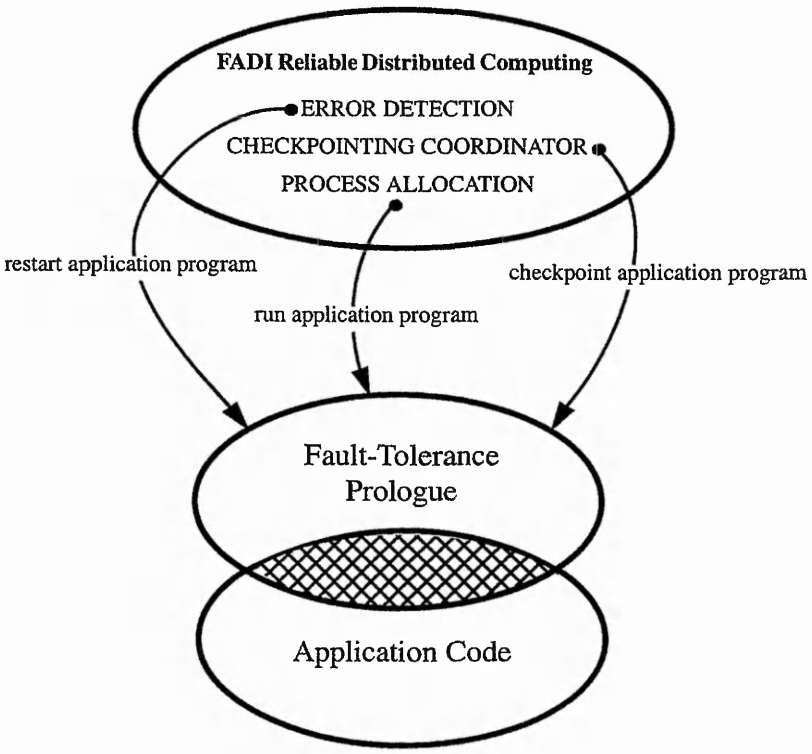


FIGURE 6-1 **Interaction between Fault-Management Procedures and the Application**

FADI application domain is mainly long-running scientific/engineering applications. Hence, emphasis is put on evaluating the performance in the failure-free case where the accumulation of the checkpointing overhead can tangibly affect the application run-time.

Because of the long MTBF of modern computer systems, failures are considered an exception rather than a rule, therefore the overhead of application rollback can hardly affect its execution time. However, for on-line distributed applications (e.g flight reservation systems, industrial decision support systems), the disruption of distributed services must be within acceptable limits to the system operator. The average rollback time for the synthetic applications experiment was *6.5* seconds.

The initialisation of the checkpointing/rollback protocols is performed once before the start/restart of the application program and does not affect the run-time overhead of the application. Therefore, the overheads to be studied for the performance analysis are:

a) The message logging overhead: it includes the overhead of tracking the dependencies of sent/received messages exchanged between the application processes (e.g. augmenting bookkeeping information to messages, processing this information to maintain the consistency of the communication channels, etc.), and the overhead of logging inconsistent messages into stable storage.

b) The checkpointing overhead: it includes the overhead of taking the checkpoint (obtaining and saving the process image: open-files, communication sockets state, signal handling information, process data and stack segments), and the time spent on writing the checkpoint to a disk.

6.1.3.2 Experimental Results

Figure 6-2 shows the affect of varying the rate of messages exchanged between the application processes on the failure-free overhead of the application. Incrementing the message rate mainly affects the time spent on augmenting *each* message with bookkeeping information (12 bytes containing: State interval, sender ID, send sequence number), and tracking the dependencies of the exchanged messages, i.e keeping records of the last sent and received message for every application. This overhead is persistent as long as there are messages sent/received between the application processes, while the message logging overhead affects the execution of the application only if a message crosses the recovery line due to our selective message logging policy. Subsequently the failure-free overhead gradually grows, but even with a considerable message exchange rate (6.4 Kbyte/sec) the overhead was measured at *2.9%* which comfortably falls within the widely acceptable *"10%"* overhead on application execution time [Plank 94].

Sens in [Sens 95] implemented an independent checkpointing scheme with pessimistic message logging, and for a much less communication-demanding application (0.06 Kbyte/ sec) the overhead was *3.0%* with a similar checkpointing interval of *2min.*

From Figure 6-2 it can also be noticed that changing the size of the application processes while varying the message rate has little affect on the failure-free overhead, the curves for various heap sizes of the application process almost overlap.

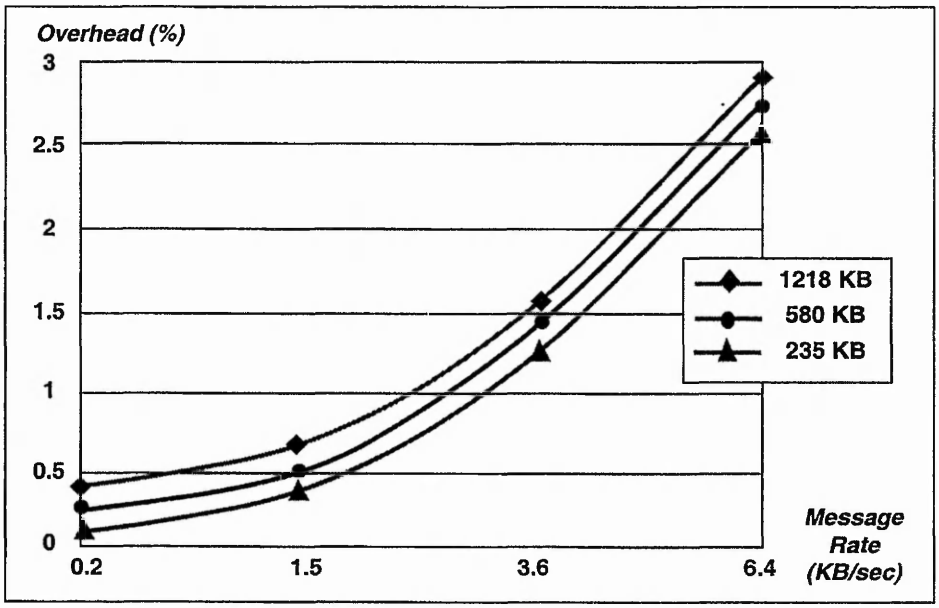


FIGURE 6-2 **Failure-Free Overhead as a Function of Message Exchange Frequency**

Figure 6-3 illustrates the checkpointing overhead of the application for three image sizes versus the frequency of taking the checkpoints.

For the three variations of the application the overhead is observed as the checkpointing interval is reduced, the reason being that with smaller intervals (higher checkpointing frequencies), more checkpoints are taken. When employing the non-blocking checkpointing policy, the checkpointing procedures are *virtually* performed by a forked thread while the main program concurrently continues execution. However, some execution time is still lost when the OS kernel makes the context switch and swaps one of the processes out.

This slight increase in the overhead escalates with large image sizes (more data to write to disk) and hence the deviation in the overhead of the three curves in Figure 6-3 as the checkpointing interval decreases. Figure 6-4 highlights the advantage of *non-blocking checkpointing*. For the most moderate overhead conditions (362 KB applications size and 10

min. checkpointing interval) the overhead of sequential checkpointing was *5.72%* as opposed to *0.12%* for non-blocking checkpointing.

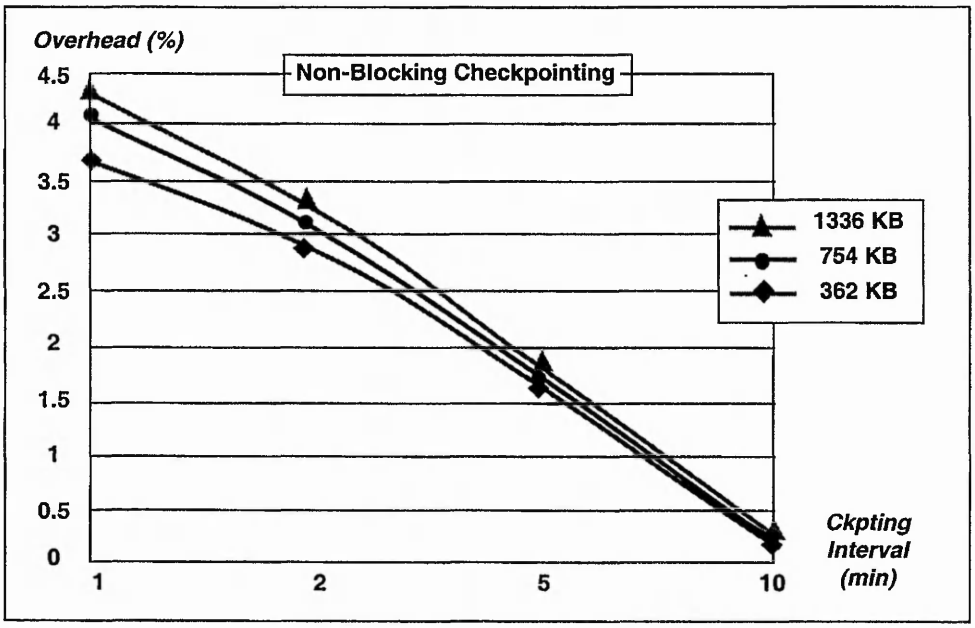


FIGURE 6-3 **Failure-Free Overhead as Function of the Checkpoint Interval (a)**

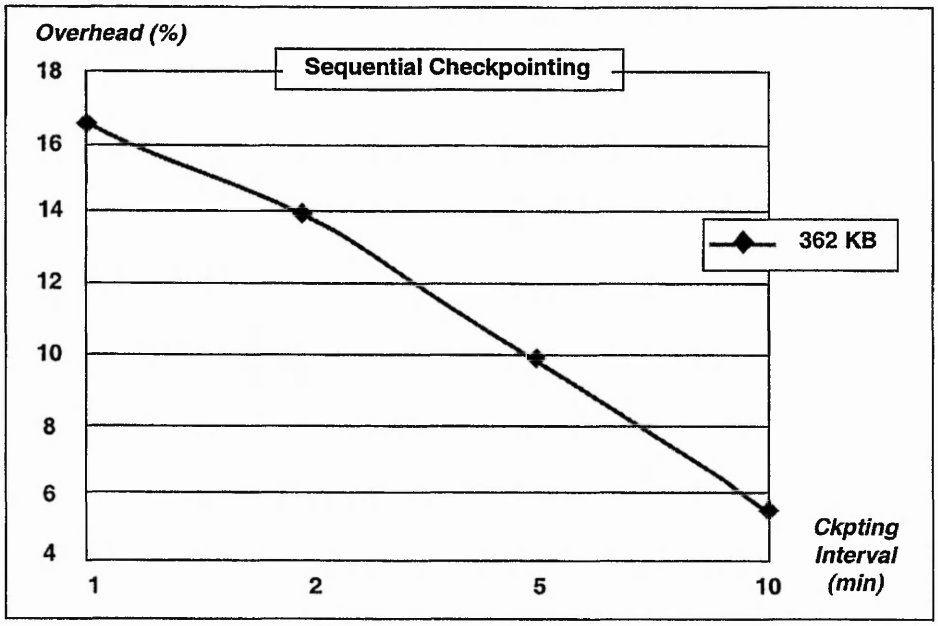


FIGURE 6-4 **Failure-Free Overhead as Function of the Checkpoint Interval (b)**

In general the overhead recorded in the second experiment is higher than the first. Taking more checkpoints increases the probability of messages crossing the recovery line, and subsequently the number of logged messages (disk I/O) which takes significantly more execution time than computation cycles for tracking the dependencies for higher message exchange rates.

Under similar message rate and image size conditions, Elnozahy and Zwaenepoel's *message logging with coordinated checkpointing* algorithm [Elnozahy 94] scored less overhead for higher checkpointing frequencies (*1.8%* compared to ours *2.8%* for 2min interval), but our algorithm performed better as the checkpointing frequency decreased (*0.2%* overhead compared to Elnozahy's *0.3%* for 10min checkpointing interval). This is despite the added complexity to our algorithm caused by catering for failures occurring whilst interprocess messages are in transit [Taha(2) 97].

Figure 6-4 highlights the advantage of non-blocking checkpointing. For the most moderate overhead conditions (362 KB applications size and 10 min. checkpointing interval) the overhead of sequential checkpointing was *5.72%* as opposed to *0.12%* for non-blocking checkpointing.

6.1.4 Conclusions

The performance study on synthetic applications indicates that FADI's reliable distributed computing protocol incurs low overhead even with high message exchange rates, thus making it suitable for communication-intensive distributed applications. The largest overheads (although still acceptable "<< *10%*") were measured for shorter checkpointing intervals. Because of the transparency of the implementation of our reliable distributed computing algorithm, the checkpointing frequency can be tuned without any modification to the application code. The requirements of FADI scientific/engineering application domain (extended execution time) allow for higher checkpointing frequencies and will therefore bear low overhead when run under FADI. The experimental results have also shown that our implementation has improved performance over related algorithms for reliable distributed computing.

The stable-storage reliability can be improved by introducing a replicated file system. This can be achieved at hardware level by using duplicate hot-swappable disk drives (e.g

RAPID™ high security drives from Digital Interactive Solutions™ [Digital 97]) without extra overhead or modifications to the software.

We conjecture that these results will hold on modern hardware platforms as well. The communication rate of the test environment was 10 Mbit/sec but *Fast Ethernet* can deliver a rate up to a Gigabit per second and the latest Sun SPARCstation *"ULTRA 2/1200"* clocks 200 MHz as opposed to the 60 MHz of the experiment's SPARC_20 server.

6.2 FADI Evaluation By Applying a Real-Life Distributed Processing System

Most of the performance evaluation work described in the literature [Elnozahy 94] [Bernard 94] [Li 94] have chosen number-crunching applications for classical engineering problems: (matrix multiplication, Gaussian distribution, FFT applications, etc.) to evaluate the performance of their distributed computing systems. Although valid, these tests often do not reveal the actual behaviour of the fault-tolerant distributed environment when deployed to run realistic engineering applications. This research has undertook to evaluate the performance of FADI also on a real-time telemetry application to complement the evaluation described in the previous section. Here, we seek to demonstrate the practicality of the use of FADI for providing a fault-tolerant platform for real-life distributed systems.

6.2.1 Background to the Industrial Application

The selected application is concerned with *Computer-Assisted Control of Water Distribution Networks.* Original work on this control system was carried out by A. Bargiela [Bargiela 84]. The developed on-line monitoring system comprises of a number of concurrent software modules, including network simulator, telemetry system estimator, state estimator(s) and the operator's interface. The principal task of this system is to process redundant, noise-corrupted telemeasurements in order to supply a real-time data base with reliable estimates of the current state and structure of the network.

The measurements are processed on a continues basis (1min scan rate) and the state estimation module identifies discrepancies between the mathematical model of the network and the actual meter readings. These discrepancies are then analysed so that their causes, such

as the presence of leakages, closed valves, or erroneous transducer data, are found and remedied [Bargiela 95].

Two methods of state estimation, with different numerical characteristics, were implemented. The first method uses an augmented matrix formulation of a classical least-squares problem, and the second is based on a least absolute value solution of an over-determined set of equations. Two water systems, one of which is a realistic 34-node network, were used to evaluate the performance of the proposed methods.

The realisation of such complex monitoring and control system as for most industrial decision-support systems constitutes the utilisation of distributed computing environment. The main reasons for this approach are:

1. On-line monitoring and control requires that a number of computational tasks execute simultaneously in parallel or in pipe-line fashion (e.g bad data processing, state estimation and calculating optimal valve controls), which is more effectively implemented in a true distributed system than in a time-shared uni-processor environment.

2. The price of the adoption of such numerical techniques such as state estimation incurs a considerable computational load. Parallel and distributed algorithms are seen as an answer to the computational complexity problems of such techniques [Bargiela 93].

3. With the increase of the system size to several hundred nodes, the topological decomposition, which maps well onto distributed computing, becomes a natural way of describing the system.

6.2.2 Functional Description of the Monitoring and Control Application

There are three main groups of programs in the package (Figure 6-5). The programs of the first group simulate the behaviour of the real network and provide measurement information which in practice is retrieved using some telemetry system. This data is effectively the only source of information for the second group of programs monitoring the network.

A major role of the monitoring programs is to supply information about the system state both for the human operator and control algorithms, Since the telemetered data is being updated without the intervention of a human intermediary the monitoring programs are said to be on-line to the process.

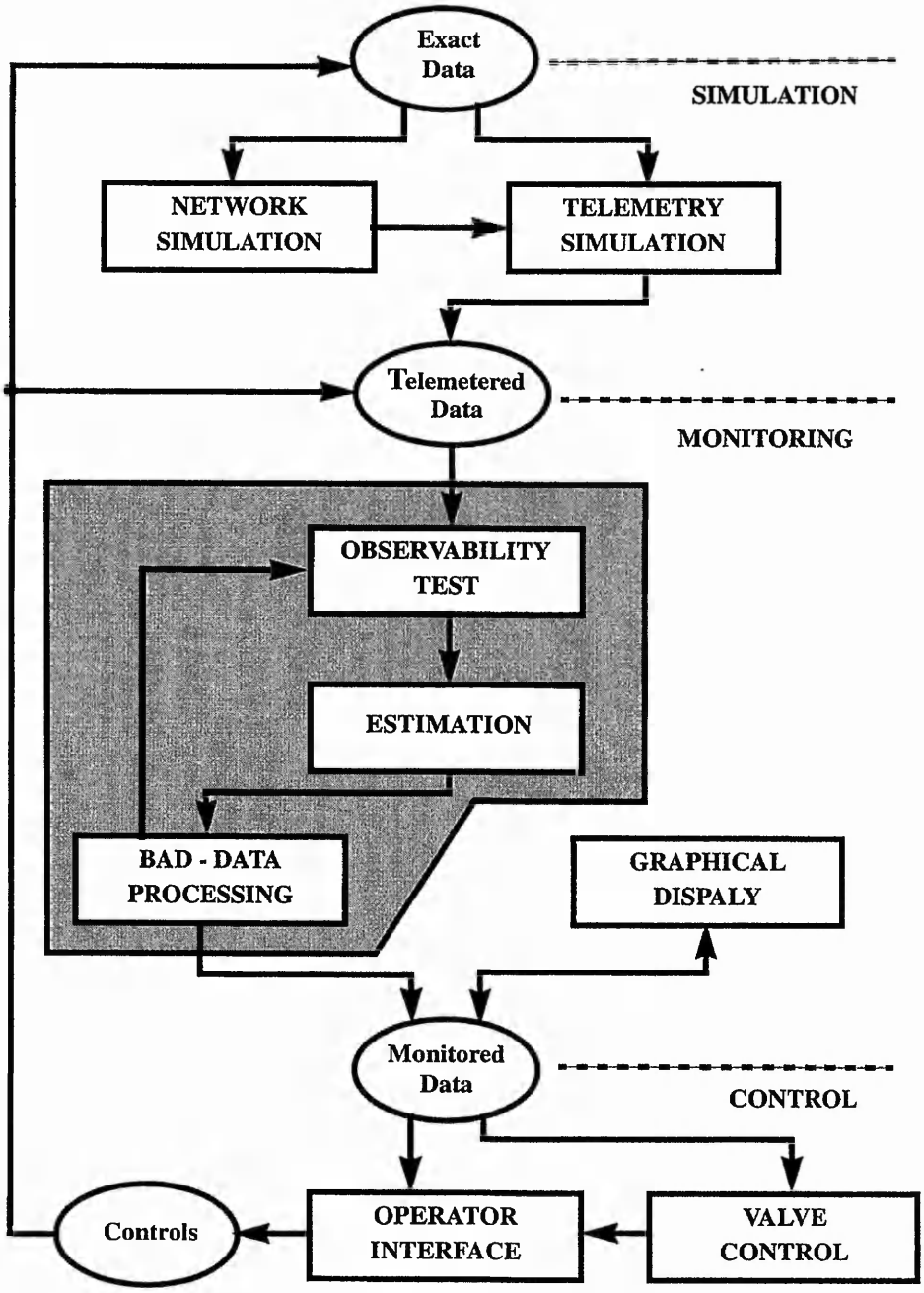


FIGURE 6-5 **Water Network On-Line Monitoring and Control Scheme**

After checking topological observability of the system, with respect to the current set of valid measurements, the estimates of the state vector are calculated. This is followed by identification of bad data points which were not found during the preprocessing stage,

Depending on the state estimation algorithm employed, the monitoring procedure involves either an iterative elimination of bad data from the set of valid measurements and re-computation of the state vector, or it simply marks erroneous measurements having rejected them in the course of the estimation. The results obtained with the monitoring programs are made available to the operator in the form of a print-out, graphical display and data file which is also used by control algorithms.

The third group of programs closes the control loop by devising and implementing control action, The flow of information between the programs implies that algorithmically calculated controls are off-line to the process since they are implemented by a human operator. Such a structure is natural at the initial stage of the computerised monitoring and control of a water network. However, it must be emphasized that the computer assisted control can be easily converted into a full on-line control scheme since the system is monitored on-line.

i) Network Simulation Module

The water network simulation program (SYSSYM) provides a facility to carry out on-line monitoring studies without recourse to a real-life telemetry system, The input data for the network simulator represents exact information about the system and, as such, are not available to the monitoring programs. This data can only be modified by the control action of the operator.

The simulator calculates an exact state vector, by applying a Newton-Raphson iterative procedure to the square set of non-linear mass-balance equations, and passes it to the telemetry simulation program which calculates the values of the measurements.

ii) Telemetry Simulation Module

Using an exact state vector, supplied by the network simulation program, and information about the meter positioning the telemetry simulation program (SYSTEL) calculates the exact values of the measurements. In order to obtain realistic set of telemeasurements, a pseudo-random measurement noise is superimposed on the meter readings. The program also enables the simulation of manufacturing of telemetry or instrumentation by making provision for the corruption of the measurement set by gross measurement errors and/or topological errors.

iii) State Estimation Module

The state estimation program plays a key role in the network monitoring package. It processes raw telemetered data augmented by pseudo-measurements which are generated by the observability routine, and calculates an estimate of the state vector. The output of the state estimator also includes estimates of the measurement residuals, thus enabling detection and identification of bad data points. Two state estimators based on the augmented matrix method (SYSESTLS) and on the linear programming approach (SYSESTLP) are implemented.

iv) Operator Interface Module

Operator interface program (OPERATOR) enables the operator to select and implement controls using information provided by the monitoring programs and optimal valve control algorithm. It also allows modification of the set of measurement points, the changing of Gaussian noise parameters, and the simulation of the occurrence of bad data by corrupting the values of the telemeasurements.

6.2.3 Inter-process Communication between the Application Modules

Program organization and interprocess communication between the application modules in the original implementation of the water system monitoring and control software program is described in [Bargiela 88]. Each task communicates with others through shared memory areas with specified access privileges. The timing of task execution and synchronization has been achieved by reference to semaphores and event keys in shared data. The shared memory segments appear as common blocks within FORTRAN programs. The original implementation has been targeted for two 32-bit minicomputers Perkin Elmer and DEC-VAX, and consequently the process communication and synchronization have been implemented using the facilities of the OS32 and VMS operating system.

The water system monitoring and control software program was re-implemented for a modern computing platform consisting of a cluster of UNIX - based workstations connected via an *Ethernet* network. No changes were made to the algorithmic computing modules (the FADI application programming interface supports FORTRAN as well as C/C++ applications), but the interprocess communication and synchronisation was restructured to adjust to the facilities of the new environment. PVM was used for interprocess communication and synchronisation. A centralised control scheme was adopted. The central control

task holds the common areas and grants access rights through a *request-acknowledgment* message exchange with the requesting task on a FIFO basis.

6.2.4 The Test Environment

The hardware setup is similar to that used for the performance studies with synthetic applications in section 6.1.1 on page 100. The five water system monitoring and control tasks (central synchronisation and control, simulation, telemetry, estimation, and operator) were distributed between two diskless SPARCstation *IPC's* and a Sun SPARC_10 workstation. PVM-TCP/IP was used for communication over a 10 Mbit/sec Ethernet. FADI error-detection and fault management tasks were executed on a SPARC_20 central server.

6.2.5 Experimental Results

The water system monitoring and control application is both computation and communication-intensive. The sophisticated algorithms used for the determination of observability, state estimation and bad data detection average a CPU load of over *30%* during execution time, and for five iterations of estimation - telemetry - control continuing for approximately 2 minutes, the average message exchange rate was 42 messages per second. Therefore, this distributed system, in addition to being a functional on-line distributed control system, represents quite a challenge from the computation and communication overhead point of view. Figures 6-6 and 6-7 show the distributed application operating in FADI fault tolerant environment.

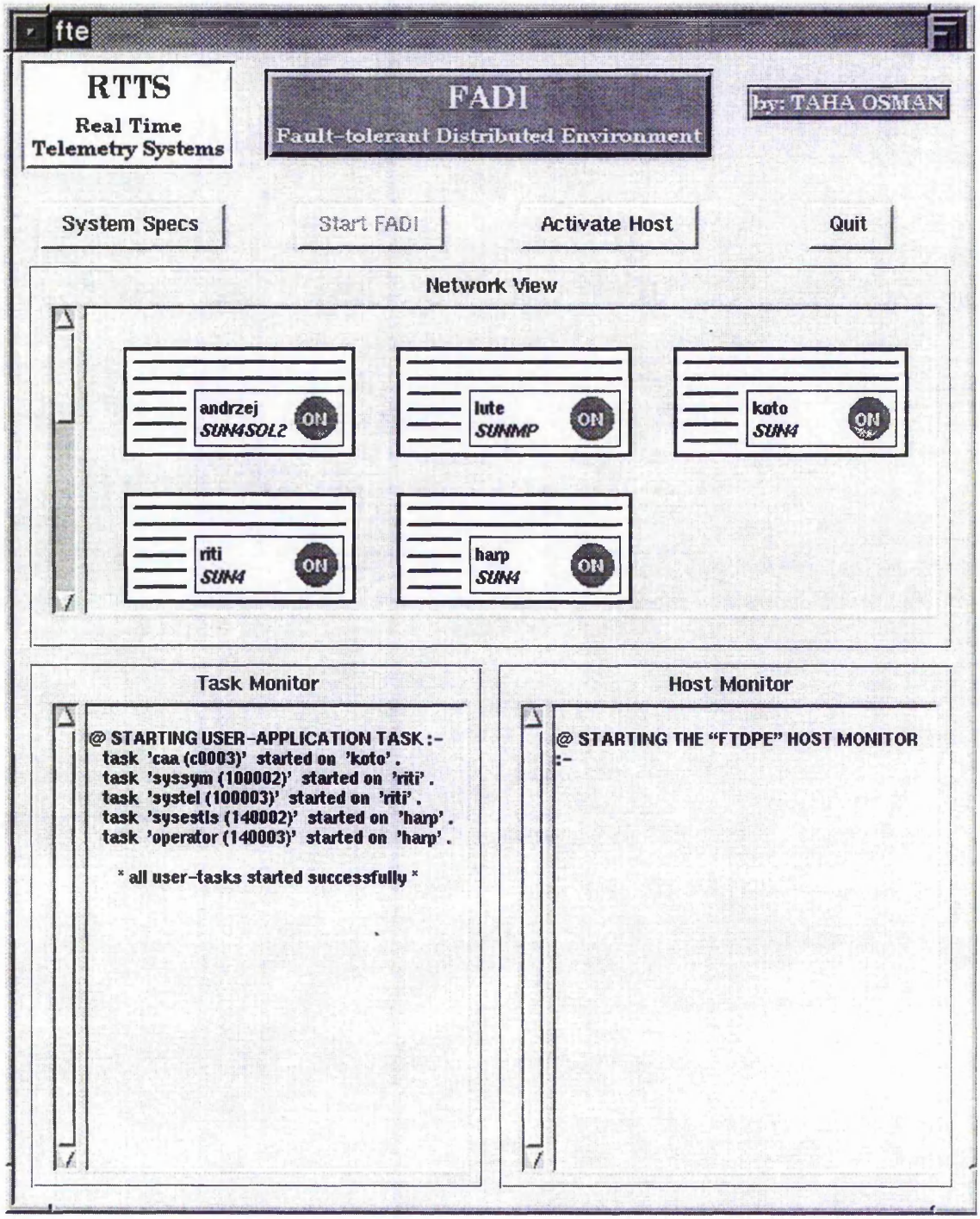


FIGURE 6-6 **FADI Running the Water Systems Monitoring and Control Application**

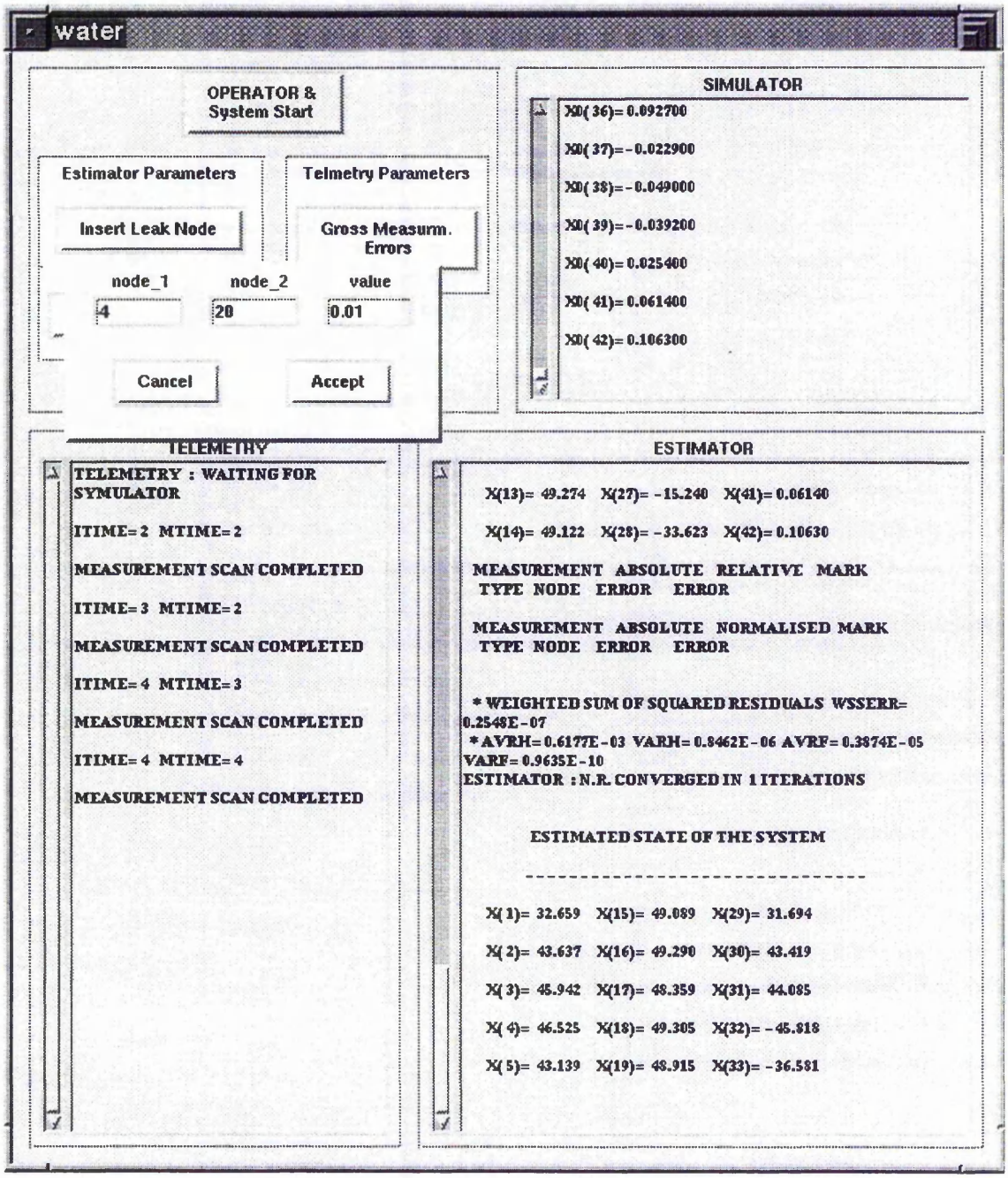


FIGURE 6-7 **The Water Systems Monitoring and Control Application**

Unlike synthetic applications, the application parameters that can affect the overhead of the fault-tolerant environment (e.g heap size, message rate, etc.) can not be regulated to test the system performance in various conditions. Hence the performance study is confined to examining the influence of varying the checkpointing interval on the application failure-free overhead, as shown in Figure 6-8.

The graphs indicate that *non-blocking checkpointing* significantly reduces the failure-free overhead of FADI (up to *80%* reduction with short checkpointing intervals). The reduction is more significant than that measured for non-interactive applications in section 4.7.2 on page 66 (a maximum reduction of *30%*) mainly because of the extra overhead of sending records of sent/received messages at each checkpoint to the checkpointing coordinator task.

The failure-free overhead at checkpointing intervals of half a minute is relatively high (although still within 10% of application running time for non-blocking checkpointing), but for the default measurement scan rate it is below *5%*. This emphasizes the importance of understanding the dynamics of the system when choosing checkpointing intervals. Clearly with the highest checkpointing rate the FADI system was interfering with the normal measurement collection cycle thus making it more difficult to maintain the checkpoints.

The Average recovery time for the water distribution networks application was *8* seconds.

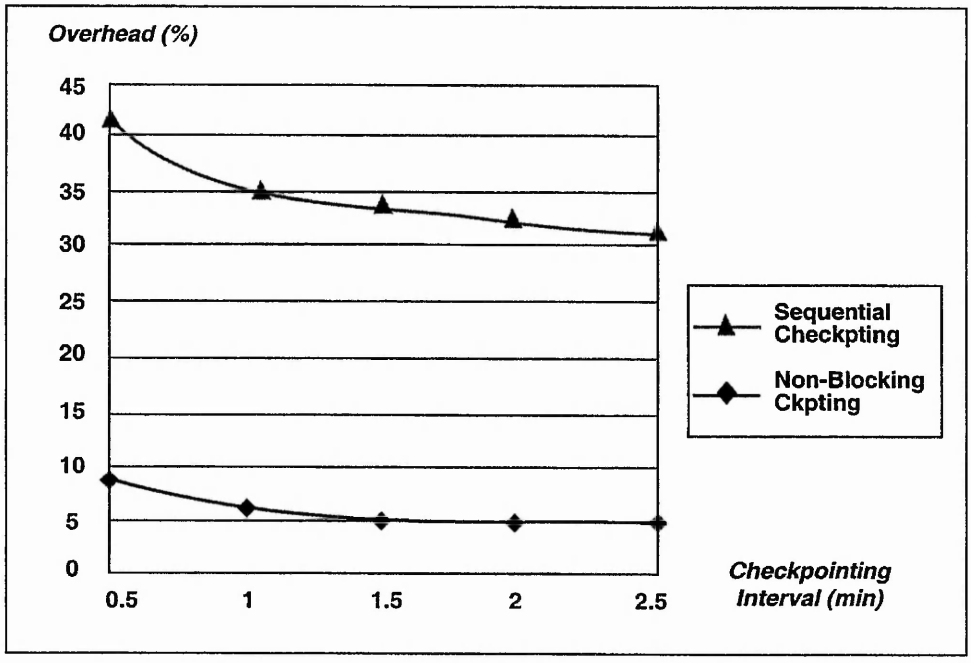


FIGURE 6-8 **Overhead of the Water-Systems Monitoring and Control Application**

6.2.6 Conclusions

This section presented a performance study of FADI by applying it to an advanced water systems monitoring and control software system. This application exemplifies a class of industrial systems where the FADI fault-tolerant environment can be utilised. The software system is inherently distributed and it needs to execute on continuous basis. It can tolerate a small delay in the operation of one or more of its tasks (while the are rolled back and restarted from checkpoints backup), but a complete halt of the system can lead to a critical failure of the decision-support system. FADI represents a low-cost and efficient alternative to hardware-redundancy based fault-tolerant computing environments to implement such systems.

The experimental results confirm that, due to the non-blocking checkpointing methodology, FADI achieves low overhead on the running time of applications.

The performance study has also highlighted the importance of a joint consideration of synthetic and real-life application when evaluating software environments such as FADI. In this context, we are confident to recommend the use of FADI to provide fault-tolerance for a broad class of computation-intensive distributed applications.

CHAPTER 7 FADI's Application Programming Interface (API)

This chapter addresses the user-interface issues associated with the FADI environment. The first section explains how to prepare the application programs for execution under FADI, highlighting the environment constraints and advantages. The second section introduces Tcl/Tk, the software package used to built FADI graphical user interface. It explains how the interface is used to input the distributed application specifications into FADI and to monitor both the progress of the application execution and the hardware platform it is running on.

7.1 Programmer's Guide to Using FADI

7.1.1 Pre-processing on the Application code

Some proprietary operating systems such as Sprite [Douglis 91] and KeyKOS [Landau 92] have built-in fault-tolerance mechanisms, where process models are carefully defined and implemented to accommodate checkpointing and migration. In these systems, fault-management is performed on the application executable (binary) code without the involvement of the user.

The FADI *generic* processing environment was built on the top of the UNIX general purpose operating system where the absence of kernel-based implementations for saving the process execution state, meant that minor alterations have to be made to the application source code to enable FADI fault-management procedures to perform recovery related procedures. In order to achieve user-transparency, a special pre-processor was designed to automatically link the application-source code with FADI fault-management libraries as illustrated in Figure 7-1.

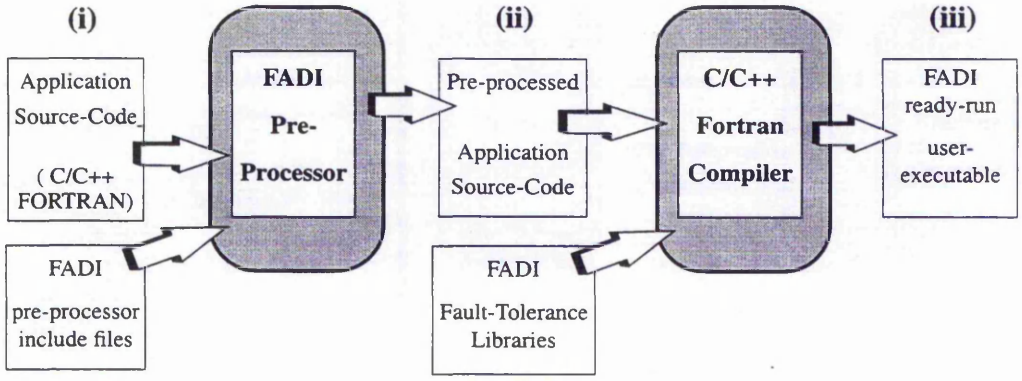


FIGURE 7-1 **Building FADI Applications**

7.1.2 Instructions to Building the Application Programs

The fault-tolerant environment was originally developed for application programs written in the *"C"* programming language. *"C"* is the most widely-spread programming language and most of the UNIX kernel code is written in *"C"*. The latter is important because it provides access to the UNIX-kernel internals that is needed to record the execution state of the process (take checkpoints). However, within the application domain of FADI (long-running, computation-intensive scientific/engineering applications) a significant proportion of programs were written in *FORTRAN*. Moreover, there is a wealth of algorithmic and mathematical libraries, that are extensively used in large scientific applications which are written in *FORTRAN* (e.g HARWELL, NAG, etc. [Harwell 97] [Kendall 95]). The PVM project [Geist 94] reached similar conclusions and they provide an analogous *FORTRAN* interface for their *"C"* libraries.

Hence, it was decided to provide both FORTRAN and C/C++ application programming interface (API) for FADI.

7.1.2.1 The *"C"* Interface

To build application programs written in *"C"* the user should follow the next steps:

1. Run FADI applications preprocessor from the command line:

```
fadi_pp   -c <file1.c, file2.cpp, ...>
```

The preprocessor edits the source code to perform four tasks:

i. scan the source code for the use of FADI-restrained UNIX and PVM system calls and procedures and inform the user accordingly.

ii. Change the *main* function call to *MAIN* to allow the checkpointing prologue to take over process execution upon start or rollback of the user application.

iii. Insert delimiters for unpacking PVM messages *(end_unpack)*. This is necessary to instruct the checkpointing prologue to unmask the checkpointing signal and complete processing the bookkeeping information of the received message.

iv. Include type definitions in the program code for augmenting certain PVM functions and UNIX system Calls to enable the checkpointing prologue to perform recovery related functions before calling the original procedures. Figure 7-2 presents an example of such augmentation for the PVM function performing the sending of interprocess messages:

2. If the first pass is successful, then it is recommended that the user copies the sample *Makefile* (Figure 7-3) from FADI source directory and follow the included instructions to modify the *Makefile* in order to compile the application programs. Next the user executes a UNIX script (provided with the FADI distribution) that verifies the programming environment (e.g existence of libraries, mode of environment variables, etc.), then triggers the compilation of the source file(s):

mfadi -c <main user-executable>

3. Once the application is successfully linked with FADI libraries, the user/programmer can use the GUI to run and monitor the distributed applications as detailed in the section 7.3.

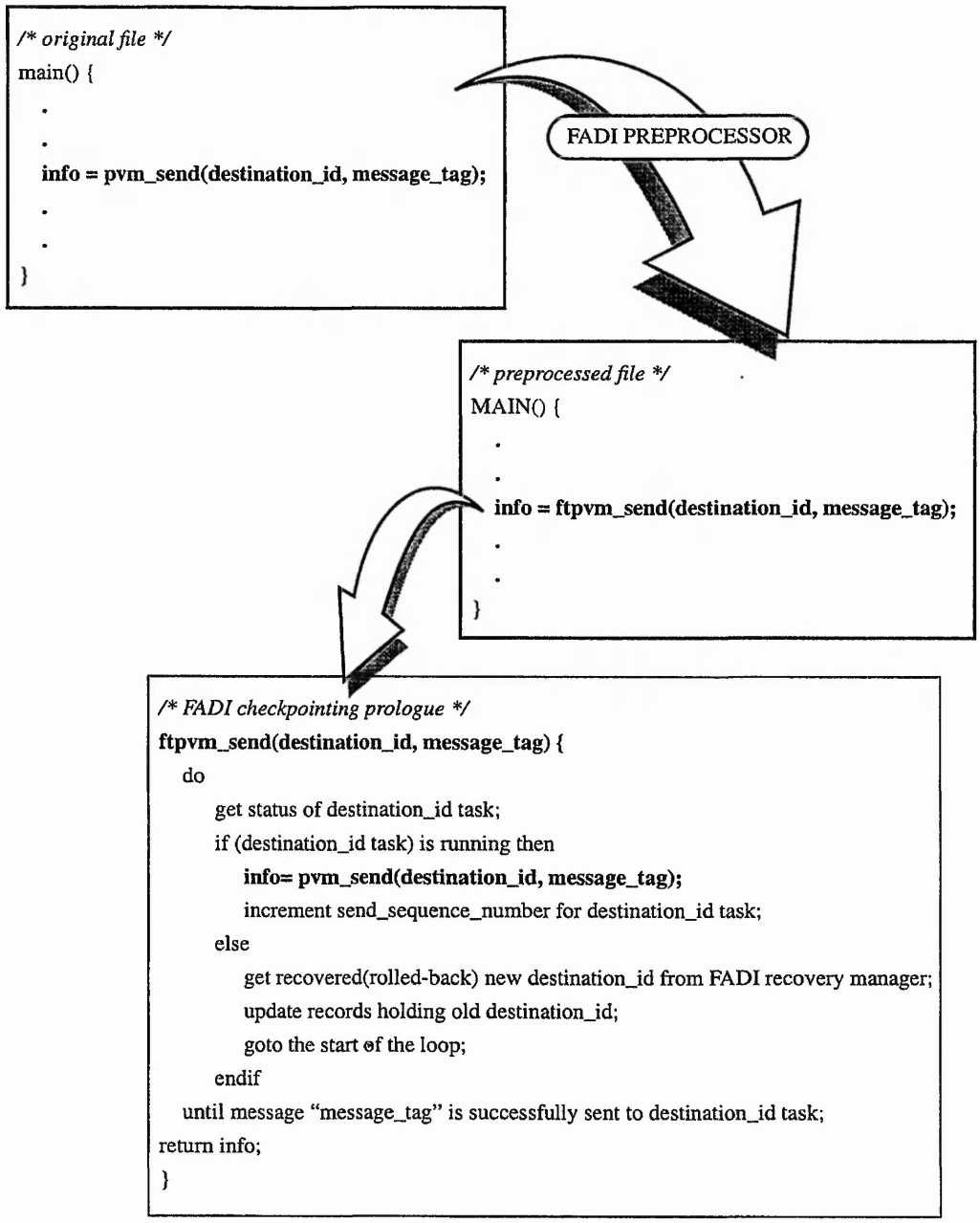


FIGURE 7-2 **Pre-Processing *"C"* Application Programs in FADI**

```
#Make file for FADI C/C++ Application Programs

# For source code compilation:
# 1) PVM must be installed on the system and the environment vars PVM_ROOT
#     and PVM_ARCH need to be set;
# 2) Enter the appropriate information when required (indicated by #++);

#++ Enter FADI Installation Directory
FDIR            =
#++ Enter Application Source Directory
ADIR            =
#++ Enter Executables Directory (default is applications directory)
XDIR            =               $(ADIR)

#++ Enter C/C++ compiler (default g++)
CC              =               g++

# Libraries & includes
#FADI
FADI_LIB=       -L$(FDIR)/lib -lfadi
FADI_INCLUDES=-I$(FDIR)/include
#PVM
PVM_LIB=        -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lpvm3 -lgpvm3 $(ARCHLIB)
PVM_INCLUDES=-I$(PVM_ROOT)/include
PVM_FLAGS=      $(ARCHCFLAGS)

#++ To change the compiler switches, for example to change from -O to -g, change the
#++ the following line:
CFLAGS = -O

CC_SWITCHES = $(CFLAGS)  $(PVM_FLAGS)

#++ Enter names of object files (user_main.o, file1.o, file2.o, etc.)
 OBJECTS        =               user_main.o, file1.o

# Main user-executable file name is extracted from the command line: "mfadi [exe_file]"
$(USER_MAIN): $(OBJECTS)
        $(CC) $(CFLAGS)  $(FADI_INCLUDES) $(PVM_INCLUDES) $(OBJECTS) $(PVM_LIB) \
        -lm -o $(USER_MAIN)
        cp $(USER_MAIN) $(XDIR)/

#++ Enter dependencies of the user_main and other object files (if any). Replace
#++ user_main.*, and file1.cpp, etc. with appropriate source file names.
user_main.o: $(ADIR)/user_main.c
        $(CC) $(CFLAGS) -c $(FADI_INCLUDES) $(PVM_INCLUDES) $(ADIR)/user_main.c

file1.o: $(ADIR)/file1.cpp
        $(CC) $(CFLAGS) -c $(FADI_INCLUDES) $(PVM_INCLUDES) $(ADIR)/file1.cpp
```

FIGURE 7-3 **Sample FADI "C" *Make* File**

7.1.2.2 The *FORTRAN* Interface

In order to avoid rewriting FADI fault-management procedures for the *FORTRAN* API, the skeleton of the application programs was retained in *"C"*. This skeleton performs all the required initialisation of FADI checkpointing and rollback procedures before calling *FORTRAN* main routines.

The following instructions should be followed to link *FORTRAN* programs to FADI:

1. The user should make sure that the *FORTRAN* main program has a *"PROGRAM"* and *"STOP"* statements. Some compilers such as the *SPARCworks F77* compiler [Sun 94] assume them by default, but they need to be declared explicitly because they are necessary for FADI preprocessor to identify main *FORTRAN* application modules.
2. Run FADI preprocessor from the command line:

   fadi_pp -f <file1.f, file2.f, ...>

the preprocessor edits the source to perform the following tasks:

   i. scan the source code for the use of FADI-restrained UNIX and PVM system calls and procedures and inform the user accordingly.
   ii. edit *the FORTRAN* text in order to change the *FORTRAN* declaration for main program to a subroutine that can be called from within a *"C"* code. Next the preprocessor augments some of the *FORTRAN* PVM and UNIX procedures to cross-call their corresponding fault-tolerant versions implemented in *"C"*. Diagram 7-4 illustrates the process of building FORTRAN applications for use in the FADI environment.

3. If the first pass is successful, then it is recommended that the user copies the sample FORTRAN *Makefile* (Figure 7-5) from FADI source directory and follow the included instructions to modify the *Makefile* in order to compile the application programs. Next the user executes a UNIX script (provided with the FADI distribution) that verifies the programming environment (e.g existence of libraries, mode of environment variables, etc.), then triggers the compilation of the source file(s):

   mfadi -f <main user-executable>

4. Once the application is successfully linked with FADI libraries, the user/programmer can use the GUI to run and monitor the distributed applications as detailed in the next section.

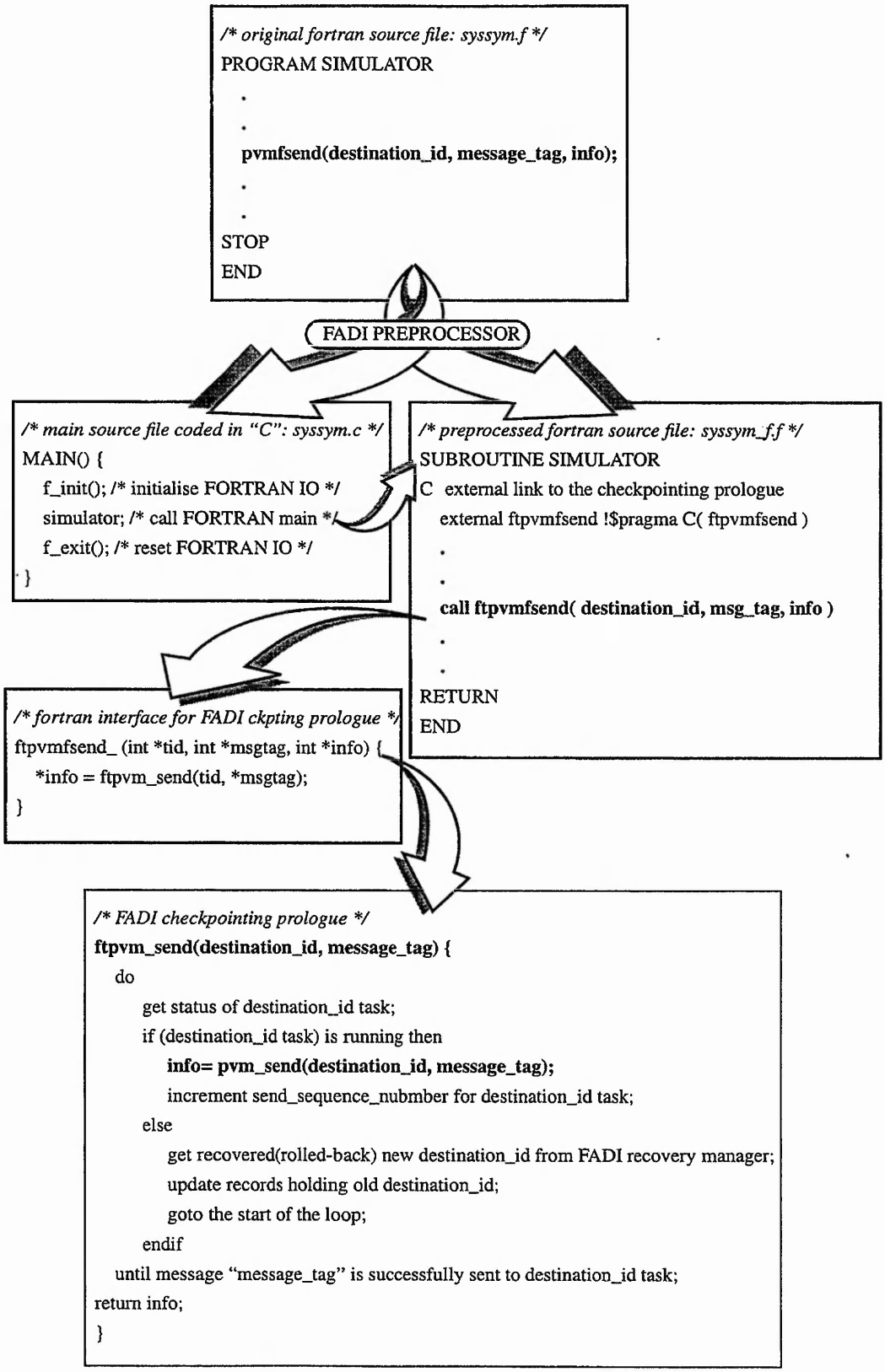


FIGURE 7-4 **Pre-processing *FORTRAN* Application Programs in FADI**

```
#Make file for FADI FORTRAN Application Programs

# For source code compilation:
# 1) PVM must be installed on the system and the environment vars PVM_ROOT
#    and PVM_ARCH need to be set;
# 2) Enter the appropriate information when required (indicated by #++);

#++ Enter FADI Installation Directory
FDIR=
#++ Enter Application Source Directory
ADIR=
#++ Enter Executables Directory (default is applications directory)
XDIR=           $(ADIR)

#++ Enter C/C++ compiler (default g++)
CC=             g++
#++ Enter FORTRAN compiler (default g77)
F77=            g77 -f_init-local-zero      #g77 does not set uninitialised local vars to 0

# Libraries & includes
#FADI
FADI_LIB=       -L$(FDIR)/lib -lfadi
FADI_INCLUDES=-I$(FDIR)/include
#PVM
PVM_LIB=        -L$(PVM_ROOT)/lib/$(PVM_ARCH) -lfpvm3 -lpvm3 -lgpvm3 $(ARCHLIB)
PVM_INCLUDES=-I$(PVM_ROOT)/include
PVM_FLAGS=   $(ARCHCFLAGS)

#++ To change the compiler switches, for example to change from -O to -g, change the
#++ the following line:
CFLAGS = -O

CC_SWITCHES =$(CFLAGS)  $(PVM_FLAGS)

#++ Enter names of object files (user_main.o, file1.o, user_main_f.o, file2.o, etc.)
 OBJECTS=       fpp.o, user_main.o, user_main_f.o, file2.o, ...

# Main user-executable file name is extracted from the command line: "mfadi [user_main]"
$(USER_MAIN): $(OBJECTS)
     $(CC) $(CFLAGS)  $(FADI_INCLUDES) $(PVM_INCLUDES) $(OBJECTS) \
     $(PVM_LIB) -lm -o $(USER_MAIN)
     cp $(USER_MAIN) $(XDIR)/

#++ Enter dependencies of the user_main and other object files (if any). Replace
#++ user_main.*, and file2.f, etc. with appropriate source file names.
user_main.o: $(ADIR)/user_main.c
     $(CC) $(CFLAGS) -c $(FADI_INCLUDES) $(PVM_INCLUDES) $(ADIR)/user_main.c

user_main_f.o: $(ADIR)/user_main_f .f fpp_f.h
     $(F77) $(CFLAGS) -c $(FADI_INCLUDES) $(PVM_INCLUDES) $(ADIR)/$user_main_f.f

file2.o: $(ADIR)/file2.cpp
     $(CC) $(CFLAGS) -c $(FADI_INCLUDES) $(PVM_INCLUDES) $(ADIR)/file2.cpp
# FADI preprocessor file, no change necessary
fpp.o: fpp.c
     $(CC) $(CFLAGS) -c $(PVM_FLAGS) $(PVM_INCLUDES) fpp.c
```

FIGURE 7-5 **Sample FADI "FORTRAN" *Make* File**

7.2 The Tcl and the Tk GUI Development Toolkit

Tcl and Tk are software packages that provide programming system for developing and using graphical user interface applications. Tcl is a simple scripting language for controlling and extending applications; its name stands for "tool command language" [Ousterhout 94]. Tcl provides generic programming facilities, such as variables and loop procedures, that are useful for a variety of applications. Furthermore, TCL is *embeddable.* Its interpreter is a library of *"C"* procedures that can easily be incorporated into applications, and each application can extend the core Tcl features with additional commands for that application.

One of the most useful extensions of Tcl is Tk, which is a toolkit for the X Window System [Nye 90]. Tk extends the core Tcl facilities with commands for building user interfaces, so that one can construct Motif-like user interfaces by writing Tcl scripts instead of *"C"* code. Tcl/Tk is already utilised in a variety of serious graphical and communication software packages, the IMIS system [Thiran 96] uses Tcl/Tk to implement tools for telediagnoses and 3D medical image processing.

Together Tcl and Tk has many advantages to the application developers and users:

- The main benefit of programming in Tcl/Tk is rapid application development. Many interesting GUI applications can be written as scripts, using a windowing shell called *wish.* This allows to program at much higher level than in *"C"* or *C++*, and Tk hides many of the details that *"C"* programmers must address. Compared to toolkits where programming is in *"C"*, such as the Motif toolkit, there is much less to learn in order to use Tcl and Tk and much less code to write. New Tcl/Tk users can often create interesting user interfaces after just a few hours of learning, and many people have reported reductions in the code size and development time when they switched from other toolkits to Tcl/Tk.

- Tcl is an interpreted language. When using Tcl applications such as *wish*, one can generate and execute new scripts on the fly without re-compiling or restarting the application. For example to change the font of a text frame tagged "T_Frame" while it is active, merely one line of script has to be executed from within the *wish* shell:

  " T_Frame configure -font <*new_font*> "

  This is particularly useful at the prototyping stage of the GUI. The GUI developer can make changes on the fly to suit the customer requirements, get instant feedback, make few more modifications, etc.

- Tcl makes it easy for applications to have powerful script languages. To create a new application, all that needs to be done is to implement a few new Tcl commands that provide the basic features of the application. Then these commands can be linked with the Tcl library to produce a full-function scripting language that includes both the commands provided by Tcl (called the *Tcl core*) and those written by the programmer. This allowed Tcl to include many different library packages, each of which provides an interesting set of commands as in Figure 7-6. Tk is one example of such library.

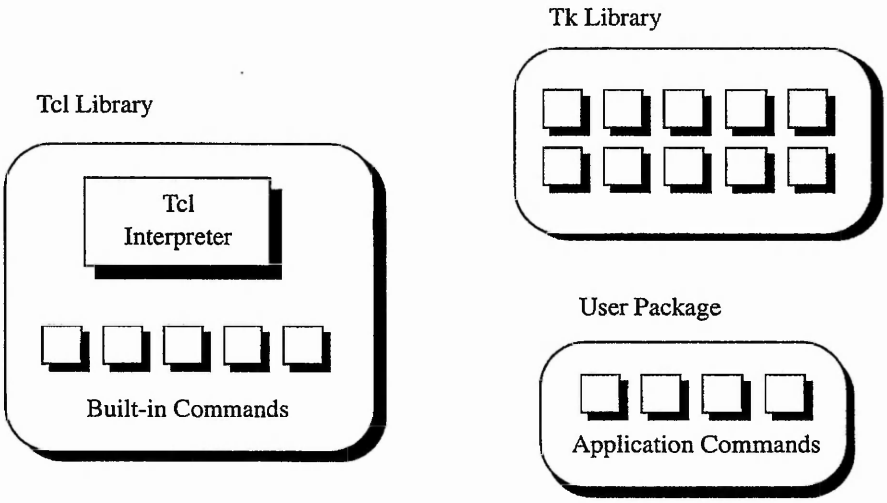


FIGURE 7-6 **Structure of a Tcl Application**

- Tcl scripts can also be used as a communication mechanism to allow different applications to work together. For example, any windowing application based on Tk can send a Tcl script to any other Tk application to be executed there.

Besides the general benefits to GUI developers, the Tcl/Tk Toolkit has features that are specifically advantageous to FADI's behavioural model:

- Tcl/Tk is built on the top of the X11 interactive window system. It provides a network transparent, graphics operating environment [Jones 90]. Most UNIX-based computer systems - which is FADI's hardware platform, utilise X11 to build graphical front-end interface to their operating systems.
- Tk offers a wide range of graphic widgets: frames, labels, buttons, radio-buttons, scroll lists, text windows, menus, text entries, canvas drawing, and many more. i.e ready-made graphical tools to support FADI user-input and system monitoring tasks.
- The Tcl/Tk Toolkit provides a high-level bidirectional interface to programming modules written in C. The initialisation script executed when the application starts should contain declarations of *"C"* procedures which are called directly from a Tcl script. To execute Tcl scripts from within a *"C"* procedure, information about the Tcl application Tcl interpreter has to be embodied in the procedure. No further changes are required to the normal *"C"* code. Hence, FADI GUI tools can be written in Tcl, because scripts are easier to write, they can be modified dynamically, and they can be debugged more quickly because there is no need for re-compilation after each bug fix. FADI error detection and recovery modules are implemented in *"C"* because it is faster, and has access to low-level OS facilities (e.g network socket operations) that are not available to Tcl scripts.
- Once the initialisation stage is complete, the Tcl application enters an event loop to wait for user-interactions. Whenever an interesting event occurs, such as the user invoking a menu entry, moving the mouse, or a call from a *"C"* procedure, a Tcl script is invoked to process that event. This perfectly complies with the distributed behavioural model of FADI central processes which is constructed in an event-processing fashion - that is message driven.
- The interaction between the application programs and FADI Tcl/Tk GUI is implemented by exchanging interprocess messages via the PVM message passing interface. This made linking *FORTRAN* applications to the GUI effortless, without the need to cross-link the *"C"* GUI code to the *FORTRAN* application programs.

7.3 The Integrated Input and Monitoring Environment

The purpose of FADI GUI is to provide the application programmer with a user-friendly form for inputting the specifications of the distributed system, and to aid in on-line monitoring of the application tasks and the underlying hardware platform (computer nodes). Following is a description of the interface:

‡ The System Specs submenu contains entry forms for the input of the distributed system specification as shown in Figure 7-7:

- Application Checkpointing Interval;
- Specifications of the Application Tasks:
  - Task Name (executable file);
  - Spawn In (the task host): Host (by host name);
    Arch (by CPU/OS architecture);
    Default (Spawn on any available host).

‡ The Network View window displays icons representing the distributed system computer nodes. At the start of the application, the distributed system configuration is automatically detected and displayed in the icon form shown in Figure 7-8. It comprises the node name, its CPU/OS architecture, and its current state: On - operative, Off - crashed.

‡ Start FADI button launches the process allocation, error detection, and the checkpointing/rollback tasks.

‡ Activate Host invokes an entry form that allows the user to manually add new hosts to the distributed system configuration.

‡ Host Monitor and Task Monitor text windows display information about the operation status of FADI during the current run of the distributed application such as: detected hardware failures, failed-user tasks, the latency of the errors, process allocation and migration information, etc.

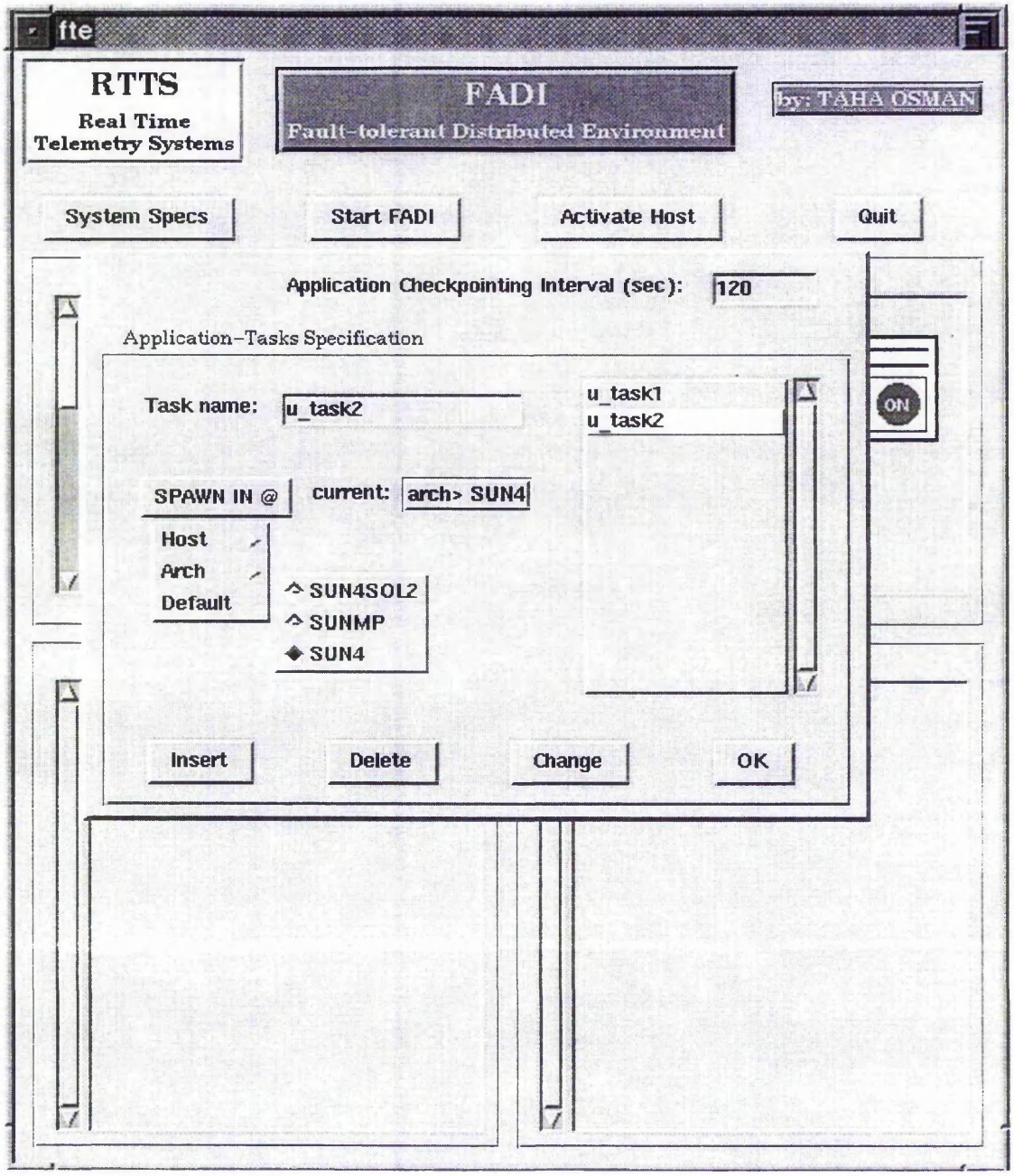


FIGURE 7-7 **Entering The Distributed Application Specifications**

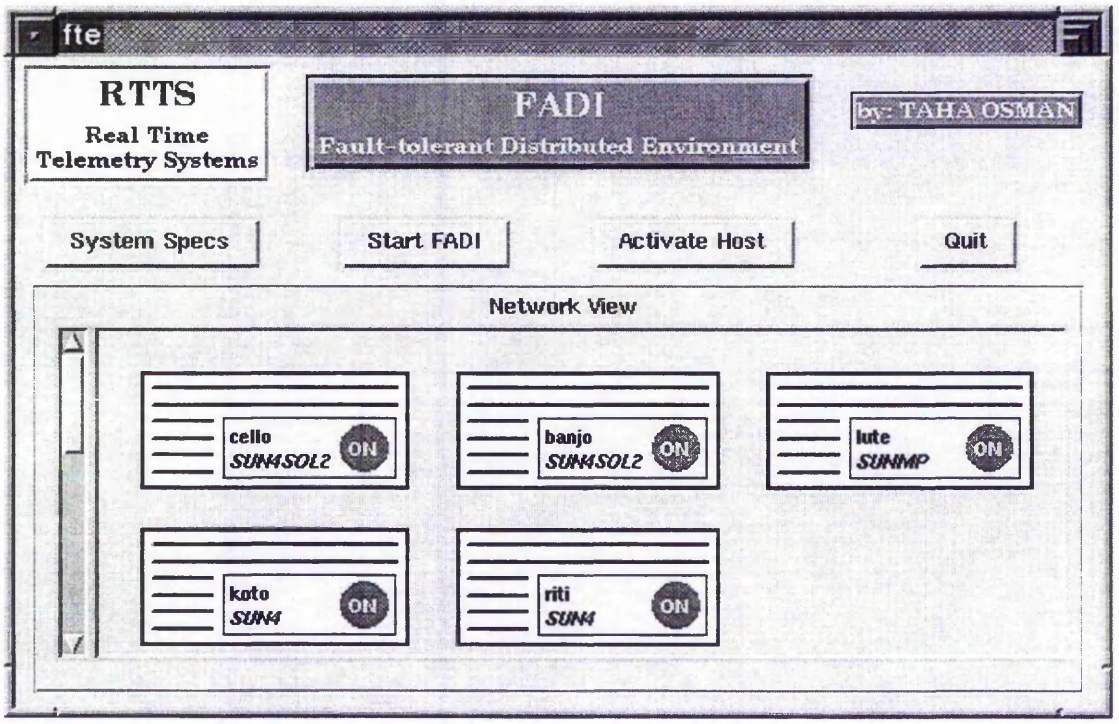


FIGURE 7-8 **Distributed System Configuration**

Figure 7-9 represents a snapshot of the system operation. The configuration of the distributed system consists of five processing nodes that have three different CPU/OS architecture.

An artificial application is initiated under FADI. The application consists of two user-application tasks (see process allocation information in the Task Monitor window). One task was allowed to run to successful termination, while an engineered hardware fault interrupted the execution of the other.

The hardware fault was simulated by powering down the host of u_task2 "riti". FADI error detection mechanism detected the fault and initiated the migration of u_task2 to host "koto" - that has the same CPU/OS architecture to "riti", when it is rolled back to the most recent saved checkpoint and continues execution until its successful termination.

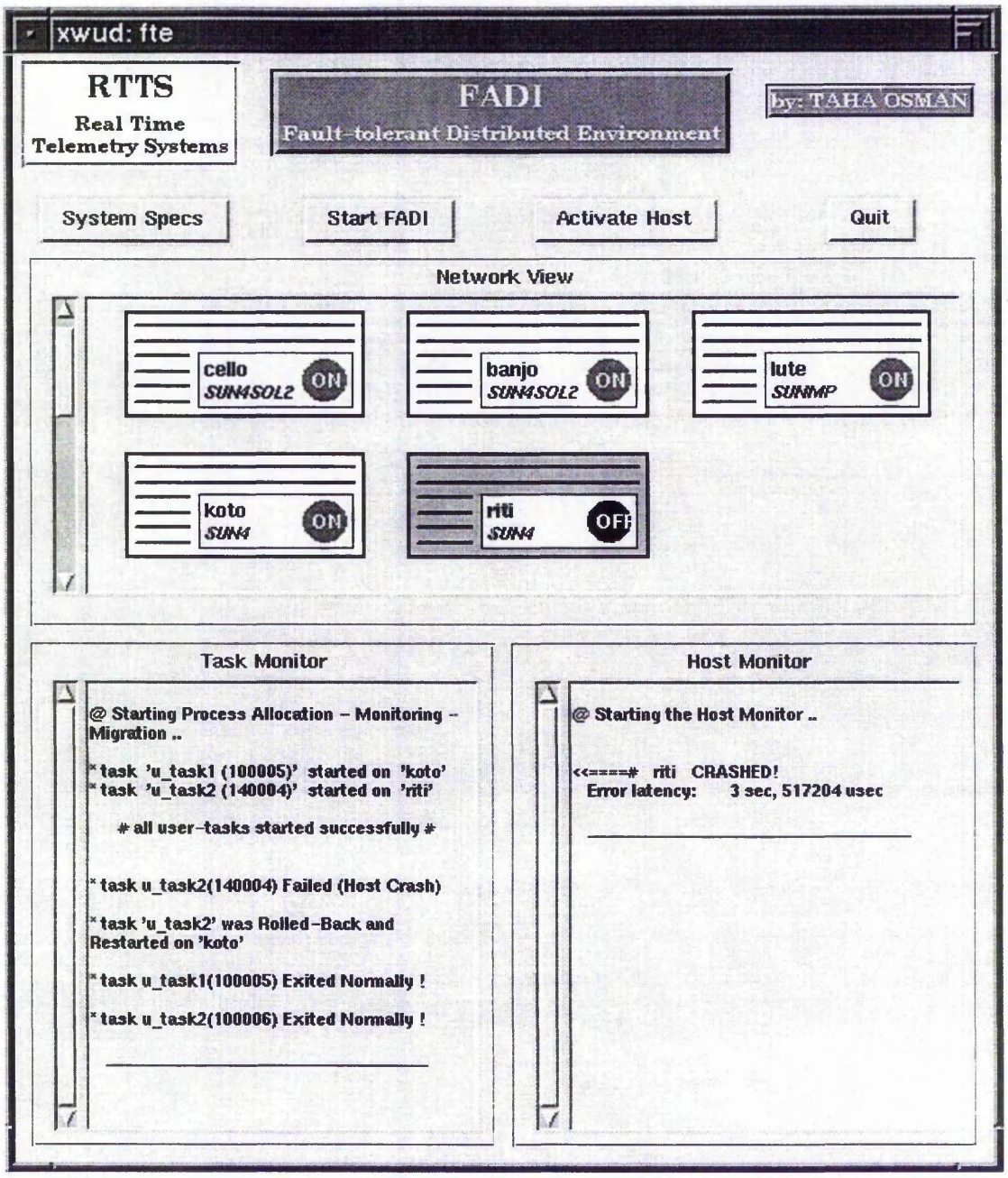


FIGURE 7-9 **A Snap-Shot of FADI in Operation**

CHAPTER 8 Conclusions and Future Work

This chapter reviews the original contributions of this research, discusses the limitations of the fault-tolerant distributed environment limitations and highlights lessons learnt throughout the development of the system. Based on the above, suggestions for future research are proposed.

8.1 Conclusions

The first stage of this research was explorative by nature. A broad spectrum of fault-tolerant systems has been examined, their characteristic features and relative strength have been compared. This stage served to identify the need for a new fault-tolerant system that was particularly well suited for the execution of computation-intensive applications on a network of workstations. The aim of the research was to develop an integrated environment that encompasses all aspects of modern fault-tolerant distributed computing: automatic remote process allocation, user-transparent detection of hardware errors, and a technique to recover executing distributed user processes from these errors. This fault tolerant environment can be used to execute long-running number-crunching scientific applications, where a fault occurring on a single computing node could incur the waste of hours or even days of computations performed in the distributed system. Another potential application area are on-line distributed programs in industrial systems, where a complete halt of the system because of a permanent or a transient hardware fault cannot be tolerated.

Based on the investigation of the variability of the round-trip time of the communication network and its effect on the error latency and the accuracy of fault diagnoses, this research has recommended a formula for the dynamic calculation of the acknowledgment time-out (resulting in more accurate fault diagnoses), and the optimisation of the associated parameters for error latency.

Various fault-tolerant solutions have been considered, and the checkpointing and rollback was adopted as the backup and recovery methodology for distributed user-applications running in FADI (**FA**ult Tolerant **DI**stributed Environment) in preference to process

replication fault-tolerance methods. The main reason for this decision is the avoidance of the heavy cost of the redundant hardware needed for the execution of the replicas.

A novel non-blocking checkpointing algorithm has been proposed. With this method, an exact copy(thread) of the checkpointed program is forked, which performs all the checkpointing routines without suspending the execution of the application code thus significantly reducing the checkpointing overhead. A problem overlooked by many checkpointing algorithms is the rollback of user files open at checkpoint time. For instance Condor bytestream checkpointing successfully rolls-back user files only if no modifications were made to them since the last checkpoint was taken. The non-blocking checkpointing algorithm introduces a module that uses a combination of copy-shadowing and file size bookkeeping to undo modifications made by append or update to user-files upon rollbacks.

Experimental results demonstrated that the performance of the developed checkpointing protocol compares well with results published of similar work in [Sens 93] and [Plank 94].

A generalisation of the checkpointing algorithm to cater for interactive (message passing) applications has been developed and its correctness has been theoretically proven. The algorithm is a hybrid of consistent checkpointing, with its low failure-free overhead, and selective logging of messages that cross the recovery line to avoid blocking the application process during the checkpointing protocol. The low failure-free overhead is at the expense of a longer rollback time which is deemed to be admissible because of the extended execution time of the targeted applications.

The main contribution of this new technique is that in contrast with similar algorithms, it is tolerant to errors occurring whilst messages are in transit, i.e messages are delivered to the destination (queued at message passing daemon or transport protocol thread), but not yet requested (consumed) by the receiving task. The algorithm requires only one global checkpoint to be recorded in a stable storage and avoids multiple rollbacks (domino effect).

This research has advanced the understanding of the overheads associated with fault-tolerant execution of applications on distributed computing resources. The theoretical considerations has been backed-up by extensive experimentation with synthetic applications using the FADI environment. The gained results showed that the system compares favourably with similar fault tolerant environments and exhibits low-overhead even with a over-estimated process memory requirements and inter-process message exchange rate. FADI

has also been subjected to a performance study while running a real engineering application: a *computer-assisted control of water distribution networks system* developed at the The Nottingham Trent University. The test-runs of this application have confirmed the practicality of employing the FADI fault-tolerant environment to reliably execute realistic distributed real-time telemetry applications.

On the practical level, this research resulted in the development of a portable fault-tolerant environment that has an application programming interface to FORTRAN and C/C++ applications. The automated pre-processing of the application and the graphical user-interface to FADI make it a user-transparent and convenient distributed processing software tool.

Due to the incompatibility of the UNIX -FADI OS platform- systems (discrepancies in the a.out format, management of the process's address space by the MMU, etc.), rolling-back (restarting) the process from the previous checkpoint on another computer system is limited to hosts with similar Operating System architecture to the failed one. FADI also does not consider the checkpointing of processes that were created dynamically - e.g using UNIX fork() or exec(). The absence of built-in fault-tolerance tools (such as in SPRITE, KeyKOS) in the UNIX OS, means that the application programs need to be staticaly linked to FADI fault-management libraries to allow for process checkpointing and recovery.

8.2 Areas of Future Research

1. Traditionally, communication among processes in a distributed system is based on copying data and using a *message-passing* model (in FADI, PVM was adopted as the message passing interface). An alternative *shared-memory* model can be used to provide application programs with a shared address space that can be used in the same way as local memory, for *read* and *write* operations. Morin [Morin 97] claims that the primary advantage of shared-memory for the application programmer is that the model for using shared data is identical to that used when writing sequential programs, allowing a natural transition from sequential to distributed applications. As an extension of the current research, It would be interesting to investigate the advantages and disadvantages of a shared-memory implementation for the FADI distributed communication model from

the point of view of the ease of development of distributed applications, the efficiency of implementation and the fault-management overhead of this technique compared to that of message-passing implementation.

2. Currently FADI supports the recovery of the application processes on hosts that have similar OS architecture to the original host where the checkpoints were taken. This is due to the incompatibility of the format of the executable processes on different operating systems. A fruitful area of research might be to explore the generalisation of the FADI environment for an architecture-independent programming platform as offered by *Java*. *Java* is an object-oriented programming language that is portable across multiple machine architectures, operating systems and GUIs [Manger 96].
   Extending the FADI API for *Java* programs will allow to take checkpoints in a machine independent format, which means that the application programs can be checkpointed and restored on heterogeneous systems.

3. A natural expansion of the FADI project can be in the area of *Distributed Object Computing*. CORBA (Common Request Object Broker Architectures) is an open standard that is considered as the leading edge in distributed object computing technology. It defines a set of components that allow client applications to invoke operations in remote object implementations. CORBA enhances application flexibility and portability by automating many common development tasks such as object registration, location and activation; demultiplexing; and operation dispatching [Maffeis 97].

   To enable the adoption of such technology as the backbone for the next-generation of distributed object services (such as electronic commerce, personal communication systems, satellite surveillance systems, distributed medical imaging, real-time data feeds and flight reservation systems), the technology must be reliable and highly available. Neither the CORBA standard nor conventional implementations of CORBA directly address the problems of reliability. Maffies and Schmidt presented an extension to the Object Management Architecture in CORBA based on the virtual synchrony model that improves support for reliability by means of expensive hardware replication [Schmidt 97]. For application that do not have stringent real-time constraints (e.g flight reservation systems and electronic commerce), FADI can provide an efficient and cost-effective reliability model for building fault-tolerance into CORBA. On an abstract level the CORBA architecture is similar to FADI: the distributed objects resemble FADI's

remotely allocated application processes and the method invocations (returned values) on these objects are similar to the message-passing communication system in FADI. The FADI non-blocking checkpointing technology can be utilised to save the execution state of active (remote) objects and the reliable message-passing algorithm can be used to safe-guard the delivery of remote method invocations on the distributed objects.

Plans for Further work in paragraphs 2. and 3. can be combined to develop an open reliable object distributed computing environment (Figure 8-1). *Java* offers the flexibility of architecture-independent programming, CORBA delivers network-transparent distributed object infrastructure, and FADI deals with the reliability issues. FADI's Tcl/Tk GUI could usefully be re-written in *Java* to build a portable FADI client that can be executed on any user's desktop workstation or PC. This will allow users (whether remote or local) to use FADI services from any web-enabled computer system.

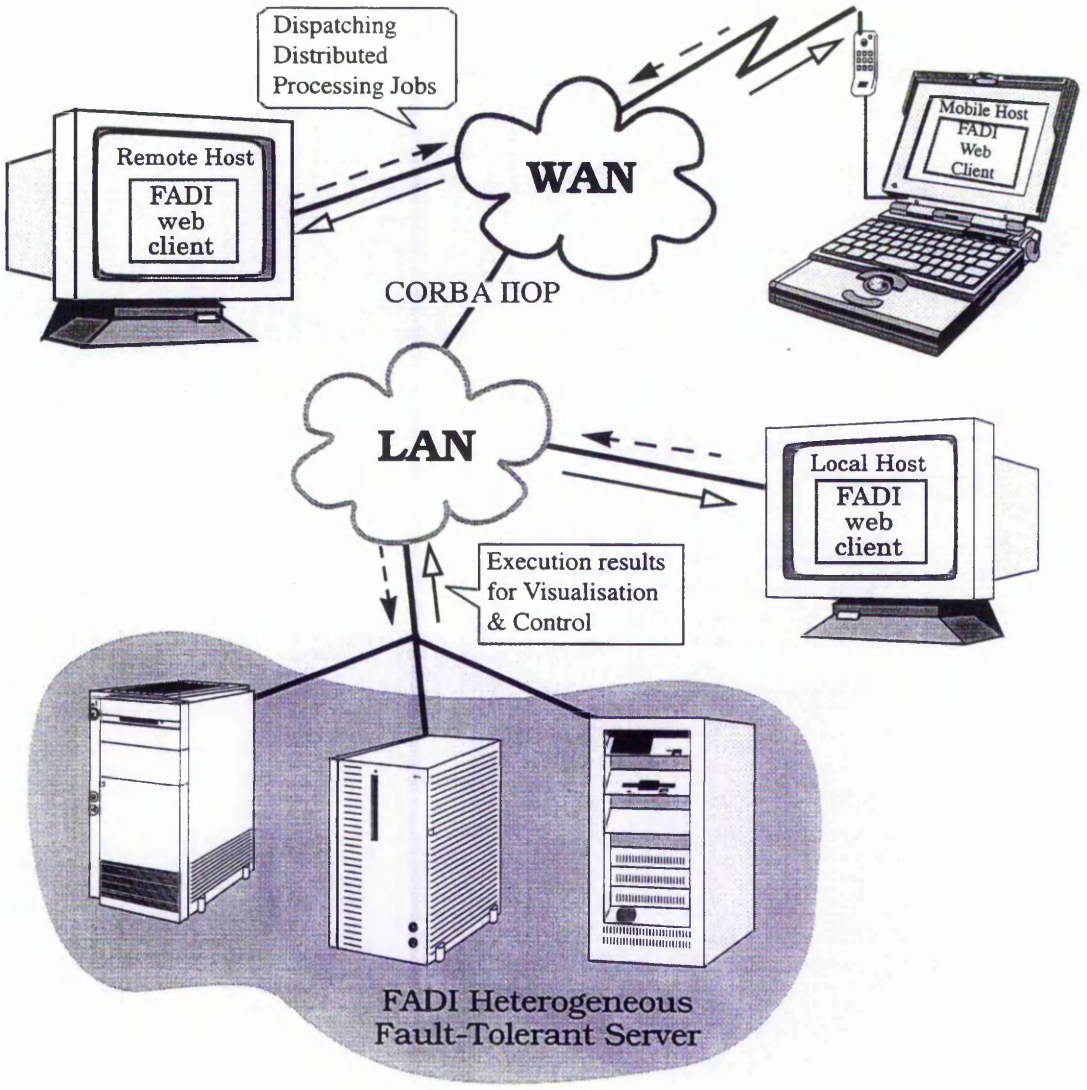


FIGURE 8-1 **Open Reliable Distributed Computing**

References

[Al-Dabass 81]

[1] D. Al-Dabass. "Partitioning and Synchronization Concepts for Computing Dynamical Systems Algorithms on Distributed Computer Control Networks", *IFAC 3rd International Workshop on DCCS, Beijing, China,* August 1981.

[Altmann 95]

[2] J. Altman, T. Bartha and A. Pataricza. "On Integrating Error Detection into a Fault Diagnosis Algorithm for Massively Parallel Computers". *Proceedings of the International Computer Performance Symposium*, p. 154-164, April 1995.

[Anderson 81]

[3] T. Anderson and P.A. Lee. "Fault Tolerance. Principles and Practice", *Prentice Hall,* 1981.

[Appel 92]

[4] B. Appel *et al.* "Implications of Fault Management and Replica Determinism on the Real-Time Execution Scheme of VOTRICS".

[Attig 93]

[5] N. Attig and V. Sander. "Automatic Checkpointing of NQS Batch Jobs on CRAY UNICOS Systems", *Proceedings of Cray User Group Meeting (Spring), KFA-ZAM-IB-9303, Montreux,* March 1993.

[Bacon 93]

[6] J. Bacon. "Concurrent Systems. "An Integrated Approach to Operating Systems, Database, and Distributed Systems", *Addison-Wesley*, 1993.

[Bargiela 84]

[7] A. Bargiela. "On-Line Monitoring of Water Distribution Networks", *PhD Thesis, Faculty of Science, University of Durham,* May 1984.

[Bargiela 88]

[8] A. Bargiela and D. Al-Dabass. "A Simulated Real-Time Environment for Verification of Advanced Water Network Control Algorithms", *Systems Science Journal, Vol. 14, No. 3,* 1988.

[Bargiela 93]

[9] A. Bargiela, A. Argile, and J. Hartley. "Parallel Processing for Probabilistic Decision Support in Water Distribution Systems", *SERC Seminar, Burnel University,* September 1993.

[Bargiela 95]

[10] A. Bargiela and J. Hartley. "Parallel Simulation of Large Scale Water Distribution Systems", *Proceedings of the 9th European Simulation Multiconference,* 1995.

[Bauch 92]

[11] A. Bauch, B. Bieker, and E. Maehle. "Backward Error Recovery in the Dynamical Reconfigurable Multiprocessor System DAMP", *Workshop on Fault-Tolerant Parallel and Distributed Systems,* Amherst, MA, p. 36-43, July 1992

[Bernard 94]

[12] G. Bernard and D. Conan. "Flexible Checkpointing and Efficient Roll-Recovery for Distributed Computing", *Proceedings of SUUG International Conference on Open Systems: Solution for Open World,* p. 25-29, April 1994.

[Bhargava 88]

[13] B. Bhargava and S. Lian. "Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems", *Proc. 7th Symposium on Reliable Distributed Systems,* p. 3-12, 1988

[Birman 94]

[14] K. P. Birman and R. V. Renesse. "Reliable Distributed Computing with the ISIS Toolkit", *IEEE Computer Society Press,* 1994.

[Borg 83]

[15] A. Borg, J. Baumach and S. Glazer, "A Message System Supporting Fault-Tolerance", *Proc. 9th Symp. on Operating Systems Principles*, p. 90-99, Oct. 1983.

[Briker 91]

[16] A. Briker, M. Litzkow and M. Livny. "CONDOR Technical Summary", *University of Wisconsin - Madison, Ver 4.1b,* Sept. 1991.

[Butler 94]

[17] R. Butler and E. Lusk. "Monitors, Messages and Clusters: The P4 Parallel Programming System". *Parallel Computing,* Vol. 20(4), p. 547-564, April 1994.

[Carriero 89]

[18] N. Carriero and D. Gelernter. "LINDA in Context", *Communications of the ACM,* 32(4), p. 444-458, April 1989.

[Chadna 96]

[19] H. Chadnaand J. Baugh. "Network-Distributed finite element analysis", *Advances in Engineering Software,* Vol. 25, Iss. 2-3, p. 267-280, April 1996.

[Chandy 72]

[20] K. Chandy and V Ramamoorthy. "Rollback and Recovery strategies for computer programs", *IEEE Transactions on Computers,* Vol. 22, p. 546-556, June 1972.

[Chandy 85]

[21] K. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer systems,* Vol. 3(1), p. 63-75, Feb. 1985.

[Chen 96]

[22] P. Chen. "Climate and Weather Simulations and Data Visualisation Using a Supercomputer, Workstations and Minicomputers", *Proceedings of the SPIE - The International Society for Optical Engineering,* Vol. 2656, p. 254-264, 1996.

[Chiu 94]

[23] J Chiu and G. Chiu. "Process Replication Technique for Fault-Tolerance and Performance Improvement in Distributed Computing Systems".

[Cin 93]

[24] M. Cin *et al.* "Error Detection Mechanisms for Massively Parallel Multiprocessors", *Proc. of Euromicro Workshop on Parallel and Distributed Processing,* p. 401-408, Jan. 1993.

[Clark 95]

[25] J. Clark and D. Pradhan. "Fault Injection. A Method for Validating Computer-system Dependability", *Computer,* June 1995.

[Comer 93]

[26] D. Comer and D. Stevens. "Internetworking with TCP/IP. Volume III. Client - Server Programming and Applications", *Prentice Hall,* 1993.

[Cooper 85]

[27] E. C. Cooper. "Replicated Distributed Programs", *In Proc. of the 10th ACM Symposium on Operating Systems Principles,* p. 206-211, December 1985.

[Davoli 96]

[28] R. Davoli *et al.* "Parallel Computing in Networks of Workstations with Paralex", *IEEE Transactions on Parallel and Distributed Systems.* Vol. 7, No 4, April 1996.

[Deconinc 93]

[29] G. Deconink *et al.* "Survey of Checkpointing and Rollback Techniques". *ESAT-ACCA Laboratory, Katholieke Universitei Leuven, Belgium,* 1993.

[Digital 97]

[30] Digital Interactive Solutions Ltd. *http://www.digital-interact.co.uk/.* 1997

[Douglis 91]

[31] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Software - Practice and Experience,* 21(8),:757-785, 1991.

[Dugan 94]

[32] J.B. Dugan and M.R. Lyu. "System Reliability Analysis of an N-Version Programming Application". *IEEE Transactions on Reliability",* Vol. 43, No 4, Dec. 1994.

[Elnozahy 92]

[33] E. N. Elnozahy and W. Zwaenepoel. "Replicated Distributed Processes in Mantheo". *In Proc. Conf. on Fault-tolerant Computing Systems,* p. 18-27, July 1992.

[Elnozahy (2) 92]

[34] Elmootazbellah Elnozahy and Willy Zwaenepoel. "Mantheo: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit", *IEEE Transactions on Computers,* Vol. 41, No. 5, May 1992.

[Elnozahy 94]

[35] E. Elnozahy and W. Zwaenepoel. "On the Use and Implementation of Message Logging", *Digest of Papers. The 24th International Symposium on Fault-Tolerant Computing,* p. 289-307, June 1994.

[Enslow 78]

[36] P.H. Enslow. "What is a 'distributed' system?", *Computer,* Jan. 1978, pp. 13-21.

[Fatoohi 94]

[37] R. Fatoohi and S. Weeratunga. "Performance Evaluation of Three Distributed Computing Environments for Scientific Applications", *Proceedings SuperComputing '94,* p. 400-409, Nov. 1994

[Frings 97]

[38] J. Frings. "LINDA - A Development Platform for the Planning of ATM networks", *ITG-Fachberichte Journal,* Iss. No.141, p. 181-189, Feb. 1997.

[Gaida 96]

[39] K. Gaida. "ATM Hits the Desktop", *Byte,* November 1996.

[Geist 88]

[40] R. Geist, R. Reynolds and J. Westall. "Selection of a checkpoint Interval in a Critical-Task Environment", *IEEE Trans. on Reliability,* 37(4), Oct. 1988.

[Geist 94]

[41] A. Geist *et al.* "PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing". *The MIT Press,* Cambridge, Massachusetts, 1994.

[Geist 96]

[42] G. Geist. "Advanced Programming in PVM", *Proceedings of the Third European PVM Conference,* p. 1-6, Munich, Oct. 1996.

[Gibbons 76]

[43] T. K. Gibbons. "Integrity and Recovery in Distributed Systems", *NCC Publications, The National Computing Centre Limited,* 1976.

[Gropp 94]

[44] W. Gropp and E. Lusk. "The MPI Communication Library: Its Design and a Portable Implementation", *Proceedings of the Scalable Parallel Libraries Conference,* CA, USA, p. 160-165, 1994.

[Gunneflo 89]

[45] U. Gunnfelo and J. Karlsson. "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", *Proceedings of the 19th Symposium on Fault-Tolerant Computing,* Chicago, p. 340-346, June 1989.

[Halfill 97]

[46] T. Halfill. "Building Network Applications. Java Gets Down to Business", *Byte, Vol. 22,* p. 87, October 1997.

[Harwell 97]

[47] Harwell Subroutine Library (HSL). *http://www.dci.clrc.ac.uk/Activity.asp?HSL.*

[Huang 93]

[48] Y. Huang and C. Kintala. "Software Implemented Fault Tolerance: Technologies and Experience", *Proceedings of 23rd Int. Symposium on Fault-Tolerant Computing,* p 2-9, Toulouse, France, June 1993.

[Hurwicz 97]

[49] M. Hurwicz. "Preparing for the Gigabit Ethernet. Byte Special Report on Extending the Enterprise", *Byte, Vol. 22 N0. 10,* p. 63, October 1997.

[Janakiraman 94]

[50] G. Janakiraman and Yuval Tamir. "Coordinated Checkpointing-Rollback Recovery for Distributed Shared Memory Multicomputers", *Proc. The 13th Symposium on Reliable Distributed Systems, p. 42-51,* CA, 1994

[Johnson 88]

[51] D. Johnson and W. Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", *Proc. of the 7th Annual Symp. on Principles of Distributed Computing,* Toronto, Canada, p. 171-181, August 1988.

[Jones 90]

[52] O. Jones. "Introduction to the X Window system. 1st edition", *Prentice Hall*, 1990.

[Kanawati 95]

[53] G. Kanawati, N. Kanawati, and J. Abraham. "FERRARI: A Flexible Software-based Error and Fault Injection System", *IEEE Transactions on computers,* Vol. 44 Iss. 2, p. 248-260, Feb. 1995.

[Kendall 95]

[54] R Kendall *et al.* "High Performance Computing and Computational Chemistry: a Review of Methods and Machines". *Reviews in Computational Chemistry 6.* p.209-316 K.B Lipkowitz, D.B. Boyd (Eds), VCH Publishers Inc., New York, 1995.

[Kim 83]

[55] K. H. Kim and A. Kavianpour. "A Distributed Recovery Block Approach to Fault-Tolerant Execution of Application Tasks in Hypercubes", *IEEE Trans. on Parallel and Distributed Systems*, p. 104-11, Jan. 1983.

[Kohl 96]

[56] J. Kohl and A. Geist. "CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications", *Internal Report. Computer Science And Mathematics Division. Oak Ridge National Laboratory.*

[Koo 87]

[57] R. Koo and S. Toueg. "Checkpointing and Rollback Recovery for Distributed Systems", *IEEE Trans on Software Engineering,* SE-13(1), p. 23-21, Jan. 1987.

[Krame 87]

[58] M. Sloman and J. Kramer. "Distributed Systems and Computer Networks", *Prentice Hall*, 1987

[Kumar 91]

[59] V. Kumar and E. Unger. "A non-FIFO Checkpointing Protocol for Distributed Systems", *Symposium on Applied Computing*, p. 266-272, April-1991.

[Lamotte 91]

[60] W. Lamotte, K. Ellens. "Surface Tree Caching for Rendering Patches in a Parallel Ray Tracing System", *Proceedings of the Conference on Scientific Visualisation of Physical Phenomena*, p. 189-207, 1991

[Landau 92]

[61] C. Landau. "The Checkpointing Mechanism in KeyKOS", *Proc. of the Second International Workshop on Object Orientation in Operating Systems,* p. 86-89, Sept. 1992

[Landis 95]

[62] S. Landis and R. Stento. "CORBA with Fault Tolerance", *Object Magazine,* Vol. 5, Iss. 7, p. 62-66, Nov. 1995.

[Lee 90]

[63] P. Lee ant T. Anderson. "Fault-Tolerance: Principles and Practice", Second Revised Edition, Series: "Dependable Computing and Fault-Tolerant Systems", *Vol. 3, Springer-Verlag,* NY. 1990.

[Lee 96]

[64] F. Hsieng Lee. "Parallel Simulated Annealing n a Message Passing Multi-Computer". *PhD Thesis, Utah State University*, Logan, 1995.

[León 93]

[65] J. leon and P. Steenkiste. "Fail-Safe PVM: A Portable Package for Distributed Processing with Transparent Recovery. *Technical Report CMU-CS-93-124, Carnegie Mellon University,* February 1993.

[Li 94]

[66] K. Li, J. Naughton, and J. Plank. "Low-Latency, Concurrent Checkpointing for Parallel Programs", *IEEE Transactions on Parallel and Distributed Computing,* Vol. 5. No. 8, p. 874-879, August 1994.

[Litzkow 90]

[67] M. Litzkow and M. Livny. "Experience with the CONDOR Distributed Batch System", *Proc. of the IEEE workshop on Experimental Distributed Systems,* Hunstville, AL October 1990.

[Litzkow 92]

[68] M. Litzkow and M. Solomon. "Supporting Checkpointing and Process Migration Outside the UNIX Kernel", Usenix Winter Conference, San Francisco, California, 1992.

[Maffeis 97]

[69] S. Maffeis. "Pirhana: A CORBA Tool for High Availability", *Computer,* Vol. 30, No: 4. April 1997.

[Manger 96]

[70] J. Manger. "Essential Java. Developing Applications for the World Wide Web", *McGraw-Hill,* 1996.

[Martin 81]

[71] J. Martin. "Computer Networks and Distributed Processing", *Prentice Hall,* 1981.

[Martin 95]

[72] I. Martin *et al.* "Distributed Parallel Computers Versus PVM on a Workstation Cluster in the Simulation of Time Dependent Partial Differential Equations", *Proceedings Euromicro Workshop on Parallel and Distributed Processing,* CA, USA, p. 20-26, 1995.

[Mattson 94]

[73] T. G. Mattson. "Programming Environments for Parallel Computing: A comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG", *Proceedings of the 27th Hawaii International Conference on System Sciences. Vol. II: Software Technology (Cat. No 94TH0607-2),* p. 586-594, 1994.

[Meakin 90]

[74] Meakin, R.L. "Overset Grid Methods for Aerodynamic Simulation of Bodies in Relative Motion", *8th Aircraft/Stores Compatibility Symposium,* Oct. 1990.

[Morin 97]

[75] C. Morin and I Puaut. "A Survey of Recoverable Distributed Shared Virtual Memory Systems", *IEEE Transactions on Parallel and Distributed Systems,* Vol. 8, No. 9, Sept. 1997.

[Mills 83]

[76] D. Mills. "Internet Delay Experiments; RFC889", *In ARPANET Working Group Requests for Comments,* No. 965. SRI International, Menlo Park, Calif., Sep. 1983.

[Mullender 95]

[77] S. Mullender. "Distributed Systems", *Addison Wesley,* 1995.

[Nye 90]

[78] A. Nye. "The Definitive Guides to the X Window System. Xlib. Volume 1: Xlib Programming Manual", *O'Reilly & Associates, Inc.,* 1990.

[Ousterhout 94]

[79] J. Ousterhout. "TCL and the TK Toolkit", *Addison Wesley,* 1994.

[Plank 94]

[80] J. S. Plank and K. Li. "A Consistent Checkpointer for Multi-computers", *IEEE Parallel & Distributed Technology,* 2(2), p. 62-67, 1994

[Plank 95]

[81] J. Plank, M. Beck and G. Kingsley. "Libckpt: Transparent Checkpointing Under Unix", *Proc. of USENIX Winter 1995 Technical Conference,* Jan. 1995.

[Plank 97]

[82] J. Plank, M. Puening. "Checkpointing Java", *Project Description at the DoC website at the University of Tennessee,* http://www.cs.utk.edu/~plank/javackp.html.

[Postel 81]

[83] J. Postel *et al.* "Transmission Control Protocol; RFC793.", *In ARPANET Working Group Requests for Comments,* No. 793. SRI International, Menlo Park, California. Sep. 1981.

[Powell 88]

[84] D. Powell *et al.* "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", *Proc. of the 18th International Symposium on Fault-Tolerant Computing, p. 246-251,* June 1988.

[Pradhan 86]

[85] D. Pradhan. "Fault-Tolerant Computing Theory and Techniques, Volume II", *Prentice-Hall,* NJ 1986.

[Pressman 92]

[86] Roger Pressman. "Software Engineering. A Practitioner's Approach", *McGraw-Hill International,* UK 1992.

[Riberio 95]

[87] L. Riberio *et. al.* "Numerical simulations of liquid-liquid agitated dispersions on the VAX 6250/VP", *Computing Systems in Engineering"*, Vol. 6, Iss. 4-5, p. 465-469, Oct. 1995.

[Russinovich 95]

[88] M. Russinovich and Z. Segall. "Application -Transparent Checkpointing in Mach 3.0/UX", *Proc. of the 28th Annual Hawaii International Conference on System Sciences,* p. 114-123, 1995.

[Schmidt 97]

[89] S. Maffeis and D. Schmidt. "Constructing REliable Distributed Communication Systems with CORBA". *IEEE Communication Magazine,* Vol. 14, No. 2, February 1997.

[Segall 88]

[90] Z. Segall *et al.* "FIAT - Fault Injection Based Automated Testing Environment", *In Proc. 18th International Symposium on Fault-Tolerant Computing,* p. 102-107, June 1988.

[Seligman 94]

[91] E. Seligman and A. Beuelin. "High-Level Fault Tolerance in Distributed Programs", *Technical Report CMU-CS-94-223, School of Computer Science, Carnegie Mellon University,* Pittsburgh, December 1994.

[Sens 93]

[92] P. Sens and B. Folliot 1993. "STAR: a Fault Tolerant System for Distributed Applications". *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, p. 656-660, Dec. 1993.

[Sens 95]

[93] P. Sens. "The performance of Independent Checkpointing in Distributed Systems", *Proc. The 28th Hawaii International Conference on Systems Sciences,* January 1995.

[Silva 92]

[94] L. Silva and G. Silva. "Global Checkpointing for Distributed Programs", *Proc. of the 11th Symposium on Reliable Distributed Systems,* Huston, Texas, p. 155-162, Oct. 1992.

[Sista 89]

[95] A. Sista and J. Welch. "Efficient Distributed Recovery Using Message logging", *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing Systems*, Aug. p. 222-238, 1989.

[Stellner 95]

[96] G. Stellner. "CoCheck: Checkpointing and Process Migration for MPI", *In Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, Hawaii, April 1996.

[Sterling 84]

[97] M. Sterling and A. Bargiela. "Minimum Norm State Estimation for Computer Control of Water Distribution Systems", *IEE Proceedings,* 131, D, 2, March 1984.

[Sterling(2) 84]

[98] M. Sterling and A. Bargiela. "Leakage Reduction by Optimised Control of Valves in Water Networks", *Transactions of The Institute of Measurement and Control".*

[Storm 87]

[99] R. Storm, S Yemini and D. Bacon. "Towards Self-Recovering Operating Systems", *The International Conference on Parallel Processing*, North-Holland, 1987.

[Sun 94]

[100] "SPARCworks & SPARCcompiler reference guide, Version 3.0.1 for Solaris", *Sun-Soft, Sun Microsystems Inc.,* 1994.

[Sun 97]

[101] M. Sun and L. Tong. "Communication Performance and Parallel Performance Research of a Networked Parallel Computing System", *Mini-Micro Systems,* Vol. 18, Iss. 1, p. 13-18, Jan. 1997.

[Taha 95]

[102] Taha Osman, and Andrzej Bargiela. "Error Detection For reliable Distributed Simulations", *In proceedings of the 7th European Simulation Symposium*, p. 385-362, 1995.

[Taha 97]

[103] T. Osman and A. Bargiela. "Process Checkpointing in an Open Distributed Environment", *In the Proc. of the 11th European Simulation Multiconference*, p. 536-541, Turkey, June 1997.

[Taha(2) 97]

[104] Taha Osman and Andrzej Bargiela. "A Selective Message Logging Checkpointing Algorithm for Reliable Distributed Computations", *submitted for publication in the IEEE Transactions on Reliability.*

[Tamir 89]

[105] Y. Tamir and T. M. Fraizer. "Application-Transparent Process-level Error Recovery for Multicomputers", *Hawaii International Conf. On System Sciences,* Jan. 1989.

[Tanenbaum 90]

[106] A. Tanenbaum *et. al.* "Experiences with the Amboeba Distributed Operating System", *Communication of the ACM,* Vol. 33, p. 46-63, Dec. 1990.

[Thiran 96]

[107] J. Thiran *et al.* "IMIS: A Multi-Platform Hardware Package for Telediagnoses and 3D Medical Image Processing", *Proceedings. International Conference on Image Processing (Cat. No. 96CH35919),* Vol. 2, p. 273-276, Sept 1996.

[Toueg 84]

[108] S. Toueg and Özlap Babaglu. "On the optimum checkpoint selection problem", *SIAM Journal on Computing,* Vol. 13, p. 630-649, Aug. 1984.

[Vounckx 93]

[109] J. Vounckx *et al.* "The FTMPS-Project: Design and Implementation of Fault-Tolerance Techniques for Massively Parallel Systems". *Katholieke Universitei Leuven,* Belgium.

[Wang 92]

[110] Y. Wang and W. Kent. "Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems",

[Ward 85]

[111] P. Ward and S. Mellor. "Structured Development for Real-Time Systems. Volume 1: Introduction & Tools", *Yourdon Press Computing Series,* 1985.

[Ward 86]

[112] P. Ward and S. Mellor. "Structured Development for Real-Time Systems. Volume 3: Implementation Modelling Techniques", *Yourdon Press Computing Series,* 1986.

[Wojcik 90]

[113] Z. Wojcik and B.E. Wojcik. "Fault Tolerant Distributed Computing Using Atomic Send and Receive Checkpoints", *Proc. 2nd IEEE Symp. on Parallel and Distributed Processing*, p. 215-222, 1990.

[Young 74]

[114] J. Young. "A First Order Approximation to the Optimum Checkpoint Interval", *Communications of the ACM,* 17(9), Sept. 1974.

[Zwaenepoel 92]

[115] E. Elnozahy and W. Zwaenepoel. "The Performance of Consistent Checkpointing", *Proc. of he 11th IEEE Symp. on Reliable Distributed Systems*, pp. 39-47, 1992.

Design Appendices

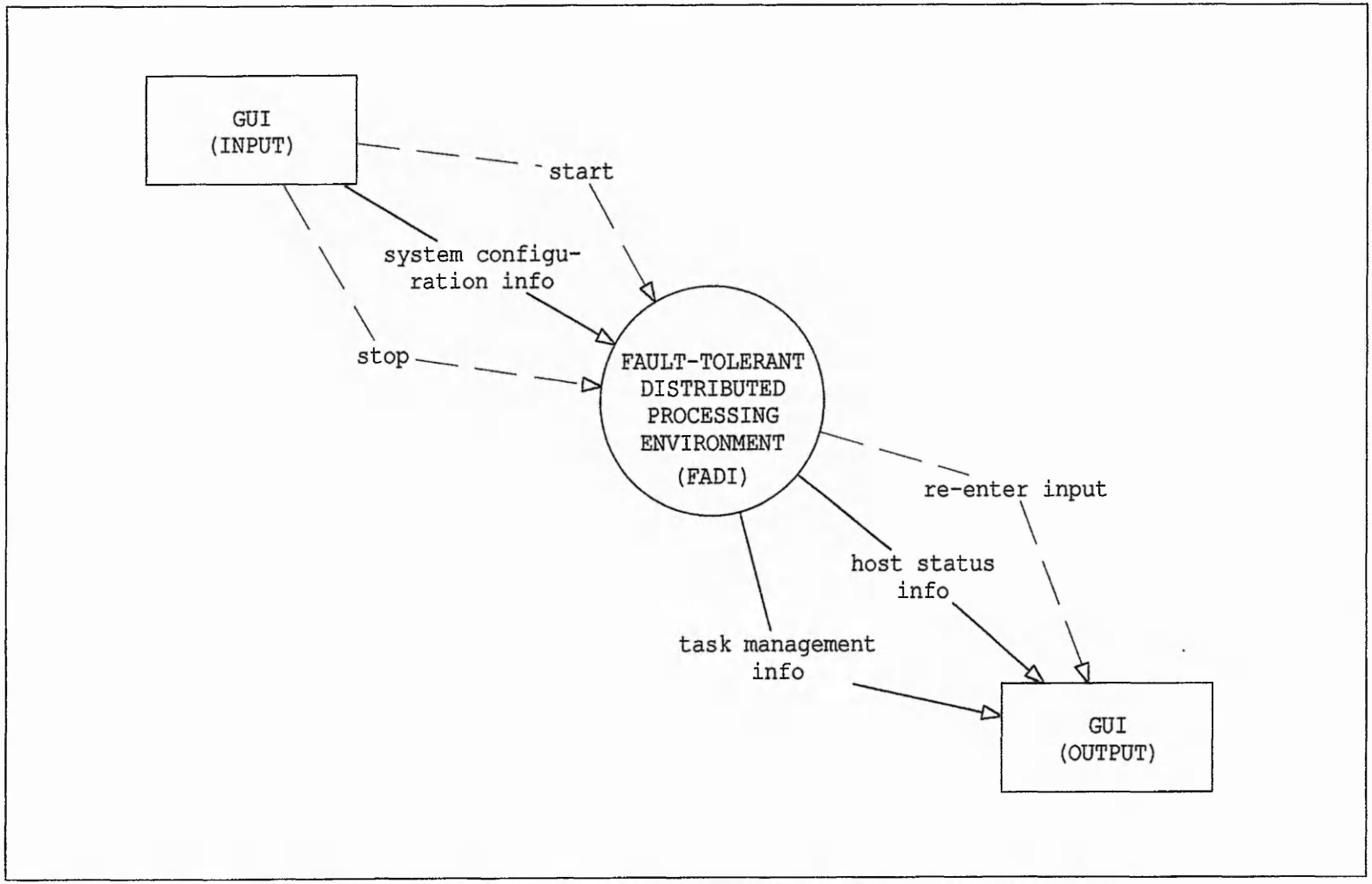
Design is the technical kernel for software engineering. In developing FADI, we took great consideration in developing a modular design, where software is logically partitioned into components(modules) that perform specific functions and subfunctions. These modules exhibit functional characteristics. This resulted in a representation of FADI that is implementation-independent to boost the reusability of its software components. The *error detection, process allocation, and the backup and recovery* are all stand-alone modules with clear definition of the interfaces between them which simplified their modification or even replacement without affecting the integrity of the whole system. For instance, replacing the checkpointing/rollback recovery method with a one based on hardware redundancy/process replication can be swiftly achieved while keeping the other modules and interface connections intact. The same concept applies for reusing FADI components for incorporating fault-management in the rapidly growing field of internet-based distributed object computing as for *CORBA's Object Request Brokers (ORB)* [Landis 95].

The abstract view that the design diagrams provide also improves the readability of the system for fast generation of quality code on one hand, and aid in giving a detailed but simple representation of the system on the other.

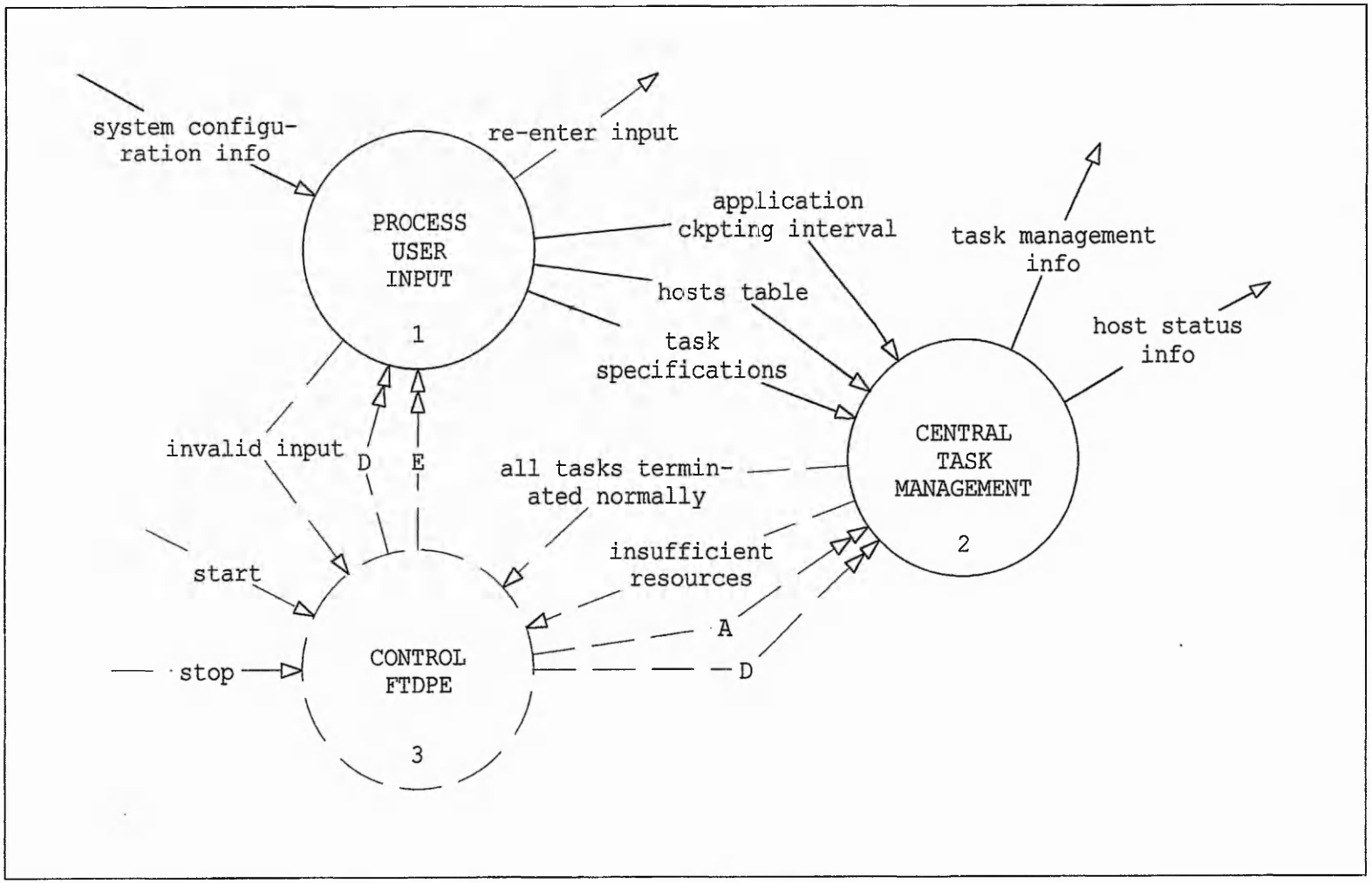
FADI design was implemented following the Data-Flow/structured design methodology. For details of the modelling techniques and notation information of this methodology refer to Ward & Mellor valuable document on structured software development [Ward 85]. Listing of all the software design stages are provided in appendices A-D.

APPENDIX A Data-Flow Design

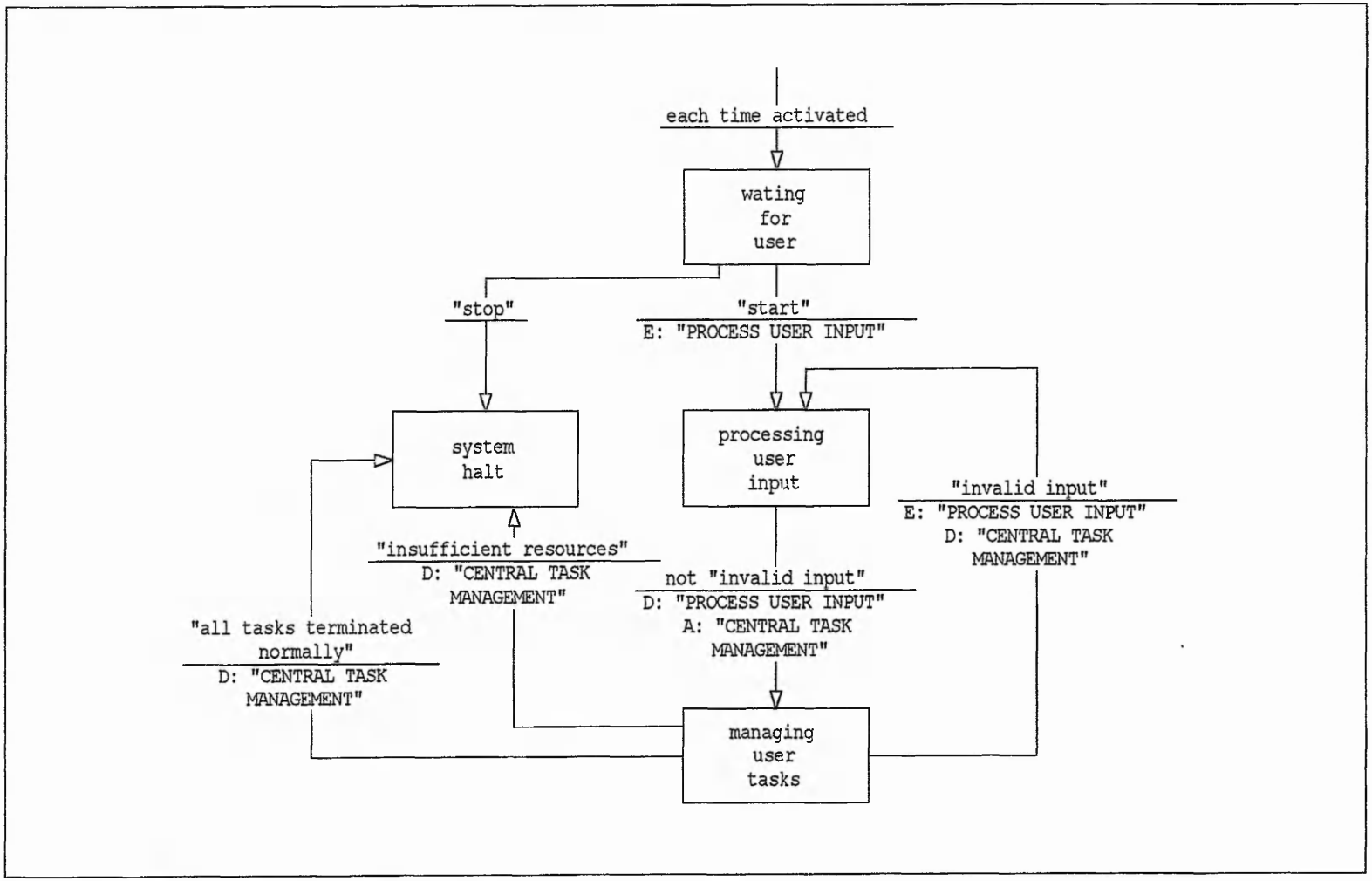
This appendix contains listing of the data-flow diagrams representing the processes (data transformations) of FADI modules.



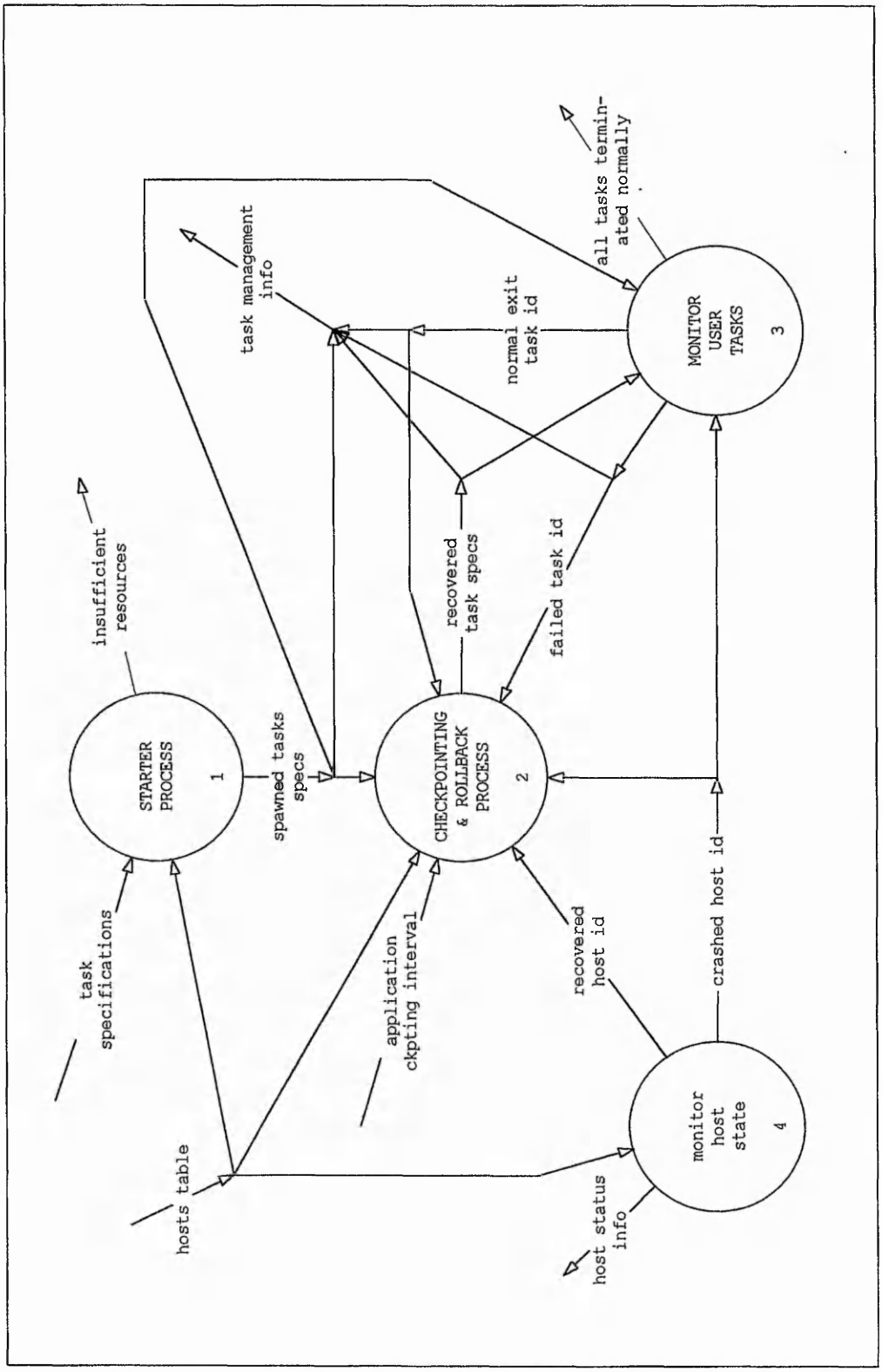
...am - FAULT-TOLERANT DISTRIBUTED PROCESSING ENVIRONMENT, CONTEXT.DFD
...-Tolerant Distributed Processing Environment TAHA OSMAN 21-Aug-97
Page 1 of 1



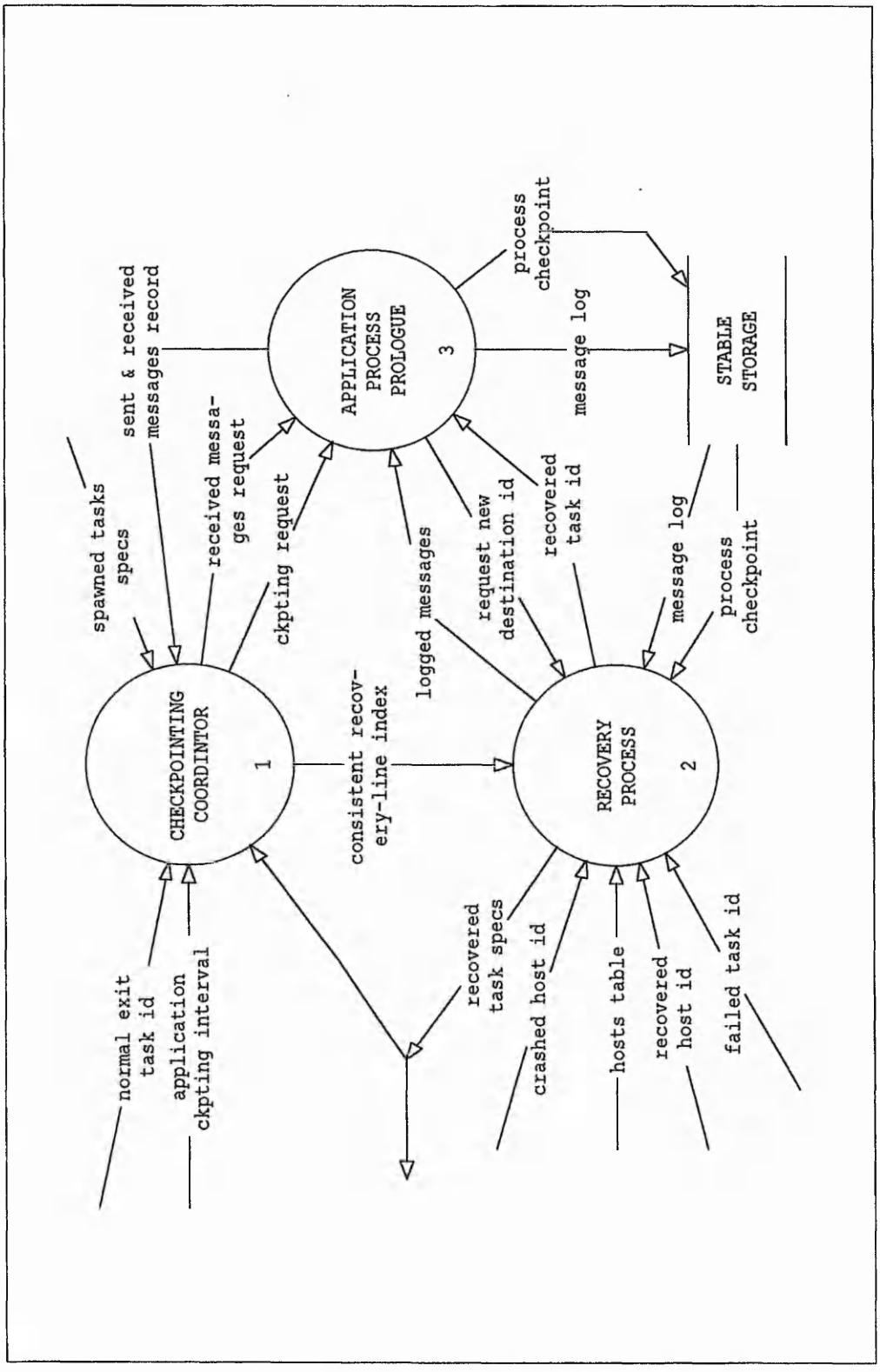
0 - FAULT-TOLERANT DISTRIBUTED PROCESSING ENVIRONMENT, MAIN.DFD
Fault-Tolerant Distributed Processing Environment TAHA 3-Jun-97
Page 1 of 1



0-s3 - CONTROL FTDPE, CONTROL.STD
Fault-Tolerant Distributed Processing Environment TAHA 3-Jun-97
Page 1 of 1



2 - CENTRAL TASK MANAGEMENT, TASKMAN.DFD
Fault-Tolerant Distributed Processing Environment TAHA 3-Jun-97
Page 1 of 1



2.2 - CHECKPOINTING & ROLLBACK PROCESS, CK&ROLL.DFD
Fault-Tolerant Distributed Processing Environment TAHA 3-Jun-97
Page 1 of 1

APPENDIX B Data Dictionary

This appendix presents a dictionary depicting the flow of information between the system processes.

Project: H:\CC416350\FADI_DGN\

Title : FAULT TOLERANCE IN DISTRIBUTED COMPUTER SYSTEMS V

Date: 6-Nov-97 Time: 11:45

Report: Flow Decomposition Summary (BNF)

This report generates an alphabetical data dictionary listing of all items which may have a BNF clause.

Name: all tasks terminated normally

Type: Control flow

Bnf: \ The distributed application tasks were succssesfully executed \

Name: application ckpting interval

Type: Discrete flow

Bnf: \ time interval used by the system timer to periodically raise a checkpoiniting interrupt \

Name: ckpting request

Type: Discrete flow

Bnf: \ a signal sent to the application process at every application checkpointing interval. It interrupts the application process and calls the checkpointing handler \

Name: consistent recovery-line index

Type: Discrete flow

Bnf: \ An indexes set of consistent checkpoints of all the distributed application processes, to which processes can rollback in case of node failure. \

Name: crashed host id

Type: Discrete flow

Bnf: \ hosts that failed to fulfil the aknowledgment request at the current moment of time \

Name: failed task id

Type: Discrete flow

Bnf: \ id of tasks that exited with erroneous status \

Name: host status info

Type: Discrete flow

Bnf: \ operation status of hosts running in the network environment \

Name: hosts table

Type: Discrete flow

Bnf: \ list of available hosts on the net (host id + arch) \

Name: insufficient resources

Type: Control flow

Bnf: \ not enough processsing power to run all user tasks \

Name: invalid input

Type: Control flow

Bnf: \ entered host and task specifications are not compatible

Name: logged messages

Type: Discrete flow

Bnf: \ messages retrieved from stable storage and re-sent to the restarted application processes\

Name: message log

Type: Discrete flow

Bnf: \ Messages logged to stable storage. These messages will not be re-sent after the processes rollback, therefore they must be replayed from stable storage \

Name: normal exit task id

Type: Discrete flow

Bnf: \ id of tasks that exited without failure \

Name: process checkpoint

Type: Discrete flow

Bnf: \ an image of the processes state (stack, data,CPU,etc.) at the current moment of execution. \

Name: received messages request

Type: Discrete flow

Bnf: \ interrupt-request to the application process to send info about undelivered messages\

Name: recovered host id

Type: Discrete flow

Bnf: \ process ID of recovered host to be re-instated in active hosts pool \

Name: recovered task id

Type: Discrete flow

Bnf: \ process ID of a recovered task requested by a destination process \

Name: recovered task specs

Type: Discrete flow

Bnf: \ specifications of checkpoint processes that are to replace the failed tasks \

Name: request new destination id

Type: Discrete flow

Bnf: \ request of process ID of a restarted task \

Name: re-enter input

Type: Control flow

Bnf: \ user to re-enter invalid input \

Name: sent & received messages record

Type: Discrete flow

Bnf: \ bookkeeping information about messages received and sent by the process. It allows the ckpting coordinater to verify the consistency of the taken checkpoint. \

Name: spawned tasks specs

Type: Discrete flow

Bnf: \ (task-id) + (task-host id) + (task checkpointing interval) \

Name: Stable Storage

Type: Store

Bnf: message log + process checkpoint

Name: start

Type: Control flow

Bnf: \ run the FTDPE master daemon \

Name: stop

Type: Control flow

Bnf: \ terminate FTDPE system operation \

Name: system configuration info

Type: Discrete flow

Bnf: \ task id + task name + task-host id + application checkpointing interval \

Name: task management info

Type: Discrete flow

Bnf: (spawned tasks specs) + (failed task id) + (recovered task specs) + (normal exit task id)

Name: task specifications

Type: Discrete flow

Bnf: \ task name + task-host name + checkpointing interval \

---- End of report ----

APPENDIX C Process Specification

This appendix contains listing of the process specification (pseudo-code) in structured english (PDL - Program Description Language).

@IN = system configuration info

@OUT = hosts table

@OUT = invalid input

@OUT = re-enter input

@OUT = task specifications

@OUT = application ckpting interval

@PSPEC Process User-Input

whenever enabled,
 get "system configuration info"
 get list of available hosts on the net
 if (user-specified host not operative)
    or (syntax error in task specifications) then
  issue "re-enter input"
  set "invalid input" to true
 else
  set "invalid input" to false
  send "task specifications" + "hosts table" +
   "application ckpting interval"+ to task management
 endif
@

```
@IN  = hosts table
@IN  = task specifications
@OUT = insufficient resources
@OUT = spawned tasks specs

@PSPEC Task Allocation Process

upon receipt of "task specifications" and "host table"  do
 if (hosts are specified by the user), then
  repeat
   spawn user task on user-specifiefied host with \
            argument: checkpointing interval
       if (failed to spawn task on user-specified host), then
          repeat
            spawn user task on a host with similar arch. from\
             the host table with argument: checkpointing interval
          until (spawn O.K or end of hosts table)
   endif
   if (user task still not spawned) then
    issue "insufficient resources"
    terminate the process
   endif
  until (all user tasks are spawned)
 else
    repeat
  repeat
        spawn user task on a host from host table with \
        argument: checkpointing interval
      until (spawn O.K or end of hosts table)
```

```
      if (user task still not spawned) then
     issue "insufficient resources"
     terminate the process
      endif
    until (all user tasks are spawned)
endif
send "spawned tasks specs" to task monitoring processes
send task spawning information to GUI

@
```

```
@IN  = application ckpting interval
@IN  = recovered task specs
@IN  = spawned tasks specs
@IN = normal exit task id
@IN = sent & received messages record
@OUT = consistent recovery-line index
@OUT = received messages request
@OUT = ckpting request
@PSPEC Checkpointing Coordinator

upon receipt of "spawned tasks specs" do
 set checkpointing flag
 unset received messages flag
 at every "application ckpting interval" do
  if checkpointing flag is set then
        send "ckpting request" to all application processes
        unset checkpointing flag
  endif
  if received messages flag is set then
        send "received messages request" to all application\
            processes
        unset received messages flag
  endif
 until[all tasks exited normally]

 case received message of:
  [sent & received messages record] :
   if the balance of "sent & received messages record" is\
            correct then
      send "consitent recovery-line index" to recovery task
      set checkpointing flag
```

```
        else
            set received messages flag to wait for undelivered \
            messages
    endif
  break
  [recovered task specs]
    update id of failed task in checkpointed tasks list
    if recovered task is in debt of acknowledgment then
            re-send checkpointing or received messages request
    endif
  break
  [normal exit task id]
        remove exited task from checkpointed tasks list
  break
 endcase
@
```

```
@IN  = consistent recovery-line index
@IN  = crashed host id
@IN  = failed task id
@IN  = hosts table
@IN  = message log
@IN  = process checkpoint
@IN  = recovered host id
@IN = request new destination id
@OUT = recovered task id
@OUT = logged messages
@OUT = recovered task specs
@PSPEC Recovery Process

on receipt of "hosts table" do
 case received message of:
  [crashed host id] :
   remove "crashed host id" from hosts table
  break
  [recovered host id] :
   add "recovered host id" into hosts table
  break
  [request new destination id] :
   send "recovered task id" to requesting task
  break
  [failed task id] :
   get "process checkpoint" belonging to "consistent\
          recovery-line" of failed task from stable storage
   get "message log" of the failed task from stable storage
   restart failed task from consistent checkpoint
   replay logged messages to the recovered task
      send "recovered task specs" to task monitoring & GUI
  break
 endcase

@
```

@IN = ckpting request

@IN = logged messages

@IN = received messages request

@IN = recovered task id

@OUT = sent & received messages record

@OUT = message log

@OUT = process checkpoint

@OUT = request new destination id

@PSPEC = Application Process Prologue

```
on receipt of ckpting request do
  take a local checkpoint of the process
  save "process checkpoint into stable storage"
  send "sent & received msg record" to ckpting coordinator
end

on receipt of received messages request do
  send "received messages record" to ckpting coordinator
end

upon (message send) do
  if message cannot be sent because destination failed
    request new destination id from recovery task
    await for receipt of recovered task id
  endif
end
```

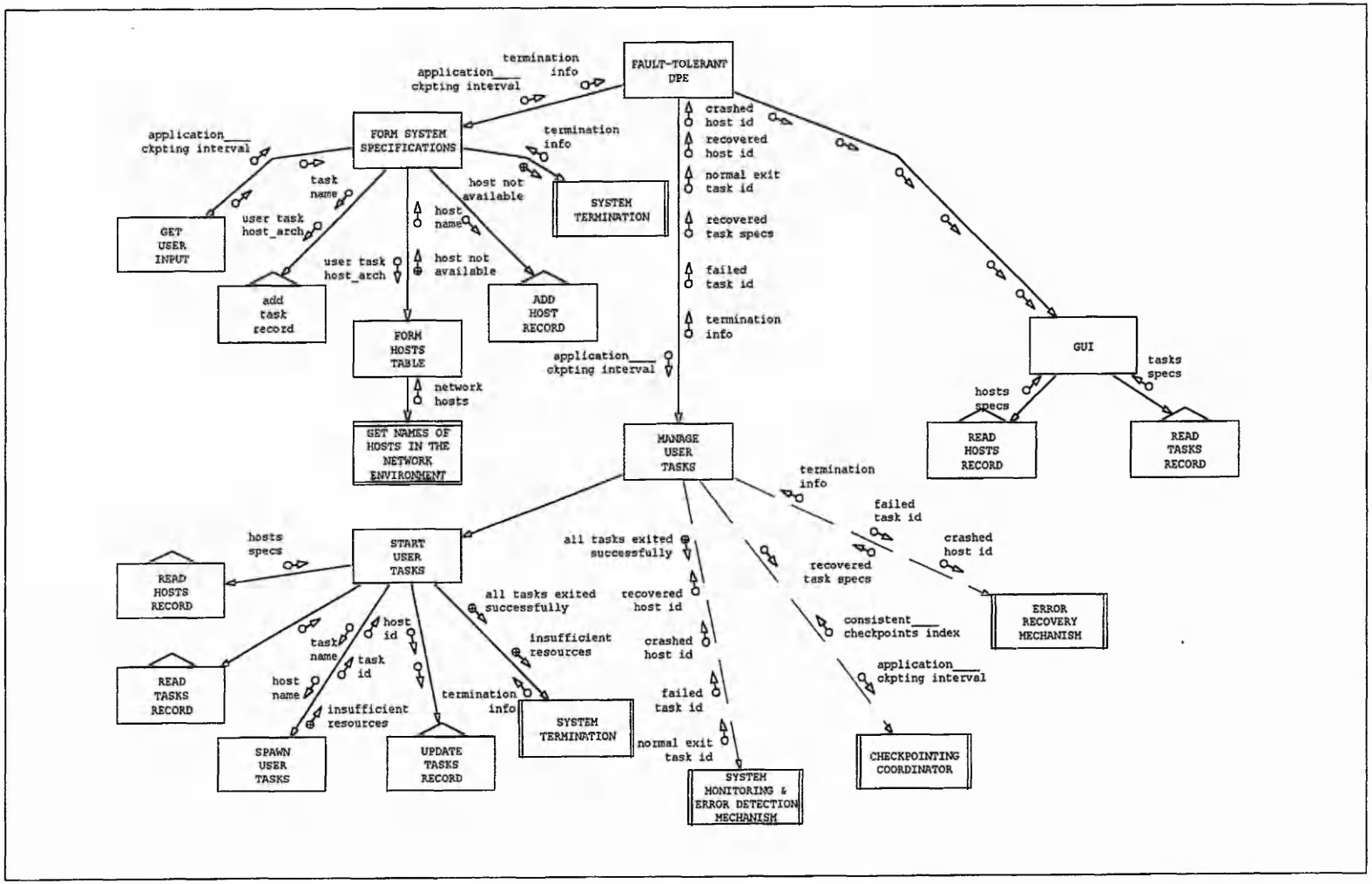
```
upon (message receipt) do
 if ((received msg was not replayed "logged messages") and
   (it cannot be resent upon rolled-back)) then
     record "message log" into stable storage
 elseif (message is duplicated)
          ignore received message
 endif
end

@
```

```
@IN  = recovered task specs
@IN  = spawned tasks specs
@IN  = crashed host id
@OUT = failed task id
@OUT = normal exit task id
@OUT = all tasks terminated normally

@PSPEC Monitor User Tasks

upon receipt of "spawned tasks specs" do
repeat
 upon receipt of wait-exit message do
  add any "recovered task specs" to the spawned tasks list
  if ('exited task id' is in spawned tasks list), then
   case (exit status) of :
    [normal exit] :  send "normal exit task id" to
                                          ckpting coord & GUI
    [abnormal exit] : send "failed task id" to recovery
                                            process
   endcase
   remove exited task id from spawned tasks list
  endif
 upon receipt of "crashed host id" message
  send "failed task id" of all tasks running on crashed \
              host to recovery task
until (all tasks exited normally)
issue "all tasks terminated normally"

@
```

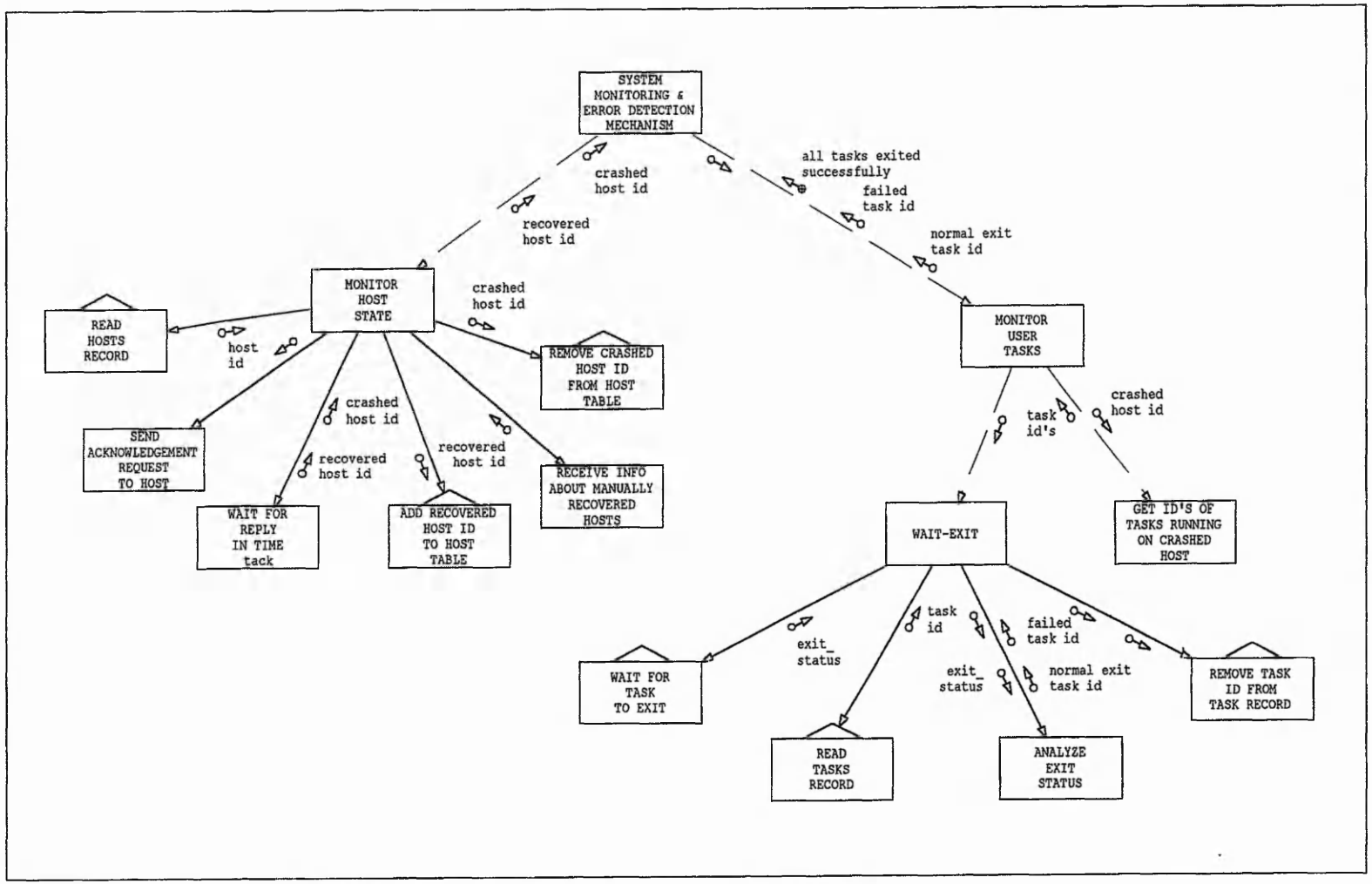
```
@IN  = hosts table
@OUT  = recovered host id
@OUT = crashed host id
@OUT = host status info

@PSPEC Monitor Host State


upon receipt of "hosts table" do
 at 'pre-defined intervals' repeatedly do
  send request of aknowledgment to all hosts on "active\
            hosts table"
  if (host fails to acknowledge), then
   remove crashed host id from "active hosts table"
   send "crashed host id" to user-task monitoring & \
              recovery processes
  else, do
   if (host not in 'hosts list'), then
    send "recovered host id" to the recovery process
   endif
  endif
  report "host status info" to GUI
 until (stopped)

@
```

APPENDIX D Data-Structured Design

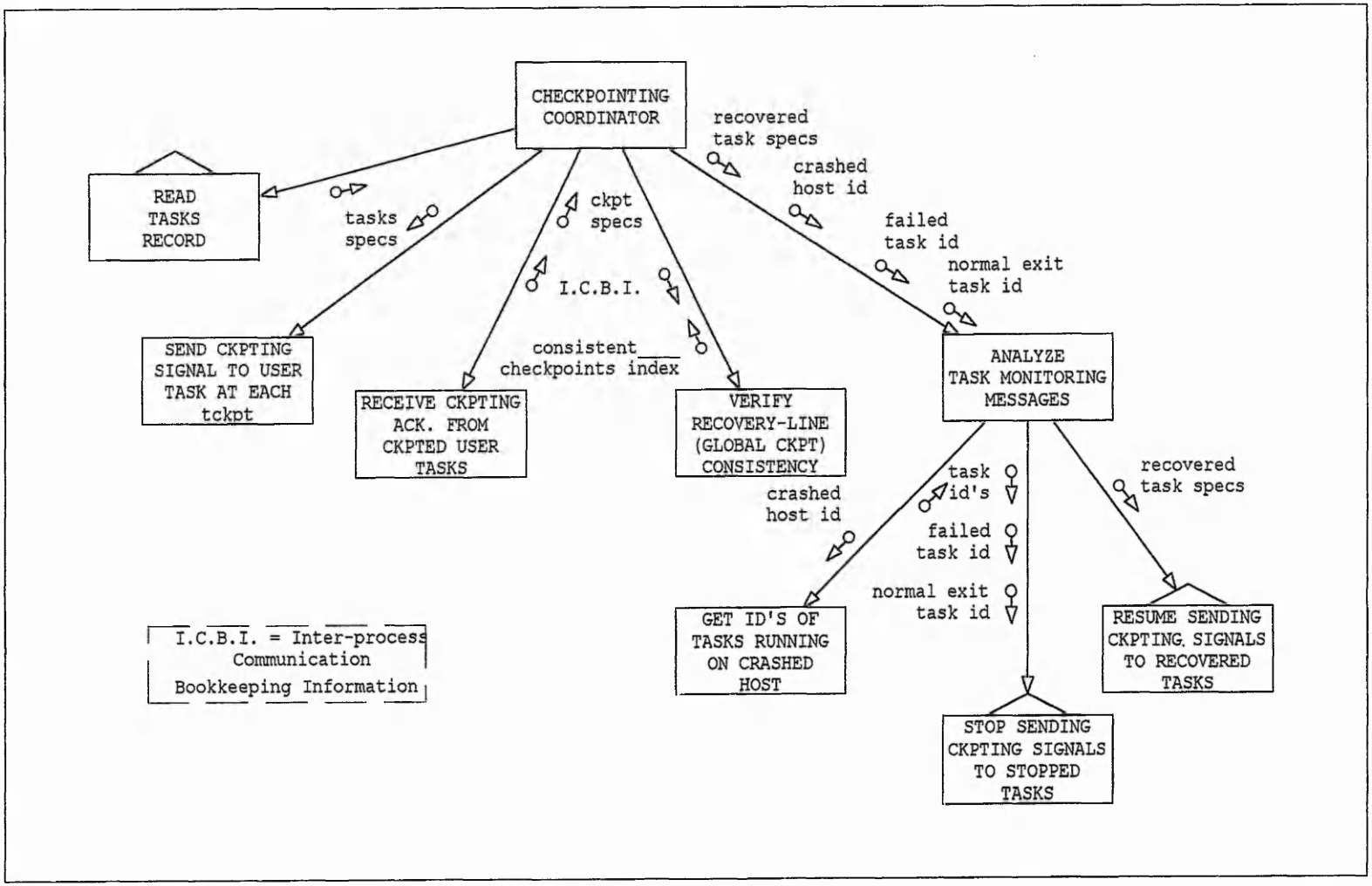
This appendix contains data-structured design diagrams. Their main purpose is to aid in the implementation analysis of the system [Ward 86].



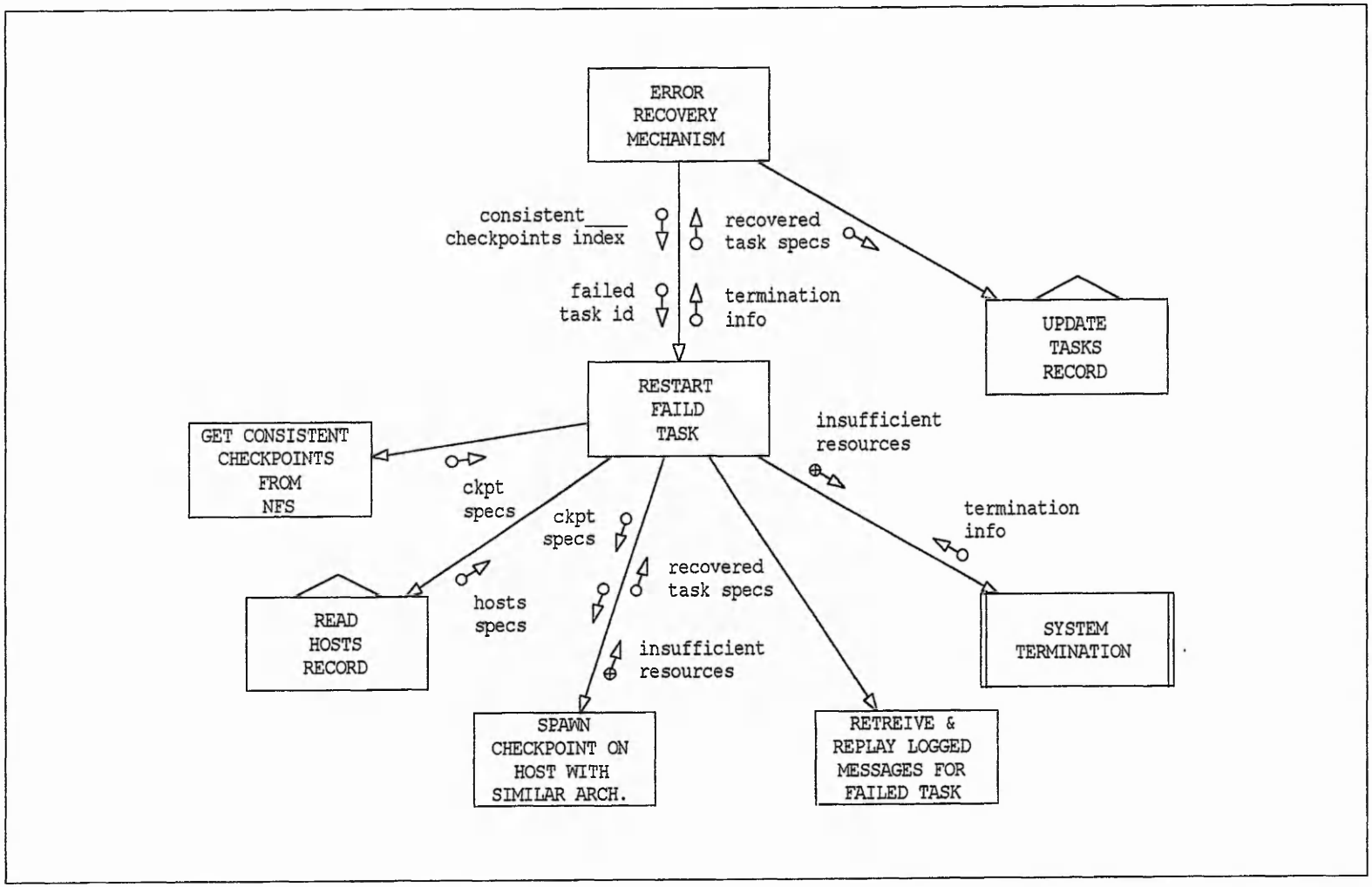
FAULT-TOLERANT DPE, MAIN.CSD
...rant Distributed Processing Environment PC191 9-Jul-96
Page 1 of 1



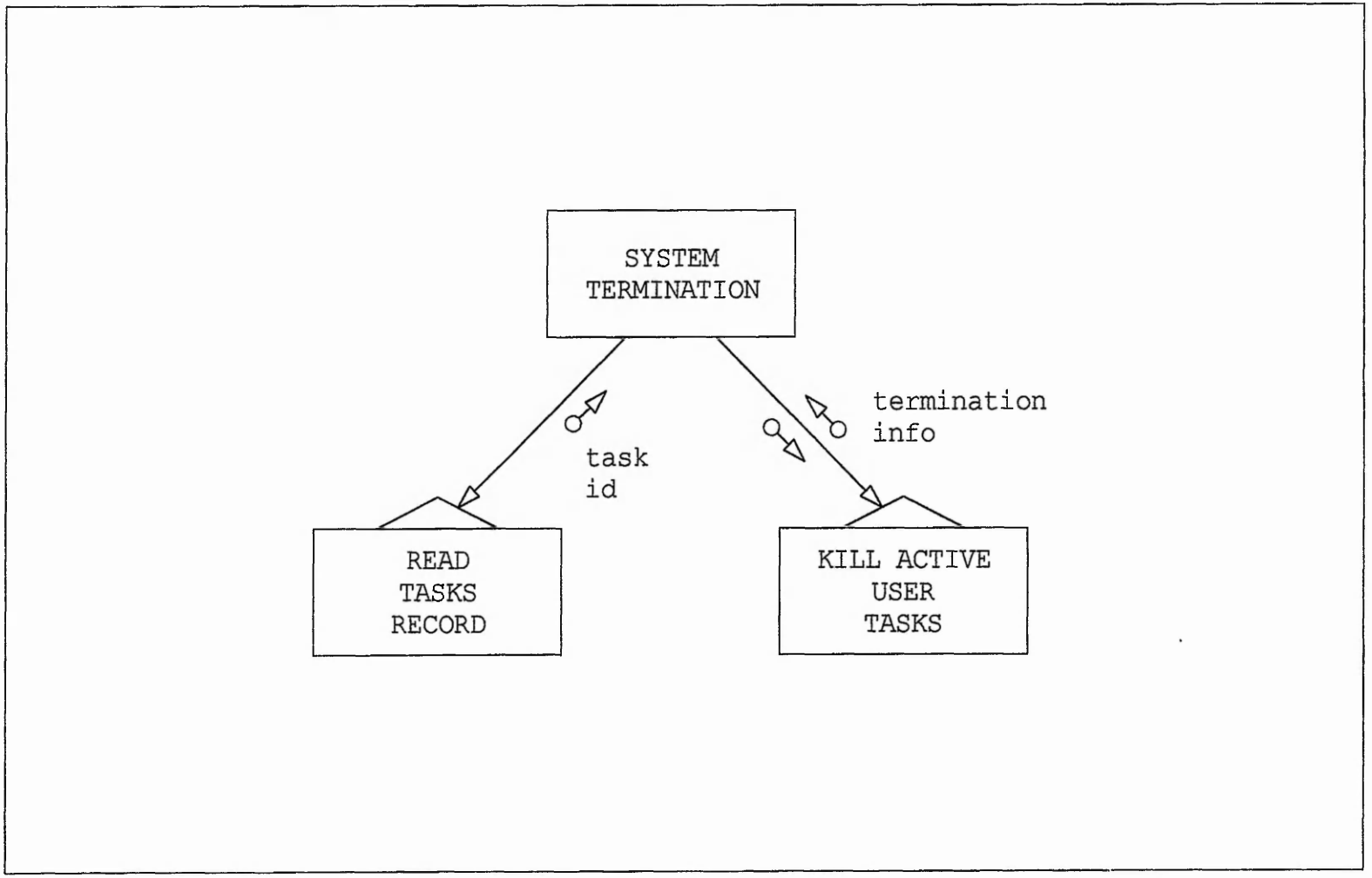
SYSTEM MONITORING & ERROR DETECTION MECHANISM, DETECT.CSD
...-Tolerant Distributed Processing Environment TAHA OSMAN 7-Feb-96
Page 1 of 1



CHECKPOINTING COORDINATOR, COORD.CSD
...-Tolerant Distributed Processing Environment TAHA OSMAN 10-Mar-98
Page 1 of 1



ERROR RECOVERY MECHANISM, RECOVERY.CSD
...rant Distributed Processing Environment PC191 9-Jul-96
Page 1 of 1



SYSTEM TERMINATION, TERM.CSD
...-Tolerant Distributed Processing Environment TAHA OSMAN 15-May-95
Page 1 of 1