Implementing Highly Available, Highly Reliable Virtual Processors

by

Robert Noël Macdonald

B.Sc., University of Victoria, 1988

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science
in the Department of Computer Science

We accept this thesis as conforming to the required standard

[redacted]

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

[redacted]

Dr. Mantis H. M. Cheng
Departmental Member (Department of Computer Science)

[redacted]

Dr. Warren D. Little
Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Fayez El Guibaly
External Examiner (Department of Electrical and Computer Engineering)

Supervisor: Dr. Gholamali C. Shoja

Abstract

A fault-tolerant distributed facility called a Halt on Failure Processor (HFP) and its performance in a network of workstations are described. Process replication and n-modular redundancy are used to achieve fault tolerance in a general purpose workstation environment.

A blacklisting mechanism is used to differentiate between slow and crashed workstations. The system achieves high availability by keeping a list of healthy workstations. The HFP will halt rather than deliver the results from an erroneous calculation to its users. The design of the HFP is presented along with the type and number of errors it is capable of handling. The implementation using the existing Remote Execution Manager is discussed.

Extensive performance studies were carried out within a network of Sun SPARC workstations running UNIX. Performance results are presented and the costs of performing fault management at various levels are exposed. Flaws in the way UNIX reports load information and their implication on load-balancing are pointed out.

It is shown that HFPs can achieve high availability and fault-tolerance using the idle cycles of workstations in a local area network with little performance degradation.

Examiners:

[redacted]

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

[redacted]

Dr. Mantis H. M. Cheng
Departmental Member (Department of Computer Science)

[redacted]

Dr. Warren D. Little
Outside Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Fayez El Guibaly
External Examiner (Department of Electrical and Computer Engineering)

Contents

List of Tables

List of Figures

Acknowledgements

I gratefully acknowledge the patience and support of my supervisor, Dr. Gholamali C. Shoja. His knowledge, guidance and persistence were primary factors in the completion of this research.

I thank the Faculty of Graduate Studies and the Department of Computer Science for their support.

This work was supported in part by an NSERC Scholarship (PGS1 and PGS2) and by a President's Research Scholarship from the University of Victoria.

Dedication

This work is dedicated to the memories of my father, William Harold Macdonald, and my grandfathers, Noël William Joseph Macdonald and Robert Hewitt Calvin Birdsell.

Chapter 1

Introduction

Networked computer systems abound in modern academic, industrial, commercial and government sites. Most of these organisations have hundreds, if not thousands, of powerful workstation computers. A large proportion of these computers are generally idle and have some, or even a major portion, of their CPU cycles going to waste.

Despite the enormous computational power of newer workstations, they can barely keep pace with the increasing demands that can be put on them by some new applications. These applications are computationally complex and CPU intensive. With proper programming techniques, multiple processors can compute faster than a single processor. A workstation in a local network can be likened to a node in a massively parallel multiprocessor. In this light, load sharing and remote execution are totally justifiable and need more development.

In addition to providing increased performance, the use of multiple workstations on a network adds the possibility of partial failure to applications. Fortunately, a network of multiple workstations is also an excellent environment for achieving fault-tolerance through the use of n-modular redundancy. Applying these techniques precludes partial failure and provides a level of fault-tolerance unavailable in

a uniprocessor environment. For a large proportion of applications, this approach can be more flexible and economical for reliable computing than special purpose machines such as Tandem [1].

1.1 Objectives

The prime objective of this research has been to provide a high performance, highly available, fault-tolerant computing environment using parallel remote processors and adaptive load sharing. The virtual fault-tolerant unit is called a Halt on Failure Processor (HFP). An HFP must guarantee delivery of correct responses to a user as long as a specified error threshold is not exceeded. A user is to be permitted a multiplicity of HFPs to allow parallel processing in computationally complex applications.

HFPs must have the ability to distinguish overloaded or slow remote workstations from crashed ones. The system must provide healthy, available workstations to which new tasks can be assigned. Faulty workstations should be able to rejoin this list after repair or when a backlog of tasks is cleared.

Portability and simplicity are used as guidelines for this system. It should run entirely in the user space of an operating system and use standard communication mechanisms.

1.2 Fault Tolerance

Fault tolerance is the ability of a system to withstand faults in some of its parts. There are many types of computer faults and they have been defined, characterized and classified in different ways [2–7]. Detecting and masking faults in a distributed computing environment has been addressed in many papers [8–13]. Faults that are

of interest in this research are defined in Section 2.1.

1.3 Halt on Failure Processor

A processor that will halt rather than deliver an erroneous response to a user has a *halt on failure* property. If a user can detect that a processor has failed and is halted then the processor has a *failure status* property [14]. A processor that has both of these properties is called a Halt on Failure Processor (*HFP*). These properties are defined more thoroughly in Chapter 4, along with HFPs.

A virtual HFP was implemented in a distributed environment. It provides a configurable level of fault tolerance and frees the user from managing replicated processes on remote workstations. The HFP detects and masks errors from faulty workstations and allows them to rejoin processing if and when they are repaired. It also recognizes the difference between overloaded and failed workstations.

The rationale for proposing and implementing HFPs are:

1. They are the key element in the implementation of fault-tolerant distributed systems.
2. The halt on failure and failure status properties are crucial to the operation and realization of many distributed algorithms [12,15–17].
3. The ability to mask occasional or transient errors from the user is more desirable than simply reporting the failure and stopping.

1.4 Computing Model

A client/server computation model was assumed. The client generates a request from a host computer. The request is processed by a server on one or more remote computers and a single response is returned to the client.

This model is consistent with the models assumed for fault classification in [3, 4,6]. Workstations can fail by stopping, sending incorrect responses, sending late responses, duplicating responses, and by responding when no requests were made.

1.5 Organisation of Thesis

Chapter 2 provides background and literature review on computer faults, distributed agreement protocols, and remote execution and fault-tolerant systems. Chapter 3 is an exposé of the Remote Execution Manager REM, developed at the University of Victoria, which forms the basis of the implementation of HFPs.

Chapter 4 shows a logical model of an HFP and provides details of an implementation under REM. Descriptions of the other modules necessary for this implementation are in sections in Chapter 4. Chapter 5 provides details on the calibration and tuning of HFPs after the development was completed. Several graphs of timing results are presented therein. Chapter 6 explains testing procedures and provides performance results. Chapter 7 states the conclusions of this research and describes some of its possible applications and future directions.

Extensions and changes to the application programming interface of REM and the source code of most of the system are provided in Appendices.

Chapter 2

Background

2.1 Computer Failures and Faults

Computer failure modes have been loosely grouped together as administrative, software, hardware, and environmental [2].

Administrative failures comprise errors from on-line maintenance, operations and configuration. Software failures are caused by errors in operating systems and application programs that manage to stop the system. Hardware failures are caused when components such as CPUs, disk drives, controllers or power supplies fail. Environmental failures are events such as power outages, loss of communication lines or facilities failures.

Differentiating among these factors may be important for design, repair or statistical purposes, but it serves no purpose from a distributed viewpoint. To a remote, unaffected computer waiting for a reply, these failures appear in the form of incorrect, late or missing results. In a distributed environment, the failure can be detected and compensated for.

2.1.1 Fault types and definitions

Fault types have been defined differently by a number of authors. Additionally, the faults have been classified using different mechanisms.

Ezhilchelvan and Shrivastava classify computer faults by their characteristics into timing, timely, commission and omission faults [3]. The faults from their basic classification are of interest:

1. a correct value that was late (timing),
2. an incorrect value that was on time (timely),
3. an incorrect value that was late (commission),
4. a value when none was expected (commission),
5. no value when one was expected (omission).

The authors fashion a lattice from the characteristics of the faults. An omission fault can be viewed as a special case of either a timing fault or a timely fault. As a timing fault, it is assumed to return a correct value at infinite time. As a timely fault, it is assumed to return an incorrect (null) value on time.

A commission fault is any failure of a computer to meet its specified behaviour. Hence, timing and timely faults are special cases of commission faults. If a computer fails as in item 3 above, it is considered only as commission fault. In an asynchronous environment, it is important to recognize the independence of the incorrect value from the late delivery.

Barborak et al [4], on the other hand, created an ordered classification of faults. Each classification encompasses the previous class and adds another type of fault. A brief description of each fault class follows. The ordering of the classifications are given in a Venn diagram in Figure 2.1.

Fail-Stop Fault is characterised by a processor that ceases operation and informs other processors of its state. This failure model was defined in [13] and is described in some detail in Section 2.2.2.

Crash Fault occurs when a processor loses its internal state and halts.

Omission Fault is characterised by a processor failing to meet a deadline, that is, it never responds to a particular request. This fault is described in detail in [5].

Timing Fault occurs when a processor completes a task before or after its specified time frame, or never[5].

Incorrect Computation Fault occurs when a processor does not produce the correct results in response to correct inputs.

Authenticated Byzantine Faults are malicious or arbitrary faults where the processors are always able to determine if messages are authentic [9].

Byzantine Faults are the set of all possible faults including processors being able to disguise themselves as other processors when communicating [9]. The Byzantine Generals Problem is discussed in Section 2.3.1.

The definitions of these errors is carefully worded so that each classification is a superset of the previous classification. At the heart of the classification system is the **Fail-Stop Fault**. This fault is unlikely to be generated by anything but a special purpose device [14]. The faults addressed by this research are encompassed by the **Incorrect Computation Fault**. By the use of replication and a software mechanism, these faults are all mapped to a **Fail-Stop Fault**, as in Figure 2.2. This basic objective allows a user of the system to be notified of a failure in the computing device without handling the intricacies.

In this research, three fault types are explicitly handled:

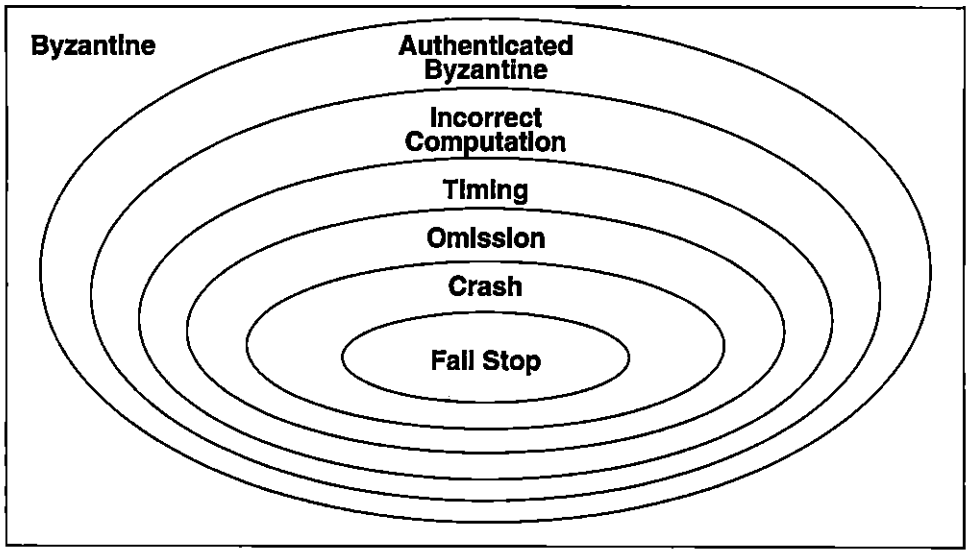

Figure 2.1: An ordered fault classification.

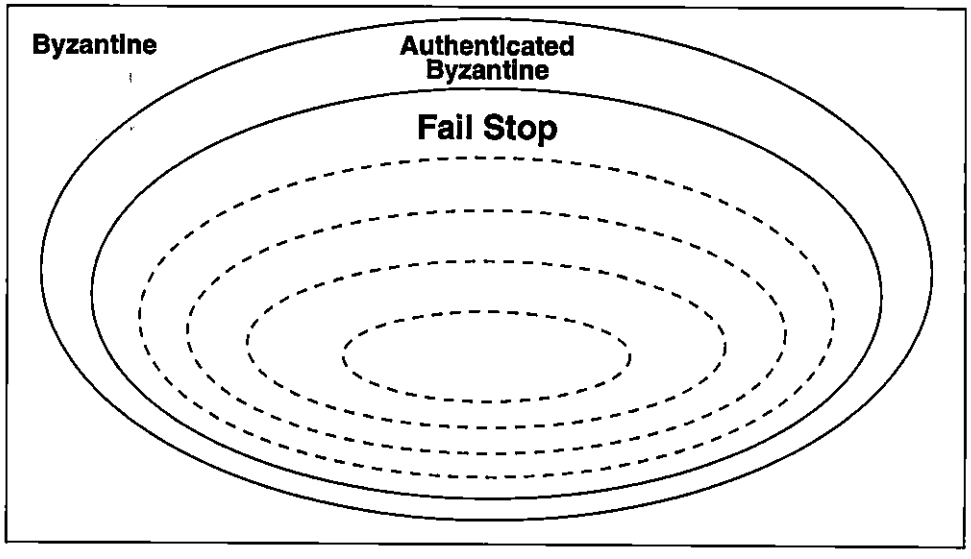

Figure 2.2: Remapped ordered fault classification.

Timing Errors occur when a workstation completes a task after a deadline, or never. Early timing errors are not dealt with in this research. Correctness of a computation has no effect on the determination of a timing error.

Timely Errors are faults in calculation and state changes. These faults are detected by the comparison of replicated computations. This error type is similar to Cristian's *response failure* in [6]. A workstation has to complete a task and return a result before this fault can occur.

Commission Errors are the other types of faults detected by the system. In particular, they are duplicated results and unrequested results.

This fault classification is used for maintaining a list of fault-free processors. The reaction of the system to a faulty workstation is dependent on both the number and type of errors that have occurred.

2.2 Related Work

Many systems that perform remote execution have been proposed, implemented and tested [11,12,14,18–26]. Some of these systems perform fault tolerance through primary/standby schemes or through replication. The remote execution facilities are discussed first, followed by a section on more advanced fault tolerant systems.

2.2.1 Distributed and Remote Execution Systems

Distributed and remote execution facilities allow processing to be performed by remote processors. In general, they do not have fault-tolerant capabilities.

Some remote execution facilities provide fault-tolerance at a very rudimentary level. The fault-tolerance is a mapping of a partial failure to a total failure, emulating

a uniprocessor environment. This generally takes the form of a notification that the remote host has died or a UNIX style signal, such as SIGCHLD is sent to the originating process or client. In one case, the client program is aborted by the operating system when a remote failure is detected [20].

The form of the notification is unimportant. The significant common factor amongst these systems is that the computation that was underway is terminated in a way that is difficult to recover. A failure of some part of the distributed system will cause a total failure of the user application. While this is reminiscent of a uniprocessor environment, it is an undesirable behaviour and is avoidable in a distributed system.

Amoeba

The Amoeba [18] distributed operating system was developed at the Vrije Universiteit (VU) in Amsterdam, Netherlands and is currently under joint development with Centrum voor Wiskunde en Informatica (CWI) also in Amsterdam. This research project is designed to seamlessly connect multiple computers to present the user with single large virtual machine to perform computing. All interprocess communication is handled using an implementation of RPC[25].

Users interface with the Amoeba system using fast diskless workstations dedicated to running editors or other tasks that require fast interactive service. The CPU pool consists of a group of processors under the control of the process server and are used as compute servers. Process assignment to servers is static. All compute servers are under the control of the process server, so load anomalies are rare. There is no need to migrate processes, since users are not running processes on each others' machines.

Butler

Butler [21] was an ambitious distributed execution facility at Carnegie-Mellon University. It was run in 1987 on a network of 350 heterogeneous workstations running the 4.2 BSD release of UNIX. It allows the use of remote workstations for CPU intensive tasks. Butler will run application programs compiled for non-distributed environments.

Butler does not perform process migration. It claims an idle machine for use and starts a process on it. If a user of a machine reclaims it by logging on at the console, guest processes are evicted promptly. Butler uses the network window management facilities of Andrew [27] to notify the user who spawned the guest process. The guest process is aborted and needs to be restarted by the user.

Parallel and concurrent execution were reported as being in an experimental stage in [21]. There is no process replication or primary/standby recovery to complete aborted processes.

Condor

The Condor [23] system provides remote execution capability with process migration. Its scheduling algorithm distributes remote capacity by identifying idle workstations and scheduling background jobs on them.

Process migration is accomplished by checkpointing a process' state. The process' state consists of the executable image, data segments, any dynamically allocated memory, stack, CPU registers, the state of all open files (and devices) and any outstanding messages. If the process is terminated due to the shutdown of a remote workstation or intentional termination for system management, it can be restarted from the last checkpoint on some other machine.

Condor provides users with the ability to perform computationally intensive

background jobs that consume months of CPU capacity. Jobs can be moved, stopped and restarted with a minimal loss of intermediate calculations.

Condor does not provide for fault tolerance except in the case of a crashed workstation. Checkpoints are saved to secondary storage, allowing processes to be restarted even in the event of a complete system shutdown. A user of the system may be able to explicitly perform replication for fault tolerance but issues of interprocess communication and local agreement have to be dealt with at the application level. Additionally, no facilities existed for the parallel execution of processes or concurrency control.

2.2.2 Fault Tolerant Systems

The systems or protocols described in this section have some degree of fault tolerance. They use replication, primary/standby, or probes to detect crashes. Primary/standby or probing systems are capable of recovering from crashes but do not explicitly perform error detection. Replicated systems have the additional ability to detect computational errors through the comparison of results.

Remote Procedure Calls

Remote Procedure Calls (RPC) were developed by Birrel and Nelson [25]. RPCs do not allow for direct error handling or explicit concurrency by the user. The design is well suited for replication, as it can be easily hidden from the user but there is no allowance for parallel or concurrent processing. The calling process must wait until a call returns in order to preserve the semantics of a procedure call.

Fault tolerant RPCs have been proposed and implemented [28]. This work requires replicated copies of a remote server to be aware of each others' presence and that they communicate with each other to determine replica functionality. The

authors also created an ordering amongst the replicas. The primary replica is responsible for communicating with the caller and returning the result of a calculation. If the primary replica fails, the next replica becomes the primary and performs the function of the primary.

The return value from a fault tolerant RPC is the result of one calculation, that of the primary. The results of all other replicas are discarded and the opportunity to perform fault detection and masking is lost. The overhead associated with communication between replicas shows a linear slowdown in response time as the number of replicas grows.

Fail Stop Processors

Schneider describes an implementation of a *fail-stop processor (FSP)* in [14]. Schneider's FSPs have the following properties: Halt on Failure, Failure Status, and Stable Storage. The concept of an FSP was first proposed in [13] and has been cited as a failure model by many authors [4,12,15–17].

Schneider's proposed implementation of FSPs uses a redundancy of physical processors in a synchronous environment to detect failures. When an error is detected, the user is notified and the FSP is stopped. The Stable Storage property provides partial results in the form of a task state if the user wishes to restart the computation.

Schneider's implementation uses two types of processes: *s-processes* and *p-processes*. S-processes are used to store non-volatile data produced and consumed by p-processes. *Interactive consistency* [9] is maintained amongst the s-processes for the requests from p-processes. P-processes perform the processing of the application task and run synchronously on processors with synchronized clocks. In order to withstand k faults, a single FSP needs to use $k+1$ p-processes and $2k+1$ s-processes.

There are impediments to the implementation of FSPs since they require extensive hardware support and use expensive interactive agreement protocols to store results and distribute information. These factors alone preclude the practical implementation of FSPs in most computing environments available today. In addition, there is an undesirable side effects that will occur in the implementation of Schneider's FSP. A single fault in one of the p-processes of an FSP will cause it to fail. The failures will become more frequent as k, the fault tolerance level, increases [29].

ISIS

Birman describes implementing *k-resilient* objects in the *ISIS* system [15]. Such an object is distributed across sites and is guaranteed to remain available despite up to k site failures. Additionally, the object's operations are guaranteed to progress to completion. The implementation is generated automatically from the object specifications and requires no more effort to program than a nonresilient nondistributed object.

The method of the replication uses cold standby processes and checkpoints. The active site is called the coordinator and the cold, or inactive, standbys are called cohorts. When the system is running, checkpoints and retained results are passed to the cohorts from the coordinator. When a coordinator fails, one of the cohorts takes over as the new coordinator and, using the retained results, continues execution from the last checkpoint. Standbys use very little system resources when they are inactive and, in the absence of failures, are highly efficient.

This work assumes "processors satisfy a *fail-stop* assumption: that they fail by halting and do not produce spurious messages, and that active sites are notified of the failure ..." and references [13]. A footnote on this work hints at implementing software failure detectors to obtain an abstraction of fail-stop processors. However, the user manual accompanying a subsequent release of the *ISIS* system contained

no mention of an implementation of FSPs [30].

Replicated Distributed Programs

Replicated distributed programs by Cooper [12] use a many-to-many relationship between servers and clients. A troupe is a set of replicas of a module where each replica executes on a separate machine. Members of a troupe are unaware of each others' existence and do not intercommunicate.

Each *calling* module can be replicated independently of the *called* module. Each called replica is accessible by each of the calling replicas. Calling replicas have many called replicas and called replicas have many calling replicas, hence the many-to-many relationship. All interaction between troupes is performed by replicated procedure calls and returns.

Cooper assumes each replica runs on an FSP and references Schlichting and Schneider [13] as the definition of the processor. He claims that if the processors were not fail-stop, replicas would have to engage in expensive Byzantine agreement on the contents of incoming messages, but states Byzantine agreement could be added to the algorithms presented. Unfortunately, that would contradict his model of independant, non-communicating troupe members. There are two other contradictions that arise out of the assumption of FSPs.

FSPs, as described by Schneider in [14], use Byzantine agreement to decide on the contents of their input and to perform other actions. This contradiction may be dispelled by *assuming* that processors will fail only by stopping rather than actually using an FSP, as in Birman's work.

Second, the default behaviour of the client of a server troupe is to collect all of the responses from the replicas and then check if they are identical. Cooper terms this *unanimous* behaviour and claims it provides both error detection and transparent error correction. However, under the assumption that replicas run on

fail-stop processors, collating returning messages is meaningless. If each replica received identical inputs and behaves deterministically, then each will return the same answer or, if it has failed, no answer at all.

2.3 Decision Making

2.3.1 Distributed Agreements

A large amount of research has been done on distributed agreement algorithms [9, 31–39]. Distributed agreement allows participants to come to a common conclusion in an autonomous manner. There are three shortcomings present in most of these algorithms:

1. a high number of messages must be exchanged;
2. each participant must be able to communicate with all other participants;
3. the algorithms are complex.

A selection of some of the distributed agreement algorithms is presented below for completeness.

Byzantine Generals Problem

The classic distributed agreement problem and some provably correct solutions were presented by Lamport et al [9]. The protocols can withstand arbitrary faults which include malicious manipulation by participants or intruders.

The solution requires recursion or repetition of the algorithm based on the maximum number of faults tolerated. The number of faults tolerated is also limited by the number of participants. In order to withstand k faults, there must be at least $3k + 1$ participants, in general.

More efficient solutions to this formulation of the problem have also been researched [31]. Solutions to reduce the number of messages exchanged have also been proposed. Randomized byzantine [32–35] and early stopping [36] algorithms have been presented. The randomized solutions use probabilistic protocols instead of deterministic and have lower expected order of execution but are not guaranteed to terminate in that order. The early stopping algorithms are only effective under some conditions and, hence, may not stop early.

All of these solutions may still require a large number of messages to be exchanged. They all still require explicit knowledge of all participants and their intercommunication.

Distributed Consensus

Distributed consensus is not as robust as solutions to the Byzantine agreement. Application of this approach to distributed agreement is usually accompanied by assumptions as to the robustness of the basic hardware and software. Typically, the authors [37,38] assume a reliable network delivery system, a stable file system and that the object being agreed or operated upon was already in some consistent state before the current operations began.

None of the authors explicitly permitted the data items in question to become corrupted. The concept of two supposedly identical pieces of data with identical timestamps actually being different was never entertained.

The papers quoted below deal explicitly with databases and mutual exclusion. The principles of operations can be applied equally well to consistency amongst distributed processing elements rather than distributed data elements or critical sections.

Weighted Voting Weighted voting [37] allows a system to maintain multiple copies of data. Users can make and keep both local and remote copies to deal with security and performance issues. Each copy is called a representative and is assigned a variable number of read and write votes. The mechanism allows remote copies to remain consistent with local copies through the use of quorums. A read or write quorum is established by summing the pertinent votes of each representative. If quorum is established the operation can proceed, otherwise the user is informed that the operation has failed.

Majority Consensus Majority consensus [38] allows multiple copies of databases stored at remote locations to remain consistent. The majority consensus algorithm also allows multiple users to read and update the distributed database. This work uses a series of rules as its algorithm to maintain consistency. It is not capable of withstanding Byzantine faults and assumes that applications and other participants only make real requests. All data in the system is timestamped and the timestamps recorded with the data. Most importantly, when variables are read from a local copy, their timestamps are also read and passed along with any subsequent update requests.

The most important rules are the *voting* and the *request resolution* rules. Voting rules allow participants to *reject, accept or pass* on an update request. After voting, participants perform the request resolution. A *reject* vote means the update is rejected and voting is stopped, ie., a veto. If a majority of *accept* votes was formed, then the update proceeds and all other participants are notified and updated. If the request was neither rejected nor a majority formed, the request and the tally of votes is passed on to the some participant who hasn't voted. The *pass* vote is used to resolve concurrent requests for updates on the same item.

Other rules deal with communication models, internal update procedures and timestamp generation[39].

2.3.2 N-Modular Redundancy

N-Modular redundancy (NMR) has been an attractive concept in the field of fault-tolerant computing since its introduction by von Neumann in 1956 [8]. His original proposal introduced triple modular redundancy and the interest and variations have grown over time. The basic concept is to run N identical modules synchronously with the same program. The final output is then collated from the N individual outputs from each module. The collection of the outputs and decision making is performed by a comparator. NMR systems are usually incapable of withstanding faults in comparators.

Several different methods of output selection are common with well known error tolerances. These variants are described below:

NMRV: the standard NMR uses simple voting to determine the correct answer. It can withstand $\lfloor (N-1)/2 \rfloor$ errors.

NMRB: has backup spares. If there are s spares, then there are $N - s = r$ running modules. NMRB can withstand $\lfloor (r-1)/2 \rfloor$ concurrent faults and $\lfloor (r-1)/2 \rfloor + s$ sequential faults.

NMRA: performs adaptive voting. After a module is determined to be faulty, it can no longer vote. This variant has excellent sequential fault tolerance, up to $N-2$ and can also withstand $\lfloor (N-1)/2 \rfloor$ concurrent faults.

Additionally, other variants have been proposed such as NMR Sift-out (NMRS) [40] and NMR with Comparison (NMRC) [41]. NMRS can withstand $N-2$ sequential or concurrent faults by using $N(N-1)/2$ pairwise comparators. NMRC is an extension of NMRS that uses a variable interconnection topology which has a configurable fault tolerance from 0 to $N-2$.

The greatest drawback to this approach to fault tolerance is its reliance on

synchronous hardware. Nonetheless, the ideas presented are attractive and were adapted to the asynchronous workstation environment.

2.4 k-out-of-n:G Systems

A k-out-of-n:G system [42] has n replicated identical components and a controller or comparator. There are several sub-classes of these system but they all have the same basic property. A k-out-of-n:G system gives satisfactory performance as long as k out of the n units are functional. This approach to reliability is widely used in civil, mechanical and electrical engineering. Some of the uses include electrical power generation systems, multiple regulated power supplies, bolts holding machine members, and hydraulic pumps and valves.

The controller in the system is not replicated and is always assumed to be functional. It combines results from the functional units and may provide a single output value. The controller may be of higher quality or performing a less arduous task than the individual components, so the assumption of higher reliability is valid for many systems.

This model of behaviour is applicable to discrete systems. A number of discrete replicated components each supplies a single output value. The comparator collects these values and applies a reduction rule or algorithm to produce a single output value. As long as at least k of the inputs to the comparator are valid, then the system is considered to be functional, otherwise it has failed. The failure model is used in this research and is returned to in Chapter 4.

2.5 Reliability and Availability

The reliability, R, of a component is a function of time. It is the conditional probability that the component will function as specified over the time interval $[t_0, t]$, given that the system was functioning at time t_0. The availability, A, of a component is an instantaneous measure. It is the probability that a component is functioning according to specifications and is available for use at some instant in time t [43].

Industry standard measures of reliability and availability for components have been determined and published [43,44]. Given that R is the reliability of a single component over some time interval, then the reliability of a k-out-of-n:G system over the same interval is [44] :

$$R_s = \sum_{i=k}^{n} \binom{n}{i} R^i (1-R)^{n-i}.$$

Note that R_s is not always strictly greater than R and R_s is a function of R, n and k [29,44]. Creating $R_s > R$ is dependent on the selection of parameters for the expected time interval.

Similarly, the availability of a system of n components each with an availability of A is given by:

$$A_s = \sum_{i=k}^{n} \binom{n}{i} A^i (1-A)^{n-i}.$$

Additionally, the same caveats stated above apply to availability.

However, this measure of availability assumes a fixed set of components make up each system. As will be seen later, the components that make up an HFP are selected dynamically from a pool of components that are known to be available. This radically improves the availability of an HFP over a system that has a fixed set of components.

Chapter 3

REM

The implementation of HFPs was performed using the Remote Execution Manager (REM) developed at the University of Victoria. REM's goals are a subset of the HFP objectives stated in the introduction. It is appropriate to describe the REM system at this point.

REM was developed with the objective of utilizing the idle processing power of local workstations [45]. It provides remote execution, process management, fault tolerance, load sharing and interprocess communication services in a transparent manner. REM provides these services through an application programming interface (API).

REM was chosen as the starting point for a highly reliable, highly available virtual processor. It already had the basic mechanisms necessary for an implementation, such as interprocess communication and process replication.

REM replicates processes for fault tolerance. It uses the first response returned from any of the replicas and ignores all others. This approach allowed for a simple and efficient implementation of distributed processing and provided protection against remote crashes.

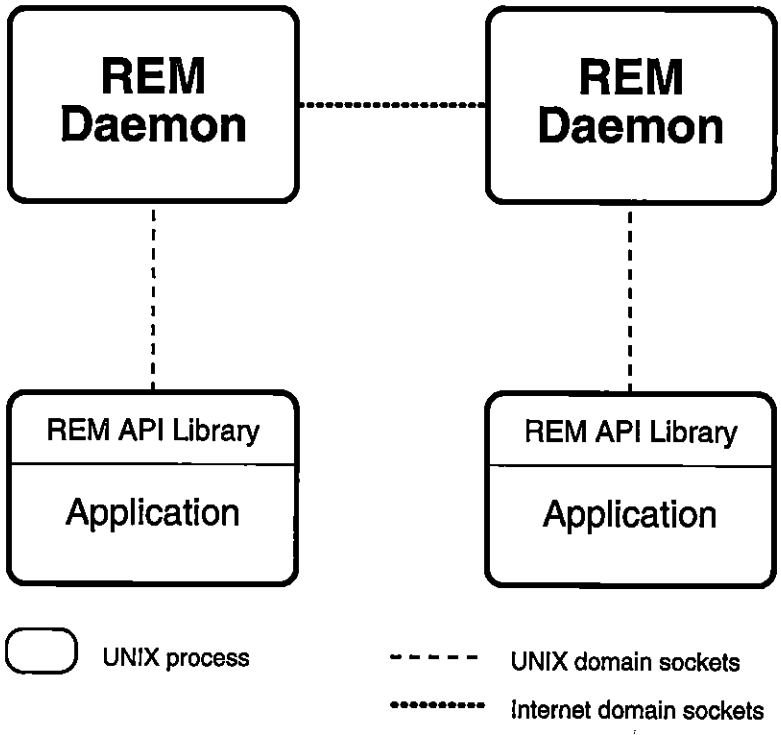

Figure 3.1: Overview of the REM system.

3.1 REM Architecture

An element of the REM system consists of two parts: a daemon that runs under UNIX on a workstation and a collection of subroutines that form the API library. The API library is bound to the user program at link time.

The daemons communicate with each other through Internet domain sockets. Each daemon is responsible for the creation, management and destruction of processes on its machine. User processes communicate with the API library through function calls. The API library communicates with the REM daemon on the local machine at execution time through a UNIX domain socket. An overview the REM system and its communication paradigm is presented in Figure 3.1.

3.2 REM Terminology

The definitions for the terminology for the REM system are given below. Some of the terms are in wide use in the literature but their definitions are given below to prevent confusion about their intended meaning.

REM application A collection of distributed, communicating processes cooperating on the completion of a task. An application consists of a parent process and any number of child processes.

parent process A UNIX process that spawns child processes and performs task allocation to those processes. The parent process is the first process to be started in an REM application. The parent process also synchronizes results from the child processes.

child process A virtual process spawned by a parent process. If a parent process has spawned more than one child process, the child processes are unaware of each other. The current REM system precludes communication directly between child processes. A child process is virtual since it actually consists of a number of replicated processes. Each replicated process runs asynchronously on a possibly separate workstation using identical data.

replica A single copy of a process executing on one workstation. The replicas that form a child process are unaware of each other. They execute in isolation returning any results by message passing to the parent process.

suite A collection of REM daemons running on separate workstations. Each suite has a name and the REM daemon is passed its suite name when it is invoked. REM daemons in the same suite are explicitly aware of each other but are unaware of REM daemons in other suites.

client workstation The workstation that runs the parent process of an REM application. Only the parent process executes on the client workstation.

remote workstation The workstation(s) that run the replicas of child processes of an REM application. Replicas of a child process may run on the same remote workstation or on separate remote workstations depending on the decisions made by the load balancing algorithm. Replicas from distinct child processes may also be run on the same remote workstation for the same reason.

Chapter 4

Virtual Halt on Failure Processor

The design of a highly available and highly reliable distributed computing system must include provisions for fault tolerance. The fault tolerance previously provided by the REM system was inadequate in some areas. The REM system did not detect errors created by faulty workstations nor did it explicitly manage crashed workstations.

REM was extended with a comparison mechanism. This addition, in conjunction with its replication features, made the system n-modular redundant with a single comparator. The features of the comparator are its voting mechanism and performance enhancements. These are discussed in Section 4.2. Additionally, the ability to detect errors allowed the system to decide if it should halt.

A **Halt on Failure Processor (HFP)** is a virtual processor that satisfies the following properties:

halt on failure A processor will halt rather than deliver an erroneous response to a user.

failure status A processor, whether failed or not, will report its failure status.

A blacklist module was added to provide the system with a history mechanism.

This module is responsible for deciding when a workstation is actually faulty and not overloaded or having suffered some isolated fault. It also provides new HFPs some assurance that they will not receive a known faulty workstation to perform calculations with.

4.1 Logical Model

An HFP is composed of two basic components, one of which may be replicated. There is one *client agent* and one or more *replicas*. The logical model is presented graphically in Figure 4.1. Each replica is unaware of any other replicas. The client agent performs replica creation, location, communication, error detection and masking, and failure detection. The client agent is not fully replicated and is assumed to be failure free, as are voters in NMR and comparators in k-out-of-n:G systems.

The replicated processes are distributed amongst the workstations and function in parallel. The parent process requests the creation of child process from the client agent. The client agent creates replicas of the child process to build the HFP. Neither the parent process nor the replicas are explicitly aware of the client agent. The parent appears to communicate with a single child process and the replicas all appear to send replies directly to a parent process.

Communication between the parent process and the child process is message based. The messages follow a request/reply paradigm. All messages are passed through the client agent. When a parent sends a request destined for a child process, the client agent sends a copy to each of the replicas. When the replicas send replies back to the parent process, they are received by the client agent and verified against each other for correctness.

The workstation hosting each replica is known to the client agent. Incorrect and late replies indicate a faulty, slow, or halted workstation. The result of each

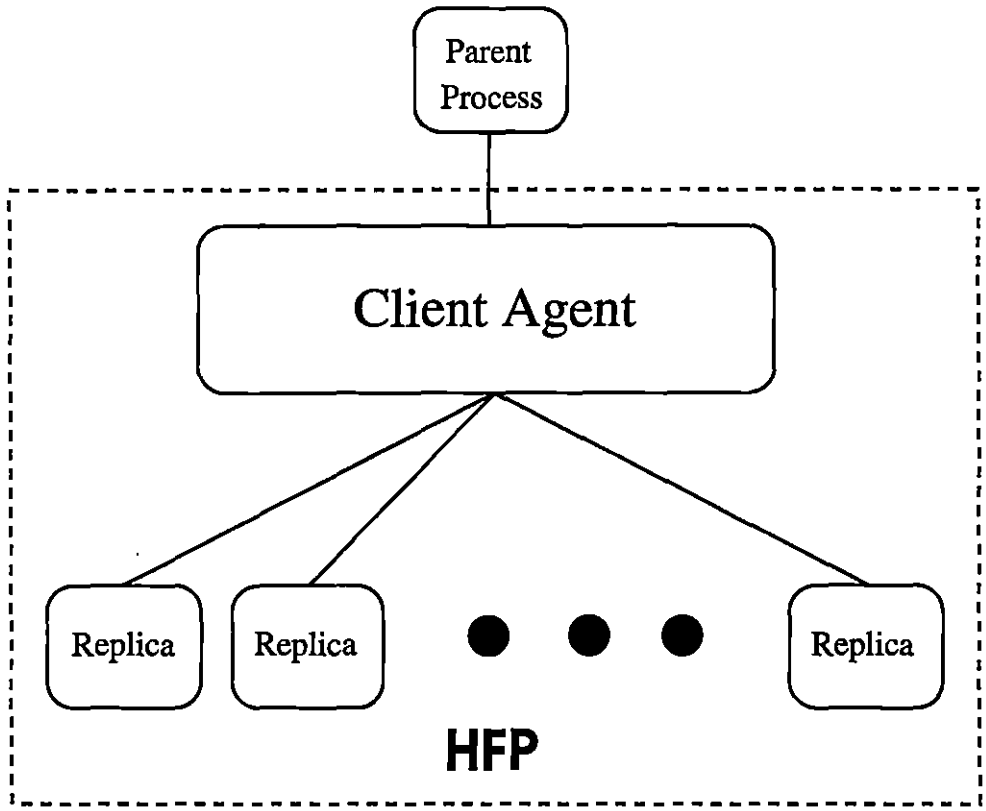

Figure 4.1: Logical view of an HFP

workstation is compared against its peers and, if it does not perform as expected, it is deleted as a candidate for hosting new replicas. If the error level, determined by the number of malfunctioning replicas, rises above a given tolerance, the HFP halts.

A halted HFP will not send messages between a parent process and its child process. A request to a halted HFP will immediately result in a failure response. After halting, replies received from remote replicas are ignored by an HFP. Remote replicas are destroyed as soon as the client agent detects a failure and the host workstation of the replica is notified. The parent process is left alive and running to perform whatever remedial action the application wishes. The application is free to request a new HFP to replace the halted one. A replacement HFP may use other non-faulty workstations not selected for the original.

4.2 Replication and Voting Mechanism

The number of replicas and the voting level in an HFP is controlled by the user. When the REM daemons are started, the user supplies n, the number of replicas, and k, the number of identical replies needed to form a valid response. These parameters affect fault detection capability, overall load factors, system response times and probability of HFP failure. Some interaction of these factors is explored in Chapter 6.

The concept of replicating the programs and running them independently is very similar to NMR with the exception that the modules run asynchronously. Due to this asynchrony, the standard voting mechanisms of NMR become inadequate. Not all of the replies will be available when the first is delivered, due to different load factors and normal communication delays. A timeout value, based on the execution time of the first reply, is determined and the replies that have been received are compared at timeout. This forces synchronization of the replicas that have replied [46].

Performance is enhanced with early synchronization established at two points. If k identical replies are received prior to timeout, consensus is deemed to have occurred, a response is returned to the user and the replicas that have replied are synchronized [46]. If n replies are received prior to timeout, then synchronization of all replicas has occurred. This approach was taken to enhance the performance of the system, shorten response times and make better use of the HFP resources. As will be seen in Chapter 5, timeout values were necessarily set considerably longer than just the execution times would indicate.

The value k is also used to determine if the system is functioning normally, as in a k-out-of-n:G system. As long as an HFP receives k identical replies from the n replicas, the HFP is considered to be functional. If less than k identical replies have been received by timeout, then the HFP has failed.

4.2.1 Error Tolerance and Voting

Error tolerance is directly affected by the user parameters n and k. In this manner, the user controls all aspects of replication and fault tolerance. The error tolerance is $d = n - k$ for $k > \lceil (n-1)/2 \rceil$. As long as d or fewer errors have occurred, the HFP will return a correct answer. If more than d errors occur, the HFP will halt and report failure.

To implement an NMRV system, for example, the user selects d, the maximum number of errors expected in the system. The system needs $k = d + 1$ correct replies to "outvote" d possibly erroneous or missing replies. The minimum number of replicas needed is then

$$\begin{aligned} n &= k + d \\ &= d + 1 + d \\ &= 2d + 1. \end{aligned}$$

Using $k = n$ causes the system to send a response iff all the replies are identical and have arrived on time. Using $k = 1$ allows the first reply received to be returned to the user as a response.

4.3 Implementation under REM

An implementation of HFPs was made using the REM system. Aside from implementing the HFP described in Section 4.1, some other objectives include backward compatibility with REM and a usable programming interface.

Definitions of the terms used to describe an HFP follow. The terms defined for REM in Chapter 3 are also used in the description of this implementation.

Service Request A message sent to an HFP from a parent process.

Reply A message sent from a replica to the client agent of an HFP destined for the parent process.

Valid Response A copy of a reply that has been validated as correct by the client agent of an HFP and returned to a parent process. A service request is said to be *satisfied* when a valid response has been sent to a parent process.

Stale Reply A reply that is received for a service request after it has been satisfied.

The descriptions of the extensions to REM system are divided into two sections. The first section is devoted to changes that affect the API and other routines that are linked to the application program. The second is devoted to the changes and enhancements made to the REM daemon.

4.3.1 Changes to the REM API

Minor changes were needed to this part of the REM system. Some message types were extended in size and created a new alignment of data structures. For the most

part, recompilation was the only step necessary to adapt the system. A new function was added to the API and an existing function was adapted to return a new value. The original interface [47] remained intact and functional.

Service Requests

The original REM API function to send a service request to a remote process was inadequate for sending requests to an HFP. A new function was created to communicate a service request to an HFP. The function is similar in syntax to the REM request for service but the code sends an HFP service request type. The function returns a value that indicates whether the HFP successfully sent the service request or not. A description of this function is in Appendix A.

Response Delivery

An HFP delivers a correct response or a failure indicator. A mechanism for delivering data was already present in REM, so this part of the API did not need many changes. The code was extended to return a compile time constant, `HFP_USER_FAILURE`, when a failure message from the daemon was detected instead of response data. Changes were necessary in several locations to check for a new message type from the daemon. Details of these changes are included in Appendix A.

4.3.2 Changes and Extensions to the REM Daemon

Process Creation

Minor changes were needed to the REM daemon routines that created and maintained remote processes. These were largely limited to changes in data structures and additions to process tables. Some new code was implemented to perform initialization of these data structures.

Replicas, and the remote daemons that manage them, have no awareness of the HFP. The replicas can be invoked and used by HFPs or standard REM applications.

Service Requests

The code to perform the reception of a service request for an HFP needed extension over the standard REM service request. A data structure, the request header in Figure 4.3, is allocated and initialized for the request and a timer slot is reserved for use. The request data is then sent to the remote replicas as per the REM system.

Reply Reception

Several types of messages are received by the REM daemons from two sources. This section deals with the reception of messages by a daemon on the client workstation from a daemon on a remote workstation. The messages of interest here are replies destined for the parent process on the client machine, sent through the remote daemon by a replica. All messages have a type associated with them. When a reply type is detected by the daemon, the message is passed to the reply reception code.

Many extensions were necessary to the reply reception code of REM. The original system simply passed on the first reply received from any of the replicas performing identical calculations. REM first checked if it had already received a reply from one of the other replicas of the child process. If it had already received one and passed it on to the parent process, the current reply was simply ignored.

The replies in an HFP have to be considered more carefully. An algorithm of the behaviour of this code is given in Figure 4.2. A reply is first tested to see if it is actually expected, i.e., to verify that a replica had been spawned on the workstation that sent the reply and a service request had been made of this replica. If no reply was expected, then a request to blacklist the originating workstation is

made. Section 4.4 describes the details of handling this request. If the reply is valid then it is compared against all the other replies received so far.

The first reply back causes an expiration timer for the request to be set. The elapsed time of the first reply and the load information of all the workstations processing the same request are used to determine the expiration time of the request. Chapter 5 contains the details on how this time was determined.

If no replies have been received, then a list of replies is started. This reply is stored along with the number of identical replies received and the replying workstation heads the list of workstations having sent that reply. If a list of replies already exists, then the reply is compared against all others received. When a match is found, the number of identical replies is incremented and the replying workstation is added to the list. If no match is found, the new reply is added to the list of replies for this request in the same way that the first reply started the list. This approach reduces the number of copies of replies that need to be stored. Figure 4.3 provides a graphical representation of the data structure used to store the replies.

A valid response is formed when the number of identical replies received for a service request attains the threshold specified by the user. If the number of identical replies has not reached this threshold, the reply reception is deemed incomplete. As soon as a valid response can be formed, a copy of the reply is sent immediately to the parent process and the service request is flagged as being satisfied. The system does not wait for all replies to be received. The list of replies is left intact for further processing by the reply reception code and at timer expiration.

Stale Replies Once a service request has been satisfied, other replicas may still return replies. Stale replies are treated in the same manner as normal replies for unsatisfied service requests, except their reception will not cause another response to be sent to the parent process. The reply and the replying workstation are added to the data structures until the timer expires on the service request.

```
HFP_receive_reply ( workstation, reply )
{
if there is no request for the reply then
   { blacklist( workstation, COMMISSION_ERROR )  }
else if the request has expired then
   {
   /* reply is from a late workstation */
   deblacklist( workstation );
   if the reply does not match the response sent to the user then
     { blacklist (workstation, TIMELY_ERROR) }
   if all workstation have sent replies then
     { free correct reply; free request }
   }
else /* reply is for an unexpired request */
   {
   if this is the first reply received for the request then
     { set the HFP timer for the request }

   if the reply matches some other reply for this request then {
     increment identical count for this reply;
     record workstation;
     }
   else /* the reply is unique for the request */
     {
     set identical count to 1 for this reply;
     record workstation;
     }

   if sufficient identical replies have been received and
     the request has not been responded to then
     { send (HFP_USER_SUCCESS, reply) to user }
   }
}
```

Figure 4.2: Reply Reception Algorithm.

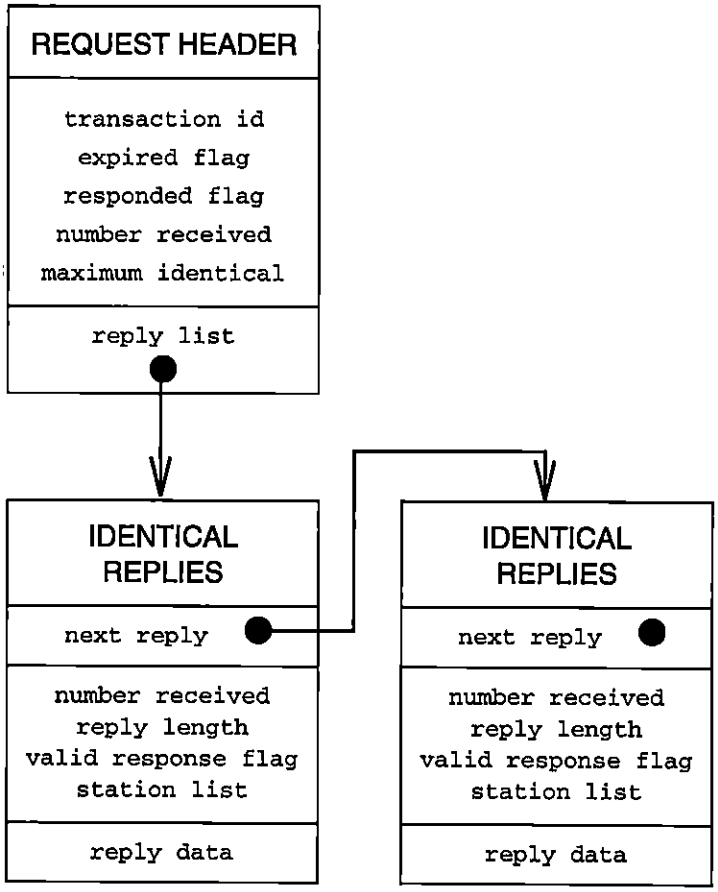

Figure 4.3: Data structure to store HFP replies

If the timer has expired on service request and a reply is subsequently received from a replica, the reply reception code sends a request to the blacklist module to remove the sending workstation from the blacklist. The stale reply is checked for correctness against the valid response already sent to the parent process. If it is different, then a request to blacklist the sending workstation for an incorrect reply is made. This process is similar to the action taken upon timer expiration, as detailed in Section 4.6. Note that the blacklist module decides whether a workstation actually goes on or off the blacklist based on the requests received from the timer module and the reply reception code.

4.4 Blacklist Module

The reliability of workstations is monitored by keeping a list of workstations that have suffered errors. When an error is detected, the reliability of a workstation is in doubt. When a workstation that was participating in an HFP becomes an undesirable candidate for future processing, it is blacklisted by the system.

The blacklist module was designed and implemented to maintain the list of undesirable workstations. Aside from maintaining the blacklist of workstations, the module also decides if and when a workstation should be blacklisted.

The blacklist module deals with the three kinds of errors described in Chapter 2:

1. commission errors,
2. timely errors,
3. timing errors.

Blacklisting is performed differently for each type of error. In addition, two different blacklists are maintained, a permanent blacklist and a temporary one.

Permanent blacklisting of workstation entails removing it from list of known workstations. The internet domain socket between the client workstation and the workstation is closed. All other workstations are informed of the decision and they also close their connections to the blacklisted workstation. This decision is unilateral on the part of the client workstation. As sockets remain open only if both processes maintain them, the blacklisted workstation can not refuse to close the communication channel. The blacklisted workstation is removed from all internal data structures such as process tables and socket lists.

Temporary blacklisting is a less severe action. A local data structure is altered to record the temporary blacklisting. All requests for a new workstation to run a replica will be diverted away from temporarily blacklisted workstations. Any replicas believed to be running on temporarily blacklisted workstations will continue to have requests for processing sent to them.

Temporary blacklisting is easily reversed and was designed to be used for workstations that become unavailable intermittently. Section 4.4.3 contains the details on implementing temporary blacklisting. This type of blacklisting was designed to compensate for workstations that suffer load spikes and other temporary load anomalies. That is, a temporary increase in load that occur after a request is dispatched to a replica.

For example, when a user logs in on the console of a workstation, it generally causes a load spike. These were a common event during testing. After logging in, the X server is started then a window manager, several shells and programs, such as Xterminals and a display clock, are all started at the same time. The load inflicted on a workstation is high for a brief period of time, one to three minutes, while new processes and shells are forked.

4.4.1 Commission Errors

When the blacklist module is informed that a commission error has occurred on a particular workstation, it permanently blacklists the workstation. This type of error is highly unlikely in correctly functioning programs. It only occurs when a reply is received for a nonexistent service request. A replica can generate this error by sending a reply to a parent process when no request has been made.

Outside of malicious intervention and programming errors, this error occurs when a halted, or extremely overloaded, workstation becomes active and sends a reply for a request that has been deleted. In this case, the workstation is possibly out of synchronization with the client workstation and is removed from the suite. In this manner, any and all replicas on the workstation are deleted from the client and remote workstation.

The workstation can rejoin the suite and becomes available for any new replicas that are spawned.

4.4.2 Timely Errors

When the blacklist module receives notification that a timely error has occurred on a particular workstation, it does not blacklist the workstation immediately. Instead, it assumes a transient error has occurred and increments a count of the timely errors committed. If the count exceeds a certain threshold, currently three, the workstation is assumed to be faulty and is permanently blacklisted.

4.4.3 Timing Errors

Timing errors proved to be the most challenging type of error to detect and recover from. Differentiating between a workstation that is so overloaded that it has not

yet responded and a workstation that has crashed is essentially the halting problem. Waiting indefinitely for a failed workstation to complete is futile. Deciding a workstation has finally failed only to discover it was simply slow is wasteful and undesirable. Deleting each workstation that suffers a temporary load anomaly from a suite would soon leave the suite barren of workstations.

However, for the purposes of performing remote computations, an extremely overloaded workstation is just as useless as a failed workstation. The delays incurred using an overloaded workstation for remote execution outweigh its usefulness. The further an overloaded workstation falls behind, the less reason there is to maintain timer table entries, message queues and various other data structures. Additionally, adding to its workload only aggravates the condition.

When a workstation suffers a timing error, it is temporarily blacklisted. While it is on this blacklist, service requests for any processes currently running on the workstation are sent as usual, allowing those processes to contribute to the HFP if the workstation clears its backlog.

The blacklist module keeps a count of the lag in responses for all late workstations. When a workstation has not sent a response in the expected time frame, the blacklist module increments the lag count for that workstation. The reception of a late reply from a workstation causes its lag count to be decremented. When the lag reaches a given threshold, currently three, the workstation is deemed to be halted, or so slow it is of no use, and is permanently blacklisted.

4.5 Timer Mechanism

In the UNIX environment, there are two choices for timer implementation for users. The first is to set an alarm clock for a specified elapsed time of t seconds. The alarm clock then "rings" after t seconds and a user defined interrupt routine is invoked.

The second method is to select a time interval of t and set an interval timer. Every t seconds thereafter the alarm clock "rings" and invokes a user specified interrupt routine.

The weakness with both of these method is that each process is only allowed one timer. With the former method, when the interrupt routine is invoked and the tasks associated with the expired service request completed, it must then calculate the length of time, t_2, to the next expiration, if any. This time, t_2, is then used to reset the alarm clock. The latter method allows the user to set the interval timer once and then, upon each interval interrupt, poll a list of data structures checking for expired service requests.

An interval timer used to check various data structures already existed inside the REM system. The interval timer calls REM's interrupt routine every two seconds. The interrupt routine performs a C `longjmp()` to the start of a message polling loop in the daemon when it completed its tasks. This effectively clears the interrupt condition.

The original REM code was altered to perform more efficient signal handling. The interval timer interrupt handler was being reset inside the main daemon loop at each iteration. Additionally, messages were being sent and received while the interval timer was enabled. Interrupting system based input or output requires special programming techniques that were not implemented in REM, so the message transmission was moved to another location in the system. The system stability has improved since these changes were implemented.

4.6 HFP Timeout Testing

A subroutine call, `check_HFP`, was added to the timer interrupt routine to service HFP timeouts. The interval of two seconds used for the REM system was deemed

an adequate granularity for HFP maintenance. Subsequent tests have shown no behavioural anomalies for this interrupt frequency despite the addition of instructions to maintain HFPs. The timeout testing routine only modifies the data structures associated with the timer and does not directly affect the data structures associated with service request.

The timeout testing routine decides if a service request has expired and initiates management of the data structures of service requests. In this manner, the remaining code need only maintain the data structures of an expired service request and is unconcerned with the notion of time. This code must perform several tasks, including notifying a parent process of failure, flagging the service request as being expired and managing the service request data structures.

4.6.1 Service Request Expiration

After determining that a service request has expired, the system checks if a valid response was returned to the parent process. If a reply was sent to the parent, the service request was satisfied. This is referred to as a normal expiration. However, if no response was sent, then the HFP has failed. The parent process is advised of the failure by the return of a predefined constant, `HFP_USER_FAILURE`, and a null message. This is the critical function of the HFP timeout routine. The remaining function is the maintenance of data structures.

In both the normal and failed cases, the data structures associated with the service request need maintenance. The service request is flagged as having expired, so that stale replies can be handled differently from normal replies in the request reply code. The maintenance of data structures is different for the two cases. Figure 4.4 provides an algorithm of the behaviour of the timer expiration code.

```
timer_expiration( request )
   {
   for each workstation in request that has not sent a reply
      { blacklist ( workstation, TIMING_ERROR ) }

   if no response was sent for this request then
      {
      /* insufficient identical replies have been
       * received to establish a correct reply
       * and nothing was sent to the user
       */
      send HFP_USR_FAILURE to user;
      free all replies; free request;
      halt HFP;
      }
   else
      {
      /* a correct reply was established and
       * a response was sent to the user
       */
      for each workstation that sent an incorrect reply
         { blacklist( workstation , TIMELY_ERROR ) }

      if one or more of the workstations have not sent replies then
         {
         /* keep the request structure for later use
          * keep correct reply to compare with late replies
          */
         free incorrect replies;
         mark the request as expired;  /* for reply reception */
         }
      else
         /* all workstations have responded */
         { free all replies; free request; }
      }
   }
```

Figure 4.4: Timer Expiration Algorithm.

4.6.2 Normal Expiration

In the normal case, a valid response has already been sent to the parent process. This response is considered the correct reply amongst all the different replies received. If any workstation returned a reply other than the one sent to the parent process, a request to blacklist the workstation for a timely error is sent to the blacklist module. The erroneous replies are deleted from the list and the memory deallocated.

If there are any outstanding replies, a request to blacklist the outstanding workstation for a timing error is sent to the blacklist module. A copy of the valid response is kept only if there are outstanding replies. Stale replies can then be checked against the valid response for correctness. The handling of stale replies was previously discussed in detail in Section 4.3.2.

4.6.3 Failed Expiration

If the HFP has failed, a valid response was not sent to the parent process. Few assumptions can be made about the individual workstations in this case. There is no correct response to compare the replies against, so workstations can not be blacklisted for timely errors. Workstations that have not responded to service requests are still blacklisted for timing errors.

The service request data structures are all deallocated. The HFP is flagged as having failed and all future requests for services from this HFP are not performed. The details of message handling by halted HFPs were given in Section 4.1.

Chapter 5

Calibration and Tuning

After inception and testing, the HFP code needed calibration for the network it was running on. The most sensitive part of an HFP is the timer expiration. In particular, if timeout values are set too low, workstations will be needlessly blacklisted and removed from active use in the suite. If timeout values are set too high, system resources, such as timer slots and reply storage, are consumed unnecessarily and user processes may have to wait for long periods of time before discovering that an HFP has failed.

Service requests are issued after a child process is initiated and the results are returned before the child process is terminated. No concern is given to the costs of initiating the remote process nor terminating it. The execution time of a service request is the time to send a message, execute the necessary instructions and return the results. This is referred to as the job execution time. The timeout value should be a function of the job execution time.

No *a priori* information of job execution time is assumed, so an estimate of this time is needed before a timeout value can be set. Several problems stood in the way of obtaining an accurate estimate. The job had to complete on one workstation to get an initial execution time. Execution times are affected by workstation load levels and

load types, so these had to be taken into account. Asynchrony of the workstations and random elements introduced by the network system create variability in the estimates so the times are adjusted accordingly.

These problems are discussed in turn and some solutions are presented. A model of execution is developed and a method of estimation is based on the model. The model's relation to empirical data is presented.

5.1 Load Factors

The UNIX operating system provides a facility to determine the amount of load on the operating system, called a load average. It is described as the average number of processes in the run queue over a time frame. Three load averages are provided by the kernel. They are the averages over the last fifteen minutes, the last five minutes and the last minute. These are referred to as load factors in this research, since they had some serious shortcomings as a measure of average load.

The REM system was already designed to obtain a load factor from each of the workstations in the suite to perform load-balancing. The load factor over the last minute is the only one used. Each daemon obtains the load factor from the operating system for its own workstation and sends it to the other daemons. The load factor is currently updated at six second intervals and only if the change in load factor exceeds a compile time constant.

This constant was changed to zero for purposes of calibration to cause updates in load information to occur every six seconds, if there was any change. The load information used to calibrate the system was as accurate and as recent as the remote host could provide.

The analysis of execution times versus load factors for this work uncovered some anomalies in the UNIX load factors. First, significant lag time occurred between

changes in load and the load factor as reported by UNIX. Second, variations in the load factor were uncovered that could not be attributed to lag time. Ultimately, these variations were traced to system execution that is not reported in the load factor.

5.1.1 Lag Time

The load factor used in the analysis was the average number of jobs in the run queue over the last minute. This is the most instantaneous measure of load that is provided by the system and it should have a maximum lag time of one minute. All other factors being constant, one minute after a new job is started the load factor should have stopped increasing and report the true current load factor.

Analysis of test runs indicate that it takes between three and four minutes for the load factor to stabilize after a constant change in the load on the operating system. For example, if the operating system has a constant background load of 1.8 jobs and a new job is started, the load factor will rise over a period of three to four minutes and eventually stabilize at 2.8 jobs. This example assumes the new job consumes only CPU cycles and does not get blocked performing system calls.

This lag affected the earlier analysis of load versus execution time analysis. Early tests results indicated a slight non-linear relationship between load factor and execution time. After detection of the lag, the relationship became highly linear, as was assumed in the early stages of the analysis.

Figure 5.1 on page 49 clearly shows the effect of lag. A series of test jobs were run on an isolated workstation that had a known load factor of 1.8 before submission of the new job. The execution time and load factor at job completion were recorded for repeated runs of the job. These test jobs did not make any system calls and, aside from the background load, were executed alone, one at a time, on the workstation. The figure shows that the reported load factor approaches the true value of 2.8 as

the execution times become longer. However, the system does not start to report the true load until a job has been present for 200 seconds.

This statistic is critical in the work that follows and was used as reported by UNIX in the calculations. An adjustment for variability is made to compensate for this potential lag.

5.1.2 Load Profiles

Aside from executing arithmetic and logic instructions, a job can also make requests to the operating system. In the UNIX environment, these requests can be made of devices, such as disk, console and communication requests, and system services such as process spawning and synchronization.

When the operating system performs these requests on behalf of a user job, the user job is halted and removed from the run queue and put in a wait queue. The system process executes on behalf of the user job and, when the system process is complete, the results are returned to the user job and it is returned to the run queue. During this time, the user process is not counted in the load average and neither is the system process. The true system load is underestimated when this occurs.

In addition, some system requests often contain some indivisible operations, during which time all other processes, including system process of a lower priority, are denied access to the CPU. An example of this is a `fork()` system call. This call causes a duplicate of the current process to be created and entered into the operating system's process tables under a new process identifier.

While the duplication takes place, the operating system consumes all of the CPU cycles and other processes are starved. This can drastically affect the amount of CPU cycles a job receives and impacts the execution time of the job.

A test was run to determine the impact of load profile on job execution time. The same job was run under similar load factors but different load profiles. The execution

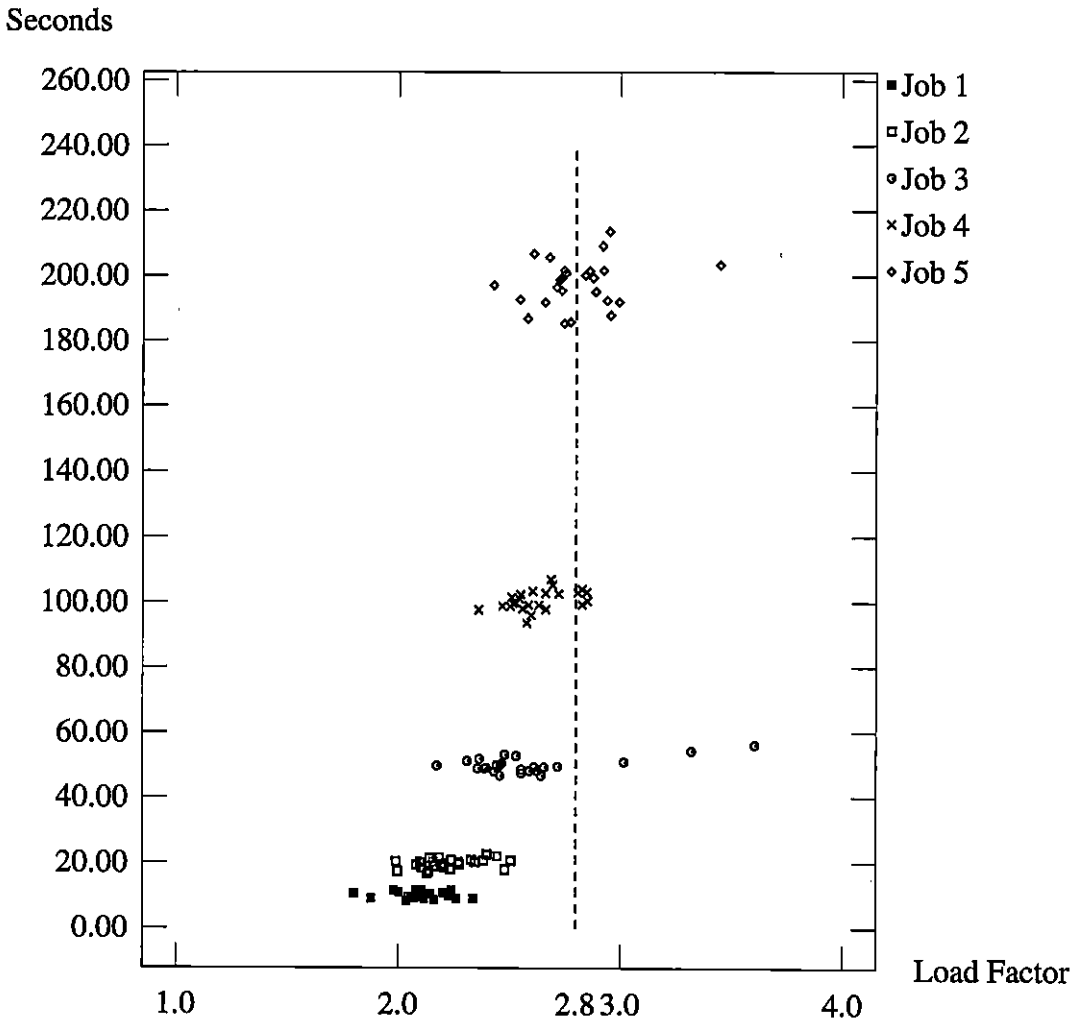

Figure 5.1: Execution time versus Load Factor for a single workstation.

time versus load factor was plotted for each run and compared. An example of this test is Figure 5.2 on page 51. The difference in the execution times of the job under the two different background load profiles can be clearly seen.

The points marked High System Calls on Figure 5.2 had background jobs running where the system consumed approximately 65 to 70 percent of the CPU cycles. The points marked No System Calls had background jobs where the system consumed no appreciable CPU cycles. Differing numbers of these jobs were started on the workstations to provide varying load factors of the same load profile. The impact on the execution time is marked. For example, at a load factor of 6.0, a job executed in approximately 21 seconds when no system calls were being made by the background jobs. In contrast, the same job took 120 to 130 seconds to complete at *the same load factor* when system calls were occurring.

Different load profiles can produce the same reported load factors but very different execution times for a job. When the same job was run on different workstations with differing load profiles, marked differences in execution time were recorded. Load factors alone are insufficient information for estimating the execution times of jobs on a workstation.

5.2 Estimating Job Execution Times

Determining the exact execution time of a job on a computer requires indepth knowledge of the job, exact performance specifications of the computer and *a priori* knowledge of all other tasks that may interrupt the job. None of these factors are immediately available for the environment that HFPs were developed in nor are they generally available to application programmers. However, enough information is available to construct an estimate of the job execution time.

The job scheduling algorithm used by SunOS, Sun's UNIX system for workstations, is plumbed. A simplified model of a job scheduling algorithm is developed

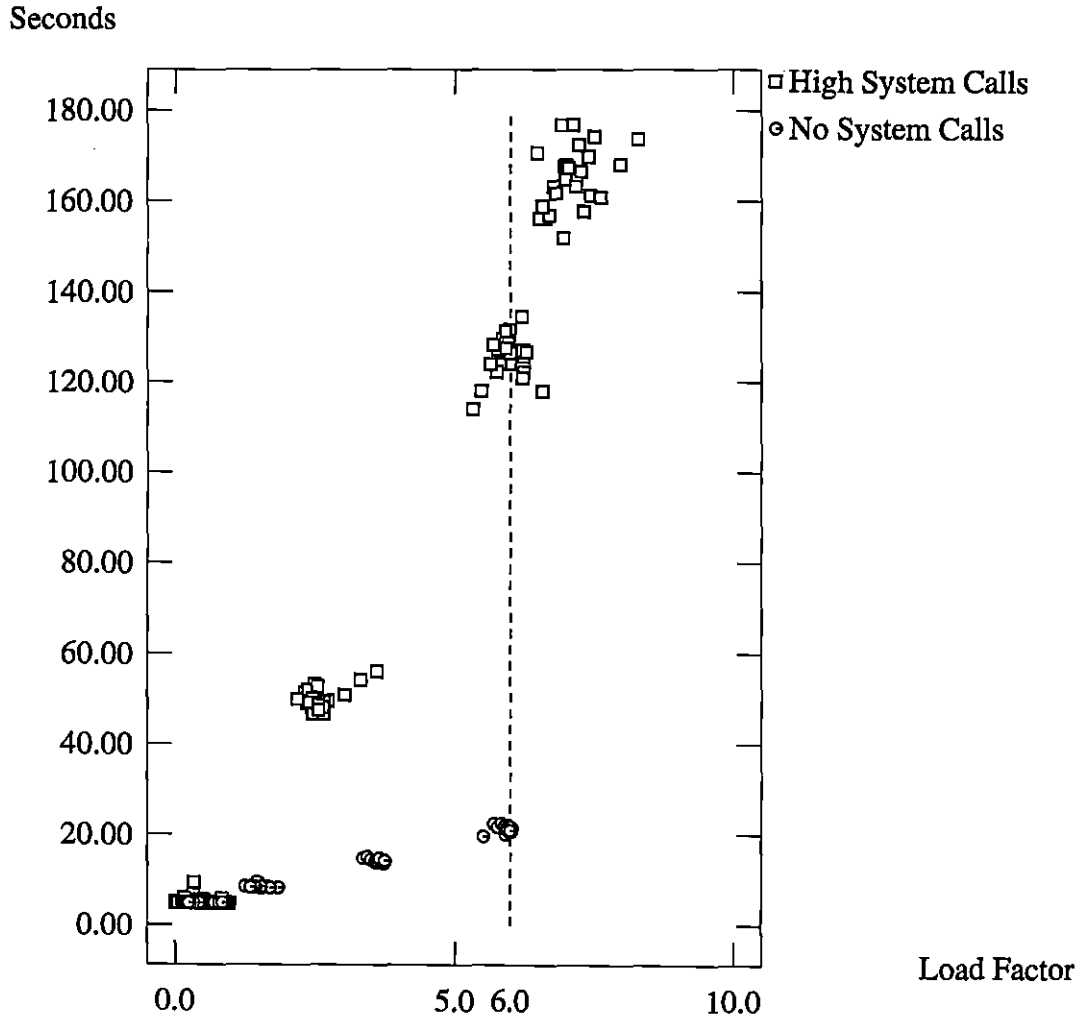

Figure 5.2: Execution time versus Load Factor of different load profiles.

that permits an estimate of the percentage of CPU cycles a job is receiving. This, along with a completed execution time, is used to predict a job's execution time.

5.2.1 UNIX Scheduling Algorithm

The SunOS operating system uses a scheduling algorithm to determine the job that warrants the use of the CPU [48]. The algorithm is designed for use on a workstation and not a multiuser timesharing system.

Jobs in the run queue have their priorities recomputed once per second. The job with the highest priority is dispatched. The algorithm uses percent of CPU cycles consumed by a job to degrade its priority. If a job has used a lot of CPU cycles relative to its peers, its priority gets reduced. The algorithm uses a decaying average of CPU cycles consumed; it forgets 90% of the current CPU use by a job in $5n$ seconds, where n is the load factor over the last minute described in Section 5.1.

Hence, CPU intensive jobs are penalized heavily when the system is highly loaded and less so when the load is light. Jobs blocked on I/O consume no CPU cycles, so when they return to the run queue they have a high priority. This gives interactive users high priority after coming back from keystrokes. This algorithm is designed for short term scheduling needs, but still allows CPU bound processes to progress.

An analysis of job progress shows that most of the priority adjustment had little effect on CPU intensive jobs. Interactive use and small jobs that use CPU cycles in a bursty manner, such as compilers and text searching tools such as grep, did not have a high impact on the execution time of other longer jobs.

A simplified CPU scheduling model is used for medium and long term jobs and the short term priority adjustments to enhance interactive use are ignored. All user processes in the system are assumed to be of equal priority and will receive an equal share of the CPU cycles that are available.

5.2.2 CPU Cycle Allocation

The distribution of the CPU cycles between the user, system and idle processes were extracted from the memory map of the operating system's kernel. The percentage of CPU cycles garnered by the operating system, P_{system}, are unavailable for use by user processes. The percentage of CPU cycles available to user processes is then $100 - P_{system}$. This percentage was combined with the load factor to determine an estimate of the percentage of CPU cycles that would be used by a single job.

The percentage of CPU cycles that would be received by a newly submitted job is given by:

$$P_{newjob} = \frac{100 - P_{system}}{LF + 1} \tag{5.1}$$

Where:

P_{newjob} is the percentage of CPU cycles to be received by a new job

P_{system} is the percentage of CPU cycles currently allocated to the system

LF is the current load factor

Alternately, the load factor and CPU distribution may be available after the completion of a job. The load factor and CPU distribution then include the completed job, assuming the statistics were recovered at job completion time. The formula for the percentage of CPU allocated to the completed job (P_{comjob}) is given by:

$$P_{comjob} = \frac{100 - P_{system}}{LF} \tag{5.2}$$

An experiment was performed to record the percentage of CPU cycles allocated to a completed job, P_{comjob}, as well as the time to complete the job, t_{comjob}. The same job was repeatedly run on workstations with different load levels, first with one load profile and then with another. The results were combined and are shown in Figure 5.3. The points marked High System Calls are execution times on a workstations that had background jobs where the system consumed approximately 65 to 70 percent of the CPU cycles. The points marked No System Calls had background jobs where the system consumed no appreciable CPU cycles.

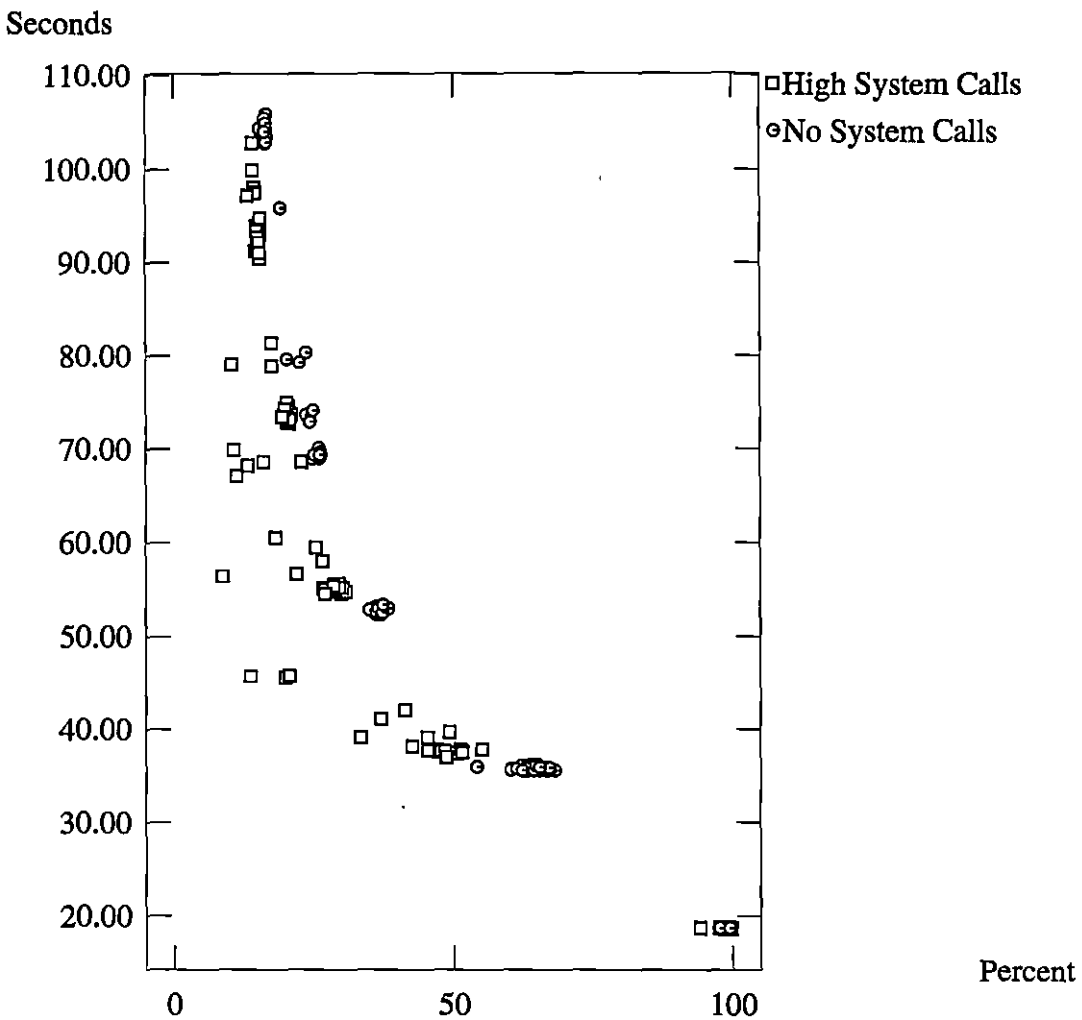

Figure 5.3: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of a test job with different background load profiles.

Figure 5.3 plots t_{comjob} versus P_{comjob} over two different load profiles and is similar to Figure 5.2. However, Figure 5.3 clearly shows there is very little bias in execution time versus percent CPU cycles for the two different load profiles, unlike the execution time versus load factor of Figure 5.2. Percent CPU cycles is a good predictor of execution time irrespective of the type and amount of load present on a workstation.

As the percentage of CPU cycles allotted to a job changes so does its execution time. It is assumed that the time, t_{comjob} will vary inversely proportional [49] to P_{comjob}, i.e., as the percentage of CPU cycles available to perform the job decreases, the time to execute the job increases proportionately.

Mathematically, this is expressed as

$$\begin{aligned} t_{comjob} &= \frac{c}{P_{comjob}} \\ t_{comjob}P_{comjob} &= c \end{aligned} \tag{5.3}$$

The two formulas, 5.2 and 5.3, can be used to predict the approximate completion time of a job, given that one has already been timed on an identical workstation. Assuming a job completed in a given time, t_{comjob}, on one workstation, the expected execution time for the same job, t_{newjob}, on another differently loaded workstation is:

$$\begin{aligned} t_{newjob}P_{newjob} &= t_{comjob}P_{comjob} \\ t_{newjob} &= \frac{t_{comjob}P_{comjob}}{P_{newjob}} \end{aligned} \tag{5.4}$$

The data collected from machines with light to medium load averages are consistent with this model. Workstations with high load averages, over six, and high system CPU utilization, over 50 percent, exhibited slightly different behaviour. Test results showed new jobs added to heavily loaded workstations received a lower percentage of the CPU than predicted by Equation 5.1. Several graphs of execution

time, t_{comjob}, versus the measured percentage of CPU cycles used, P_{comjob}, are included here as examples in Figures 5.4–5.9.

Three test jobs were run on workstations with two different load profiles. Different workstations had different load levels of the same load profile. Each of the three test jobs had different execution times. The execution times and percentage of CPU cycles used are plotted as points on the graph for each job and load profile. Additionally, a line representing Equation 5.4 is drawn on each of the graphs.

The actual formula for the line is included in the caption of each figure. The values t_{comjob} and P_{comjob} used in the formula were drawn from the fastest response back from all of the test data for that test. For example, the line plotted on Figure 5.4 is derived starting with Equation 5.4 with $t_{comjob} = 2.22$ and $P_{comjob} = 100$.

$$t_{newjob} = \frac{t_{comjob}P_{comjob}}{P_{newjob}} = \frac{(2.22)(100)}{P_{newjob}}$$

$$t_{newjob} = \frac{222}{P_{newjob}}$$

Essentially, the point (P_{comjob}, t_{comjob}) used in each of the examples is the execution time when 100% of the CPU cycles are dedicated to the test job. This is a good choice for demonstrating the effectiveness of this approach. In practice, the percent CPU cycles and execution time of the first reply back are used as the basis for timeout calculations.

5.3 Timeout Calculation

Equation 5.1 assumes that submitting the new job will not affect the percentage of CPU used by the system, i.e., the percentage of CPU cycles available for user jobs will not change. The test jobs used to generate the data conform with the assumptions made in the design of the system; they are large grain, computationally intense programs and, as such, do not make any system calls for process management or device input/output.

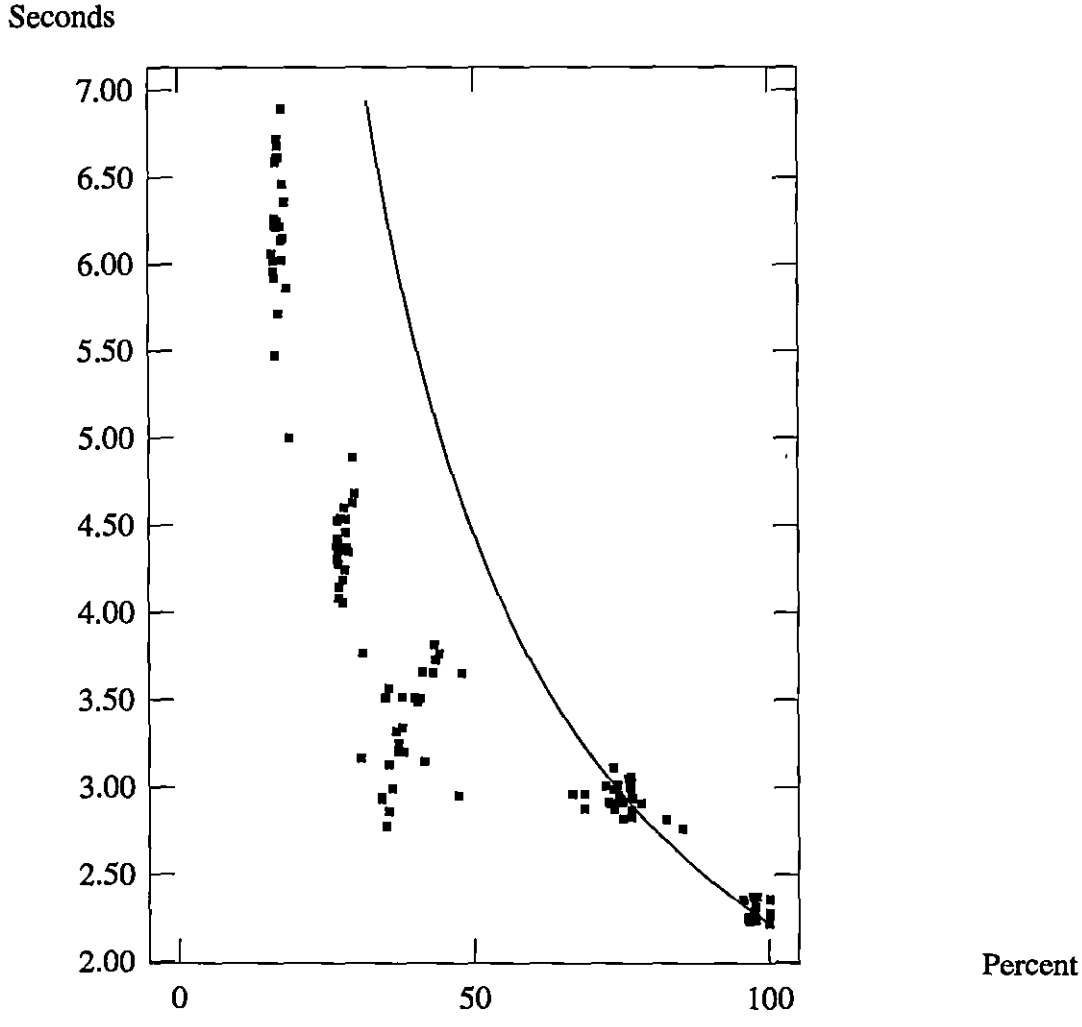

Figure 5.4: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 1 with no system calls from background jobs. The line is $t_{newjob} = 222/P_{newjob}$.

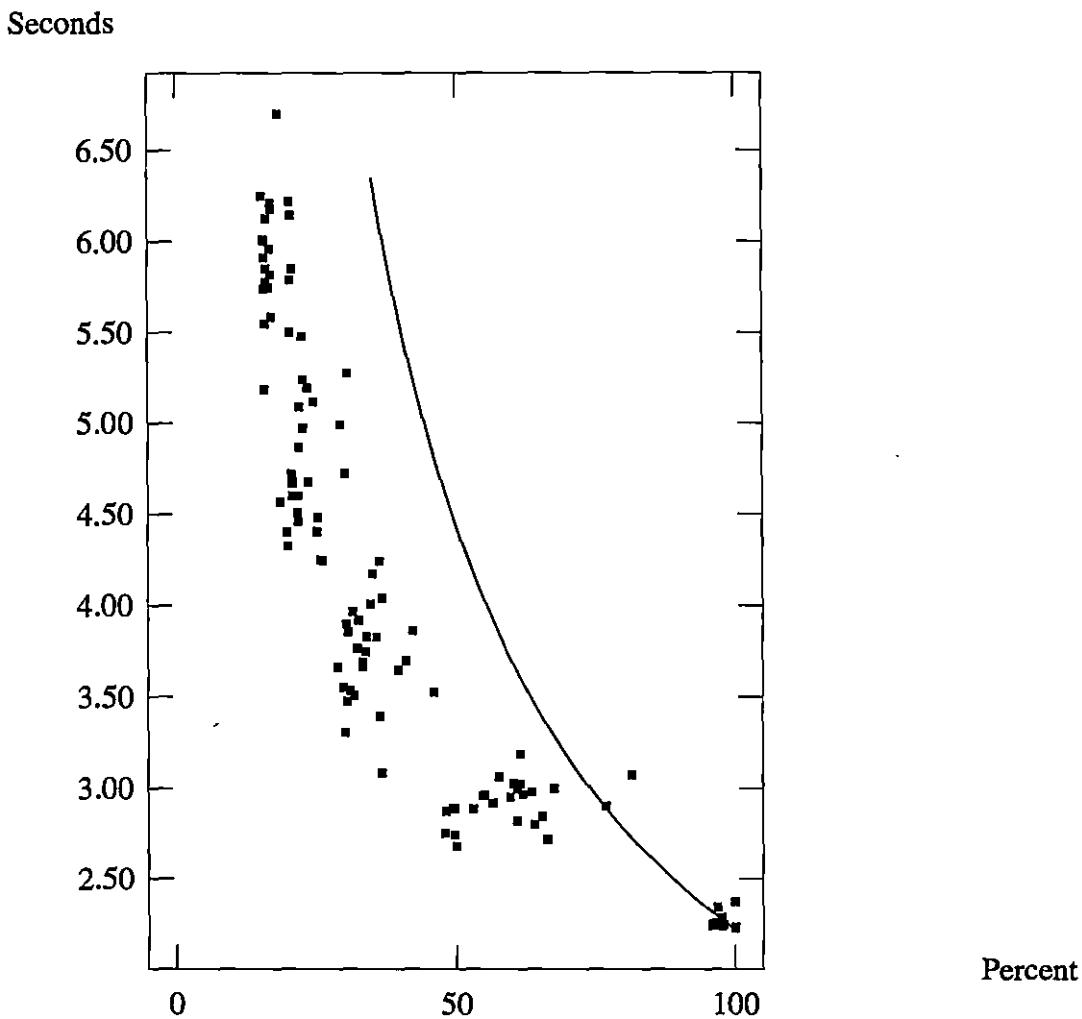

Figure 5.5: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 1 with high system calls from background jobs. The line is $t_{newjob} = 222/P_{newjob}$.

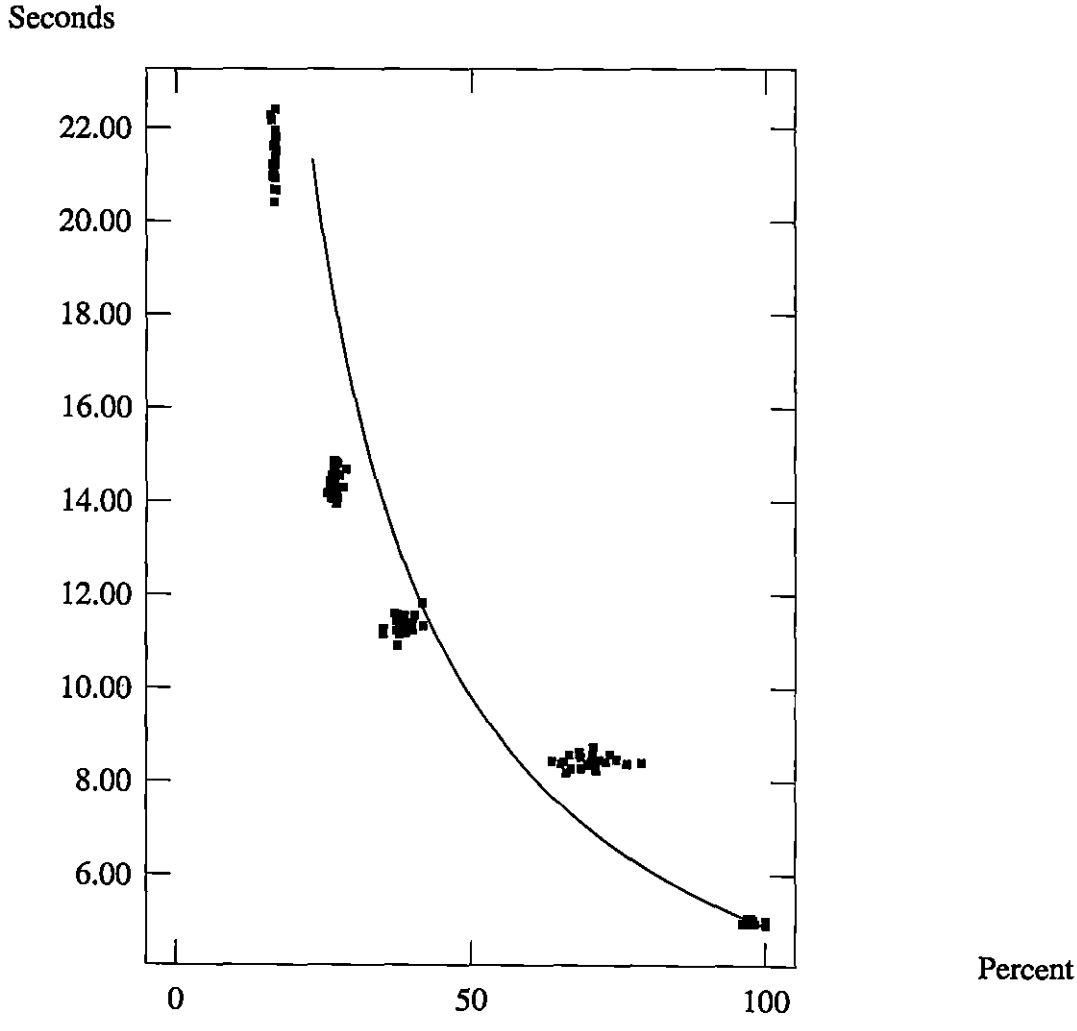

Figure 5.6: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 2 with no system calls from background jobs. The line is $t_{newjob} = 490/P_{newjob}$.

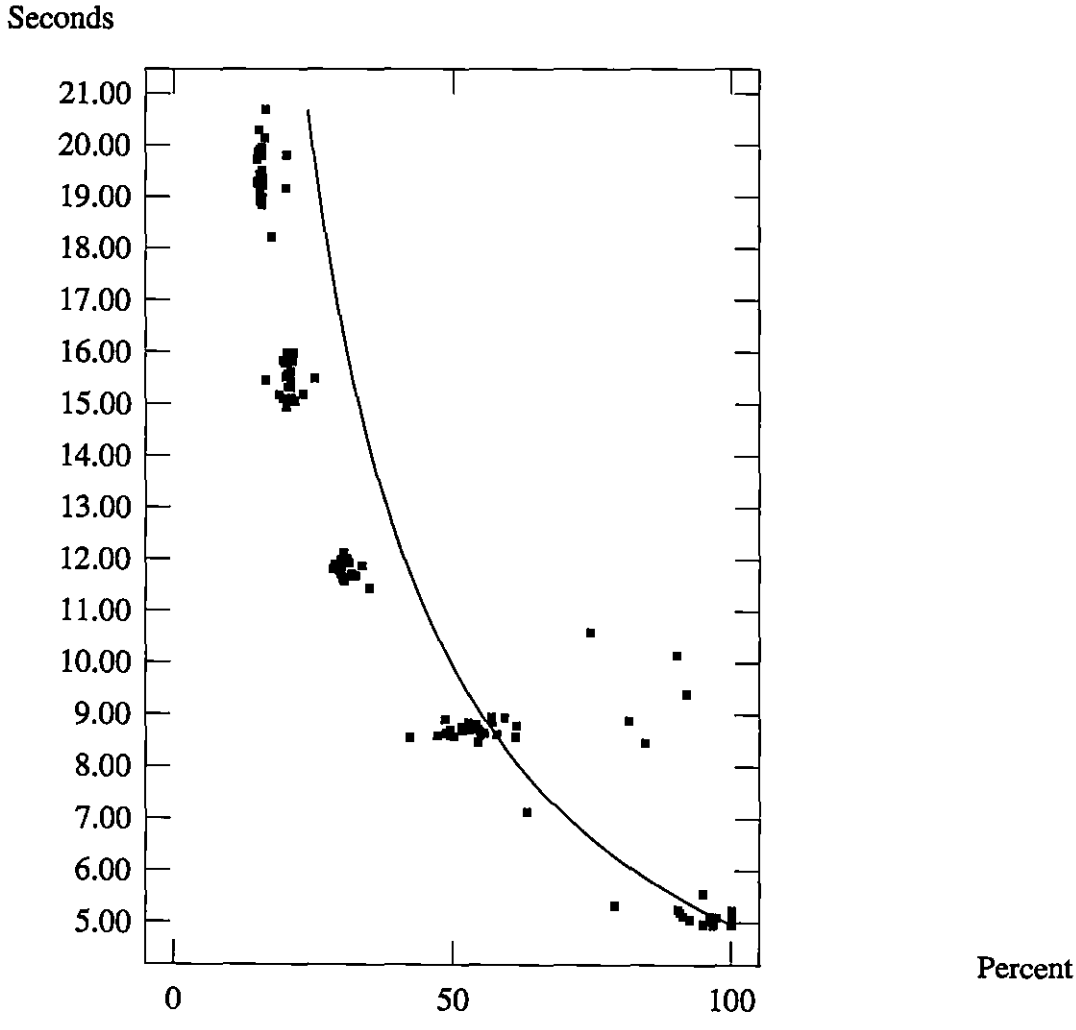

Figure 5.7: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 2 with high system calls from background jobs. The line is $t_{newjob} = 496/P_{newjob}$.

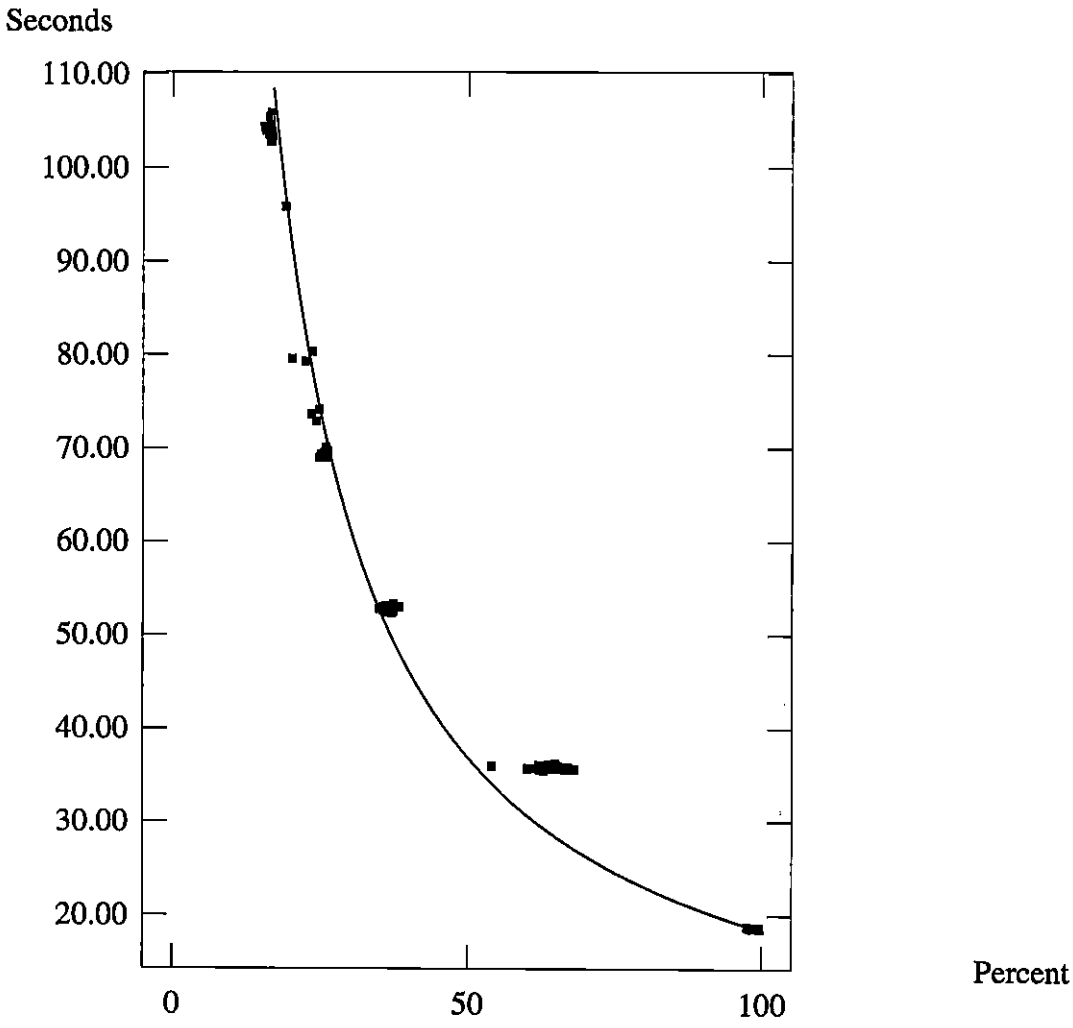

Figure 5.8: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 3 with no system calls from background jobs. The line is $t_{newjob} = 1843/P_{newjob}$.

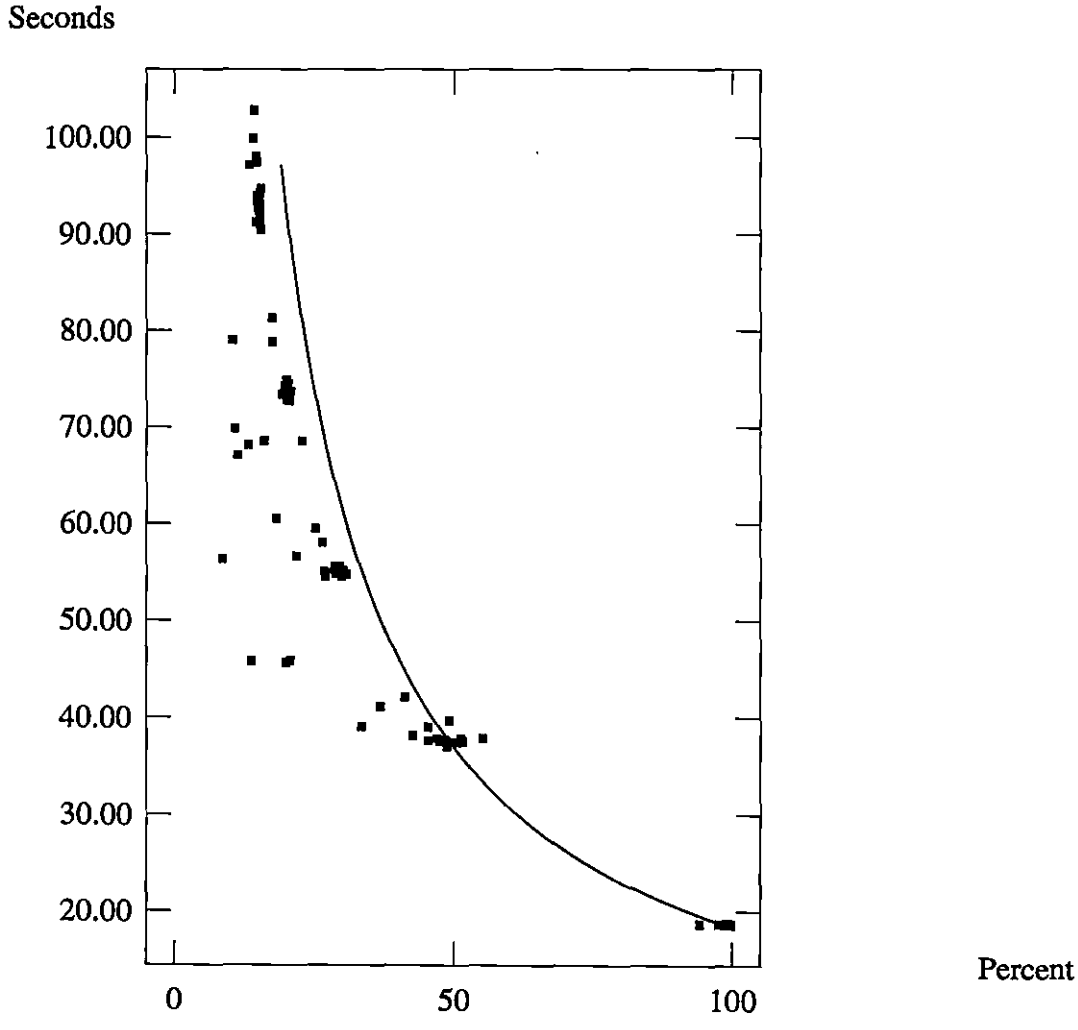

Figure 5.9: Execution time, t_{comjob}, versus percentage CPU, P_{comjob}, of test job 3 with high system calls from background jobs. The line is $t_{newjob} = 1845/P_{newjob}$.

Nonetheless, these test jobs caused a change in the percentage of CPU cycles used by the system. When the test jobs were started, the percentage of CPU cycles being used by the operating system decreased. This is attributed to the decreased percentage of CPU cycles allotted to background jobs making the system calls. As the background jobs are executed less, fewer system calls are generated.

In general, Equation 5.2, the measure of percentage for a completed job, gives a more reliable result than Equation 5.1, the estimate of percentage. So rather than using Equation 5.1 on a workstation before a job is submitted to it, the HFP uses Equation 5.2 to measure the percentage of CPU cycles being allocated to the job in progress on workstations that have not yet replied. Equation 5.4 is then applied to estimate the completion time.

In this manner, the HFP gets a more accurate measure of the load on the workstation and a better estimate of the completion time. The system is also protected from violations of the assumption that the submitted task will not affect system CPU utilization, since it has a measure of CPU cycle distribution while the job is in progress.

Aside from getting a better measure of system utilization, there is a more practical reason for not using Equation 5.1. The HFP has no completion statistics at the very start of job. It has to wait for at least one reply and perform an elapsed time measurement to obtain t_{comjob} and measure P_{comjob} on the workstation that replied. The percentage of CPU cycles being allotted to the job on each of the outstanding workstations is then estimated by Equation 5.2. The completed job statistics and the lowest percentage of CPU cycles obtained from the remaining workstations are used in Equation 5.4 to estimate the longest completion time. Individual workstations are not timed and only one timeout value is used per service request based on the longest completion time.

5.3.1 Allowance for Variation

Several sources of variation occur in the estimation of the execution time of a job. The reported load factor contains a lag and it is used in the calculation of the percentage of CPU cycles be allocated to a job. More importantly, other users are affecting the workstation while jobs are in progress. Other jobs on the workstation may become computationally intense or new jobs may start, affecting the percentage of the CPU available and increasing the time necessary to complete the job. Additionally, network transmission delays may affect the delivery time of replies.

Underestimation of timeouts will cause unnecessary and undesirable blacklisting of workstations. On the other hand, overestimating timeouts will cause delays when enough errors occur to prevent early synchronization during reply reception. By definition, users expect enough identical replies to allow a valid response to be formed and early synchronization to occur during normal operation of the system. Hence, overestimation is preferred to underestimation.

To assure that timeouts are not underestimated, the completion time calculated for a job in Section 5.3 was multiplied by a variance factor of 2. This allowance was tested on jobs with medium and long execution times, between two and 1000 seconds, and proved to be acceptable.

The allowance was designed to compensate for variation in the measurement of load factors and CPU availability, communication delay, and minor fluctuations in actual load. The blacklist module, described in Section 4.4, is designed to compensate for large changes in load after a job is submitted for computation.

Chapter 6

Performance Tests and Results

To expose the performance penalty of executing fault detection and masking, two benchmark tests for comparing the execution time of an HFP were designed and run. The first set of tests compares HFPs to the default REM environment in the absence of faults. The second set of tests compares the two when intentional omission faults are introduced into the system.

A third set of tests measures the cost of detecting a failed HFP, starting up a replacement and resending the data for computation. The original REM system did not have this capability, so no timing comparisons were made. Comparisons with the results of the second test will expose the costs for completing a job when failures occur and are trapped by the application. The code to start a new HFP resend the data and await the results is in the application itself, and is not part of the HFP.

All tests were run on a group of fifteen Sun IPC SPARCstations connected with 10 Mbps Ethernet. Two background load levels were used for these tests. The load levels were uniform across all the machines, with the exception of the client machine which had no appreciable load. One load level was light with little or no load and the second was heavy.

The light load was created by running tests in the background on workstations that were open to users. The load levels were monitored during the tests to insure that the users were not creating abnormally high loads. If a machine exhibited high load, the tests were aborted and a new test suite was established to produce test results. Minor load fluctuations were observed and allowed to influence the tests. The heavy load was created artificially with a mixture of process types running in the background.

Each test sorted an array of 9000 floating point numbers. The parent process read in the array, subdivided it into 10 sub-arrays and distributed each sub-array to a child process for sorting. The child processes used a quick sort algorithm to sort their sub-arrays. The sorted sub-arrays were returned to the parent process which used a merge sort to combine them. Each test was timed 25 times and the results averaged. The percent slow down was calculated as $\frac{HFP-REM}{REM} * 100$.

6.1 Tests with no failures

The first test was run using three and five replicas for each child process. The REM system used the first response back from any replica. The number of identical replies needed for the HFP to form a valid response was varied from one to five. The results for lightly loaded workstations are reported in Table 6.1 and the results for heavily loaded ones are reported in Table 6.2.

The heavy background load for this test created a UNIX load factor of approximately 3.75 before the tests were run. The percentage of CPU utilization for these background jobs was 30% to 35% by user processes and 65% to 70% by the operating system. During the running of the tests, the load factor rose to 6.00 and the CPU utilization changed to approximately 50% by user processes and 50% by the operating system.

Number of Replicas	HFP Identical Replies Needed	HFP Average Sort Time (sec)	REM Average Sort Time (sec)	Percent Slow Down
3	1	7.93	7.82	1.41
3	2	8.32		6.93
3	3	8.44		7.93
5	1	9.98	10.09	-1.09
5	2	9.97		-1.19
5	3	10.78		6.84
5	4	10.97		8.72
5	5	10.82		7.23

Table 6.1: Execution times of fault free tests at light load.

Number of Replicas	HFP Identical Replies Needed	HFP Average Sort Time (sec)	REM Average Sort Time (sec)	Percent Slow Down
3	1	18.46	17.45	5.79
3	2	19.32		10.72
3	3	20.17		15.59
5	1	20.18	20.15	0.15
5	2	20.66		2.53
5	3	21.51		6.75
5	4	22.63		12.31
5	5	23.39		16.08

Table 6.2: Execution times of fault free tests at heavy load.

6.2 Tests with failures

In the second test, omission faults were introduced into the sorting replicas on two remote workstations. The affected replicas did not return a sub-array to the parent process. The omission faults also accurately simulate the effect of timing faults on the HFP. The second test was run using five replicas per child process. The number of identical replies required to form a correct response was configured to one, two and three.

The lightly loaded workstations were configured as above but the high load stations were not as heavily loaded with system jobs. The heavily loaded stations had a load factor of 3.0 and CPU utilization by the system was approximately 10%. The load factor rose to 5.0 during the tests and system CPU utilization dropped to 5%. The timing results from running on lightly loaded workstations are in Table 6.3 and the heavily loaded workstation results are in Table 6.4.

6.3 Tests with failures and retries

The third set of tests measured the cost of starting a new HFP when one had failed. Omission faults were introduced into two workstations as in Section 6.2. Five replicas were used for each child process but the number of identical replies required to form a correct response was set to four and five.

These tests were run only on lightly loaded or empty workstations. Two different tests were performed. In the first, the REM daemon was started and 25 repetitions of the sorting program with retries was run. Invariably, the HFP discovered the faulty stations during the first run and permanently blacklisted them. The subsequent 24 runs had no HFP failures and did not have to start any new HFPs, as the first run had to. The results of this test are reported in Table 6.5.

Number of Replicas	HFP Identical Replies Needed	HFP Average Sort Time (sec)	REM Average Sort Time (sec)	Percent Slow Down
5	1	9.71	9.69	0.21
5	2	9.93		2.48
5	3	10.33		6.60

Table 6.3: Execution times of fault tests at light load.

Number of Replicas	HFP Identical Replies Needed	HFP Average Sort Time (sec)	REM Average Sort Time (sec)	Percent Slow Down
5	1	10.48	10.20	2.75
5	2	10.49		2.84
5	3	10.42		2.16

Table 6.4: Execution times of fault tests at heavy load.

Identical Replies Needed	Sort time of First run (sec)	Number of HFP Retries	Average Sort Time of Remaining Runs (sec)
4	17.20	1	11.58
5	27.61	5	12.04

Table 6.5: Execution times of fault tests using a single daemon.

Identical Replies Needed	Average Number of HFP Retries	Average Sort Time (sec)
4	1.04	16.24
5	6.08	26.00

Table 6.6: Execution times of fault tests using multiple daemons.

The second test started the REM daemon and made one run of the sorting program with retries. The timing results were recorded and the daemon killed. This test was repeated 25 times and the average execution times are reported along with the average number of failed HFPs per execution. Starting a new daemon assured the faulty workstations were included in the suite for each run of the sorting program. The results are reported in Table 6.6.

Chapter 7

Conclusions

7.1 Failure Detection and Adaptive Load Sharing

Virtual Halt on Failure Processors provide a highly available and highy reliable computing environment on a local network of workstations.

HFPs provide a responsive, adaptive load sharing ability. When a workstation begins to miss processing deadlines, new tasks are diverted away from it. If the workstation catches up to its peers, it is reinstated in the processing pool. Otherwise, it is permanently removed. This was achieved by a blacklisting mechanism which differentiates between temporarily overloaded and halted workstations.

Extensive performance measurements were taken to establish the relation between execution times and load factors of remote workstations under different job mixes. An effective, dynamic job timeout mechanism was developed, based on the results from a single job execution on one remote workstation.

The performance results showed that adding the HFP code to REM had very little impact on the execution times of the test cases, hence the overhead of perform-

ing verification and comparison of replies is small. Despite their efficient operation, HFPs provide a significant level of fault tolerance. Failed workstations were detected and when errors exceeded the specified threshold, users were notified of the halted virtual processor. The system has enough flexibility to allow users to restart processing on another HFP.

HFPs are well suited for distributed computing in a contemporary workstation environment as they run in an asynchronous environment and have a reasonably simple user interface. System awareness is encapsulated in a daemon and the user programmes are freed from the intricacies of managing replicated distributed processes. HFPs also meet most of the needs of distributed systems that require or assume the presence of Fail Stop Processors (FSPs). HFPs are faster and use less resources than a full implementation of an FSP can be, since HFPs perform no interactive consistency checking.

7.2 Future Work

Checkpointing is a desirable feature for long term computations and is absent from the current system. It is incumbent upon the application to perform steps of this nature and to start a new virtual HFP in the event of a failure. It is dependent upon the application whether the checkpoint data can be used to reset the state of the HFP. Checkpointing can be implemented through the HFP and HFP verified intermediate results can be logged.

Checkpointing a replica's state could be added to the system. When a failure occurs, the system could restart a copy of the last checkpointed replica state on a healthy processor. Messages sent since the last checkpoint may also be stored and replayed. In this manner, extremely large grain computations could be carried out to completion despite the presence of partial failures and without the intervention of the user application.

HFPs can be used with systems that provide reliable broadcasts. There is no explicit dependency on a communication paradigm in their design. Currently, reliable point-to-point communication is maintained between daemons, reliable broadcasting should improve efficiency and result in simpler communication code.

7.3 Other Applications

With the HFP's ability to detect late replies, REM can be used to implement soft real-time systems with only minor modifications. Job information could read from a database when the timeout is to be set. Jobs are passed by name through the system, so this information is generally available.

As can be seen in Section 5.2, estimated job completion times are calculable from the information currently in the system. If the initial job completion time is available from a database, rather than from the results of the first reply, an absolute timeout can be established before execution begins.

Bibliography

[1] J. Bartlett, "A NonStop Kernel," *Proceeding of the Eighth International Symposium on Operating Systems Principles* (December 1981), 22–29.

[2] Jim Gray, "Why Do Computers Stop and What Can be Done About It," *Fifth Symposium on Reliability in Distributed Software and Database Systems* (January 1986), 3–12.

[3] Paul D. Ezhilchelvan & S.K. Shrivastava, "A Characterisation of Faults in Systems," *Fifth Symposium on Reliability in Distributed Software and Database Systems* (January 1986), 215–222.

[4] Michael Barborak, Miroslaw Malek & Anton Dahbura, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys* 25 (June 1993), 171–220.

[5] Flaviu Cristian, Houtan Aghili, Ray Strong & Danny Dolev, "Atomic Broadcast: From Simple Diffusion to Byzantine Agreement," *Proceedings of the Fifth Annual International Symposium on Fault-Tolerant Computing Systems* (June 1985), 200–206.

[6] Flaviu Cristian, "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM* 34 (February 1991), 56–78.

[7] Ray Strong, "Problems in Maintaining Agreement," *Fifth Symposium on Reliability in Distributed Software and Database Systems* (January 1986), 20–27.

[8] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components.," in *Automata Studies*, C. E. Shannon & J. McCarthy, eds., Princeton University Press, 1956, 43–98.

[9] L. Lamport, R. Shostak & M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems* 4 (July 1982), 382–401.

[10] D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms* 3, 14–30.

[11] Kenneth P. Birman, "Replication and Fault-tolerance in the ISIS System," *Proceedings of the Tenth ACM Symposium on Operating System Principles in Operating Systems Review* 19 (December 1985), 79–86.

[12] E.C. Cooper, "Replicated Distributed Programs," *Proceedings of the Tenth ACM Symposium on Operating System Principles in Operating Systems Review* 19 (December 1985), 63–78.

[13] R.D. Schlichting & F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems* 1 (August 1983), 222–238.

[14] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems* 2 (May 1984), 145–154.

[15] Kenneth P. Birman, Thomas A. Joseph, Thomas Raeuchle & Amr El Abbadi, "Implementing Fault-tolerant Distributed Objects," *IEEE Transactions on Software Engineering* SE-11 (June 1985), 502–508.

[16] Rivka Ladin, Barbara Liskov, Liuba Shrira & Sanjay Ghemawat, "Providing High Availabilty Using Lazy Replication," *ACM Transactions on Computer Systems* 10 (November 1992), 360–391.

[17] Jie Wu & Eduardo B. Fernandez, "Fault-tolerant distributed broadcast algorithms for cube-connected-cycles.," *Computer Systems Science & Engineering* 8 (1993), 224–233.

[18] Andrew S. Tanenbaum et al., "Experiences with the Amoeba Distributed Operating System," *Report IR-194, Department of Mathematics and Computer Science, Vrije Universiteit* (July 1989.).

[19] Marvin M. Theimer, Keith A. Lantz & David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the Tenth ACM Symposium on Operating System Principles in Operating Systems Review* (December 1985), 2–12.

[20] Robert Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," *Proceedings of the Sixth International Conference on Distributed Computer Systems* (1986), 260–267.

[21] David A. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Proceedings of the Eleventh ACM Symposium on Operating System Principles in Operating Systems Review* 21 (November 1987), 5–12.

[22] Jiubin Ju, Gaochao Xu & Jie Tao, "Parallel Computing Using Idle Workstations," *Operating Systems Review* 27 (July 1993), 87–96.

[23] Micheal J. Litzkow, Miron Livny & Matt W. Mutka, "Condor - A Hunter of Idle Workstations," *Proceedings of the Eighth International Conference on Distributed Computer Systems* (1988), 104–111.

[24] Phillip Krueger & Rohit Chawla, "The Stealth Distributed Scheduler," *Proceedings of the 11th International Conference on Distributed Computer Systems* (May 1991), 336–343.

[25] A. Birrell & B. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* 2 (Feb 1984), 39–59.

[26] Henry Clark & Bruce McMillin, "DAWGS – A Distributed Compute Server Utilizing Idle Workstations," *Journal of Parallel and Distributed Computing* 14 (1992), 175–186.

[27] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S.H. Rosenthal & F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM* (March 1986), 184–201.

[28] Kiam S. Yap, Pankaj Jalote & Satish Tripathi, "Fault Tolerant Remote Procedure Call," *Proceedings of the Eighth International Conference on Distributed Computer Systems* (1988), 48–54.

[29] Y.C. Tay, "The Reliability of (k,n)-resilient Distributed Systems," *Fourth Symposium on Reliability in Distributed Software and Database Systems* (October 1984), 119–122.

[30] K. Birman et al, "The ISIS System Manual, Version 1.1," December 21, 1988.

[31] N. Lynch, M. Fischer & R. Fowler, "A Simple and Efficient Byzantine Agreement Algorithm," *Second Symposium on Reliability in Distributed Software and Database Systems* (1982).

[32] Gabriel Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Protocol," *Journal of the ACM* 34 (October 1987), 910–920.

[33] M. Rabin, "Randomized Byzantine Generals," *Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (1983), 403–409.

[34] Kenneth J. Perry, "Randomized Byzantine Agreement," *Fourth Symposium on Reliability in Distributed Software and Database Systems* (October 1984), 107–119.

[35] Benny Chor & Brain A. Coan, "A Simple and Efficient Randomized Byzantine Agreement Algorithm," *Fourth Symposium on Reliability in Distributed Software and Database Systems* (October 1984), 98–106.

[36] D. Dolev, R. Reischuk & H. R. Strong, "'Eventual' is Earlier than 'Immediate'," *Proceedings of the 23rd Symposium on Foundations of Computer Science* (1982), 196–203.

[37] David K. Gifford, "Weighted Voting for Replicated Data," *Proceedings of the Seventh Symposium on Operating Systems Principles* (December 1979), 150–162.

[38] Robert H. Thomas, "A Majority Consensus Approach to Concurrency for Multiple Copy Databases," *ACM Transactions on Database Systems* 4 (June 1979), 180–209.

[39] L. Lamport, "Times, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM* 21 (July 1978), 552–565.

[40] P.T. de Sousa & F. P. Mathur, "Sift-out Modular Redundancy," *IEEE Transactions on Computers* C-27 (1978), 624–627.

[41] Yinong Chen & Tinghuai Chen, "Implementing Fault-Tolerance via Modular Redundancy with Comparison," *IEEE Transactions on Reliability* 39 (June 1990), 217–225.

[42] Jiajun Shao & Leonard R. Lamberson, "Modeling a Shared-Load k-out-of-n:G System," *IEEE Transactions on Reliability* 40 (June 1991), 205–209.

[43] Barry A. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[44] Kishor Shridharbhai Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall of India Private Limited, New Delhi, India, 1982.

[45] G. C. Shoja, "A Distributed Facility for Load Sharing and Parallel Processing Among Workstations," *Journal of Systems and Software* 14 (1991), 163–172.

[46] Alexander B. Romanovsky, "Fault Tolerance: Synchronization of Redundancy," *Operating Systems Review* 27 (October 1993), 58–66.

[47] Robert S. Side, "REM Interface Routines," Univeristy of Victoria, Technical Report, 1990.

[48] Harvey M. Deitel, in *An Introduction to Operating Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990, 582–583.

[49] W. Gellert, H. Küstner, M. Hellwich & H. Kästner, eds., *The VNR Concise Encyclopedia of Mathematics*, Van Nostrand Reinhold Company, New York, New York, 1977.

Appendix A

HFP User Interface

The user interface routines added to REM or modified for the implementation HFPs are described here.

The following function was added to REM. It allows a user to make a service request of an HFP.

Syntax:

int **send_dist_HFP** (Vpid dest_HFP, char * request, int request_len)

Description:

This function sends a service request to an HFP identified by `dest_HFP`. The service request is identified by `request`, a pointer to the request data, and `request_len`, the number of bytes to send.

The call causes the data to be put in message and sent to the local REM daemon. Once the data has been dispatched to the local REM daemon, control returns to the calling program.

The local daemon initializes data structures associated with the HFP and sends the data to the remote replicas. It then awaits replies from the replicas.

Return Values:

The function returns -1 if it was unable to send the data, otherwise it returns the number of bytes sent.

The following function was modified to look for `HFP_FAILURE` type messages as well as returned data. Modifications were needed in two locations. First, the search of queued messages for this user was modified to find and return failure messages. If there is no queued message, this routines blocks waiting for a new messages. When one is received, the message is verified to be from the correct sender and of

the correct type. A check for a failure message was also added to this part of the routine.

This routine still serves its original purpose and is syntactically identical to its original form. The call is still in use by other REM programs that do not use the HFP services.

Syntax:

int **us_recv_dist** (Vpid source_HFP, char * response, int response_len)

Description:

This function retrieves a response from an HFP identified by `source_HFP`. The response is stored in `response` to a maximum of `response_len` bytes.

Return Values:

The function returns the number of bytes received if the HFP completed a computation. Otherwise it returns `HFP_USR_FAILURE` to indicate the HFP failed and is halted.

Appendix B

Source Code

B.1 Blacklist Module

```
typedef enum {
   COMMISSION_ERROR,
   TIMELY_ERROR,
   TIMING_ERROR
}              ERROR_TYPE;

  *
/*
 * blacklist.c
 *
 * author: Robert (Mac) Macdonald
 *
 * purpose: maintain a blacklist of processors or stations
 *
 *
 */

#include "global_defines.h"
#include "net_types.h"
#include "blacklist.h"

#define CHECK_UE


static struct {
   char            commission, timing, timely;
}              error_count[MAX_STATIONS];

static struct {
   char            temp, perm;
}              blist[MAX_STATIONS];

/*********************************************************
 *                                                       *
 * routine to check if a station has been blacklisted    *
 *                                                       *
 *********************************************************/
int
is_blacklisted(station)
   int             station;
```

```
{

#ifdef CHECK_UE
   if (station < 0 || station >= MAX_STATIONS) {
      fprintf(stderr, "is_blacklisted: out of range station number:%d\n", station);
      return -1;
   }
#endif

   if (blist[station].temp || blist[station].perm)
      return 1;

   return 0;

}

/*********************************************************
 *                                                       *
 * add a station to the blacklist for some reason        *
 *                                                       *
 *********************************************************/

void
blacklist(station, reason)
   int             station;
   ERROR_TYPE      reason;
{

#ifdef CHECK_UE
   if (station < 0 || station >= MAX_STATIONS) {
      fprintf(stderr, "blacklist: out of range station number:%d\n", station);
      return -1;
   }
#endif

   switch (reason) {

   case COMMISSION_ERROR:   /* some odd error has occurred */
      /* flag the station */
      blist[station].perm = 1;
      /* initiate de-connection with station */
      /* unplug (station) */
      break;

   case TIMING_ERROR:   /* late or lost packet error has occurred */
      {
         register int    count = ++error_count[station].timing;

         if (count > 2) {
            blist[station].perm = 1;
            /* initiate de-connection with station */
            /* unplug (station) */
         } else if (count == 1) {
            blist[station].temp = 1;
         }
      }
      break;

   case TIMELY_ERROR:   /* some station has returned a packet deemed
            * to be in error */
      {
         register int    count = ++error_count[station].timely;
         if (count > 2) {
            blist[station].perm = 1;
```

```
            /* initiate de-connection with station */
            /* unplug (station) */
         }
      }
      break;

   }

}

/*********************************************************
 *                                                       *
 * de_blacklist a station for returning a late           *
 * packet                                                *
 *                                                       *
 *********************************************************/
void
de_blacklist(station)
   int             station;
{

#ifdef CHECK_UE
   if (station < 0 || station >= MAX_STATIONS) {
      fprintf(stderr, "de_blacklist: out of range station number:%d\n", station);
      return -1;
   }
#endif

   if (--error_count[station].timing <= 0)
      blist[station].temp = 0;
}

void
init_blacklist()
{
   register int    i;

   bzero(error_count, sizeof(error_count));
   bzero(blist, sizeof(blist));
}

void
reset_blacklist(station)
   int             station;
{

#ifdef CHECK_UE
   if (station < 0 || station >= MAX_STATIONS) {
      fprintf(stderr, "reset_blacklist: out of range station number:%d\n", station);
      return -1;
   }
#endif

   bzero(&error_count[station], sizeof(error_count[station]));
   bzero(&blist[station], sizeof(blist[station]));

}
```

B.2 HFP Module

```
typedef struct HFP_message {
   int             num;
```

```
   int             len;
   int             right_one;
   char            station_list[MAX_REPLICATE_NUMBER];
   char           *message;
   struct HFP_message *next_mesg;
}              HFP_MESSAGE, *HFP_MESSAGE_PTR;

typedef struct HFP_head {
   int             mesg_num,   /* currently used as a transaction id */
                   timer,   /* timer number of this request */
                   expired, /* has the timer wound down on this request */
                   responded,   /* true if the HFP has responded to
                                 * the client */
                   num_rec,/* total number of messages received with
                            * this message number */
                   max_ident;   /* largest number of identical
                                 * responses seen so far */
   struct timeval  start_time;
   float           cpu_index;
   struct {
      int             station;
      long            system, total;
   }               start_cpu_ticks[MAX_REPLICATE_NUMBER];
   HFP_MESSAGE_PTR mesg_list;   /* list of messages received */
   struct HFP_head *next_head;   /* pointer to the next message header */
}              HFP_HEAD, *HFP_HEAD_PTR;

typedef struct HFP_struct {
   int             replicates,   /* number of copies believed to be
                                  * executing */
                   identical_needed,   /* number of identical
                                        * replies needed to form a
                                        * valid response */
                   failed,   /* state of the HFP */
                   response_level;   /* last request number responded to */

   HFP_HEAD_PTR    head_list;
}              HFP, *HFP_PTR;

extern HFP_HEAD_PTR get_HFP_header();

#define HFP_INC


/*
 * hfp.c
 *
 * contains code to maintain a Halt on Failure Processor
 *
 * Author: Robert (Mac) Macdonald
 *
 *
 *
 */

#include "globals.h"
#include "net_include.h"
#include "HFP.h"
#include "net_process.h"
#include "blacklist.h"
#include "loadbal.h"

#define MAX_HFP 100

typedef struct {
```

```
   long            expiration_time;   /* negative to reflect unused
                   * entry */
   NET_PROC_ENTRY_PTR proc_ptr;
   HFP_HEAD_PTR    hfp_head_ptr;
}               TIMER_ELEMENT;

static TIMER_ELEMENT timer_list[MAX_HFP];

void
init_HFP_timer()
{
   register int    i;

   for (i = 0; i < MAX_HFP; i++)
      timer_list[i].expiration_time = -1L;
}

void
unset_HFP_timer(i)
{

   timer_list[i].expiration_time = -1L;
}

static float
get_percent_cpu(station, hfp_head_ptr)
   int             station;
   HFP_HEAD_PTR    hfp_head_ptr;
{
   register int    i;
   typ_loadmsg     load;


   lb_g_load_struct(station, &load);

   if (load.load_factor < 1.0)
      load.load_factor = 1.0;

   for (i = 0; i < MAX_REPLICATE_NUMBER; i++)
      if (hfp_head_ptr->start_cpu_ticks[i].station == station)
         break;

   if (i == MAX_REPLICATE_NUMBER)
      return -1.0;

   if (load.total_cpu == hfp_head_ptr->start_cpu_ticks[i].total)
      return (1.0 / load.load_factor);

   return ((1.0 - ((float) load.system_cpu - hfp_head_ptr->start_cpu_ticks[i].system) /
       (load.total_cpu - hfp_head_ptr->start_cpu_ticks[i].total)
       )
      / load.load_factor
      );
}

int
set_HFP_timer(station, proc_ptr, hfp_head_ptr)
   int             station;
   NET_PROC_ENTRY_PTR proc_ptr;
   HFP_HEAD_PTR    hfp_head_ptr;
{

   register int    i;
   typ_loadmsg     load;
```

```
for (i = 0; i < MAX_HFP; i++)
   if (timer_list[i].expiration_time < 0)
      break;

if (i < MAX_HFP) {
   struct timeval  tp;
   float           min_cpu = 100.0, tf, percent_cpu, expected_time,
                   elapsed_time;

   /* get the load factor of the responding workstation */
   percent_cpu = get_percent_cpu(station, hfp_head_ptr);

   /* get the lowest cpu factor of the children */
   {
      register int    i = 0;
      REPLICATE_PTR   kid;
      float           temp_cpu;

      kid = proc_ptr->child;

      do {
         /* get the station number of this kid */

         temp_cpu = get_percent_cpu((GET_STATION(kid->proc)), hfp_head_ptr);
         if (temp_cpu < min_cpu)
            min_cpu = temp_cpu;
         ++kid;
         ++i;
      } while (i < REPLICATE_NUMBER);
   }

   /* calculate the time factor for the first reply */
   gettimeofday(&tp, (struct timezone *) NULL);

   /* time factor is ( elapsed time ) * ( cpu factor ) */
   elapsed_time = (tp.tv_sec - hfp_head_ptr->start_time.tv_sec);
   elapsed_time += ((float) (tp.tv_usec - hfp_head_ptr->start_time.tv_usec) / 1000000.0);

   /* keep the computation index for this job */
   hfp_head_ptr->cpu_index =
      tf = elapsed_time * percent_cpu;

   /*
    * calculate the maximum expected time and double as fudge
    * factor
    */
   expected_time = (tf / min_cpu) * 2;

   /* set up the time data structures */

   /* wake up call set to start time plus maximum expected time */
   timer_list[i].expiration_time = hfp_head_ptr->start_time.tv_sec + (long) expected_time;
   /*
    * if the wake up call is before the current time, make it
    * for 2 secs from now
    */
   if (timer_list[i].expiration_time <= tp.tv_sec)
      timer_list[i].expiration_time = tp.tv_sec + 2;
   timer_list[i].proc_ptr = proc_ptr;
   timer_list[i].hfp_head_ptr = hfp_head_ptr;
   hfp_head_ptr->timer = i;
   return i;
} else
```

```
    return -1;
}

void
HFP_failure(proc_ptr, hfp_head_ptr)
   NET_PROC_ENTRY_PTR proc_ptr;
   HFP_HEAD_PTR    hfp_head_ptr;
{

   int             stat;

   if (proc_ptr == NULL) {
      return;
   }
   /* send a failure message to the user */
   stat = send_local(proc_ptr->proc, GET_PID(proc_ptr->owner), NULL, 0,
          HFP_FAILURE, NULL);

   if (stat < 0) {      /* local process is down !  */
      delete_proc_entry(proc_ptr->proc);   /* delete entry  */
      delete_owner_entry(proc_ptr->proc);   /* delete ophanded
                     * kids.. */
   }
   /* deallocate space for the HFP */
   {
      HFP_PTR         hfp_ptr = proc_ptr->HFP_params;
      HFP_HEAD_PTR    head_ptr;
      HFP_MESSAGE_PTR mesg_ptr, temp_list;
      char            received[MAX_STATIONS];
      register int    i;

      for (i = 0; i < REPLICATE_NUMBER; i++) {
         received[GET_STATION(proc_ptr->child[i].proc)] = 0;
      }

      if (hfp_ptr == NULL) {
         return;
      }
      while ((temp_list = hfp_head_ptr->mesg_list) != NULL) {
         hfp_head_ptr->mesg_list = temp_list->next_mesg;
         /* flag the stations as having sent a reply */
         for (i = 0; i < temp_list->num; i++)
            received[temp_list->station_list[i]] = 1;
      }

      /* now flag off the stations that did not send a reply */
      for (i = 0; i < REPLICATE_NUMBER; i++)
         if (!received[GET_STATION(proc_ptr->child[i].proc)])
            blacklist(GET_STATION(proc_ptr->child[i].proc), TIMING_ERROR);

      while (hfp_ptr->head_list != NULL) {
         head_ptr = hfp_ptr->head_list;
         while (head_ptr->mesg_list != NULL) {
            mesg_ptr = head_ptr->mesg_list;
            if (mesg_ptr->message != NULL)
               free(mesg_ptr->message);
            head_ptr->mesg_list = mesg_ptr->next_mesg;
            free(mesg_ptr);
         }
         hfp_ptr->head_list = head_ptr->next_head;
         free(head_ptr);
      }
      /* flag the HFP as having failed */
      hfp_ptr->failed = 1;
```

```
    }
}

void
HFP_cleanup(proc_ptr, hfp_head_ptr)
    NET_PROC_ENTRY_PTR proc_ptr;
    HFP_HEAD_PTR    hfp_head_ptr;

{

    register HFP_PTR hfp_ptr = proc_ptr->HFP_params;

    if (hfp_ptr == NULL) {
        return;
    }
    if (hfp_head_ptr == NULL) {
        return;
    }
    /* have we received all replies ? */
    if (hfp_head_ptr->num_rec == hfp_ptr->replicates) {
        /* yes, dispose of everything */
        HFP_MESSAGE_PTR mesg_list = hfp_head_ptr->mesg_list;

        while (mesg_list != NULL) {
            /*
             * did we receive enough of these messages to form a
             * valid response ??
             */
            if (!mesg_list->right_one) {
                /* no, so black list these stations */
                register int    i;
                for (i = 0; i < mesg_list->num; i++) {
                    blacklist(mesg_list->station_list[i], TIMELY_ERROR);
                }
            }
            mesg_list = mesg_list->next_mesg;
            free(hfp_head_ptr->mesg_list->message);
            free(hfp_head_ptr->mesg_list);
            hfp_head_ptr->mesg_list = mesg_list;
        }

        /*
         * replace the head_ptr in the pointer chain and free the
         * head_ptr
         */
        if (hfp_ptr->head_list == hfp_head_ptr)
            hfp_ptr->head_list = hfp_head_ptr->next_head;
        else {
            HFP_HEAD_PTR    temp_head_ptr = hfp_ptr->head_list;
            while (temp_head_ptr->next_head != hfp_head_ptr)
                temp_head_ptr = temp_head_ptr->next_head;
            temp_head_ptr->next_head = temp_head_ptr->next_head->next_head;
        }

        free(hfp_head_ptr);
    } else {
        /*
         * we have not received all the replies so keep the right
         * answer for future verification
         */

        register HFP_MESSAGE_PTR temp_list;
        HFP_MESSAGE_PTR right_one = NULL;
```

```
        char            received[MAX_STATIONS];

        register int    i;

        for (i = 0; i < REPLICATE_NUMBER; i++) {
           received[GET_STATION(proc_ptr->child[i].proc)] = 0;
        }

        /* find the answer we returned to the client */
        while ((temp_list = hfp_head_ptr->mesg_list) != NULL) {
           hfp_head_ptr->mesg_list = temp_list->next_mesg;

           /* flag the stations as having sent a reply */
           for (i = 0; i < temp_list->num; i++)
              received[temp_list->station_list[i]] = 1;

           /* is this the message returned to the user ? */
           if (temp_list->right_one) {   /* yep */
              right_one = temp_list;
              right_one->next_mesg = NULL;
           } else {/* nope */
              /* blacklist senders of the "wrong" messages */

              for (i = 0; i < temp_list->num; i++)
                 blacklist(temp_list->station_list[i], TIMELY_ERROR);

              /*
               * return dynamically allocated memory to the
               * pool
               */
              free(temp_list->message);
              free(temp_list);
           }
        }
        hfp_head_ptr->mesg_list = right_one;
        /* now flag off the staions that did not send a reply */
        for (i = 0; i < REPLICATE_NUMBER; i++)
           if (!received[GET_STATION(proc_ptr->child[i].proc)])
              blacklist(GET_STATION(proc_ptr->child[i].proc), TIMING_ERROR);
     }
  }


  void
  check_HFP()
  {
     struct timeval  tp;
     long            curr_time, *tmp_time;
     int             i;

     gettimeofday(&tp, (struct timezone *) NULL);
     curr_time = tp.tv_sec;

     for (i = 0; i < MAX_HFP; i++) {
        /* is there a valid entry at this location */
        if (*(tmp_time = &timer_list[i].expiration_time) >= 0) {
           /* has the timer expired for this slot */
           if (*tmp_time <= curr_time) {
              timer_list[i].hfp_head_ptr->expired = 1;
              if (!timer_list[i].hfp_head_ptr->responded) {
                 HFP_failure(timer_list[i].proc_ptr, timer_list[i].hfp_head_ptr);
              } else
                 HFP_cleanup(timer_list[i].proc_ptr, timer_list[i].hfp_head_ptr);
              *tmp_time = -1L;
```

```
        }
      }
   }
}

HFP_HEAD_PTR
get_HFP_header(HFP_ptr, mesg_num)
   HFP_PTR         HFP_ptr;
   int             mesg_num;
{
   register HFP_HEAD_PTR temp_ptr;

   if (HFP_ptr == NULL)
      return NULL;

   temp_ptr = HFP_ptr->head_list;

   while (temp_ptr != NULL) {
      if (temp_ptr->mesg_num == mesg_num)
         break;
      temp_ptr = temp_ptr->next_head;
   }

   return temp_ptr;

}
/*************************************************************/
HFP_HEAD_PTR
new_head(mesg_num, list)
   int             mesg_num;
   HFP_HEAD_PTR    list;
{
   HFP_HEAD_PTR    ptr = (HFP_HEAD_PTR) malloc(sizeof(HFP_HEAD));

   bzero((char *) ptr, sizeof(HFP_HEAD));
   ptr->mesg_num = mesg_num;
   ptr->next_head = list;

   return ptr;

}
/*************************************************************/

/*************************************************************/
HFP_MESSAGE_PTR
new_message(message, len, list, station)
   char           *message;
   int             len;
   int             station;
   HFP_MESSAGE_PTR list;
{
   HFP_MESSAGE_PTR ptr = (HFP_MESSAGE_PTR) malloc(sizeof(HFP_MESSAGE));

   ptr->num = 1;
   ptr->station_list[0] = station;
   ptr->len = len;
   ptr->message = (char *) malloc(len);
   bcopy(message, ptr->message, len);
   ptr->next_mesg = list;

   return ptr;

}
/*************************************************************/
```

```
/*************************************************************/
/* */
/* adds a newly received message to an HFP and returns the    */
/* number of identical messages received thus far             */
/* */
/* will return 1 if a messgae is added after it has           */
/* received IDENTICAL_NEEDED copies of a message              */
/* */
/*************************************************************/
int
add_message(HFP_ptr, HFP_list_ptr, from_station, message, len)
   HFP_PTR         HFP_ptr;
   HFP_HEAD_PTR    HFP_list_ptr;
   char           *message;
   int             len;
   int             from_station;
{

   register HFP_HEAD_PTR head_list = HFP_list_ptr;
   register HFP_MESSAGE_PTR mesg_list;
   HFP_MESSAGE_PTR save_mesg_list;
   int             return_value = 1;

   ++(head_list->num_rec);
   /* check if we need to look for matching messages */

   /* point the message list to the first message from the header */
   mesg_list = head_list->mesg_list;

   /* look for a matching message */
   while (mesg_list != NULL) {
      if (mesg_list->len == len) {
         if (bcmp(mesg_list->message, message, len) == 0) {
            mesg_list->station_list[mesg_list->num] = from_station;
            if (++(mesg_list->num) > head_list->max_ident)
               head_list->max_ident = mesg_list->num;
            if (head_list->max_ident == HFP_ptr->identical_needed &&
                !head_list->responded)
               mesg_list->right_one = 1;
            return_value = mesg_list->num;
            break;
         }
      }
      mesg_list = mesg_list->next_mesg;
   }
   /* end while mesg_list != NULL */

   /* it must be that we didn't find a matching message */
   if (mesg_list == NULL) {
      head_list->mesg_list = new_message(message, len, head_list->mesg_list,
                     from_station);
      if (head_list->max_ident < 1)
         head_list->max_ident = 1;
      if (head_list->max_ident == HFP_ptr->identical_needed && !head_list->responded)
         head_list->mesg_list->right_one = 1;
      return_value = 1;
   }
   return return_value;

}
/*************************************************************/
```

B.3 Reply Reception Code

```
/*****************************************************************************/
void
in_usr_msg(message, len)
/*****************************************************************************/
   typ_msg         message;
   int             len;   /* length of user's message */

/*
 * recieve a user message, just too long to place into the message handler..
 */
{
   NET_MSG_HEADER  reply_header;
   Vpid            from, too;
   int             type, stat = -99, which_station;
   register NET_PROC_ENTRY_PTR entry;

   /* variables for HFP support */
   register int    mesg_num;
   int             d, r, m, max, k;

   from = message.net_header.from;
   too = message.net_header.too;
   type = message.net_header.type;
   which_station = GET_STATION(from);
   entry = find_proc(too);

   if (entry == NULL) {
      /*
       * do something, man.  we just got a message for a process
       * that we don't have.  The process may have died or
       * soemthing
       */
      /* return kid died EXCEPTION */
      reply_header.len = size_of_net_header;
      /* set up exception message */
      reply_header.too = from;
      reply_header.from = too;
      reply_header.type = UNKNOWN_PROC;   /* unknown process flag */

      stat = send_station(which_station, (char *) &reply_header,
               size_of_net_header);
      if (stat != OK) {   /* oops station went down ! */
         Fprintf(stderr, "user msg dropping station>%d\n", which_station);
         drop_station(which_station);
      }
      return;

   }
   if (type == USR_MSG_TO_PARENT) {   /* a message for a parent... */

      /* local variables for HFP support */
      register HFP_PTR HFP_params;
      register HFP_HEAD_PTR HFP_header;


      /* get the HFP parameters */
      HFP_params = entry->HFP_params;

      /* if the HFP has failed, ignore the message */
      if (HFP_params->failed) {
         return;
      }
```

```
mesg_num = inc_msgs_rec(entry, from);

if (mesg_num == NON_EXISTING_NET_PROC) {
   /*
    * we have received a message for a proc that is no
    * longer with us we should black list the station
    * that sent it
    */
   blacklist(GET_STATION(from), COMMISSION_ERROR);

   return;
}
/* get the HFP header for this message number */
HFP_header = get_HFP_header(HFP_params, mesg_num);

/*
 * retrieve the necessary number of identical messages that
 * constitute a valid response
 */
d = HFP_params->identical_needed;

/* is this a special case ? */
if (d == 1 && HFP_header == NULL) {

   /*
    * if number of identical responses needed is set to
    * one, we accept the first answer back always and
    * don't need to do a lot of message comparison and
    * other overhead
    */

   /*
    * check to see if the call has already been
    * responded to
    */
   if (mesg_num <= HFP_params->response_level)
      return;
   else
      /*
       * flag the call as being serviced so other
       * messages will be discarded.
       */
      HFP_params->response_level = mesg_num;

   /*
    * and return the message to the parent by executing
    * the remaining code under this if statement
    */

} else {

   /*
    * otherwise we must check the message for
    * correctness and subject it to HFP tests before we
    * can pass the mesg on
    */

  /*
   * are we expecting a message back for this message
   * number?
   */

  if (HFP_header == NULL) {
     /* unexpected message */
```

```
   /* blacklist the sending station */
   blacklist(GET_STATION(from), COMMISSION_ERROR);
   return;
}
/* is the timer set for this request ? */
if (HFP_header->timer == -1) {
   /* no, so set up the expiration timer */
   set_HFP_timer(GET_STATION(from), entry, HFP_header);
}
/* has the time expired on this request ? */
if (HFP_header->expired) {
   /*
    * stale reply has arrived, do some blacklist
    * management
    */
   de_blacklist(GET_STATION(from));
   /*
    * compare the message to the one we returned
    * to the user
    */
   if (len != HFP_header->mesg_list->len ||
       strncmp(message.message, HFP_header->mesg_list->message, len) != 0) {
      /*
       * different message, so blacklist
       * the sending station
       */
      blacklist(GET_STATION(from), TIMELY_ERROR);
   }
   if (++HFP_header->num_rec == HFP_params->replicates)
      HFP_cleanup(entry, HFP_header);
   return;
}
/*
 * add this message to the list of all messages
 * received returns the number of messages received
 * that are identical to this one, r
 */

r = add_message(HFP_params,   /* current HFP machine */
      HFP_header,   /* current message  */
      which_station,   /* sending station
            * number */
      message.message,   /* the actual message */
      len   /* message length */
   );

if (HFP_header->responded) {   /* but not expired */
   /*
    * do not return anything to the caller. but
    * we have added the message to the list of
    * received ones for processing at clean up
    * time
    *
    */

   if (HFP_header->num_rec == HFP_params->replicates) {
      /*
       * lets do a little cleanup, esp free
       * a timer entry
       */
      unset_HFP_timer(HFP_header->timer);
      HFP_cleanup(entry, HFP_header);
   }
   /*
```

```
            * don't send anything back to the user
            */
           return;
        }
        /*
         * do we now have enough identical messages to
         * constitute a valid answer ? is r == d ?
         */
        if (r == d) {
           /* flag the call as being serviced */
           HFP_params->response_level = mesg_num;
           /*
            * flag the data structure so the timer
            * interrupt knows that a valid response has
            * been sent
            */
           HFP_header->responded = 1;
           if (HFP_header->num_rec == HFP_params->replicates) {
              /*
               * lets do a little cleanup, esp free
               * a timer entry
               */
              unset_HFP_timer(HFP_header->timer);
              HFP_cleanup(entry, HFP_header);
           }

           /*
            * we can safely return the current message
            * it must be right since it triggered the
            * change to the response level, if some
            * other message had already done this, the
            * response level check would have prevented
            * entering this code
            */

        } else
           return;


     }      /* end if d != 1 */

     stat = send_local(too, GET_PID(entry->owner), message.message, len,
             message.msg_type, message.VT);


} else {      /* a message for a kid... */
     ++entry->mesg_sent;

     stat = send_local(entry->owner, GET_PID(too), message.message, len,
             message.msg_type, message.VT);

}


if (stat < 0) {      /* local process is down !  */
     delete_proc_entry(too);   /* delete entry    */
     delete_owner_entry(too);   /* delete ophanded kids.. */
     /* return kid died EXCEPTION */
     reply_header.len = size_of_net_header;
     /* set up exception message */
     reply_header.too = from;
     reply_header.from = too;
     reply_header.type = DEAD_KID;   /* died of unknown causes    */
```

```
        stat = send_station(which_station, (char *) &reply_header,
                size_of_net_header);
        if (stat != OK) {   /* oops station went down ! */
           Fprintf(stderr, "user msg dropping station> %d\n", which_station);
           drop_station(which_station);

        }
     }
  }
```

Appendix C

Glossary of Acronyms

API Application Programming Interface

BSD Berkley Software Distribution

CPU Central Processing Unit

FSP Fail-Stop Processor

HFP Halt on Failure Processor

NMR N-Modular Redundancy

NSERC National Sciences and Engineering Research Council

REM Remote Execution Manager

RPC Remote Procedure Call

TMR Triple Modular Redundancy

VITA

Surname:	Macdonald	Given Names:	Robert Noël
Place of Birth:	Montréal, Canada	Date of Birth:	June 17, 1958

Educational Institutions Attended:

John Abbot College	1979 to 1982
University of Victoria	1983 to 1994

Degrees Awarded:

B.Sc. (Honours) University of Victoria 1988

Awards :

University of Victoria	
The President's Scholarship	1984-85
The President's Research Scholarship	Jan. 1991 to Dec. 1992
Natural Sciences and Engineering Research Council of Canada	
Postgraduate Scholarship 1	1991
Postgraduate Scholarship 2	1992

Publications:

1. R. N. Macdonald and G. C. Shoja, "Implementing Halt on Failure Processors" *Proceedings of the 1993 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, vol. 1, May 1993, pp. 272 - 275.

Partial Copyright License

Title of Thesis:

Implementing Highly Available, Highly Reliable Virtual Processors

Author: [redacted]
Robert Noël Macdonald
September 25, 1994