Failure Detector for Somersault Distributed System

by

Michael Davidson

Submitted to the Department of EECS
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 26, 1995

© Massachusetts Institute of Technology, 1993. All Rights Reserved.

Author ....
Department of EECS
May 26, 1995

Certified by ....
Assistant Professor Frans Kaashoek
Department of EECS
Thesis Supervisor

Accepted by ....
Professor Frederic R. Morgenthaler
Department of EECS
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 17 1995

LIBRARIES

ARCHIVES

Failure Detector for Somersault Distributed System

by

Michael Davidson

Submitted to the Department of EECS on May 26, 1995, in partial
fulfillment of the requirements for the degree of Master of Science
in Computer Science

Abstract

This thesis describes design, implementation, and performance of a Failure Detector (FD) for Somersault distributed system. Somersault FD algorithms take a global view of failure detection in order to increase their accuracy and to distinguish between process and connection failures. The FD is divided into two functional parts. One, Foreground FD, provides on-demand detection of failures. Another, Background FD, reduces the vulnerability of Somersault by finding the hidden failures that may accumulate in the system over time. Foreground and Background FDs differ in their performance trade-offs. Foreground FD guarantees short detection times at the expense of reduced resolution between process and connection failures. Background FD provides higher accuracy of failure detection at the expense of long detection times. Scalability of FD algorithms does not limit the scalability of Somersault. Foreground FD requires only a small constant number of messages for every detection initiated. Background FD requires a quadratic number of messages; however, even for the largest feasible configurations of Somersault its overhead is acceptably low.

Thesis Supervisor: Frans Kaashoek
Title: Assistant Professor of EECS

HP Company Mentor: Paul Harry
Title: Member of Technical Staff

Acknowledgments

I have to thank Paul Harry of HP Labs in Bristol for his patient mentorship and technical expertise he shared with me over the period I worked in England, and Professor Frans Kaashoek of MIT for his speedy reading and insightful comments that helped me rethink the work I did in Bristol, and put this thesis together. I owe many ideas appearing in this thesis to the members of technical staff at HP Labs in Bristol who shared their knowledge with me. In particular, I have learned a lot about systems form with Steve Hinde, about fault-tolerance from Roger Flemming and about UNIX internals from Andrew Thomas. I also have to thank my fellow students Zohar Sachs and Adam Feder for their valuable comments on the style and content of this thesis.

Table of Contents

1 Introduction 9
1.1 Overview 9
1.2 Major issues addressed 10
1.3 Measuring Performance of Failure Detection 11
1.4 Proposed Failure Detection Algorithms 15
1.5 Thesis Structure 19

2 Related Work 21
2.1 A Simple Timeout-Based Approach 21
2.2 Kernel-Level Timeout Method 23
2.3 A Fully Distributed Symmetric Approach 24
2.4 Isis Asymmetric Site View Management 28
2.5 Algorithm comparison 32

3 Foreground Failure Detection 35
3.1 Overview 35
3.2 Performance considerations for failure detection 35
3.3 Assumptions 37
3.4 Method 37
3.5 Special Cases 43
3.6 Analysis 45

4 Background Failure Detection 47
4.1 Assumptions 48
4.2 Background Failure Detection Architecture 48
4.3 The Leader Algorithm 52
4.4 Analysis 54

5 Effects of Failure Detection on System Scalability 57
5.1 Overview 57
5.2 Architecture of Somersault 58
5.3 Assumptions and the Model 61
5.4 Analysis 82
5.5 Important trends 92
5.6 Ways to Increase Fault Tolerance of Somersault 94

6 Experiments 97
6.1 Experimental Setup 97
6.2 Experiments 103
6.3 Summary 112

7 Discussion 115
7.1 Future work 117

References 119

Chapter 1

Introduction

1.1 Overview

This thesis describes the requirements, design and implementation of a failure detector for Somersault, an asynchronous distributed system. The failure detector has a low detection latency, a low false positive rate and is scalable. These performance characteristics are achieved without imposing a high overhead on the system.

The majority of research on distributed fault tolerant computing concentrates on issues involving system availability (e.g. MTTF, number of communication or process failures that a system can survive), state coherence of the member processes, and scalability. Even though research in this area deals directly with failures, authors usually concentrate on the high level design issues, assuming that failure detection mechanisms are readily available to them. In this thesis we address the problem of building such failure detection mechanisms.

We show that the accuracy of failure detection can be increased by using the global information collected from different parts of the system. The improvement is not only quantitative but is qualitative as well. By using global information we can distinguish between various failure types (e.g. process, link and machine failures).

Our main conclusion is that in order to build an effective Failure Detection (FD) subsystem that is accurate, has a low overhead and scales well, it is necessary to divide it into two parts. One part, the Foreground Failure Detector, reacts to the immediate problems that occur during the system operation. It is designed to reduce the failure detection latency. The second part of the FD subsystem is the Background Failure Detector. It is

designed to reduce the number of hidden failures, which do not immediately affect the system's performance, but may cause a problem in the future. Background Failure Detection increases the MTTF of large systems by preventing the accumulation of multiple failures that may lead to a total system crash.

Our work is applicable to any distributed systems where processes are fail-silent. The greatest benefit of using our Failure Detector will be achieved in systems that maintain connections between all the processes in the system. This thesis discusses the design of the Foreground and Background failure detectors, and their implementation in Somersault, a distributed system being developed at HP Labs in Bristol, England.

1.2 Major issues addressed

There are several major difficulties in doing Failure Detection in Distributed systems. Let us consider some of them. First of all, there is no certainty in distributed systems. If a process A can not communicate with a process B across the network it may mean a number of things. For instance:

- Process B is dead
- The connection between A and B is broken
- B's machine is down
- The whole network is down

Deciding among these possible causes requires additional information. This information can be derived from looking at other failure indications in the system.

Next, there is a circularity problem involving the Failure Detection and the View management subsystems. The View Manager is responsible for maintaining a consistent view of the system, so that all the processes that are in the system agree on who the members are. The failure detector depends on the view manager to tell it what processes are there in

the system. The view manager expects the failure detector to tell it about the problems with any of those processes. As a result, there maybe various undesirable interactions between the view manager and the failure detector that may lead to the loss of constant system view. This is a serious problem and it has to be addressed in design of the failure detector.

Finally, scaleability is crucial for building Failure Detectors. Namely, if the Failure Detector generates too much traffic, it will slow the system down. As a result, more failure detections will be initiated, generating even more traffic. As a result, the whole system may collapse under the increased failure detection load. On another hand, if the failure detection is performed infrequently, failure detection traffic is not going to be a problem. However, multiple failures may accumulate in the system, thus increasing its vulnerability to total failure. Therefore, Failure Detectors have to be carefully designed to meet both requirements of not overloading the system and conducting the failure detection frequently enough to avoid the accumulation of multiple failures. This task becomes even more difficult for larger systems.

1.3 Measuring Performance of Failure Detection

Effective failure detection system is essential for obtaining the high levels of system availability. Here we consider the criteria for evaluating failure detection schemes and determine the desirable characteristics.

1.3.1 Semantic Guarantees

Failures Detected

Failures in a distributed system can be classified as follows [16]:

- **Value failures:** The value returned in response to a request for information is incorrect.

- **Crash failures:** The component just stops working, without bad side-effects, outputting nothing.
- **Performance failures:** The response occurs, but too early or (more usually) too late.
- **Omission failures:** There is no response to a particular request for service.
- **Duplication failures:** There are too many responses to a particular request for service.
- **Ordering failures:** Responses are ordered incorrectly.
- **Byzantine failures:** Unpredictable erroneous behavior.

Failure detection mechanisms are able to cope with certain subsets of these failures, usually not including the Value and Byzantine failures, because detecting those requites application level knowledge, which is normally not available to fault tolerant systems.

It is also important that the failure detection is "intelligent" about what it is doing. For instance in the case of faulty link, it is wrong to declare both processors on the ends of the link dead; instead, only one should be killed, and one should survive. It is desirable that the failure detection deals gracefully with LAN glitches and isolated processes. Most importantly, if failure detection involves selecting a leader, it is important to insure that there is only one leader selected, otherwise the processes maybe partitioned into separate groups, and the split-brain syndrome will result without the total network failure. It is impossible to enumerate all the things that can go wrong with a distributed system, however, the above examples are the ones that are of biggest concern to us; therefore they will be addressed in the algorithm analysis in the following chapters.

Consistency and Order

Detecting a failure of a process or a link by means of a time-out is relatively straightforward. The difficult part is notifying all the processes in the system about the failure in a consistent way that presents a system with a coherent membership picture. The most important parameter which defines the semantics of failure notification is ordering of the

membership events. There may be no ordering guarantees at all, or there may be a partial, casual, or total ordering. The stronger consistency is harder to achieve, but it allows for a simpler overall system design.

1.3.2 Performance Metrics

Apart from the "quality" of service it is important to know what is the overhead of a failure detection scheme both during normal operation and when there are failures in the system. It is important to know wether any additional messages are required or if the failure detection information is piggybacked onto other messages. If additional messages are indeed required, it is necessary to know the order of growth of these messages, i.e. whether the number of messages is linear, quadratic, etc. to the number of the processes in the system. Finally, knowing how the Failure Detector behaves under different loads may be important in fine tuning the system.

1.3.3 Service Guarantees

There are several dimensions for classifying failure detection schemes. Ordering by *completeness* and *accuracy* is proposed in [8]. Completeness relates to identifying the faulty processes. It is said to be *strong* if "every process that crashes is permanently suspected by every correct process"; it is said to be **weak** if "every process that crashes is permanently suspected by some correct process." Similarly, accuracy is said to be **strong** if "correct processes are never suspected" and it is said to be **weak** if "some correct process is never suspected" [8].

For the purpose of building a working system, however, a different set of classification is required. It is critical to be able to find the failed processes fast to avoid long recovery times. However, because recovery maybe expensive, it is important to not declare correct processes dead, for such declaration will lead to unnecessary recovery overhead. There-

fore, for the purpose of our analysis we adopt the following metrics — **error detection latency** and a **false positive rate**. The latency characterizes a mean delay required to detect a failure of a fail-silent process. False positive rate characterizes the percentage of the processes that were suspected to have failed, but were in fact correct.

Finally, we should be cautious of the system accumulating undetected errors, which, when discovered, may lead to a total system failure. Therefore, it is important that all failures are discovered in a bounded amount of time.

1.3.4 Realistic Failure Detector Requirements

The Ideal Failure detector would provide the ordering of failures, would be accurate, complete, would have a small overhead and a bounded detection time, and would not have a single point of failure, would not partition and would not generate the bursts of traffic when failures are detected.

The ideal failure detector cannot be built, however, because its requirements can not be implemented in one system. First, let us note that in order for all processes to get a consistent view of failures in the system, it is necessary to perform a broadcast every time a failure is discovered. Therefore, if we want consistency we must accept having traffic bursts at failure. Second, we can not guarantee accuracy, that is that only the faulty components of the system will be declared dead. Finally, one can derive form the result by Fischer, Lynch and Paterson [14], that there is no way of reliably checking weather a process is dead or is merely slow. Moreover, same result shows that reaching consensus about the faulty process is impossible [14].

Taking this into account, we conclude that the closest approximation to the ideal failure detector would be a failure detector that provides ordering of failures, it may not be accurate, but will be complete, would have a small overhead and a bounded detection

time, would not have a single point of failure, and would not partition, however, it may generate traffic bursts when failures are detected.

1.4 Proposed Failure Detection Algorithms

Here we discuss the Failure detection algorithms that meet the performance requirements outlined above, and work well within Somersault.

1.4.1 Somersault

Somersault is a distributed fault-tolerant system for supporting telecom services. It provides fault-tolerance with respect to hardware, software and communication failures. It is expected to operate continuously for periods as long as 20 years, and support a real-time processing capability. It is based on a system called Manetho [13], [18]. Somersault runs on top of UNIX, using TCP/IP stacks on a LAN.

Somersault provides fault-tolerance through the use of active process replication. Somersault is guaranteed to survive any single link or process failure. However, it may not be able to survive multiple failures, as some of the state information that is needed for recovery may be lost.

Somersault consists of processes and links between them. Unlike many other fault-tolerant systems, Somersault is connection oriented. Connections are elements of the system, just like processes are. Every process maintains connections with every other process in the system. Most of the connections are idle most of the time. However, during failure recovery, most of connections carry recovery traffic. Broken connections are fixed by killing off one of the end processes, and then taking a recovery action for that process. Because process recovery in the extreme cases may involve sending Gigabytes of state information, it is expensive, both in terms of latency and network resources.

In order to support the normal operation of Somersault over an extended period of time, failure detector must find both process and link failures. Regardless of the application traffic it must find failures fast enough in order to prevent the system from accumulating multiple failures and crashing.

In order to support Somersault's real time processing capabilities, the failure detector has to have a low latency (under 300ms). With a 300ms failure detection time, it would be able to detect failures and notify the fault handling mechanisms in time to take a fault-handling action without disrupting the quality of real-time service.

This information is sufficient to justify and understand the failure detector design. A more comprehensive description of Somersault is provided in Chapter 5, where a detailed description of Somersault architecture is needed in order to analyze the scalability of the system.

1.4.2 Design

We solve the FD problem by introducing two classes of Failure Detection Algorithms. Both use time-outs, and both are not absolutely accurate. The best we can do is to approximate ideal behavior using the assumptions we can make about our system.

Both approaches to failure detection take a global view of the system, in order to increase their accuracy, and to distinguish between the process and link failures. Both assume that there exists an independent view management mechanism that insures that all surviving processes in the system see the failures they discover in the same order (we describe such a view manager in Section 2.4). Whenever a failure has been discovered by either detector, it is reported to the view manager which insures that it becomes known to the rest of the system.

The Foreground Failure Detection responds to failures that occur when a process tries to communicate with another process and does not succeed. In this case Foreground FD attempts to determine the cause of failure accurately and quickly. Foreground Failure detection is subject to trade-off between the speed and the accuracy of failure detection. When it is very fast its accuracy is low. In order to increase the accuracy it is necessary to increase the duration of failure detection.

Background FD addresses the long term viability of the system. Its task is to prevent the system from accumulating failures that are not encountered during the normal operation, but may cause a system crash during process recovery.

1.4.3 Foreground (event driven) scheme

When a process can not communicate with another process the immediate objective is to determine whether the process or the channel to that process are dead. This can be done by asking some other process to ping the suspect. We call this *arbitration*, and the process that is doing the additional pings an *arbiter.* Arbiters are not special. Any process in the system can be asked to play the arbiter role at some point

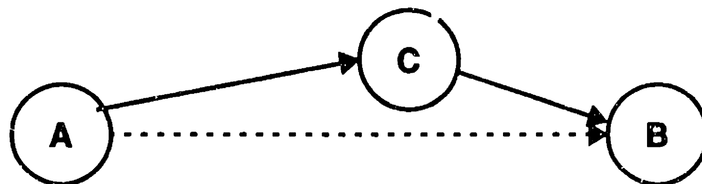

Figure 1.1. Distinguishing between a channel and a process failure. An arbiter C pings B on A's behalf. If C is able to communicate with B, then the AB link is broken; otherwise B is dead.

Consider the Figure 1.1. Process A sends a message to process B, but does not receive an acknowledgment. After a while A begins to suspect that either process B is dead or the link leading to B is broken. A asks C to be an arbiter and ping B. After pinging B, the arbi-

ter returns the result of the ping back to A. Now A can decide whether the process B is dead, or the AB link is broken. If C reports that pinging B was unsuccessful, A decides that because other processes can not communicate with B as well, B must be dead. Alternatively, if C reports that B acknowledged its ping, A decides that it is likely that the AB link is broken, but B is still up and running. Either way A reports its conclusion to the view manager which than notifies the remaining processes.

1.4.4 Background FD mechanism

Background failure detection finds failures in components (both links and processes) that are temporarily idle, and thus are not subject to Foreground Failure detection. Compared to Foreground, Background Failure Detection operates under much looser time constraints, but potentially has many more components to check (if most of the links in a large system will be idle most of the time). Thus the Background FD algorithms trade the speed of failure detection for reduced traffic generated by it.

The Background Failure Detection algorithm runs on two types of processes — a leader process and normal processes. There should be only one leader process in a system, the rest of the processes are normal. Any normal process can be chosen to become a leader. The algorithm also defines the maximal period of time after which any failure should be detected. We call it a self-checking period.

Normal processes in Background FD ping all other processes in the system, spacing pings to every process randomly within an interval from half to one self-checking period. Whenever a ping is not acknowledged, a *suspicion* report is sent to the leader process.

The leader process keeps track of all the suspicion reports it received during the sliding window of one self-checking period. Using a sliding window allows the leader to filter out transient failures that were caused by variation in communication latency. Whenever a

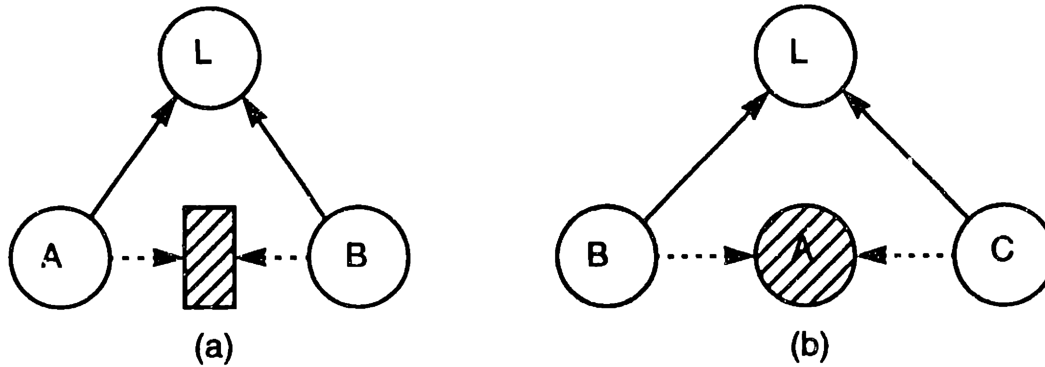

Figure 1.2. Background Failure Detection Scenarios. Dotted lines are pings, solid lines are suspect messages. L is a leader process; A, B and C are the normal processes. (a) Link failure detection. (b) Process failure detection.

leader gets a report saying that process A suspects process B, and a report that process B suspects process A within the same window, the leader decides that the link AB is broken. Alternatively, if two processes report that they suspect a process A during the same window, leader decides that the process A is dead.

1.5 Thesis Structure

In this chapter we outlined the requirements for failure detection and the algorithms developed. The following chapters elaborate the description of the failure detection subsystem presented here. In Chapter 2 we discuss the existing failure-detector designs, and evaluate their applicability to Somersault. In Chapter 3 we describe the Foreground Failure detection mechanism in detail. In Chapter 4 we concentrate on Background Failure detection. Chapter 5 analyzes the relationship between the reliability of components, performance of the failure detector and scalability of the system. In Chapter 6 we present the experiments verifying our failure detection algorithms. Finally, in Chapter 7 we discuss the lessons learned form building the failure detector and suggest the possibilities for future work.

Chapter 2

Related Work

This chapter analyzes four different approaches to failure detection according to the criteria suggested in Section 1.3 above. Each of these approaches is discussed in turn. For each method we analyze it's algorithm, the semantic guarantees it provides, it's complexity, service guarantees and overall rating. The comparison of all these schemes is summarized in table Table 2.1 at the end of this section.

2.1 A Simple Timeout-Based Approach

A simple minded approach to failure detection was described by Loques and Kramer [25]. Their system detects failures of machines participating in computation by using a timeout mechanism.

Algorithm Description

A special class of processes, *stand-by managers*, periodically send messages to the rest of the processes in the system. If the reply from a particular process does not arrive within a certain time limit, the stand-by manager decides that the process is dead and invokes a stand-by copy for that process.

Semantic Guarantees

Stand-by managers, however, may fail to detect a machine failing and recovering quickly. If a machine goes down and recovers in between two consecutive messages from the stand-by manager, the manager would never suspect that the machine has crashed. This is unacceptable if the stand-by manager has to take some corrective action whenever a machine crashes. Therefore, this method requires some additional mechanism to insure that all failures get detected, for example the use of version numbers. Even if a machine

fails and recovers between the two messages from its manager, the manager will notice that the machine in question responded with the new version number, and this must have crashed and subsequently recovered.

More importantly, this method of failure detection does not provide a mechanism to insure that all the processes in the system have a consistent view of the system. For example, processes may receive notifications of other machines failing in different order. Even worse, this method is subject to a single point of failure. Namely, no one checks on the status of stand-by managers. Therefore, if a stand-by manager fails, there is no way to detect this failure, and consequently any other failure among the processes it was checking on.

Complexity

In its simplest form the algorithm described in [25] hardly incurs any overhead. However, it has to be extended with a mechanism for consistently informing surviving processes about the discovered faults. In order to insure a minimal consistency requirement of all surviving processes knowing about a failure, two-phase commit should be added to this algorithm. Running a two-phase commit will incur an $O(n)$ messages overhead for every failure. Providing a stronger consistency of every surviving process observing the failures in the same order is likely to incur additional costs.

Service Guarantees

Service guarantees of this scheme are as follows. The failure detection latency is proportional to the number of time-outs required on the "ping" messages sent by the stand-by managers to declare a process dead, and the false positive rate is inversely proportional to this number. The number of additional "ping" messages is proportional to the number of process in the system. The time to detect failures is proportional to the period of the ping messages.

Overall rating

The advantage of this algorithm is its simplicity. However, it comes at a cost of poor semantic guarantees. It would not be acceptable to use this algorithm in Somersault, because it does not even attempt to provide a consistent view of the system. Modifying it to suffice the consistency requirements of Somersault will result in building a whole new voting layer on top of it. Therefore, some of the algorithms considered below maybe better suited for modification and use in Somersault.

2.2 Kernel-Level Timeout Method

A similar timeout-based failure detector is implemented in the ROSE operating system [26].

Algorithm Description

In ROSE, the monitoring processes are part of the kernel and can be selectively turned on and off. They do not, however, make decisions about what to do after the failure is detected, instead they notify the application processes.

Semantic Guarantees

This may be a more efficient solution because running the monitoring processes in the kernel minimizes the number of context switches required to perform failure detection. However, this approach does not resolve any of the drawbacks the previous approach had. In fact, it introduces a new problem. The communication channel which is being tested is a kernel-to-kernel multicast, and is different from the communication channels used by the applications. Therefore, it may detect communication faults which do not affect the applications, or ignore some faults which do not manifest themselves on the kernel level [30].

Complexity

The number of messages is at worst proportional to the square of the number of physical machines in the system.

Service Guarantees

Failure detection latency for the Kernel Level Time-outs is similar to the one in the previous method. However, the false positive rate is worse because even though the methods are very similar, this method tests the kernel-to-kernel channels that are different from the ones used by the applications that need the failure detection results. The failure detection times are unbounded, because some channels are never tested before they have to be used by the system.

Overall rating

This is probably an efficient practical solution. However, it will not work for Somersault for two reasons. First it involves altering the kernel, which is out of the scope of this project. Second, it is little more than a trivial timeout-based failure detector. Therefore, it does not provide the semantic guarantees required by Somersault.

2.3 A Fully Distributed Symmetric Approach

The algorithms considered above offered little more than local timeout-based failure detection. In order to find an algorithm with better semantic guarantees, we turn to considering distributed algorithms which take a more global view of the system. A fully distributed symmetric protocol is discussed in [7]. The protocol consists of two interacting protocols: Node Up and Node Down.

Algorithm Description

The Node Down protocol is decentralized. It is invoked by a process (process A in Figure 2.1) which can not communicate with some other process (process C) in the system. Process A broadcasts C's identifier (ID) and the version number to every other process in the system. Upon receipt of the first Node Down message other processes mark that process as being down if the version number in the message is greater or equal to the locally cashed version number. Then the processes also run the Node Down protocol rebroadcasting the message to all the processes in the system. In the original description of the algorithm broadcast is implemented as a series of synchronous point-to-point communications. A process first communicates with his "right neighbor" in a virtual ring, waits for an acknowledgment; then communicates with the next process, all the way around the circle until it communicates with and gets reply from its leftmost neighbor. However, there is no apparent reason why Ethernet broadcast or Internet multicast could not be used.

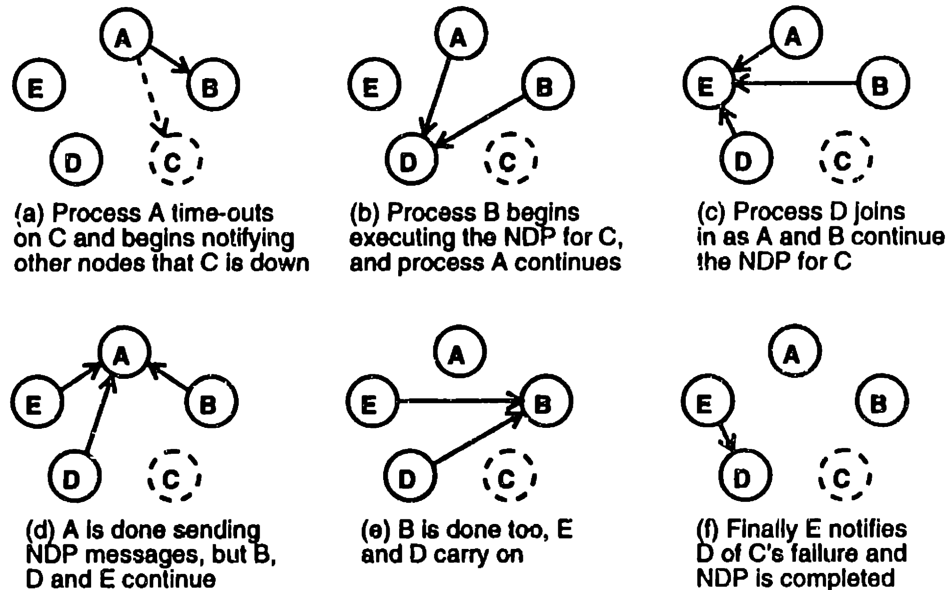

Figure 2.1. Node Down Protocol is highly redundant, and involves every process talking to every other process in the system

This scheme ensures that all surviving processes hear about the process failure. When a process recovers from a crash it runs a Node Up protocol, broadcasting its new version number to all other processes. Those processes mark the recovering process as being up if its version number is higher than the locally cashed one. Then they reply with their current version numbers, thus allowing the recovering process to construct a consistent view of the system.

Multiple failures can be discovered during the execution of the Node Down Protocol, when processes fail to reply to Node Down messages. Interactions between the Node Down and Node Up protocols are resolved by using version numbers. So, for example, if processor B believes that processor C version 3 is up, then when B gets a Node Down message for C version 1 or 2 it will take no action; version 3 or greater will cause B to mark C as being down.

If a process can not communicate with any other processes in the system it declares itself isolated and instead of trying to run a Node Down Protocol for every other process it declares itself isolated. Isolation ends when a process receives a message from another process in the system.

Semantic Guarantees

This algorithm detects crash failures of processes and communication links. In some cases, like process isolation, it allows a process to wait until it is reconnected with the rest of the system. This mechanism allows the system to survive LAN glitches, when all possessors get disconnected from each other for a short period of time. Moreover, there is no danger of splitting the processors between several leaders, because the algorithm is fully symmetric and it has no leaders.

There is no notion of process views in this algorithm, instead there are process joins and crashes. Because different processes may see events in a different order, this algorithm does not provide a consistent view of the system at all the times, and because of that the images of the system that processes have, can not be ordered. For example, in Figure 2.1, suppose that D decides that E has failed at the same time as A decided that C has failed. Then, the order of process crashes as seen by A and D will be different, and therefore their views of the system will not be consistent and can not be ordered.

Complexity

This algorithm consists of two protocols. The Node Down Protocol requires every processor to communicate with every other processor in the system and receive a reply. So even if an efficient multicast mechanism is used, the necessity to get replies brings the total number of messages to $O(n^2)$, where n is the number of processes in the system.

Node up protocol is centralized and requires a processor to communicate with all other processes in the system once, and get a reply. Again, because the reply is required, the total number of messages is $O(n)$.

Therefore the total complexity of this algorithm is determined by the number of messages sent during the Node Down stage, and is $O(n^2)$ messages.

Service Guarantees

The Failure Detection Latency is determined by the number of retries necessary for timing a connection out, plus the time it takes to propagate the Node Down messages to all the processes in the system. Depending on the implementation of the broadcast mechanism the later may take anywhere from one broadcast to $O(n^2)$ messages. Failures may go unnoticed for a long time, until one failure is found and the whole system is re-checked.

The False Positive rate is again dependent on the number of retries required for a timcout, and is slightly improved by a special case handling of a total process isolation case.

Overall rating

The performance of the algorithm can be slightly improved by using an Ethernet-like multicast mechanism. However, the number of messages required will still be quadratic to the number of processors. Therefore, this algorithm is not likely to scale well. Failure detection latency maybe long too. In addition, the processor views provided by this protocol are inconsistent and may not be ordered. Therefore, this algorithm in its original form cannot be used by Somersault.

The key to adapting this algorithm to Somersault would be providing a way to make the view of all joins and leaves consistent across the system. Another aspect that would need significant improvement is performance. Every process hears about every join or leave from every other process in the system. There are more economical ways of reliably distributing the information across the system

2.4 Isis Asymmetric Site View Management

The main drawbacks of the symmetric algorithm described above are the great number of messages required in order to propagate the view information, and an inconsistent order in which this information is delivered to processes. An asymmetric approach attempts to cope with this problem by selecting a leader which coordinates the activity of other processes, thus reducing the amount of communication required and introducing more order to the view change process [4].

Algorithm Description

Processes are ordered according to their unique IDs. A process with the highest ID is said

to be a leader. Every process maintains a view of the system containing the member process IDs and their incarnation numbers. When a system is cold-booted all the cite views are initiated to a consistent state. Site views change in an ordered sequence V_i, V_{i+1}, V_{i+2}, etc., and the content of views is consistent across all the processes in the system.

View change is carried out in a form of a two-phase commit protocol. When a process joins in or fails the leader process is notified. The leader increments his view number and stops accepting the messages from the processes with lower view numbers. Then the leader sends out a broadcast message to all other processes requesting a view change and notifying them of the events included into it.

Upon receipt of a view change request processes stop accepting messages from other processes not in the proposed view. Processes reply positively to the request if they have not seen the proposed view before, or the previous view is completely contained in the new one. Otherwise, the process has seen some event which is not included into the proposed view. In this case the process responds negatively, and includes such events into the response.

If the responses to the view change request were all positive, the leader commits the view and sends out a commit message to all the processes in the view. If there are some negative responses, or any new events had happened, the leader includes them into a proposed view, increments its number and sends out the view change request again.

Finally, if a leader crashes, the process with a next highest ID becomes a leader, includes the failure of the old leader into a proposed view, and initiates the view update by sending out the view change request.

Semantic Guarantees

Unless processes keep on continuously joining or leaving the system, a coherent view will

be achieved across the system. If a leader survives during the voting process, it will eventually hear about all the events in the system, and therefore will commit an appropriate view. If a leader crashes during the voting, a new leader will still be able to obtain all the necessary information about events in the system and will eventually commit too.

Because the view change process is centralized, the sequence of views will be exactly the same at all the processes in the system. The leader also can take a more global view on the failure information coming to it. For instance, it may be able to distinguish between the link failures and the failures of processes at the endpoints.

Still, there exists a problem with this algorithm. It assumes that the next leader is always selected uniquely and in order derived from the process IDs. However, it is possible that two processes will attempt to become the leaders at the same time. This will result in partitioning of the system, without having an actual system failure. Consider the scenario in Figure 2.2.

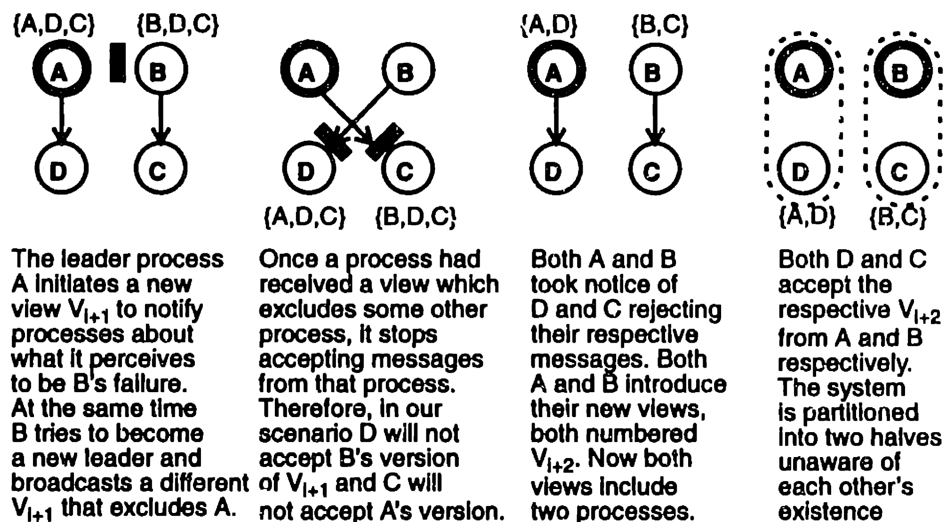

Figure 2.2. Partitioning due to a single communication channel failure: during the view V_i a link between the leader process and the next process to become a leader fails.

This situation can be easily avoided by using an ordered multicast to present the new system view. In fact, ISIS has such primitive, named GBCAST [4], for group broadcast.

the problem is that a complicated protocol like GBCAST requires a working failure detector. Therefore, to avoid circularity it can not be used in Failure Detection.

ISIS gets around this problem by discarding messages form the processes that are not in the current view and introducing a new message type "you are dead." This message is sent in response to a message received from a process not in a current view. When a process receives a "you are dead" message it terminates. Here is how this mechanism works in the competing leaders scenario. A process receives a view from one of the leaders, and when the other leader sends it a view, the process responds with "you are dead" message, causing the other leader to die. It may be the case that both leaders will get killed, and then the next candidate or a set of candidates will try to become a leader. Thus, ISIS avoids partitioning, but may kill processes, increasing the recovery cost.

Complexity

The algorithm uses a two-phase commit involving two rounds of $O(n)$ messages. Thus even if multicast is implemented as a sequence of point-to-point communications of the leader process with the rest of the system, the whole failure detection-notification cycle will be completed in $O(n)$ time.

Service Guarantees

Failure detection latency of Isis's Asymmetric Distributed Approach is proportional to the number of pings required to suspect that the process is dead, plus the cost of two system-wide broadcasts. False positive rate is inversely proportional to the number of failed pings required to suspect a process. Failures may go unnoticed for a long time, because only the channels forming the logical rings are checked regularly.

Overall rating

This algorithm offers a relatively low overhead, while providing robustness. Its major drawback is the possibility of process partitioning, even if the network is operating correctly. It is a rather unlikely event, but because it will lead to a "split brain" syndrome, it has to be taken into account.

2.5 Algorithm comparison

Now let us compare the performance of the failure detection strategies we have discussed above against the optimal failure detection requirements outlined in Section 1.3.4. Recall that the optimal failure detector should provide the ordering of failures, it may not be accurate, but should be complete, has a small overhead and a bounded detection time, and would not have a single point of failure, should not partition, however, it may generate the bursts of traffic when failures are detected

	Semantics	Service Guarantees				Performance		
	Ordering of failures	Accuracy	Completeness	Single point of failure	Danger of partitioning	O(msg) per failure	Traffic Bursts	Bounded detection
Simple Timeout	●	●	●	●	○	○	○	○
Kernel Level	●	●	●	●	○	○	○	●
Symmetric	●	●	○	○	○	●	●	●
Isis Asymmetric	○	●	○	○	●	○	●	●
Optimal	○	●	○	○	○	○	●	○

Table 2.1: Comparative evaluation of four failure detection schemes. ○ = good, ● = bad (for message order in the last column ○ = O(n), ● = $O(n^2)$). It is important to notice that while there are various good and bad aspects for every method presented here, none provide an accurate detection.

If we compare the characteristics of the failure detectors considered above to those of the Optimal failure detector from Table 2.1, we see that the Isis's Asymmetric failure detector provides the best match to the requirements of the Optimal detector. Isis deviates from the Optimal Failure detector in that it may partition and does not have a bounded detection time. Other schemes have more drawbacks, most indictably, they do not provide ordering of failures, which is necessary in our case.

If we could fix the drawbacks of the Isis Failure detector, we would be able to use it in Somersault. The danger of partitioning arises because the mechanism for determining the state of the processes is simplistic, and does not take into account the fact that communication failures may affect the system just as much as process failures. The detection time is not bounded because the failure detection is event driven. If there are no external inputs all Isis components may break one by one, and this would not be noticed until the system is needed, at which point there maybe too many latent failures in it to run properly.

In the following chapters we consider two algorithms that fix the above problems with Isis Failure Detection. These algorithms take a globalized approach to failure detection to distinguish between process and communication failures. Consequently, they improve the accuracy of Isis Failure Detection and reduce the danger of system partitioning. These algorithms also guarantee a bounded failure detection time regardless of the traffic patterns in the system. Therefore, they reduce the vulnerability of the system to hidden failures.

The algorithms described in the following chapters, however, do not replace the Isis View Manager. They improve on Isis View manager failure detection, but do not provide the rest of its functionality. Moreover, they relay on the View Manager's ability to maintain the consistent view of the system and inform the processes in the system about membership changes.

Chapter 3

Foreground Failure Detection

3.1 Overview

This chapter presents a Foreground failure detection algorithm that distinguishes between the process, link, machine, and network failures. The algorithm is based on the assumption that multiple simultaneous process and link failures are unlikely; and when multiple link or process failures are present in the system, they are caused by the machine or network crashes. The inputs to the algorithm are the suspecting and the suspected processes; the outputs are the state of the suspected process, the state of its link to a suspecting process, plus possible conclusions about the state of other processes or links as well as machines and the network as a whole. The algorithm works by asking independent processes to judge the status of a process in question by communicating with it.

The algorithm incurs a low overhead of seven messages per suspected failure, and is able to detect machine failures quickly, thus reducing the number of failure detection messages in the system. However, its' merit is limited by requiring a high network bandwidth and low latency, and fail-silence of machines and processes. When combined with the Isis view management algorithms discussed in Section 2.4, our algorithm performs as well as the Optimal Failure detector described in Table 2.1, on page 32.

3.2 Performance considerations for failure detection.

Doing timely failure detection is important because it decreases the vulnerability of the system due to multiple components being "broken" at the same time. However, fast failure detection is also key for high performance of Somersault. Suppose the system is working

on processing a request, and one of the primary processes does not respond to the messages sent to it. The system has to decide whether that process is dead, the machine it is running on has crashed or the link connecting to that process is broken. Based on the type of failure the system has to work out the recovery strategy, and carry it out. All of this has to happen without distracting normal service provided by the system.

From the external point of view every incoming request has to be processed by the system within 250 ms. (This is a telecom requirement). The typical actions, which happen during this time if a component failure is suspected are:

- Identify the type of failure and the faulty component;
- Initiate the recovery of surviving processes in the affected recovery units;
- Bring the recovery units up to date (this does not mean a complete recovery with replication, but rather having an up to date state in one of the processes);
- Continue servicing the external request.

Most of the time in failure handling is likely to be taken by bringing the state of the surviving processes up to date. In a Somersault paradigm a secondary process that is trying to catch up with the state of the dead primary has to communicate with every other recovery unit in the system requesting antecedent information from them. For a system of a reasonable size sending out all these requests and collecting and processing the replies is likely to take most of 250 ms. allotted for coping with failures. Therefore, Foreground failure detection has to be relatively fast, on the order of 10 ms.

In the process of Foreground failure detection there are several possible outcomes we are looking for. They are

- Process failure
- Link failure
- Machine failure
- Total network failure

Let us consider how we could distinguish between them using a limited number of messages in a limited amount of time.

3.3 Assumptions

The quality of failure detection strongly depends on the properties of the network and the protocols running over it. Currently we do not know what they are, however, we can make several assumptions that are true for the networks and protocols that Somersault is likely to use.)

- Communication links are lossless FIFO channels
- Network does not delay messages by more than t_{del}, unless the messages are sent on a broken link.
- Network does not partition

Using the above assumptions we can build a scheme that satisfies the timing requirements stated in the Section 3.2.

3.4 Method

The method presented here extracts the information about the likely nature of failures in the system from the communication patterns between processes. It concentrates on discriminating between the process, link and machine failures, but detects total network failures as well. In order to differentiate between the above failure modes, the method utilizes communication between processes on different machines.

3.4.1 Concept

If a process A tries to communicate with a process B and fails to, there could be several reasons for it. Communication failures could be due to any of the reasons listed in Section 3.2 above. Process and link failures are the easiest to detect, while classifying

machine and network failures requires more information. The key in all scenarios is to ask an independent process to arbiter the state of the system component in question. In our example (Figure 3.1) of A not communicating with B an additional process C can be asked to communicate with B. If C is successful, then B is alive and the link between A and B must be faulty. Otherwise, B is likely to be dead.

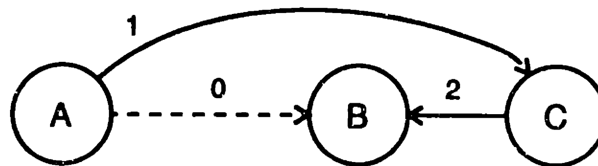

Figure 3.1. An arbitration process for classifying process and link failures. When A tries to communicate with B and fails, C is asked to communicate with B. If C is successful, then B is OK, and the AB link is broken, otherwise B is dead

3.4.2 Topology

The above arbitration works as long as it is possible to get in touch with a process C. However, the relative positions of A, B, and C affect the outcome of arbitration. If all three processes are on the same machine, then there is no possibility of a machine or network failures interfering. Therefore the conclusions reached using the above method are likely to be correct. If processes A and B reside on different machines, then there are there choices for placing C: on A's machine, on B's machine, and on the machine that hosts neither A or B.

- If C is on A's machine, C's inability to communicate with B implies one of the following: B is dead, CB link is broken, B's machine has crashed, or there is a global network failure.
- If C is on B's machine, C's inability to communicate with B implies that most likely B is dead.

- If C is on a machine that hosts neither A or B, then C's inability to communicate with B implies on of the following: B is dead, CB link is broken, B's machine has crashed, or there is a global network failure.

The above cases point to the fact that for the most definitive results the arbiter process should reside on B's machine. However, just picking the arbiter on B's machine may not solve the problem. Namely,

- If A can not contact C on B's machine, this may mean one the following: the machine hosting B and C is dead, C is dead, AC link is dead, or there is a global network failure.

This complication suggests that more information is needed in order to distinguish among the possible causes of failure. This information can be obtained by using an additional arbiter D on a different machine. When an additional arbiter is used, the following logic applies. If A cannot contact either of the arbiters C or D, it is likely that there is a network failure. If either of C or D can contact B, then B is running, and consequently B's machine is up, but the AB link is broken. Finally, if A cannot contact C, but it can contact D, and D can not contact B, then it appears that both B and C are dead. Consequently, there is a suspicion that B's machine has crashed, because B and C run on the same machine, and the only common failure mode we consider is machine failure.

3.4.3 Refutation

The scheme described above will work quite well. It uses shorter time-outs than a simple minded failure detector, because it subdivides the time available for failure detection into several stages. Instead of just waiting on a link for some time **T** and then declaring the process on the other end dead, our failure detector waits for a *fraction* of **T** and then launces an inquiry to find the nature of the failure that caused the timeout. In case the network is heavily loaded using longer time-outs could be essential for providing accurate detection. Therefore, we propose the following amendment to the above algorithm.

- In case of a timeout, run the failure detection, but before declaring the suspected process/link dead, check if during failure detection the missing reply was received. If yes, disregard the conclusions about the state of the suspected process. If no, report the results of the failure detection to the view manager.

The amended algorithm now uses the timeous which are effectively as long as the time-outs of a simple-minded algorithm, but in case of failure it is able to better determine its cause.

3.4.4 Algorithm

According to the above observations the arbitration is structured as follows. First A tries to pick an arbiter C on B's machine. If successful, C's conclusions are directly translated into the results of failure detection. Otherwise, an arbiter D is picked on another machine in order to distinguish between the link and the process failures for C, and to differentiate between the machine and network failures (see Figure 3.2).

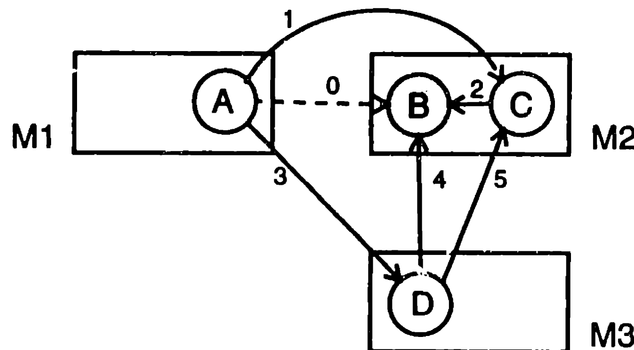

Figure 3.2. Detection mechanism. A tries to communicate with B, but fails to.Then A asks a process C on B's machine M2 to arbitrate the failure. If A can communicate with C, then C's conclusions are taken to form decisions about the failure of B/AB. If A can not communicate with C, an additional arbiter D is picked on a machine M3 which hosts neither of the processes in question. D helps to decide on the status of C, and to distinguish between the failure of machine M2 and a total network failure

Let us consider the steps that Foreground Failure Detection thread takes on timeout of a message *msg* sent by a process A to a process B. Assume that A is running on machine M1, and B is running on machine M2, and there are more than two machines in the system

(see Figure 3.2). These following actions are taken by the failure detection thread created on A to handle the suspected failure:

1. Select an arbiter process C on B's machine M2. Any process running on M2, except for B, can be selected to be an arbiter.
2. Select a machine M3 that is different from B's machine M2, and A's machine M1. Randomly select another arbiter process D among the processes running on M3.
3. Send an *arbitration request* to processes C and D, asking them to ping B.
4. Suspend this failure detection thread on A for the time allocated for arbitration
5. When the arbitration time is over, first check if B has acknowledged the *msg* that triggered the start of this failure detection thread. If B has indeed acknowledged during the arbitration, there is no failure. Therefore, terminate failure detection on B.
6. In case B has not acknowledged the *msg*, check if C has acknowledged the arbitration request. If yes, check C's conclusion:
   — If C contacted B successfully, then B is up, but the AB link is broken;
   — Else B is dead.
7. If C has not acknowledged the arbitration request, check if D has acknowledged. If D has indeed acknowledged, check D's conclusion:
   — If D contacted B successfully, then B is up, but the AB link is broken.
   — Else, suspect that M2 has crashed or is isolated; for immediate purposes conclude that B is dead.
8. If D has not acknowledged the arbitration request either, we do not have enough information to make any conclusions about the state of B, except that the network might have failed

Table 3.1 presents a summary of decisions and justifications made by the arbitration algorithm described above. The conclusions in this table are based on the assumption that multiple process or link failures happening at the same time are less likely than single failures, unless they are caused by the machine or network failures. For example consider a case when A is not able to communicate with C, and D is able to communicate with B but failed to communicate with C.

First consider B. A was not able to communicate with it, because either B was dead, the AB link was broken, B's machine has crashed, or the whole network was down due to a temporary glitch or a permanent failure. A was able to communicate with D, therefore

the network is up. D was able to communicate with B, therefore B is alive and B's machine has not crashed. This leaves us with the following conclusion: the AB link is faulty.

A C	C B	A D	D B	D C	Possible causes of failure	Most likely cause	Decide B/AB	Sus-pect
1	1	-	-	-	$(B+AB) \cdot (\neg B)$	AB	AB is broken	-
1	0	-	-	-	$(B+AB) \cdot (B+CB)$	B	B is dead	-
0	-	1	1	1	$(B+AB) \cdot (C+AC) \cdot (\neg B) \cdot (\neg C)$	$AB \cdot AC$	AB is broken	AC is bro-ken
0	-	1	1	0	$(B+AB) \cdot (C+AC) \cdot (\neg B) \cdot (C+DC)$	$AB \cdot C$	AB is broken	C is dead
0	-	1	0	1	$(B+AB) \cdot (C+AC) \cdot (B+DB) \cdot (\neg C)$	$B \cdot AC$	B is dead	AC is bro-ken
0	-	1	0	0	$(B+AB) \cdot (C+AC) \cdot (B+DB) \cdot (C+DC)$	$B \cdot C$	B is dead	B's machine
0	-	0	-	-	$(B+AB) \cdot (C+AC) \cdot (D+AD)$	$(B \cdot C) \cdot (D+AD) + AB \cdot AC \cdot AD$	Not enough info	network failure

Table 3.1: An arbitration summary. The first five columns show the outcomes of communications on the links. "1" is OK, "0" is fail, and "-" stands for the communication that does not happen, and therefore does not matter. The equations in the cells use boolean operators: "·" is AND, "+" is OR, and "¬" is NOT.

Now consider C. Let us note that C's status was not essential for Somersault's normal operation at the point of suspected failure of the process B. Therefore all decisions about C's status are more of recommendation and should be referred to when C is essential for Somersault's normal work. A was not able to communicate with C, and we know that the machine C is on is running, and the network works. Therefore, C is either dead, or has a broken AC link. D was not able to communicate with C, thus either C is dead or the DC link is broken. It is more likely that C is dead than that two links leading to it are broken at the same time. Therefore, C is likely to be dead.

When the only communication that is successful is on a link from A to D, it is likely that B's machine is dead, because all attempts to communicate with two different processes (B,C) on it from two different machines, that can communicate with each other have failed.

Finally, in an extreme case when no communication is possible, it is likely that there is a total network failure, and the arbitration should be tried later, preferably choosing D on a different machine.

3.5 Limitations and system requirements

It is very important that the network used to run Somersault has a sufficient bandwidth. Otherwise, under high loads the messages will be delayed for long times and result in many time-outs during the arbitration process that will produce the wrong detection results.

Another important issue is fail silence. If processes and machines are not fail silent, there is no guarantee that failure detection will work at all, because any single failure may flood the network without leaving any chance to do the failure detection with the short time-outs required to meet the constraints in Section 3.2.

3.6 Special Cases

In this section we will consider a number of special circumstances that require a separate treatment. We will show how the algorithm described above applies to each particular situation.

3.6.1 Network Overload and Adaptive Time-outs

In this chapter we have assumed that the network has a sufficient bandwidth and therefore the absolute majority of the messages are not delayed for more than t_{del}. However, there is

no way to guarantee this. For instance if a machine crashes and several large processes are attempting to transfer state to the newly created secondaries, the amount of network traffic will be high. The network delays are likely to become quite large. The failure detector than has to either declare very process dead, oı adjust the time-outs. Increasing the time-outs may lead to violation of a 250ms performance constraint, but will keep the system intact.

From the system point of view increasing the time-outs may have some benefit over trying to rigidly meet a 250ms constraint. If the timeous are not relaxed, more processes will be declared dead, which will lead to an increased recovery traffic. More traffic will lead to more time-outs, more false positives and even more traffic from the processes trying to recover. Finally, the system will lose so much volatile state, that it will not be able to recover at all.

Therefore, we can state two requirements for reducing the false positive rate during the network overload.

- Use adaptive timeous for failure detection, increasing the time-outs with an increased network load
- Avoid the network overload. Apart from buying a faster network use a software mechanism which will reduce the number of messages sent on the network in case the traffic becomes too heavy.

3.6.2 A Two Machine System

In case Somersault is running on two machines, the failure detector described in Section 3.4 will work, except it will not be able to detect the difference between the network failure and the machine failure. To obtain the most reliable results the first arbiter should be placed on B's machine (see Figure 3.2), and if the use of the second arbiter is necessary, it should placed on A's machine.

3.6.3 A Single Machine System

In case of a machine failure Somersault may end up running on a single machine. This configuration can not tolerate machine failures anymore, however, failure detection is still necessary to detect process and link failures. Because machine and network failures are not a concern anymore, failure detection process is reduced to its basic principle presented in the Figure 3.1. A single arbiter process is consulted and its decision is final.

In case there are only two processes on a single machine, the failure detection algorithms presented above cannot be used anymore, because of the lack of potential arbiters. Instead, simple time-outs can be used. Alternatively, information about the process status can be obtained from the kernel. However, kernel may not have the up-to-date information, and asking the kernel violates the end-to-end argument.

3.7 Analysis

The algorithm presented above has a constant cost of seven short messages. It can help diagnose machine failures, so that the whole machine can be declared dead before every other process in the system is trying to communicate with the processes on that machine. Therefore, the algorithm allows to reduce the number of failure detection messages per machine failure dramatically from $O(n^2/r)$ to $O(1)$ (as usual n is the number of processes, r is the number of machines).

The reliability of the algorithm is heavily dependant on having a sufficient network bandwidth to guarantee a bounded communication latency and on having fail-silent machines and processes.

If used as a failure detector for a more complex system like Isis sight view manager the above algorithm avoids the danger of partitioning due to a single link failure (a link between primary and secondary view managers), since this link failure will be detected

and the situation in which the primary and the secondary view manages suspect each other will be avoided.

Chapter 4

Background Failure Detection

The Background Failure Detector (Background FD) is a preemptive failure detector. Its purpose is to find failures of links and nodes that are not actively involved in system operation, but may be needed in the future; for example, during process recovery. By finding these hidden failures Background FD insures that multiple failures do not accumulate in the system over long periods of operation, so the system is less vulnerable.

Background FD is implemented as a distributed asynchronous algorithm. It operates by generating messages independent from application traffic, and analyzing the failure patterns that are observed on these messages. Background FD algorithm operates on two classes of processes: leader and normal. There is one leader processes; the rest of the processes in the system are normal. All processes generate Background FD traffic messages; if these messages are not acknowledged, failure suspicions are sent to the leader. The leader analyzes failure suspicions by comparing their relative timings. It filters out transient faults and determines the presence and the nature of permanent failures.

Having a centralized algorithm in which all participants generate traffic is suboptimal from the performance point of view. However, in Chapter 5 we show that without comprehensive centralized Background FD, Somersault will not be able to reach the required availability levels. Additionally, we show that Background FD remains efficient even for systems with hundreds of nodes in them.

The rest of this chapter is structured as follows. It begins by presenting the assumptions under which Background FD operates. Then it examines the architecture of Background FD, considers how it filters out transient failures, and the mechanisms used to

distinguish between link and node failures. This chapter concludes by describing the algorithm used by the Background FD leader, and analyzing its performance.

4.1 Assumptions

Background FD solves the same problem as the Foreground FD: finding of and distinguishing between process and link failures. However, where Foreground FD sacrifices accuracy for real time performance, Background FD makes a different trade-off. Background FD does not have to detect failures in real time; in fact, detection on the order of tens of seconds or even minutes is sufficient. Background FD must have an extremely low false positive rate, since finding non-existent failures will lead to killing off useful processes, and will easily override all the benefits of preemptive failure detection. Thus, Background FD trades timeliness for accuracy.

In particular, it is important that transient failures that arise during the normal system operation are not mistaken for permanent ones. In order to filter out the transient failures and to find all the permanent ones, Background FD requires multiple suspicions before declaring a failure.

4.2 Background Failure Detection Architecture

Background FD algorithm tests every link and every process in the system in order to detect failures. The difficulty arises making these tests accurate while minimizing the communication overhead of these tests. In this section we describe how the tests are generated, and how results of these tests are processed.

Generating Test Messages

In order to make Background Failure Detection work, there has to be a method for generating the test messages. It has to test every link in the system on a regular basis. It

also has to avoid a situation in when a single process gets periodically overloaded by such testing. For example, a situation in which there is a stable pattern where every process first sends a test message to process A, then sends a test message to process B, etc. is undesirable. Instead the test-message load should be uniformly distributed throughout the system. Also, generating this load should require only a minimal amount of coordination, so that it would stay uniform regardless of the failures present in the system.

We achieve the goal of balancing the Background FD test traffic by using a randomized round-robin approach. We introduce a notion of a *self-checking cycle* or *period.* The self-checking cycle is the maximum interval between two consecutive times any link is checked.

Every process keeps a priority queue of timers ordered by expiration time, where each timer corresponds to a particular neighbor of that process. Initially all timers are assigned random values between the current time and the current time plus the self-checking period. Whenever a timer expires:

1. Test message is sent to the neighbor process corresponding to that timer;
2. The timer is reset to its current value plus a random number in the range form half to a full self-checking period
3. The timer is reinserted into the appropriate slot in the priority queue
4. This process is repeated every time a timer expires.

The method presented above tests every link in the system with a period varying from half to a full self-checking cycle. The interval between the two successive tests of a link is randomized within this period, and tests of different links are independent of each other. Therefore, every process generally receives a balanced load of test messages from its links. It is highly unlikely that any process will get flooded by the test messages arriving at

its links at the same time. It is even less likely that any process will get flooded periodically.

Distinguishing Between Process and Link Failures

Now that we know how the test messages are generated, let us consider how to use them in order to find failures in the system. We begin by considering scenarios where link or node failures are present in the system. For those scenarios we analyze failure suspicions generated due to time-outs of Background FD test messages. Then we determine how to use these suspicions in order to accurately diagnose link and node failures.

When a link is faulty, its two end-processes find that test messages they send to each other are not being acknowledged. Therefore both processes will send their suspicions about the other process to the background FD leader (see Figure 4.1.a).

When a process is faulty, all other processes in the system will find that their test messages sent to that process are not acknowledged. Consequently, all other processes will send their suspicions about that process to the leader process (see Figure 4.1.b).

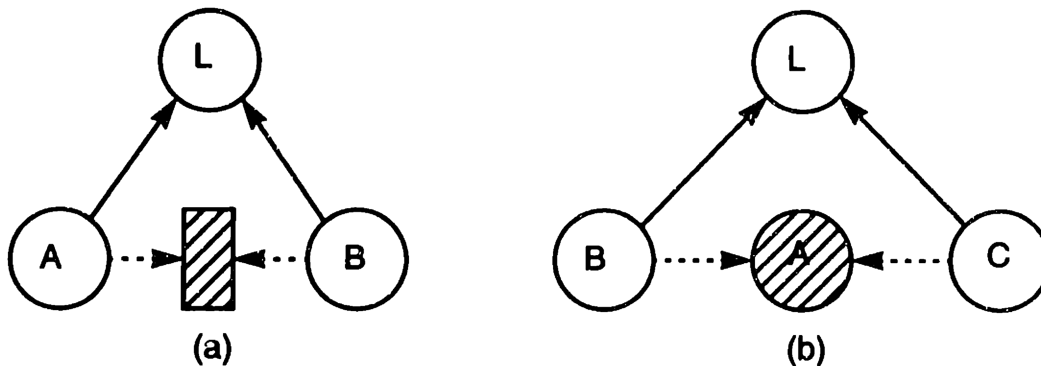

Figure 4.1. Background Failure Detection Scenarios. Dotted lines are pings, solid lines are suspect messages. L is a leader process; A, B and C are the normal processes. (a) Link failure detection. (b) Process failure detection.

Therefore, we employ the following method for distinguishing between the link and the process failures:

- If two processes repeatedly suspect each other, then the link between those two processes is broken.
- If multiple processes suspect the same process, then the suspected process is dead.

Discarding Transient Failures

In the previous section we outlined a method for detecting of and distinguishing between process and link failures. This method, however, is sensitive to transient failures. For instance, consider a link between two processes that is fully functional, but occasionally delays messages. Over a long period of time this link will delay enough background test messages, and thus generate enough suspicions, to convince the leader process that it is broken.

The main difference between transient and permanent failures from the Background FD point of view is that suspicions about transient failures are infrequent, while permanent failures are reported during every self-checking cycle.

In order to eliminate the influence of transient performance failures, Background FD leader considers only the failure suspicions that are present in the sliding window (Figure 4.2), and to declare a failure it requires multiple suspicions.

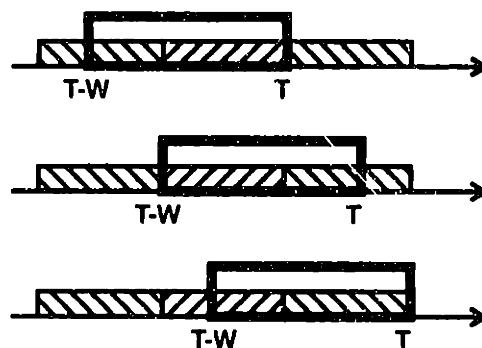

Figure 4.2. A sliding window filtering of the relevant events. Only the failure detection messages arrived during the most recent self-checking cycle are considered. The size of the sliding window W should be large enough to cover one system checking period and to allow for the decision making and network delays.

If there really is a failure, the leader will receive a number of failure indications within one window, and will declare a component dead. If there is only a performance glitch, it will receive only one suspicion, which will not be sufficient to declare a component dead. A notable exception is the case of a "persistent" performance glitch caused by an overload of a machine where the suspected process is running. However, under the telecom requirement of components being loaded at no more than 40% of their capacity during normal operation [20], we assume that machine overload is not going to happen often enough to justify changing this algorithm.

To make this approach work we need to make a sliding window as small as possible, so that the leader gets the smallest number of transient failures per window. However, we need to make the window large enough, so that the leader collects enough suspicions to reach an accurate conclusion. Let us consider the minimal number of suspicions a leader needs to observe in order to distinguish between the node and the link failures.

In order to accurately diagnose a link failure, leader needs to observe at least two messages from the processes on the ends of the link. The interval between these messages is not going to exceed one self-checking cycle. To diagnose a process failure, the leader needs at least two suspicions about the same process coming from different processes. These suspicions will also be sent no more than a self-checking cycle apart.

Assuming that performance glitches are infrequent and the transmission times are small compared to the self-checking cycle, a sliding window spanning two self-checking cycles should be sufficient for filtering out performance glitches and detecting the real failures.

4.3 The Leader Algorithm

Here we describe the algorithm that implements the heuristics described above. We look at

inputs, outputs and the side-effects of the algorithm and analyze its behavior. The algorithm is run every time the Background FD leader receives a suspicion. A sliding window is used to allow the leader to filter out transient failures that were caused by variation in communication latency. Whenever a leader gets a report saying that process A suspects process B, and a report that process B suspects process A within the same window, the leader decides that the link AB is broken. Alternatively, if two processes report that they suspect a process A during the same window, the leader decides that the process A is dead.

Data Structures

The Background FD leader keeps a sliding window containing the suspicions that are valid during the current windowing period. Each suspicion contains two elements: a suspected node and a suspected link. The content of the sliding window is updated with the flow of time. The new suspicions are added in and the out-of-date ones are deleted.

The leader also keeps the list of previously detected failures. This list allows the leader to discard suspicions about the dead processes and links without altering its internal state and without effecting its conclusions about the status of other components.

Inputs

- A sliding window (*sliding-window)* containing the previously reported suspicions. The algorithm uses one sliding-window to keep suspicions about all components.
- A process that sent a suspicion message (*suspecter*);
- A process indicated in the suspicion message (*suspected*);
- A list of previously discovered process and link failures (*already-detected*).

Output

- Failure status: *Link-Failure*, *Process-Failure* or *OK*.

Side-effects

- *sliding-window* is updated with respect to current time, suspicions received and failures detected;

- *already-detected* is updated to reflect any new failures discovered.

Actions

The following steps are taken every time a suspicion message is received by the leader process:

1. If the *suspecter* node, the *suspected* node, or the link connecting the two is found in the *already-detected* list, then there is no new failure in the system. Therefore return *OK*.

2. Otherwise, advance the *sliding-window* in accordance with the *current-time*.

3. Put suspicion {node(*suspectee*), link(*suspector, suspected*)} into the *sliding-window*.

4. If the number of (*suspecter, suspected*) links (the order in which the endpoints are listed does not matter here), is equal to 2, there is a link failure;
   - Insert link(*suspector, suspected*) into the *already-detected* list;
   - Delete all the suspicions that contain the link(suspector, suspected) from the sliding-window;
   - return *Link-Failure*.

5. Else if the count of nodes named *suspected* in the *sliding-window* is equal to 2, there is a process failure;
   - Insert node *suspected* into the already-detected list;
   - Delete all the suspicions that contain the link(*suspector, suspected*) from the *sliding-window*;
   - return *Process-Failure*.

6. Else return *OK*.

4.4 Analysis

Under the assumptions that we made in this chapter: namely, that Background Failure Detection can take a long time in order to increase accuracy of detections, and that the network transmission times are bounded and in fact are small compared to the self-checking cycle, we can make the following conclusion. The algorithm presented here has the following useful properties:

- Once a component has been declared dead, it is not considered in any more failure calculations.

- Because links are checked once a system self-checking cycle, and processes are checked more frequently, every process failure will be reported several times before

the same link will be suspected twice. Therefore, if a link and a process were suspected and, a cycle later, no one else suspected the process, but the same link is suspected again, the process is OK, but the link is broken.

- By the same argument if two links leading to the same process are suspected, then the links are OK, but the process is dead.
- If two processes on the ends of the same link suspect each other, the link will be declared dead, and the decision as to which end process to kill will be left to the view manager.

Chapter 5

Effects of Failure Detection on System Scalability

5.1 Overview

Scalability of distributed systems is a key factor in determining their success. Usually scalability implies the ability of a system to maintain a certain level of performance as the number or the size of its components increase. However, for fault tolerant systems there is another important aspect of scalability. It is concerned with the changes in availability of the system as the number of its components changes.

In this chapter we consider the scalability of Somersault with respect to fault tolerance. We conclude that reliable communication is key to scalability of Somersault. In the absence of perfect communication, Somersault needs a failure detector that distinguishes between the process and the link failures, and is guaranteed to find them in a bounded amount of time. Given a high reliability of the network and a high speed of failure detection, Somersault can scale up to tens, possibly hundreds of processes, and is expected to meet the telecom requirement of 99.9994% availability. In addition, fault tolerance of Somersault can be increased by putting a smaller number of processes on each machine, organizing the machines in a logical ring and using a faster network to connect them.

This chapter is structured in the following way. First we discuss the architectural aspects of Somersault that affect its reliability. Then we propose a probability model that allows to quantize the reliability of the system. Using this model we analyze the possible causes of a total Somersault system failure. Having seen the causes of Somersault's total failure, we suggest a machine configuration and recovery unit layout that minimizes the probability of total system failure and allows the system to withstand multiple machine

failures without loosing much reliability. Once we establish the optimal system layout we analyze the acceptable number of processes and speed of failure detection required for a typical Somersault system with availability required by telecoms. We conclude by reiterating the requirements for the increased availability of Somersault.

5.2 Architecture of Somersault

In this section we discuss the aspects of Somersault's architecture that are relevant to understanding its scalability. The design of Somersault is based on Manetho, a distributed fault tolerant system that uses a sender-based message logging algorithm [13]. We first introduce the fundamental architecture of Manetho, and then discuss modifications made to it in Somersault.

Manetho processes log the messages they *send* onto stable storage. This approach is different form the one taken in traditional "log and replay" systems (see [2], [5], [22] and [28]) that log the messages they *receive*. Another important feature of Manetho is that when recovering after crash, processes roll forward from their latest checkpoint, as opposed to rolling back the rest of the processes in other systems.

The central idea of Manetho is very simple: if the system is entirely message based, and the state of processes is entirely determined by the messages they receive, then it is possible to reconstruct the state of a crashed process by replaying to it the messages it received. Using this idea, process recovery is done by rolling a process forward from its latest checkpoint. This roll-forward is driven by the recovering process asking the rest of the processes in the system to replay the messages they sent to it during the original execution [13].

In order to allow asynchronous logging to stable storage, processes append information about their uncheckpointed state onto the outgoing messages. When recovering, a

process retrieves its uncheckpointed state along with the messages that was sent to it from the rest of the processes in the system (see Figure 5. 1).

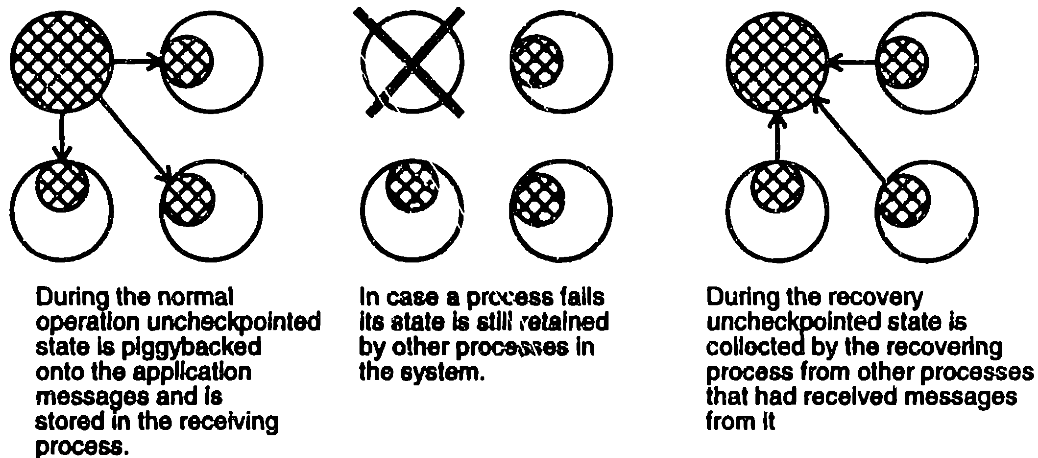

Figure 5. 1. One of the key ideas of the Manetho algorithm is to allow asynchronous logging by distributing the uncheckpointed state across the processes in the system, and by piggybacking the volatile state information onto the application messages.

During failure free operation, Manetho's overhead is low, checkpoints are inexpensive and have the benefits of optimistic logging. Recovery, however, is complicated and involves a large amount of communication. Normally, a crashed process would have to communicate with every other process in the system, thus making the amount of communication required linear to the number of processes. In the worst case if all the processes crash and are trying to recover, every process has to talk to every other process, thus the amount of communication is quadratic to the number of processes. Therefore, Manetho is not likely to scale well. However, catastrophic failures that occur when all the processes crash at the same time should be rare in a well designed system.

A serious problem with Manetho is its handling of non-determinism. Manetho is a roll-forward system, so all non-deterministic choices like *time,* or order of message receipt have to be recreated during process recovery exactly the same way as they happened during the initial execution. Therefore, all non-deterministic choices have to be logged. This

limitation introduces additional complexity into the system and requires an increased amount of information to be passed around in order to insure the consistent state of the system after crashes.

5.2.1 Active Replication

Somersault replaces the stable storage, used in Manetho, with a secondary process [18]. A primary process together with its secondary forms a Recovery Unit (RU). Somersault can recover from any combination of process failures, granted that at least one process from every RU survives.

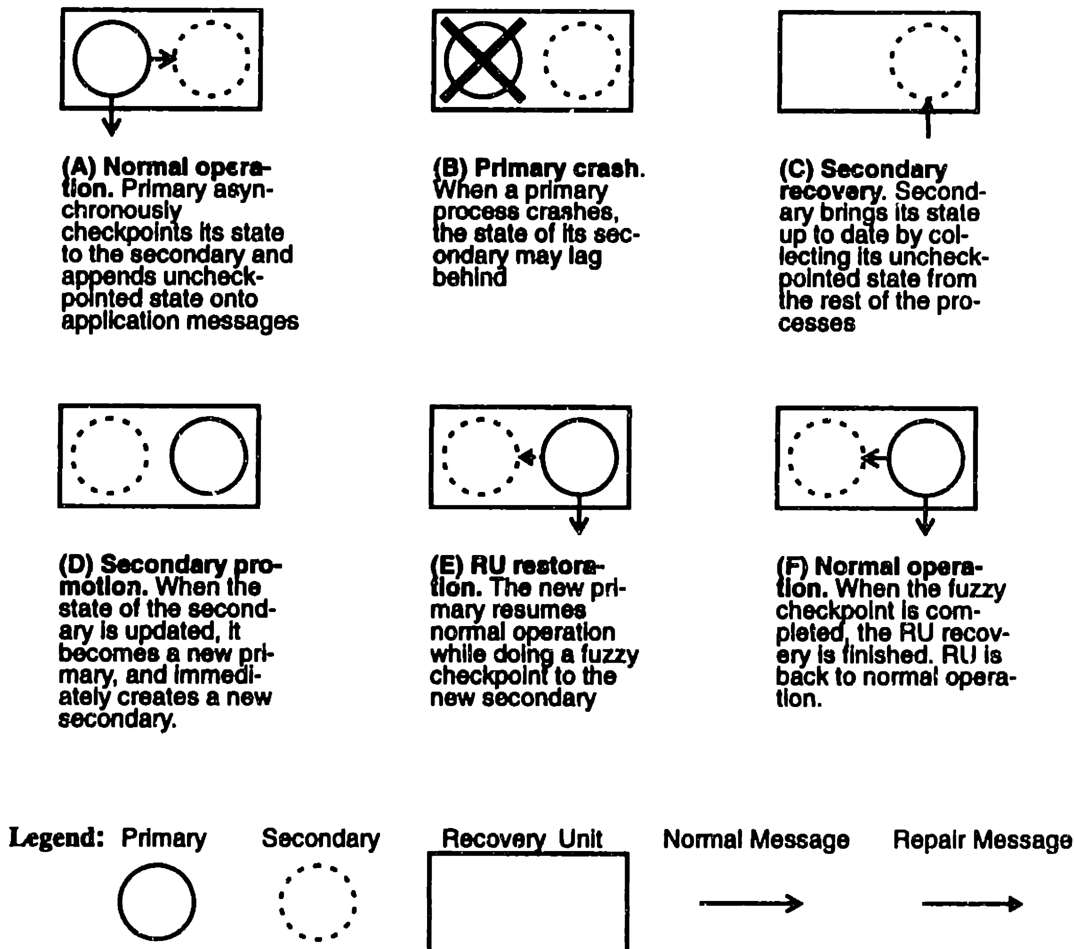

Figure 5. 2. fail-over of a Somersault Recovery Unit (RU). Each recovery unit consists of a primary and a secondary process (A). When a primary crashes (B), the secondary process updates its state (C), becomes a new primary and creates a new secondary process. Finally, the new primary transfers its state to the new secondary (E), and the RU is back to its normal state (F).

In failure-free mode primary process asynchronously checkpoints its state to the secondary, just as it would to a disk [24]. In case a primary fails, its secondary process collects the state that the primary piggybacked onto application messages since the last checkpoint. When the secondary reconstructs the state of the primary before the crash, it becomes a primary, and a new secondary is created (see Figure 5. 2). This algorithm benefits from switchover recovery, where instead of reconstructing all the state of a failed process from the disk, execution is instantaneously switched to a secondary process. However, until the new primary collects its state information from the rest of the system and completes the state transfer to the new secondary, the system remains vulnerable to a single failure of the new primary.

5.3 Assumptions and the Model

In this chapter, we concentrate on considering MTTF instead of availability of the system, because it is easier to work with and the latter can be easily derived from it.

$$Availability = \frac{MTTF_{sys}}{MTTF_{sys} + MTTR_{sys}} \tag{5.1}$$

Assuming no need for hardware or network replacement (see Section 5.3.10), mean time to repair for the whole Somersault system ($MTTR_{sys}$) is likely to be on the order of an hour. This includes time for rebooting the machines and rebuilding the state of applications from stable storage. Then in order to meet the goal of 99.9994% availability, $MTTF_{sys}$ should be at least 20 years long. We will use this number to benchmark our further calculations.

5.3.1 The Probability Model

Mean time to failure offers an accurate estimate of the availability of the system. It will be the target of our calculations. However, computing the $MTTF_{sys}$ of the system

from the $MTTF_{comp}$ of its components is cumbersome. Instead, it is easier to use the inverse of $MTTF_{comp}$ — probability P_{comp} of a component failing during a unit time interval.

$$MTTF = \frac{1}{P} \tag{5.2}$$

This will only work if the unit time used to define the $MTTF$ is small enough, otherwise we may end up with failure probabilities greater than one. Moreover, if we assume that the $MTTF_{comp}$ are large enough, then the P_{comp} is very small. Then, for a system composed of n components, the binomial probability of at least one component failing $1-(1-P_{comp})^n$ can be approximated as $n \times P_{comp}$. Thus, from the above approximation and from Equation 5.2 we can calculate the P_{sys} of n components as

$$P_{sys}(n) = n \times P_{comp} = n \times \frac{1}{MTTF_{comp}} \tag{5.3}$$

By Equation 5.2 we can now calculate the $MTTF_{sys}$:

$$MTTF_{sys}(n) = \frac{1}{P_{sys}(n)} = \frac{1}{n \times \frac{1}{MTTF_{comp}}} = \frac{MTTF_{comp}}{n} \tag{5.4}$$

Similarly, under the same assumptions, for a system with two components **c1** and **c2** with an independent failure mode, the probability of failure of at least one component can be approximated as

$$P_{sys}(c1, c2) = P_{c1} + P_{c2} = \frac{1}{MTTF_{c1}} + \frac{1}{MTTF_{c2}} = \frac{MTTF_{c1} + MTTF_{c2}}{MTTF_{c1} \times MTTF_{c2}} \tag{5.5}$$

Thus, the $MTTF_{sys}(c1, c2)$ can be easily found according to the Equation 5.2:

$$MTTF_{sys}(c1, c2) = \frac{1}{P_{sys}(c1, c2)} = \frac{MTTF_{c1} \times MTTF_{c2}}{MTTF_{c1} + MTTF_{c2}} \tag{5.6}$$

We will make an extensive use of the probability model presented above in the derivations and analysis of $MTTF_{sys}$ for Somersault.

5.3.2 General Assumptions

This section presents the basic notions used in this chapter for analyzing the components of distributed fault tolerant systems. First, we consider the types of the components in the system, then their failure modes and finally their life-cycles.

It is important to remember that no single fault can crash the Somersault system. In fact, most of the double failures will not crash the system either. That is why we have to carefully consider the failure modes of each type of a component to determine the combinations and probabilities of failures that will bring the whole system down

5.3.3 Components

For the purpose of our analysis we will assume that Somersault operates in a non-malicious environment and we do not have to worry about the physical network being destroyed or damaged. Therefore, the failure of the system could be caused by a combination of failures of the following component types:

- Process
- Process-to-process Communication Link
- Machine

5.3.4 Failure Modes

In this chapter we do not consider Byzantine failures of components and assume that all components are fail-silent. For now we also assume that the failure detector is perfect and will not produce any false positive results.

A **process death** can be caused by a process executing an illegal instruction, a process being killed by an external source, or by that process going into an infinite loop. A dead process manifests itself by not responding to or attempting to communicate with an external world.

A **link death** can be caused either by either communication software or hardware failure. It manifests itself by dropping or delaying messages for an unacceptably long time.

A **machine death** can be caused either by a hardware or a software failure, and leads to the termination of all the processes running on that machine. It is manifested by a simultaneous death of all the processes running on that machine.

5.3.5 Life Cycles

Every component we consider here goes through the same cycle of fault free operation, failure, detection and recovery (see Figure 5. 3).

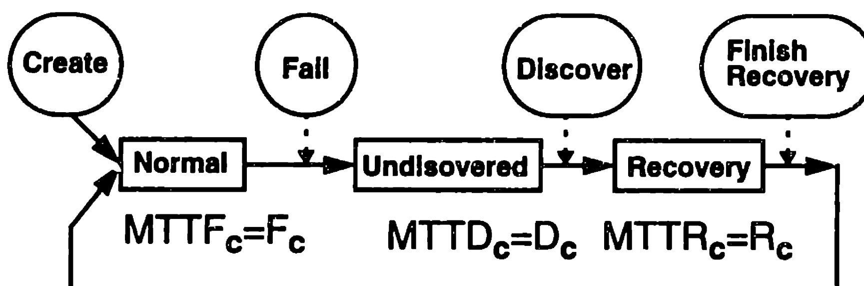

Figure 5. 3. A lifecycle of a Somersault component. The main events in the life of a component are the creation, when the component first becomes operational; Failure, when the component stops working, but the system does not know about it yet; failure Discovery, when the component recovery is initiated; and finally the completion of recovery when the component is operational again. A Somersault component repeatedly goes through a cycle of three states — Normal operation, Undiscovered failure, and of failure Recovery.

The time a component **C** spends on average in the normal state is denoted as a mean time to failure $MTTF_C$, or simply F_C. Similarly, the average time spent in the Undiscovered failure state is denoted as a mean time to discover, $MTTD_C$, or simply D_C; and the average time spent in the recovery state is denoted as $MTTR_C$ or simply R_C. The subscripts used to identify the types of

components are **l**, **p** and **m** for a link, process, and machine respectively. So F_p is a process mean time to failure, and D_m is the mean time it takes to discover a machine failure.

In Somersault the first two stages of the component lifecycle may differ in their duration for different components, but they are similar otherwise. A component works correctly, and after a while it breaks. For a while this failure goes undetected, but then it is discovered. Once the failure is discovered, the recovery is started. This is where the difference between various components shows. The following paragraphs describe how various components can be recovered. For all component types we will assume that $MTTF \gg MTTD$ and $MTTF \gg MTTR$.

Recovery for a process is conducted according to the Somersault algorithm. If a process that has failed was a secondary, then a new secondary is created and a primary transfers its state to the new secondary. If a process that has failed is a primary, then the remaining secondary is promoted to a primary, it collects uncheckpointed state, sent by the crashed primary, from the rest of the system; then a new secondary is created, and the state of the new primary is transferred to it. The second recovery scenario will take a longer time than the first one. In our derivations, however, we will use an overall average value for recovering both the primary and the secondary processes.

Recovery for a link consists of the simplest, but not the most efficient action. When a link failure is discovered, Somersault decides which end process to kill. The decision process uses a simple algorithm which helps to avoid an accidental killing of both members of a recovery unit. Later, Somersault takes care of recovering that process and rebuilding all the links to it, including the one which was initially broken.

Recovery for a machine is equivalent to the recovery of Somersault processes that were running on that machine prior to its failure. The physical machine that crashed does not have to be introduced back into the system right away. Instead, the processes that ran

```
Recover_Link (Process End1, End2) {
    if (alive (RU_Mate (End1))) {
        Kill_Process (End1);
        Recover_Process (End1);
    } else {
        Kill_Process (End2);
        Recover_Process (End2);
    }
}
```

Figure 5. 4. A recovery unit with one member process can still recover, but if both member processes are killed, then the whole system will crash. Therefore, before killing a process Somersault checks if the other process in its recovery unit is alive.

on it may be restarted elsewhere in the system. Meanwhile, the faulty machine will be rebooted, or perhaps replaced; when it joins the system some of the existing processes will be migrated onto it.

5.3.6 Fault Scenarios

Being a fault tolerant system, Somersault can withstand any single failure, and in fact a variety of multiple failures as well. Here we consider what combinations of failures of components discussed above do cause failures of the whole system. Of course, there is an infinite number of possible combinations. Our goal, however, is to estimate the MTTF of the system, thus we will limit the scope of failures considered here to failures of the second order, where two components are broken at the same time. Again, we assume independence of failures.

We have three component types: a link, a process, and a machine. Thus, we will have nine second order scenarios to examine, taking into account the algorithms and the recovery procedures described above. The common scheme for all scenarios will be as follows. As an example consider what happens when a process **P** breaks and, while its failure is discovered and repaired, the system is left vulnerable to other failures which may kill the other process **Q** in **P**'s RU. The system will fail if **Q** is killed before **P** has restored its state

to the pre-crash state completely. Possible failure scenarios caused by two component failures are classified in the Table 5.1.

first fault	state	following link failure	following process failure	following machine failure
Link	Undiscovered	When the 1st link failure is discovered its recovery algorithm will not kill an unreplicated process.	When the link failure is discovered, it's recovery algorithm will not kill the mate of the dead process.	A machine that hosts both mates of the processes on the ends of the broken link.
	Recovering	The 2nd link's recovery algorithm will not kill the mate of the process killed during the recovery of the 1st link.	The end process that was not killed after the failure of the link was detected.	A machine that hosts the end process that was not killed after link failure was detected.
Process	Undiscovered	Link's recovery algorithm will attempt to communicate with the process and discover that it's dead	Processes' mate.	A machine that hosts the processes mate
	Recovering	Link's recovery algorithm will not kill the mate of a dead process.	(Same as above)	(Same as above)
Machine	Undiscovered	A link between two processes whose mates resided on the crashed machine.	Process mated with one of the processes on a crashed machine	A machine that has processes mated with processes on a crashed machine
	Recovering	(Same as above)	(Same as above)	(Same as above)

Table 5.1: Summary of the second order system failure causes. Each cell lists the system elements that can cause the critical second failure (grey fill), or explains why there are no critical failures (white).

When making an estimate of the failure probability we will use the following metrics:

- c_x: number of components of type x that may break in the system
- v_x: vulnerability period for that component, (i.e. proportion of time during which a second failure may crash the system)
- p_x: probability that one of the components of type y that may crash the system during the vulnerability period caused by the failure of x will actually break during that period

The overall probability of system failure caused by the consecutive failures of components of types x and y, assuming independence of failures:

$$P_{xy} = c_x \times v_x \times p_x \qquad (5.7)$$

According to the assumptions about the small probabilities of failure of each component (see Section 5.3.1), the probability P_{xy} is proportional to the number of components of type x in the system. For our calculations, let us use

- n: the number of processes in the system
- $n \times (n-1)/2$ links between its full interconnected processes
- r: the number of machines
- n/r processes running on each machine

The goal of this analysis is to get a realistic lower bound on the reliability of Somersault. Therefore, we will make conservative estimates of failure probabilities, trying to be as accurate as possible.

5.3.7 Link induced failures

Let us start by considering the system failure scenarios in which a link is a first component to break. After a failure, a link is recovered by killing one of its end processes and letting Somersault to do all the work on rebuilding both the processes and the links. There is a choice of which end-process to kill, and this choice is always made to avoid killing a single-process RU. Before the link failure is discovered, the choice of a process to kill has not been made yet. Thus, if one of the processes in the RUs at the ends of the link is killed while the link is in the **undiscovered** state, the Recover_Link algorithm will later choose to kill the end process in the RU which was not affected by the second failure (see Figure 5. 5).

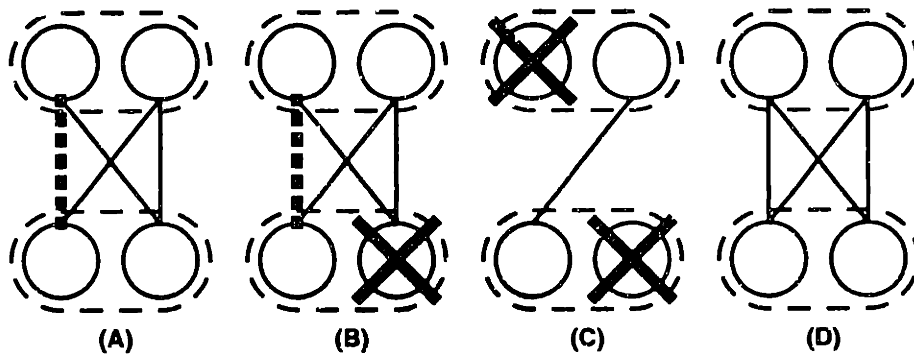

Figure 5. 5. While a link failure is undiscovered the system is not vulnerable to an additional process or link failure. First, a link failure occurs (A), but is not discovered yet. Then another failure leading to a death of one of the processes in the end-RUs happens (B), and is discovered. Later, a broken link is discovered (C). According to the Recover_Link algorithm, the link recovery is conducted via killing one of the end processes. However, if Somersault tries to kill the end-process in the lower RU, it will discover that the process is not duplicated, and thus will kill the process in the upper RU (C again). Then the recovery proceeds to restore both RUs and the links between them to the original state (D).

A system with undiscovered link failures is, however, vulnerable to machine failures (see Figure 5. 6). This happens because multiple processes die when a machine crashes. Thus, several processes can become unreplicated at once. They all need to communicate with each other during recovery, and if any of the links between them are damaged, the whole system will crash.

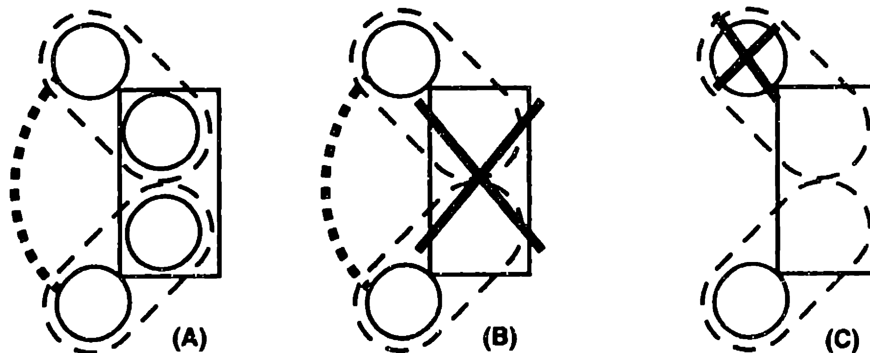

Figure 5. 6. While there is an undiscovered link failure in the system (A), a machine failure may lead to making the processes on both ends unreplicated (B), and when the link failure is discovered a complete RU is destroyed, thus the system can not recover(C).

During the link **Recovery** state, another link failure cannot crash the system, because the Recover_Link will notice that one of its end RUs is being repaired, and will choose to kill the process in another end-RU.

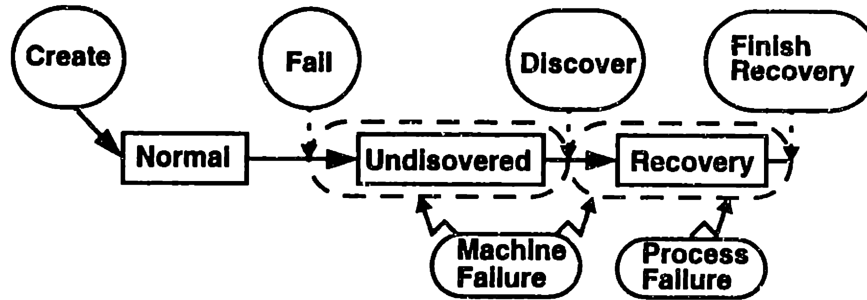

Figure 5. 7. The vulnerability period for Link-induced failures depends on both the link detection and recovery times. Some machine or process failures during the recovery time may cause the system to crash. However, due to the Recover_Link algorithm presented in Figure 5. 4 an additional link or process failure can not crash the system. A separate contribution comes from the vulnerability to machine failures while the link failure is undiscovered.

Now, let us examine the probability of the total system failure due to a link-induced failures. We will separately estimate the contribution of undetected link failures (all variables that relate to undiscovered failures are susperscripted with *undet*), and the contribution of recovering links (variables superscripted with *rec*). Calculations presented below assume that all failures in the system are independent.

First let us consider the contribution to the probability of system failures by recovering links. There are approximately $n^2/2$ links in the system, thus

$$c_l^{rec} = \frac{n^2}{2} \quad (5.8)$$

As we have discussed above, the system is vulnerable to both machine and process failures during the recovery of a broken link. Thus, the proportion of the time during which such second failure is dangerous is

$$v_l^{rec} = \frac{R_l}{F_l + D_l + R_l} \approx \frac{R_l}{F_l} \quad (5.9)$$

Finally, the system will crash, if the one remaining end-process will be killed. As we have mentioned above, this will happen either if that process fails, or if the machine on which it is running crashes. The probability of a process failure is $1/F_p$, and the probability of a machine failure is $1/F_m$. Thus, the probability of the second end-process dying is

$$p_l^{rec} = \frac{1}{F_p} + \frac{1}{F_m} \tag{5.10}$$

Therefore, the overall probability of a system crash during link recovery is according to Equation 5.7:

$$P_{lx}^{rec} = c_l^{rec} \cdot v_l^{rec} \cdot p_l^{rec} = \frac{n^2}{2} \cdot \frac{R_l}{F_l} \cdot \left(\frac{1}{F_p} + \frac{1}{F_m}\right) \tag{5.11}$$

Now let us consider the system failures that may happen while the link failures are undiscovered. Every machine has n/r processes on it. Therefore, in case a machine crashes, there will be n/r unreplicated processes in the system. They have $n^2/(2r^2)$ links between them. Because there are r machines in the system, the total number of potentially undetected failed links is

$$c_l^{undet} = \frac{n^2}{2r} \tag{5.12}$$

Every one of these links is vulnerable while it's failure is undetected, thus

$$v_l^{undet} = \frac{D_l}{F_l + D_l + R_l} \approx \frac{D_l}{F_l} \tag{5.13}$$

Finally, the probability of a second critical machine failure occurring is

$$p_l^{undet} = \frac{1}{F_m} \tag{5.14}$$

Therefore, the probability of the system failure while one of the link failures in undetected is

$$P_{lx}^{undet} = c_l^{undet} \cdot v_l^{undet} \cdot p_l^{undet} = \frac{n^2}{2r} \cdot \frac{D_l}{F_l} \cdot \frac{1}{F_m} \tag{5.15}$$

Now we can find the overall probability of the system failure due to link failures, as the sum of probabilities of failures during the undiscovered and repair periods by combining the two together.

$$P_{lx} = P_{lx}^{undet} + P_{lx}^{rec} = \frac{n^2}{2r} \cdot \frac{D_l}{F_l} \cdot \frac{1}{F_m} + \frac{n^2}{2} \cdot \frac{R_l}{F_l} \cdot \left(\frac{1}{F_p} + \frac{1}{F_m}\right) \quad (5.16)$$

5.3.8 Process induced failures

The main difference between the process-induced failure mode and the link-induced failure mode considered above is that the vulnerability period for all failure modes covers the **Undiscovered** state as well. This is because while the link is undiscovered, Somersault can still change its mind about which end to kill. With a process failure, however, there is no way of shifting the location of the fault in order to avoid simultaneous death of both processes within a RU. Link failures that follow the process failures still do not crash the system, again due to the Recover_Link algorithm of Figure 5. 4.

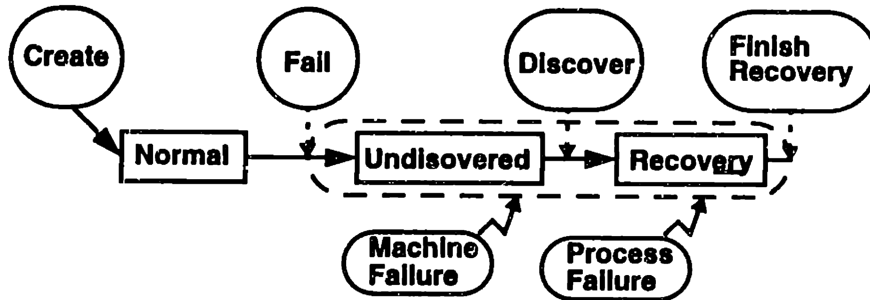

Figure 5. 8. The vulnerability period for Process-induced failures is proportional to the sum of process discovery and recovery times. Some machine or process failures during this vulnerability period may cause the system to crash. However, due to the Recover_Link algorithm presented in Figure 5. 4 an additional link failure can not crash the system.

There are n processes in the system, therefore

$$c_p = n \quad (5.17)$$

The vulnerability period for a process-induced failure is

$$v_p = \frac{D_p + R_p}{F_p + D_p + R_p} \approx \frac{D_p + R_p}{F_p} \quad (5.18)$$

Finally, the probability of the second critical failure occurring in the system is the same as for the link-induced failures

$$p_p = \frac{1}{F_p} + \frac{1}{F_m} \tag{5.19}$$

Therefore the probability of a process-induced failure is

$$P_{px} = c_p \times v_p \times p_p = n \cdot \frac{D_p + R_p}{F_p} \cdot \left(\frac{1}{F_p} + \frac{1}{F_m}\right) \tag{5.20}$$

5.3.9 Machine induced failures

The difference between the machine-induced failures and the failure modes considered above is that the machine-induced failures are common-mode failures. Several processes are killed at once, thus making a system vulnerable to a large number of additional faults that may crash it. For instance, in case of link and process induced failure modes, the second failure that crashes the system could never be a link failure. However, in the case of machine-induced failures, certain links are critical to survival of the system. These are the links are between processes in RUs that lost one of their members during the machine crash. These processes need to communicate with each other to bring their state up to date. However, if the links are broken they are unable to do so (see Figure 5. 9).

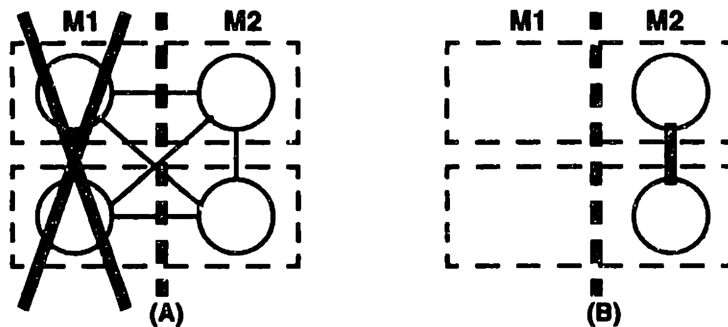

Figure 5. 9. During the machine-induced failures, links between the surviving processes in the damaged RUs are critical. When a machine M1 crashes (A), the recovery of the system as a whole depends on the reliability of the communication channels left is between the damaged RUs (B).

Therefore, all types of components contribute to critical second failures. The vulnerability period of a broken machine spans the detection and recovery time for its processes a described in Section 5.3.2. It is illustrated in Figure 5. 10.

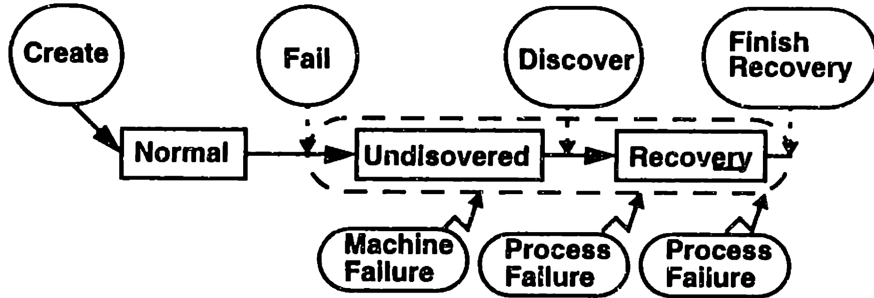

Figure 5. 10. The vulnerability period for Machine-induced failures is proportional to the time it takes to discover the failure of its processes and to repair them. Some machine, process or even link failures during this vulnerability period may cause the system to crash.

There are r machines in the system, therefore

$$c_m = r \tag{5.21}$$

The vulnerability period for a machine is a sum of the time it takes to detect the failures of its processes, and then to repair those processes, usually on some other machines. There n/r processes running on each machine, but because the detection is occurring in parallel, the time to detect n/r failures should be approximately equal to the time to detect one failure. Repair, however, is a lot more expansive. Every process that is being repaired has to receive the antecedents information from every RU in the system. Later, every recovering process has to transfer state to the newly created secondary process. Considering that the workstations are reasonably fast and the processes are likely to have Megabytes of state to transfer, the recovery information for a single process should saturate the network bandwidth. Thus, if several processes are trying to recover simultaneously, the recovery takes the time proportional to the number of recovering processes. Thus, a vulnerability period for a machine-induced failure is

$$v_m = \frac{D_p + \frac{n}{r} \cdot R_p}{F_m + D_p + \frac{n}{r} \cdot R_p} \approx \frac{D_p + \frac{n}{r} \cdot R_p}{F_m} \qquad (5.22)$$

Now let us consider the probability of the second critical failure occurring after a machine has crashed. First consider the link failures. After a machine death there are n/r unreplicated processes in the system. They have $n^2/2r^2$ links between them. Failure of any of those links will cause a death of an unreplicated process, and therefore a crash of the whole system. The probability of a link failure is $1/F_l$, therefore the total link contribution is $n^2/(2r^2 \cdot F_l)$.

Now consider the contribution of the process failures. After a machine crash there are n/r unreplicated processes. Failure of either of them will crash the system. The probability of a process failure is $1/F_p$, therefore processor failure contribution is $n/(r \cdot F_p)$.

5.3.10 Machine Level system configuration

Finally, consider the contribution of additional machine failures. The original Somersault algorithms insure that a system can survive most of the possible simultaneous independent failures. However, as machine failures cause a death of a large number of processes, they can not be classified as an independent failure mode. If caused by hardware, machine failures may take a long time to fix, and increase the vulnerability of the system. Therefore, if the special care is not taken to arrange the RUs between the machines in a most protective way, machine failures may become the major weakness for Somersault:

- The basic assumption for all the deductions presented here is that the RU member processes reside on different machines, so that a machine failure does not destroy complete RUs.

After a machine failure, a death of any of the remaining machines containing one or more of unreplicated processes will lead to a crash of the system. If n/r unreplicated pro-

cesses are spread throughout the system, then chances are that every machine has at least one unreplicated process. However, if all the RU members are shared between the pairs of machines, then there is only one machine whose failure is critical (see Figure 5. 11).

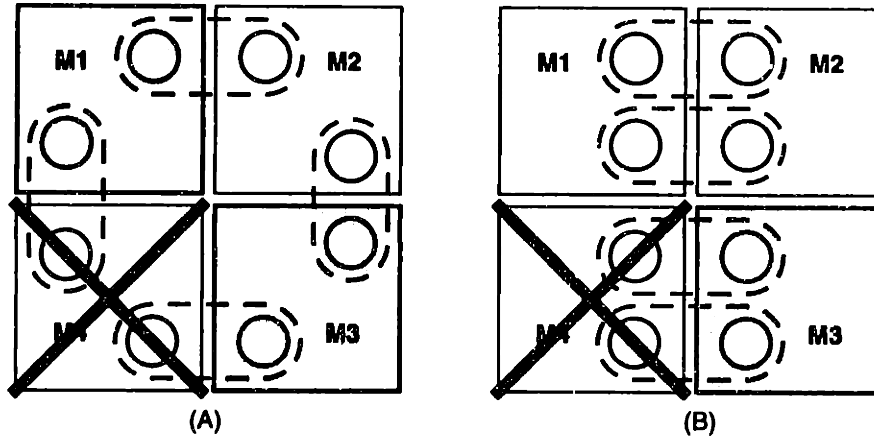

Figure 5. 11. Spreading the RU members across the system decreases the reliability of the system, while localizing the RU members minimizes the chances of total system failure. When a machine M4 crashes, in a system with a highly dispersed RUs (A) two machines M1 and M3 are critical to the survival of the system. In a system where the RUs are localized to the pairs of machines (B), only one machine M3 is critical to systems survival.

Obviously, the paired machines offer a much higher reliability. However, we have to consider the maintainability of this system as well. After a machine dies, the machine that carries the mates of the dead RU members has to choose where to restore the complete RUs. The two extreme choices are to pair up with some other machine or shear its RUs with many other machines (see Figure 5. 12). If it pairs up with only one machine that is

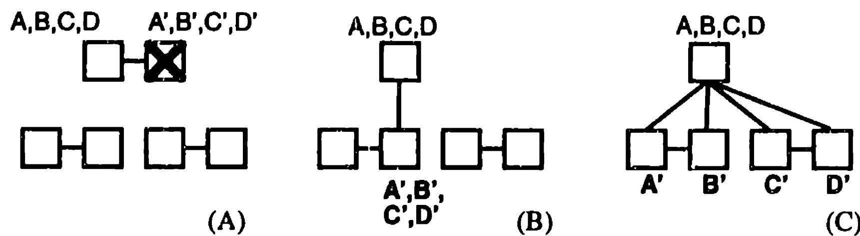

Figure 5. 12. Rebuilding the processes killed during a machine crash (A). (B) -rebuilding on a single machine requires extra processing capacity which is not normally used; (C) - rebuilding on several machines across the system increases the vulnerability to machine induced failures.

already paired up with someone else, it is essential to have enough processing and memory capacity on any machine to carry a double amount of processes. Alternatively, if the surviving machine chooses to spread its RUs across the system, it will increase the vulnerability of the surviving system to machine induced failures. Also, in this case as the system is subjected to additional machine failures, more and more RUs will be shared throughout the system, thus increasing the interdependence between machines and reducing the overall system reliability.

Neither of the above scenarios is completely satisfying. Let us consider how we could improve the fault tolerance of the system in case of machine failures. First let us consider the use of a spare machine to host the recovering processes (see Figure 5. 13).

Figure 5. 13. In case of a machine failure (A) a spare machine s is used to host the repaired processes (B).

This approach resolves the problem of overloading the surviving machines, or corrupting the paired-machines structure across the system. However, it has several drawbacks. First of all, it requires the use of an additional machine. More importantly, if the spare machine crashes, its likely to be due to hardware faults, and thus repairs may take days, leaving the system exposed to machine induced failures. The proportion of time during which a system will be vulnerable to machine failures due to the failure of the spare will be on the order of $R_{hardware}/F_{hardware}$ (vulnerability is a dimensionless quantity). Given the typical values of $R_{hardware} \approx 1$ day and $F_{hardware} \approx 2$ years, the vulnerability of such system will be on the order of 10^{-3}. This is an order of magnitude lot higher than in a system that

does not depend on spares and for the same components has vulnerability of $r \cdot R_m / F_m \approx 10\text{minutes}/2\text{month} \approx 10^{-4}$.

A combined approach that allows a system using a spare machine, in case of additional failures, to spread the RUs along the remaining machines would solve the problem. However, it is likely to make the system more complex. Instead, let us consider an approach which trades a slight reduction in the initial reliability of the system for simplicity and ability to tolerate multiple machine faults. To be precise, two machine failures in this case may still crash the system, but the vulnerability window is on the order of a process recovery time, not a physical machine repair time.

Machines are arranged in a logical ring, so that each machine shares its RUs only with its neighbors. When a machine crashes, its neighbors take over the recovering processes and pass some of them to their surviving neighbors (see Figure 5. 14). During the recovery there are only two machines that are critical, but the vulnerability period is small, and the dependency level between the surviving machines does not change.

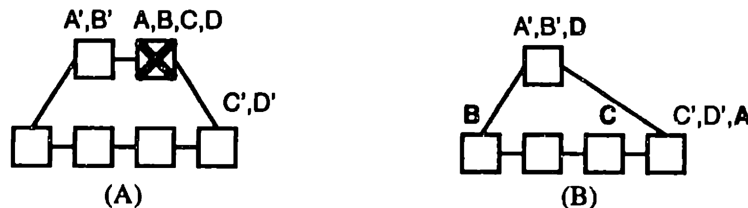

Figure 5. 14. Closure of the ring of machines under failure. When a failure of a machine carrying processes A,B,C and D occurs, some of these processes are recovered on the neighbor machines (processes A and D), the rest are resettled on the neighbors (processes B and C). The system configuration has the same structure as before the failure.

The only problem left to resolve now is how to reintroduce machines back into the ring after they are rebooted/repaired. The best place for them to be put in is where the old machines used to be before failure. Placing the machines in their original place and migrating the original processes back onto them will solve the problem of load balancing.

5.3.11 Process Migration

Process migration is closely related to introducing new machines into the system. Currently, the process migration is implemented by killing off a secondary process and then rebuilding it on another machine. However, this increases the vulnerability of the system by adding more unreplicated processes during migration. Instead, Somersault should be able to create an additional secondary process on another machine, while the original secondary is still up and running (see Figure 5. 15). Then, when the state of the new secondary is up to date with the state of the primary, the original secondary can be killed, and the RU will still be operational, and the secondary will be running on a different machine.

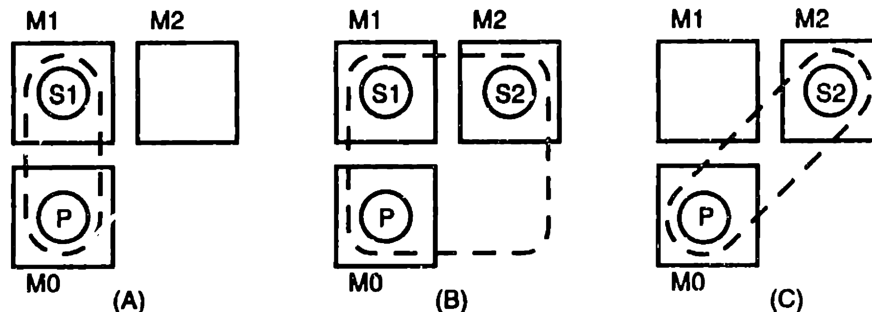

Figure 5. 15. Migrating the secondary to a different machine. A RU consisting of a primary P and a secondary S1 resides on machines M0 and M1 respectively (A). To migrate the secondary onto machine M2 a new secondary S2 is created on M2 (B). When S2's state is up to date, S1 is killed, and the completely functional RU is established on machines M0-M2 (C). The system is never vulnerable to a single failure during the whole procedure.

It seems that while moving secondaries from machine to machine is relatively easy. However primaries appear to be immobile. There is a simple solution to this problem. Synchronize the states of the primary and the secondary process within a RU, then swap their roles, so that the primary becomes a secondary and can be moved around.

From the above reasoning we shall assume that the machines are organized in a logical ring. Then, the only two machines that are critical to the system's survival after a machine

crash occurs are its two neighbors. Thus, the contribution from the second machine failure is $2/F_m$. The total probability of a second critical failure therefore is:

$$p_m = \frac{n^2}{2r^2 \cdot F_l} + \frac{n}{r \cdot F_p} + \frac{2}{F_m} \tag{5.23}$$

The overall probability of the machine-induced failure is

$$P_{mx} = c_m \times v_m \times p_m = r \cdot \left(\frac{D_p + \frac{n}{r} \cdot R_p}{F_m} \right) \cdot \left(\frac{n^2}{2r^2 \cdot F_l} + \frac{n}{r \cdot F_p} + \frac{2}{F_m} \right) \tag{5.24}$$

5.3.12 Relationship between Variables and Simplification

Finally, we have all the contributions to the system failure defined. The probability of system failure from all possible causes is (from Equations 5.11, 5.20 and 5.24) is:

$$P_{sys} = P_{lx} + P_{px} + P_{mx} \tag{5.25}$$

$$= \left(\frac{n^2}{2r} \cdot \frac{D_l}{F_l} \cdot \frac{1}{F_m} + \frac{n^2}{2} \cdot \frac{R_l}{F_l} \cdot \left(\frac{1}{F_p} + \frac{1}{F_m} \right) \right)$$

$$+ \left(n \cdot \frac{D_p + R_p}{F_p} \cdot \left(\frac{1}{F_p} + \frac{1}{F_m} \right) \right)$$

$$+ r \cdot \left(\frac{D_p + \frac{n}{r} \cdot R_p}{F_m} \right) \cdot \left(\frac{n^2}{2r^2 \cdot F_l} + \frac{n}{r \cdot F_p} + \frac{2}{F_m} \right)$$

To make sense of this result for the probability of Somersault's failure we have to simplify it. We begin by examining specific values and relationships between the variables, and then try to eliminate the parts of the equation whose contribution is negligible compared to the contributions of the other parts.

5.3.13 Specific Values

From industrial practice we know that when both hardware and software faults are taken into account [20]

$$F_p \approx F_m \approx 2\text{month} \tag{5.26}$$

Also, assuming that an average size of an application state is several Megabytes and the throughput of the network is 10 Megabits/sec we get

$$R_p \approx 10 \text{ seconds} \tag{5.27}$$

Because the links are repaired by killing off one of the end processes (see Recover_Link algorithm in Figure 5. 4), link recovery time is the same as for a process

$$R_l \approx R_p \approx 10 \text{ seconds} \tag{5.28}$$

5.3.14 Relationships between variables

Now let us examine the remaining variables in the Equation 5.25. Consider n (the number of processes), and r (the number of machines) as free variables, at least for now. As it was mentioned earlier

$$n \geq r \tag{5.29}$$

F_l varies a lot depending on the type of underlying network and the communication protocol. We will examine how different orders of its values affect the failure probability later.

The failure detection mechanism does the job of detecting failures in different parts of the system in parallel. Moreover, it judges the symptoms of failure and then decides what kind of failure it is. Because the number of possible failure types is small (process, link and machine), if it is not one kind of failure, than it is one of the other. Therefore, the detection times are approximately equal for the different types of failures

$$D_l \approx D_p \approx D_m \tag{5.30}$$

Now, let us rewrite the equation for the total probability of the system failure taking into account the relationships we have just discussed. Let us use the process variables as the common basis for simplification.

$$P_{sys} = \frac{n^2 D_l}{2rF_lF_p} + n^2\frac{R_p}{F_lF_p} + 2n\frac{D_p + R_p}{F_p^2} + \frac{rD_p + nR_p}{F_p}\left(\frac{n^2}{2r^2F_l} + \frac{n}{rF_p} + \frac{2}{F_p}\right) \tag{5.31}$$

5.4 Analysis

In this section we will attempt to build the model of reliability of the system which would give us an estimate of the optimal values for the system parameters that we can vary: the number of machines r, and the detection times D_p given the reliability of the system components, the desired number of processes n, and the desired $MTTF_{sys}$.

Our analysis is reliability driven; we do not consider the performance implications here. An important criteria we take into account is the number of machines used to run the system. It has to be as low as possible in order to make the system economical. Finally we want the detection time to be as large as possible, so that the failure detection overhead is kept as low as possible.

From the Equation 5.31 we see that the probability of system failure is proportional to the weighted sum of the link detection (D_l), process detection (D_p) and process recovery (R_p) times. We can not change the contribution introduced by the recovery time, but we can make sure that the contribution of the detection time does not outweigh it. For simplicity, let us say that we will be satisfied with the contribution of each of the detection times being equal to the contribution of the recovery time. Shortly, we shell find the maximum acceptable detection times in the form

$$D_l^{max} = f_l(R_p, n, r) \qquad D_p^{max} = f_p(R_p, n, r) \tag{5.32}$$

Knowing that while the contribution of detection times to the overall probability of the system failure is equal to the contribution of the recovery times, we can rewrite the Equation 5.31 by eliminating the detection time contributions and multiplying the recovery contribution by three:

$$P_{sys} = 3\left(n^2 \cdot \frac{R_p}{F_l} \cdot \frac{1}{F_p} + n \cdot \frac{R_p}{F_p} \cdot \frac{2}{F_p} + \frac{nR_p}{F_p} \cdot \left(\frac{n^2}{2r^2 \cdot F_l} + \frac{n}{r \cdot F_p} + \frac{2}{F_p}\right)\right) \tag{5.33}$$

Now observe that the overall probability of system failure is inversely proportional to the number of machines r used in the system. Therefore, for an optimal reliability we would have to run one process per machine. However, we only need to obtain a certain level of reliability. In the Equation 5.33 r and n are the only free variables. For any given n we can find what is its minimal acceptable value of r which gives the required system reliability.

$$r^{min} = g(P_{sys}, n, F_{comp}, R_{comp}) \tag{5.34}$$

Finally, when we know the function for r^{min} we can find out the link and process detection times corresponding to it. To do this, substitute the value of r^{min} into the Equation 5.32 for computing D_l^{max} and D_p^{max}.

In order to simplify our analysis, and to better see the trends, we will evaluate the system performance for three different ranges of link reliabilities - $F_l \gg F_p$, $F_l \approx F_p$ and $F_l \ll F_p$. They roughly correspond to using highly reliable dual LANs, using an isolated network and running across a WAN respectively.

5.4.1 Highly reliable links

If $F_l \gg F_p$, then considering Equation 5.31 we can eliminate all the terms containing F_l in the denominator, thus reducing it to

$$P_{sys} = n \cdot \frac{D_p + R_p}{F_p} \cdot \frac{2}{F_p} + \frac{rD_p + nR_p}{F_p} \cdot \left(\frac{n}{r \cdot F_p} + \frac{2}{F_p}\right) \tag{5.35}$$

The contribution of the detection times is equal to the contribution of the recovery times when

$$D_p^{max} = R_p \cdot \frac{n(4r+n)}{r(3n+2r)} \tag{5.36}$$

Considering that we are looking for a safe upper bound on the detection time and that number of machines r can not exceed the number of processes n we can approximate

$$D_p^{max} = R_p \cdot \left(0.8 + \frac{n}{5r}\right) \quad (5.37)$$

Link detection time D_l^{max} can be anything, as long as it is much smaller than F_l, which is very large in this case.

Now let us find the minimal number of machines required to run the system at a given reliability level. Considering that reliability of links in this case is a lot higher than reliability of processes, we can rewrite the Equation 5.33 for probability of system failure as

$$P_{sys} = \frac{1}{F_{sys}} = 3\left(n \cdot \frac{R_p}{F_p} \cdot \frac{2}{F_p} + \frac{nR_p}{F_p} \cdot \left(\frac{n}{r \cdot F_p} + \frac{2}{F_p}\right)\right) \quad (5.38)$$

Solving for r we get

$$r^{min} = \frac{3n^2 R_p}{\frac{F_p^2}{F_{sys}} - 12nR_p} \quad (5.39)$$

Now, knowing the equation for maximum detection time and the minimal number of machines for a given system size we can find the specific values D_p for any given n.

Here are some examples which illustrate the relationship between the variables discussed above and the size of the system. In all the following cases we consider a system with $MTTF_{sys} = 20$ years, $F_p = 2$ month and $r_p = 10$ seconds.

From the graph for the number of machines vs. number of processes used in the system (Figure 5. 16) we see that the system with highly reliable links will scale up to almost 300 processes, with the low minimal number of machines required for the systems with up to about 100 processes. The number of processes per machine is illustrated in Figure 5. 17.

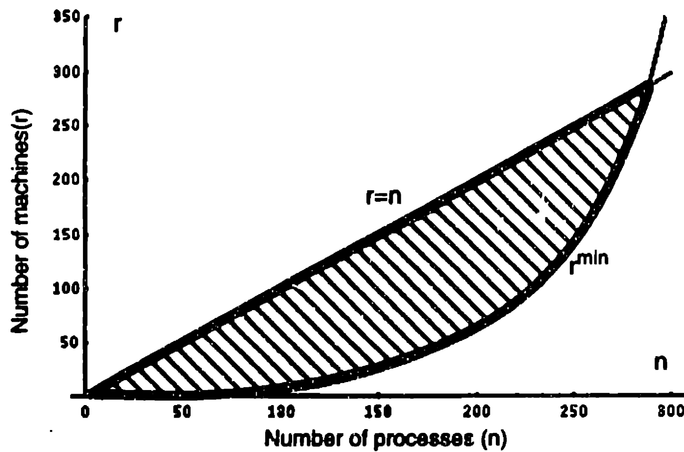

Figure 5. 16. A range of acceptable number of machines in the system. The top bound is shown here because the number of machines can not exceed the number of processes. It is important that there is an upper bound on the size of the system when the minimal number of machines required is equal to the number of processes.

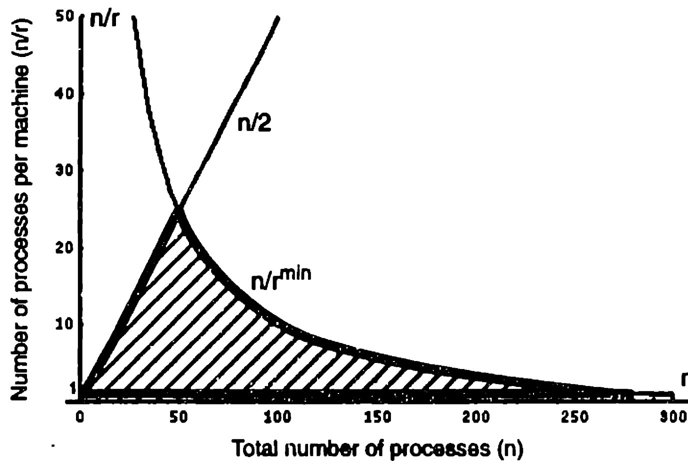

Figure 5. 17. A range of acceptable number of processes per machine. Even though for small numbers of processes our equations show that a very large number of processes can run on each machine, not more than half of the total number of processes can run on each computer.

The best processes per machine ratio is obtained at the system size of about 60 processes, and the values are quite reasonable for the system of up to about 150 processes (see Figure 5. 17).

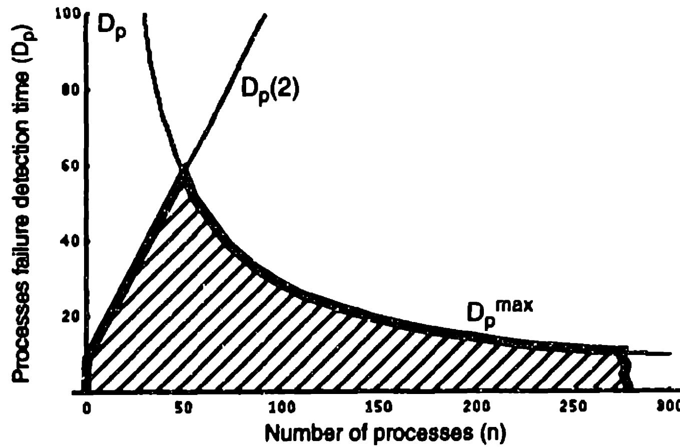

Figure 5. 18. Process detection time. For small **n** the detection time is limited by the minimal number of machines in the system equal to 2.

The most economical process failure detection time is obtained for the systems containing about 60 processes, however, the detection times are quite reasonable even for the large systems (Figure 5. 18).

Overall, from the graphs considered above, we conclude that if communication links are highly reliable, the system under constraints specified above will scale well up to about 150 processes. For larger systems, the number of machines required will be prohibitively high.

5.4.2 Links as reliable as processes

If $F_l \approx F_p$ then we can substitute F_p for every occurrence of F_l in the Equation 5.31:

$$P_{sys} = \frac{1}{F_p^2}\left(\frac{n^2}{2r}D_l + n^2R_p + 2n\,(D_p + R_p) + (rD_p + nR_p)\left(\frac{n^2}{2r^2} + \frac{n}{r} + 2\right)\right) \tag{5.40}$$

Let us find the detection times under which the contribution of failure detection will not exceed the contribution of process recovery. Link detection time

$$D_l^{max} = R_p \cdot \frac{2r}{n} \cdot \left(n + 4 + \frac{n^2}{2r^2} + \frac{n}{r}\right) \approx R_p \cdot \left(2r + \frac{n}{r}\right) \tag{5.41}$$

can be really long under two extremes. When there are very few machines, machine failures are unlikely; therefore links do not contribute to the overall probability of system failure. Alternatively, when there are so many machines that they run only a few processes each, the number of dangerous links is small.

Now let us find the process detection times whose contribution does not exceed the contribution of process recovery. From Equation 5.40 we find that process detection time is limited by

$$D_p^{max} \approx R_p \cdot \frac{n}{r} \tag{5.42}$$

Now if we use the detection times found above, we can approximate the system failure probability as

$$P_{sys} = \frac{1}{F_{sys}} = \frac{3R_p n}{F_p^2}\left(n + 4 + \frac{n^2}{2r^2} + \frac{n}{r}\right) \approx \frac{3R_p n}{F_p^2}\left(2n + \frac{n^2}{2r^2}\right) \tag{5.43}$$

Therefore, we can find the minimal acceptable number of machines

$$r^{min} = n\sqrt{\frac{3nR_pF_{sys}}{F_p^2 - 6n^2R_pF_{sys}}} \tag{5.44}$$

Again, by substituting r^{min} into Equation 5.41 and Equation 5.42 we get the maximal acceptable detection times as a function of number of processes. Here are the illustrations for a typical system.

A system whose link reliability is about the same as its process reliability scales an order of magnitude worse than the system with highly reliable links. The number of machines required stays small for systems with up to about 25 processes (Figure 5. 19).

The maximal number of processes per machine in a system with moderately reliable links under the above constraints can not exceed 8. This perhaps is not enough to keep the machines completely loaded, but may allow for use of cheaper machines. The load on

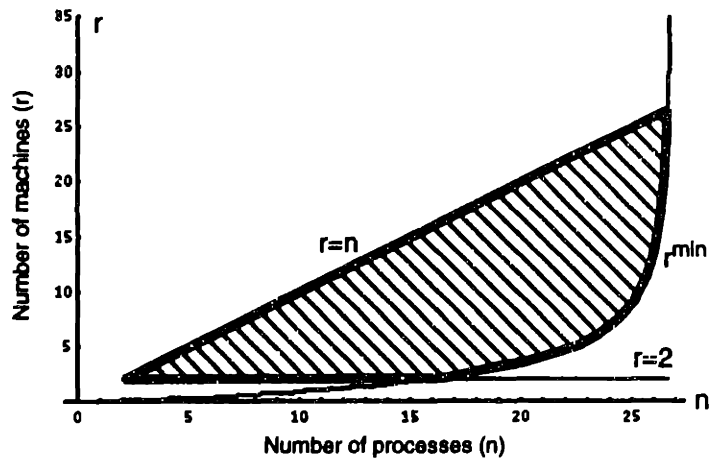

Figure 5. 19. Minimal acceptable number of machines for a system with links as reliable as processes.

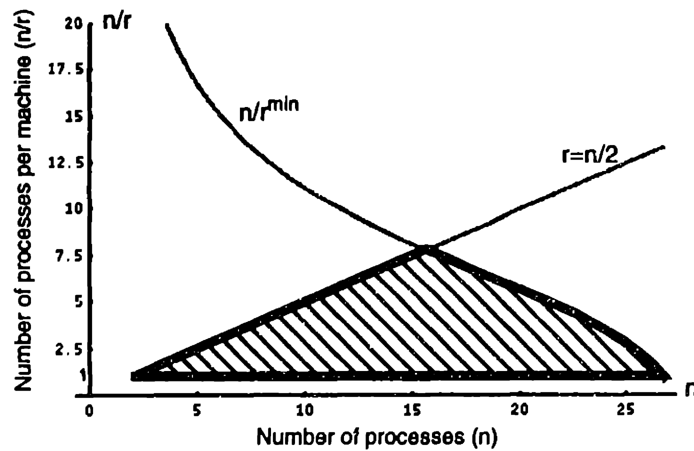

Figure 5. 20. Maximal acceptable number of processes per machine for a system with links as reliable as processes.

machines will stay reasonable only for small systems with up to about 20 processes (Figure 5. 20).

Process failure detection times are never too small, except for the systems with the number of processes close to maximum (see Figure 5. 21). However, systems with the

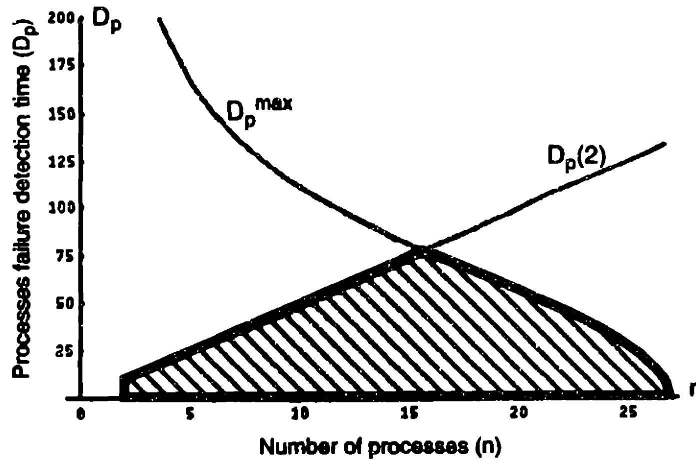

Figure 5. 21. Maximal acceptable process failure detection time for a system with links as reliable as processes.

maximal possible number of processes are not likely to be built because they will require using 1 machine per 1 process, and thus will be too expensive (see Figure 5. 20).

Link failure detection times are always quite large, so they should never become the limiting consideration for the size of the system (Figure 5. 22)

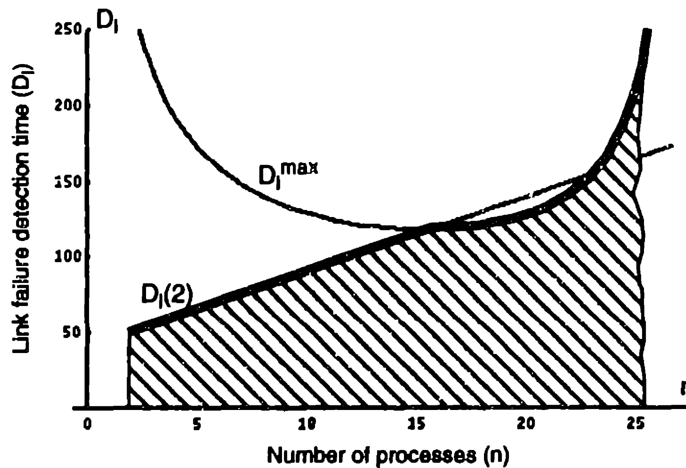

Figure 5. 22. Maximal acceptable link failure detection time for a system with links as reliable as processes.

From the graphs ([5. 19], [5. 20], [5. 21] and [5. 22]) we conclude that systems with the best operational parameters have from 10 to 20 processes, and run on a few machines, with process detection time of about 50 seconds and link detection time of about 100 seconds.

5.4.3 Unreliable Links

Finally when $F_l \ll F_p$, we can eliminate all terms that do not contain F_l in the denominator from Equation 5.31:

$$P_{sys} = \frac{n^2}{F_l F_p} \cdot \left(2rD_l + R_p + \frac{rD_p + nR_p}{2r^2} \right) \tag{5.45}$$

Again, find the expression for the detection times that give the contribution to system failure probability approximately equal to that of process recovery. We get:

$$D_l^{max} = R_p \cdot \left(\frac{1}{2r} + \frac{n}{4r^3} \right) \tag{5.46}$$

$$D_p^{max} = R_p \left(2r + \frac{n}{r} \right) \tag{5.47}$$

When the detection times are less or equal to the ones specified above in Equation 5.46 and Equation 5.47 the probability of system failure can be approximated as

$$P_{sys} = \frac{1}{F_{sys}} = \frac{3n^2 R_p}{F_l F_p} \left(1 + \frac{n}{2r^2} \right) \tag{5.48}$$

Thus, we can find the minimal appropriate value for the number of machines in the system:

$$r^{min} = n \sqrt{\frac{3nR_p F_{sys}}{2\left(F_l F_p - 3n^2 R_p F_{sys} \right)}} \tag{5.49}$$

Now we can find the actual vales for the detection times by substituting r^{min}, found above, into Equation 5.46 and Equation 5.47.

For the following graphs we use our usual timing estimates of $MTTF_{sys} = 20$ years, $F_p = 2$ month and $r_p = 10$ seconds, and estimate link reliability to be $F_l = 1$ day.

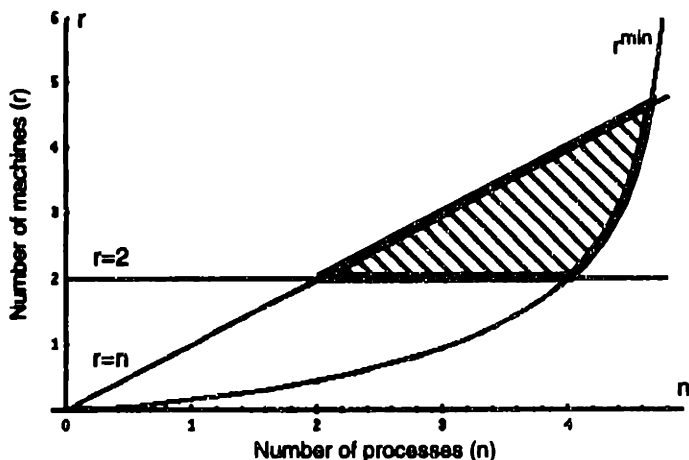

Figure 5. 23. Acceptable number of machines for a system with unreliable links

Under these constraints systems with unreliable links do not scale well at all. From the above Figure 5. 23 we see that the biggest number of processes such system can maintain is 4.

The maximal process detection time (Figure 5. 24) is dominated by the contribution of a two-machine system; process detection times are quite high, especially considering the very small size of the system.

The link detection times are very low, but still practical, considering a small size of the system. Again, the contribution of the two-machine system dominates the picture (Figure 5. 25).

Overall, it seems impractical to build systems with unreliable network connections. Still, one could imagine simple applications which need to be robust and run over WANs, for implementing which Somersault could be helpful.

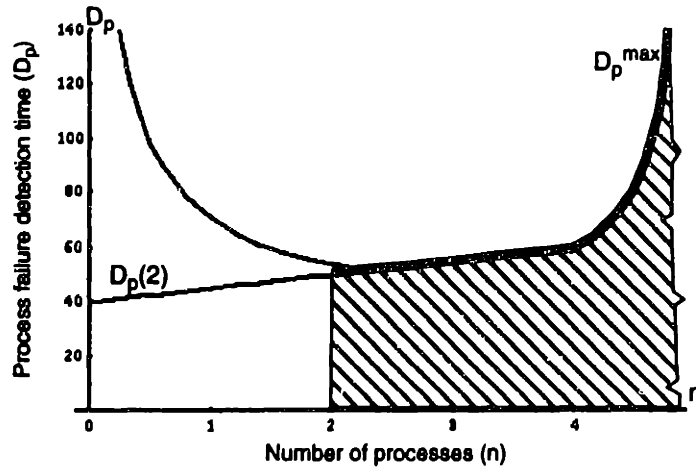

Figure 5. 24. Acceptable process failure detection times for a system with unreliable links

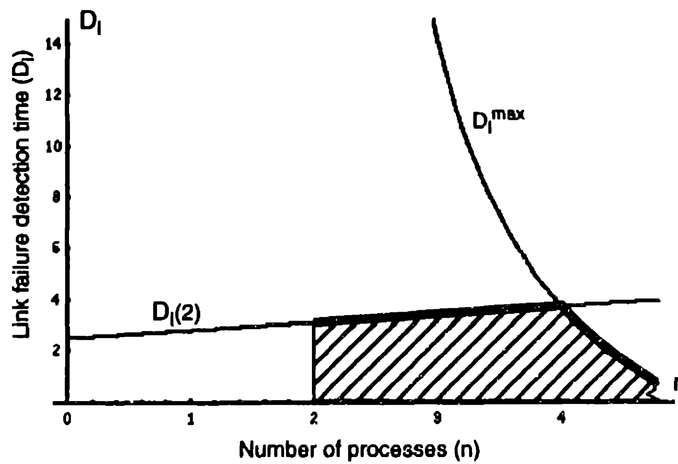

Figure 5. 25. Acceptable link detection times for a system with unreliable links

5.5 Important trends

This section summarizes some important observations that follow from the above analysis. Our conclusions, however, are optimistic. There are several caveats suggesting why in practice Somersault's availability and limits of scalability may be lower than indicated above. In the preceding calculations we made many simplifications and generalizations, plus we assumed that:

- Component failures are independent from each other. However, it maybe the case that real systems are subject to multiple simultaneous failures
- For all component types $MTTF \gg MTTD$ and $MTTF \gg MTTR$. However, this may not hold in some systems
- Machines are organized in a logical ring with respect to recovery units. This configuration is optimal for providing fault-tolerance, but may be impractical for performance reasons. Therefore, we can not expect Somersault to always operate in this optimal configuration.
- Finally, we used specific numbers in the later derivations. However, these values are only estimates, accurate to an order of magnitude

Thus, the following conclusions should be interpreted as estimates, rather than strict requirements. Even though they are not precise, we believe that they are useful because they reflect general trends rather than specific values.

5.5.1 Minimal Number of Machines

It may seem surprising that the minimal number of machines required to reach a certain level of reliability increases faster than the size of the system. Indeed, the more machines there are in the system, the more things there are to break. However, when the number of machine is large, there are fewer processes to run on each machine. Thus, when a machine crashes there are fewer unreplicated processes and critical links between those processes. A linear increase in the number of machines in the system leads to a quadratic decrease in the number of links between unreplicated processes that are critical to a survival of the system after a machine crash. That is why the minimal number of machines required to run large systems is so high.

5.5.2 Process Size

An important implication of the analysis presented here is that Somersault will not deliver the required fault tolerance, if the processes that run on it are too large. The reason is that the probability of system failure is proportional to the weighted sum of a process and link

detection times and a process recovery time. If the contribution of any one component is too big, there is no point in minimizing the others. We can control the time it takes to detect failures, however, we have no control over the process recovery time. In the above examples we assumed that the process size was on the order of several Mbytes and recovery times of ten seconds. However, if the processes were to become a lot bigger, then process recovery time would go up too, increasing the system vulnerability to additional failures. Thus, the probability of a total system failure would go up as well. As mentioned earlier, there is no way to reduce it significantly by using faster failure detection.

The only chance for running Somersault with large processes is on a Gbyte network using very fast machines. Then the process recovery, even for very large processes, will still take only tens of seconds, and the probability of the system failure will not change compared to the smaller processes on Mbit networks used in our analysis.

5.6 Ways to Increase Fault Tolerance of Somersault

The caveats related to assumptions of independence of failures, machine configuration, simplification, and imprecise data used for analysis are all applicable here as well. However, until we have a better model, the conclusions presented here are still important as they indicate the principle ways in which reliability of Somersault can be increased.

5.6.1 Component requirements

From the previous section we can conclude that in order to achieve a required availability of 99.9994% in a Somersault system consisting of tens of processes, the *reliability of the underlying network and protocols should be comparable to or greater than the reliability of Somersault processes*, which are currently rated at mean time to failure of 2 month.

In Section 5.4 and Section 5.5.2 we mentioned that smaller recovery times lead to lower vulnerability periods for the components, and thus reduce the probability of the overall system failure. We also mentioned that the process recovery time is dominated by the state transfer. Therefore, to reduce the probability of system failure we have to make the state transfers faster. In other words, *we need to use faster networks*.

5.6.2 System Configuration

In order to reduce the probability of total system failure the recovery units have to locate their member processes on different machines. The *machines should be organized into a logical ring, so that every machine shares its RUs only with its two neighbors*. This way the vulnerability of the system to multiple machine failures is reduced, while the reliability of the system does not change much after multiple failures and repairs.

When it is necessary to migrate an RU member process from one machine onto another, a *three-process recovery unit* should be created temporarily. It should include the old primary and secondary processes, plus a third process on the target machine. When the state of the third process is synchronized with the state of the primary, the old secondary process can be killed, and the newly created process on the target machine takes over its role. Therefore, the whole migration is completed without ever having a single unreplicated process.

5.6.3 Failure Detector Functionality

For highly reliable networks, the latency of link failure detection does not matter much. Somersault built on such a network probably does not require a special mechanism to do link failure detection, instead it can use the facilities provided by the point-to-point communication software. For instance TCP's failure detection mechanism guarantees to

detect any link failures within 10 minutes. This is certainly good enough for links that have a mean time to failure on the order of several month.

For the networks were link failures occur on the order of a single month the situation is quite different. Systems built on such system without a special link failure detection mechanism will never reach the required availability level.

There is however, an even more important reason to have a separate link failure detector. In the Table 5.1 we are able to eliminate about one third of all causes of total system failures, by distinguishing between the process and link failures, and applying an appropriate algorithm (see Figure 5. 4) to treat link failures separately. If we do not distinguish between the link and the process failures we will find that the reliability of the system decreases dramatically. For example, if we treat all failures as process failures, then because processes on both ends of a broken link declare each other dead, we would kill both of them, possibly completely destroying a whole RU.

Therefore, in order to be on the safe side, failure detection service in our system must have the following characteristics:

- When a failure is suspected, we *have to be able to determine weather its a process or a link failure,* and treat it appropriately.
- No failure should go undetected for an indefinite amount of time, for it increases the vulnerability of the system. Therefore all *process and link failures should be detected under some predefined period of time*. (The recommended values can be found in the previous sections of this chapter).

Chapter 6

Experiments

The goal of building the experimental system is two-fold. First, is to verify the correctness of the algorithms proposed above. Second, is to measure their performance.

The target availability rates imply that the MTTF of the whole Somersault system should be on the order of 20 years. One cannot wait this long to see if the system is working correctly. It is hard to assemble enough systems together to see if any failures occur in a shorter period of time. However, there are several options available to evaluate the FD algorithms presented in this thesis. We can test the FD algorithms under a wide range of circumstances and simulate different patterns of failure within a system to evaluate their performance. Some examples of such experiments are the following:

- Measure the time it takes to perform failure detection
- Measure the accuracy of failure detection (evaluate the percentage of false positive results produced by the failure detector)
- Evaluate the critical parameters that effect the performance of failure detection
- Check for undesirable interactions between the Foreground FD, Background FD and the view manager.

In this chapter we present the system that was built to run the types of experiments outlined above and report the obtained results.

6.1 Experimental Setup

The experiments were conducted using a distributed system that allowed us to simulate different failure modes that would occur in a real system. Failures were injected into the system from a special console process, which also collected information on accuracy and latency of failure detection we have implemented. The main benefit of this approach

was in observing the behavior of the failure detector in its "native" environment, while having a complete control over the failures occurring in the system. Needless to say that the ability to create failures on demand drastically reduced the time required for testing. Using this approach also allowed us to make repeated measurements of system performance in a large number of varied scenarios.

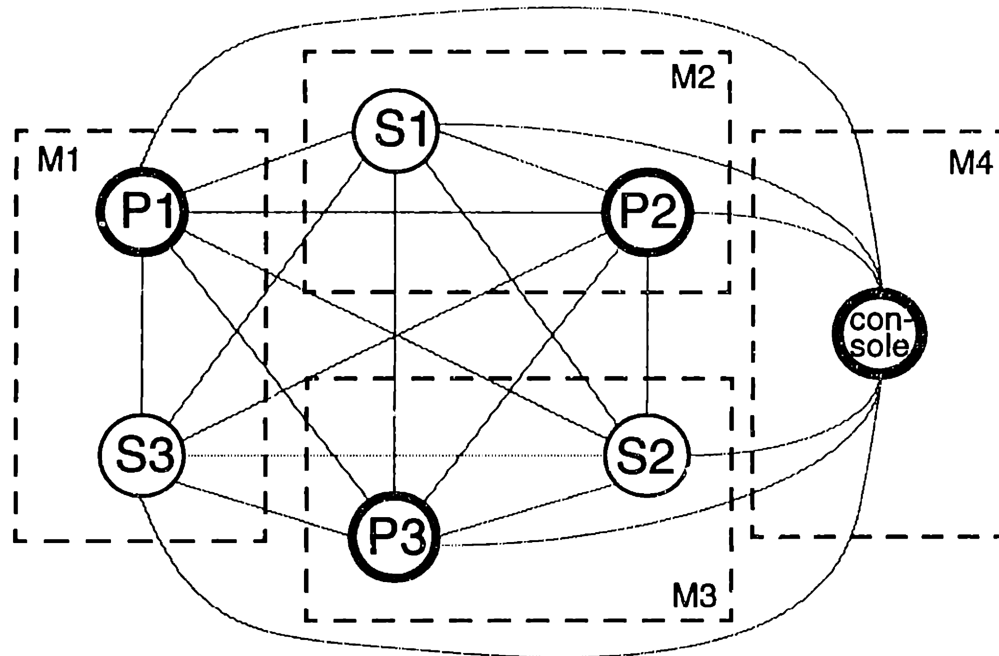

Figure 6.1. Experimental system setup. Dotted boxes are machines, circles are processes. Thick circles are primary processes, thin ones are secondary processes. Grey lines are connections. Three machines M1-M3 ran a primary and a secondary process from different recovery units each, and the forth machine M4 ran a control console process

The experiments were run on a minimal system configuration, which allowed us to test the correctness of algorithms and evaluate their performance. The system usually consisted of six processes, simulating the traffic between and within three Somersault Recovery Units, plus a console process that was used to run the test scripts and inject control messages into the system. The whole system was run on four HP-735s on a bridged-off FDDI ring. The experiments were run at night when the system activity was minimal.

However, there was still some traffic on FDDI ring, most notably generated by NFS. The process and machine configuration used in experiments is presented in Figure 6.1.

Three machines hosted two processes each, a primary and a secondary process from different RUs. All six of these processes were completely interconnected using TCP links. Primary processes ran Poisson traffic generators that sent between fifty and hundred messages per second to any other primary processes, and fifteen to thirty messages per second to their specific secondary process.

The fourth machine ran the console process. The console process was also connected to every process in the system, allowing it to send control messages to and receive reports from any process in the system. The console process executed a battery of test scripts that generated control messages. These messages caused the rest of the processes in the system to simulate various failures. Then the failure detector detected these failures them and reported its results back to the console process. The console process collected the results of the failure detection, measured the system's response time, and evaluated the correctness of failure detection results.

6.1.1 Process Structure and Control

The processes in the system has a layered structure, allowing for easy experimentation and development (see Figure 6.2). The lowest levels of the system, Naming and Process, provide means for the connection setup and naming. They also provide the low level messaging functionality in the form of *get* and *send*. All communication is asynchronous and is implemented in the Process level using the UNIX `select` call.

The Message_Ctrl level implements the failure-injection mechanism, and is critical for the experiment. Message_Ctrl provides the ability to block and unblock channels leading

to other processes on receipt of control messages, and allows dropping messages from the blocked channels.

Environment level hosts the View_Manager and the Failure Detectors. It provides Somersault system level abstractions like *primary* and *secondary* processes. The Application layer simulates the traffic that would be generated by a Somersault processes.

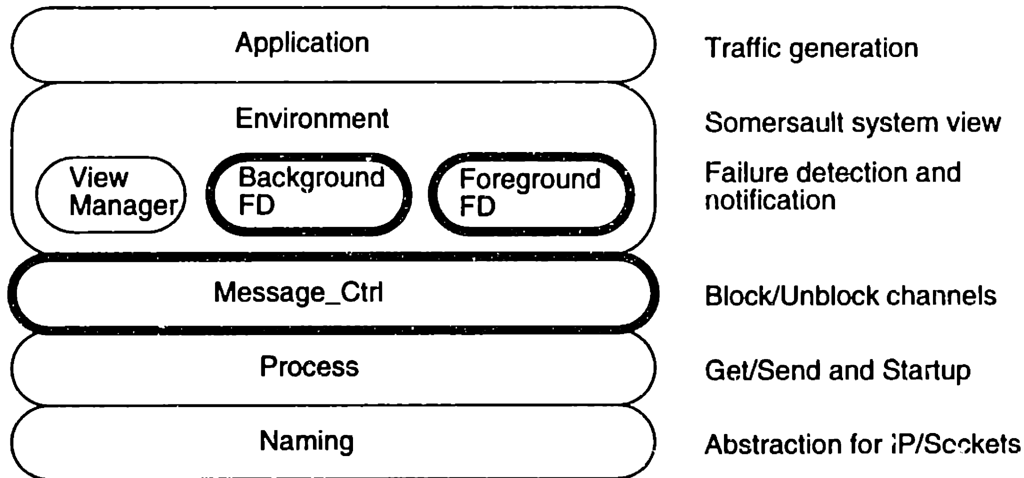

Figure 6.2. The layered structure of processes in the experimental system. We simulate failures by disabling communication channels within the Message_Ctrl layer. These disabled channels appear as failures to the Failure Detectors in the higher levels of the system.

The Message_Ctrl layer is the most interesting from the point of view of experiment design. This layer and the layers beneath it see all the processes in the system, including the console process. Layers above the Message_Control layer see only the other Somersault processes, but not the console. Thus Message_Control hides the console from Application, View Manager and Failure detectors.

Moreover, Message_Control maintains the state information about the channels of the process. Channels are either **blocked** or **unblocked**. If a channel is blocked, it looses all the messages going to and from it. If the channel is unblocked, it simply passes messages to the upper and lower levels of the system.

The state of the channel can be altered upon receipt of a Control message from the console. A control message specifies whether to block or unblock a channel or a set of channels. Because the console is invisible to the Failure Detectors, they treat a blocked channel as a communication failure in the system, and thus initiate the failure detection session.

Using this mechanism we can simulate various system failures. For instance, to simulate a link failure, we block one channel of a process. To simulate a process failure, we block all channels of that process. To simulate more complex conditions, like multiple communications failures we can inject the system with a combination of such control messages.

6.1.2 Messaging Mechanisms

The experimental system supported a large number of message classes implementing the Application traffic, View Management, Background and Foreground Failure detection, process Control and reporting (see Figure 6.3) Messages were represented as objects within both the sending and the receiving process. The methods of most message classes acted on the level of Environment, only the Ctrl_Messages acted on the Message_Ctrl level. Each message class implemented the following methods:

- Send/Receive: providing marshalling/unmarshalling and timer setup/cancellation
- Action: dispatching an incoming message to Failure Detector, View Manager, or Application
- Timeout: an action to be taken if the message has not been acknowledged after the timer has expired. Results in initiation of Failure Detection for Trigger Messages, and helps Failure Detection reach its conclusions.
- Elimination: an action to be taken if a message has been acknowledged. Important for Failure Detector making conclusions.

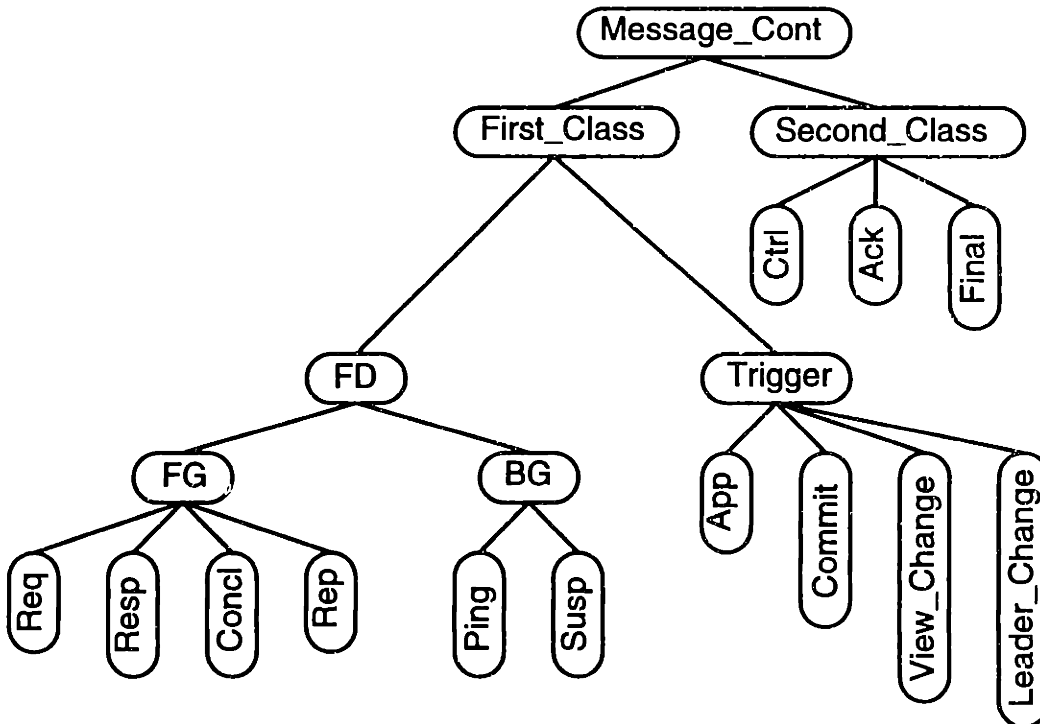

Figure 6.3. Message class hierarchy. The two main groups were the First_Class messages that generated Acknowledgments, and the Second_Class messages that did not. First_Class messages were further divided according to the use of Acknowledgments into subclasses that Triggered failure detection if acknowledgments were late, and the classes that implemented the FD algorithms using acknowledgments.

- Control: a block/unblock action taken on a channel or a set of channels at the Message_Ctrl level. This action is enabled only for Ctrl_Messages. Implements the fault simulation in the system as described in Section 6.1.1.

Here is an example of how messages are used in the system (see Figure 6.4). Assume that process P1 is the System View Manager.

(A) Console sends a *Control* message to a secondary process S2 telling it to block the channel connecting S2 with its primary P2, upon the receipt of this message S2 will start dropping all the messages going to/from P2.

(B) Then, P2 sends an *App*lication message to S2 and it is not *Ack*nowledged.

(C) After a certain time threshold, the P2 starts failure detection by *Req*uesting process P3 to arbitrate the possible failure of S2.

(D) P3 sends a message asking S2 to *Resp*ond if it is up and running. S2 *Ack*nowledges this message, because only its channel leading to P2 is down. Consequently P3 sends a *Concl*usion "S2 is alive" back to P2.

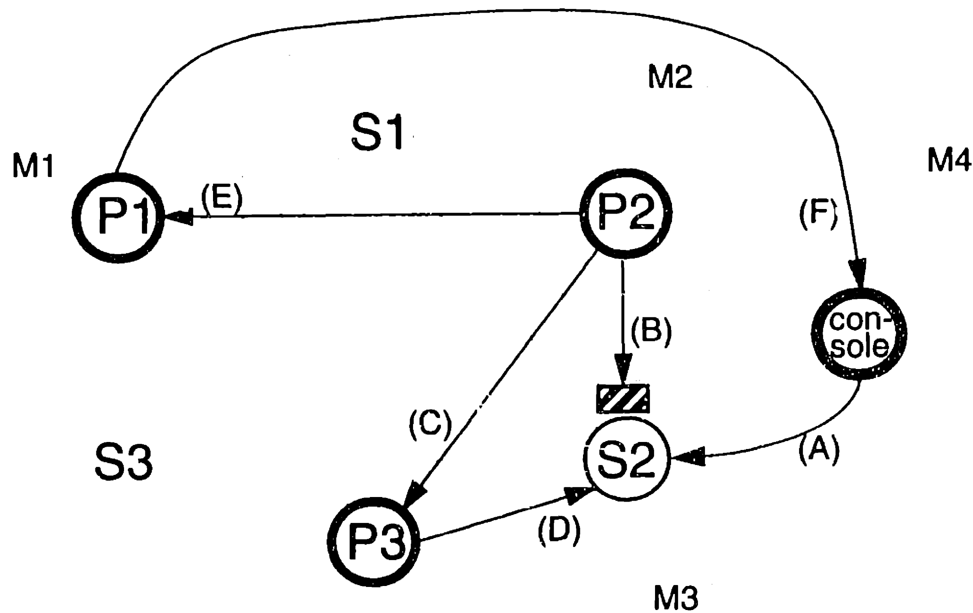

Figure 6.4. A typical experiment. (A) Console sends a control message to process S2 that causes S2 to block it's connection to P2. (B) P2 sends a message to S2 and does not get an acknowledgment. (C) P2 initiates Foreground Failure Detection by asking P3 to arbitrate. (D) P3 pings S2 and reports that S2 is up back to P2. (E) P2 concludes that S2 is up, but the P2-S2 link is broken and reports the failure to the view manager leader process P1. (F) P1 reports the failure to the console.

(E) At this point P2 knows that it can not communicate with S2, but P3, running on the same machine as S2, can. Therefore, concludes P2, the P2-S2 link is broken. Then P2 *Rep*orts this result to the View Manager process P1.

(F) Upon the receipt of this message P1 initiates the system view update by sending a *View_Change* message to all the Somersault processes in the system. Plus, P1 sends a *Final* message to the Console, to notify it of the detected link failure.

The Console then measures the time it took to get a response form the leader process and compares the detected failure with the one that was injected in the system, saving both timing and correctness results for later processing.

6.2 Experiments

This section describes the experiments conducted using the system described above and the results of these experiments. We first measure the distribution of message round trip times, then verify the correctness of Background Failure detection algorithms and then

concentrate on evaluating the performance of Foreground Failure Detection.

6.2.1 Round Trip Time Estimation

All the algorithms presented in the previous chapters are based on the assumption that the Round Trip Time of TCP packets is bounded, moreover, that it is small enough to permit the Failure Detection within the given timing constraints. We assume that even if the RTT is not uniform, the number of packets with a given RTT decreases as an inverse exponent of RTT. Using our system we ran an experiment to verify our hypothesis.

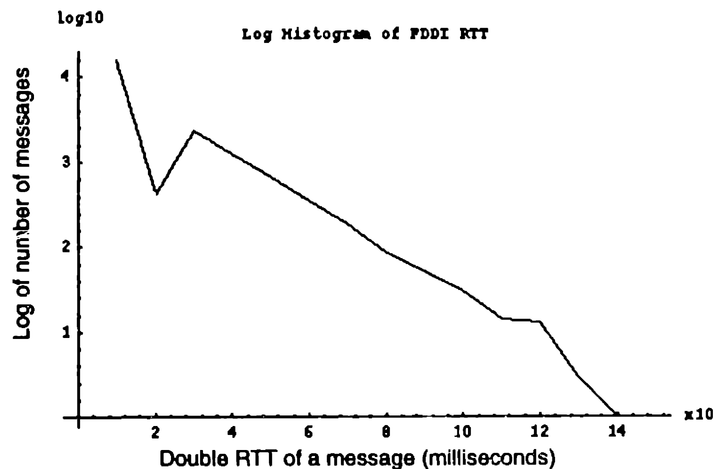

Figure 6.5. A logarithmic plot of number of TCP packets with a given RTT vs. the double of Round Trip Time in milliseconds

The above plot confirms our assumptions. However, some TCP packets made a double round-trip at around 150 ms. This implies that in order to be accurate, a multi-step failure detection will take close to or more than 300 milliseconds, thus leaving no time for recovery under the telecom requirements. Therefore, a faster network software should be used, or access to TCP acks should be provided, in order to avoid doubling the transmission times. Alternatively, we can measure the accuracy of Failure Detection with Total Dura-

tion under 300 ms. The rest of the experiments evaluate the accuracy and performance of FD algorithms under the varying timings.

6.2.2 Testing the Background Failure Detection

When testing the Background failure detection, our main objective was to check that the failures of the links and processes that do not communicate with the rest of the system are detected correctly. We ran a series of experiments in which we disabled the Foreground Failure detection, and then injected the node and link failures into the system. All the failures were detected correctly in as little time as 1.5 seconds. In order to make sure that the interaction between the Foreground and the Background failure detectors does not have a negative impact on the performance of the Background failure detector, we ran both failure detectors while injecting the failures into the links that did not carry any traffic. Again all these failures were detected correctly in 1.5 seconds.

6.2.3 Measuring the Performance of Foreground Failure Detection

According to the experiment described in Section 6.2.1, packet RTT is on the order of hundreds of milliseconds. This RTT is rather slow compared to the telecom requirement of 250 millisecond detection and recovery time. Therefore, it is important to understand how fast the Foreground Detection can run, and how its timings affect its correctness and performance.

Let us define some terms for describing the timings of Foreground Failure Detection algorithms. The important parameters are (see Figure 6.6):

- Fire Threshold: how long to wait before starting arbitration.
- Arbitration Span: how long to wait before examining the outcome of arbitration
- Total Duration: equals Fire Threshold plus Arbitration Span

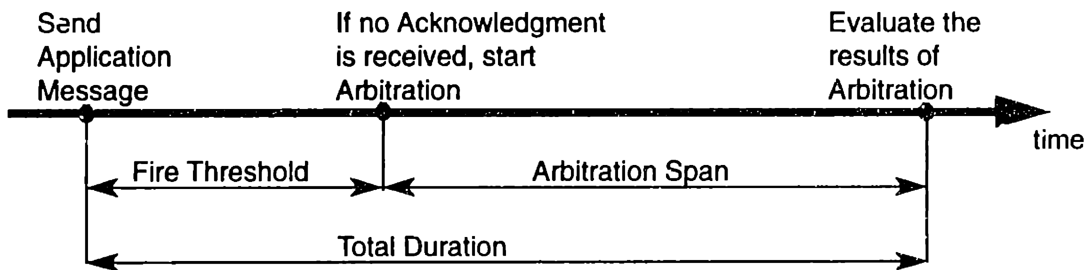

Figure 6.6. The timing parameters of Foreground Failure Detection. Total Duration = Fire Threshold + Arbitration Span.

The following experiments describe the effect each of these parameters has on the performance of Foreground Failure detection.

Fire Threshold

First let us examine the effect of varying the Fire Threshold. In this experiment we measured the number of Arbitration requests (Req_Messages) issued by a single process over the period of about 8 minutes while the system was injected with a total of 150 process and link failures. Failures were introduced one or two at a time. After detecting those failures, system was reset to its original state, and the next single or double failure was injected. During one minute the system ran without having any failures injected into it. We ran the experiment for a range of Fire Thresholds varying from 10 ms. to 280 ms.

Knowing that the message RTT distribution is the approximately the inverse exponent of the RTT, we expected to see that the number of failure detections initiated is very high for small Fire Thresholds. As the Fire Threshold is increased, the number of detections initiated should drop rapidly to approximately the number of failures injected into the system.

As expected (see Figure 6.7), the number of detections initiated declines rapidly, but never actually goes all the way down to the number of failures injected into the system,

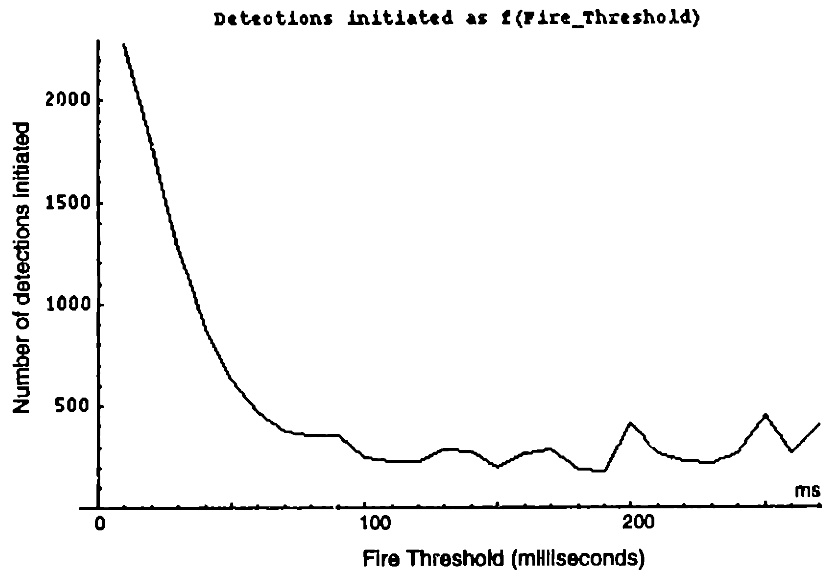

Figure 6.7. Number of initiated Failure Detections as a function of duration of Fire Threshold.

even for the very large fire thresholds. This happens because there are always packets that take a very long time to go through the network. Due to this "noise," increasing the Fire Threshold does not yield a significant reduction in the number of detections initiated past 100 ms.

The value of the Fire Threshold determines the overhead imposed by failure detection. If the Fire Threshold is low, too many failure detections are initiated, thus there is too much failure detection traffic. In the worst case scenario, there will be so much failure detection traffic that the application messages will start getting delayed in the system, thus initiating even more failure detections. Consequently, that system may come to a grinding halt under the load of failure detection traffic.

Therefore, a 100 ms value for Fire threshold is the best one, because it is the minimal latency that provides the highest available accuracy of Failure Detection initiation.

Total Duration

Now that we know the optimal value for Fire Threshold, let us consider the effect Total Duration has on the false positive rate of Failure Detection. In the following experiment we measure the percentage of failures detected as we keep the Fire Threshold at 100 ms. and vary the Total Duration from 110 ms to 330 ms. The system we run in this experiment is identical to the one we ran earlier. Over the period of about 8 minutes, it injects 150 process and link failures into the system and resets the system after each failure is detected.

If an acknowledgment is received before the end of Total Duration of failure detection, the node and the link in question are cleared of any suspicions, otherwise they are declared dead. Therefore when the Total Duration is low, we expect to get a high False positive rate.

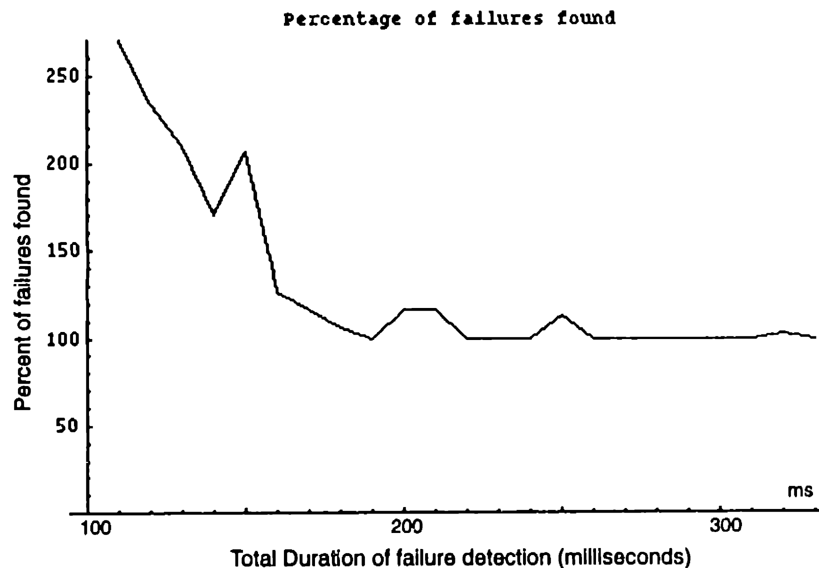

Figure 6.8. Percentage of Failures detected as a function of Total Duration, with Fire threshold set to 100 ms.

From the experiment we find that the false positive rate (node and link combined) found in the system is very high for Total Durations under 200 ms. The total false positive

rate goes down almost to zero for total durations over 260ms. However, this measurement does not differentiate between the node and the link failures.

Arbitration Span

The parameter that affects the accuracy of distinction between the node and the link failures is Arbitration Span. When Arbitration Span is small, the number of link failures interpreted as node failures is high. This happens because the process that initiates failure detection makes a decision about the nature of failure before it receives the results of arbitration. In that case, it defaults to a more conservative conclusion that the process in question, rather than the link to that process, is dead.

In the following experiment we run our usual system with a very large Fire Threshold (750ms) to insure that failure detection is initiated only when there really is a failure in the

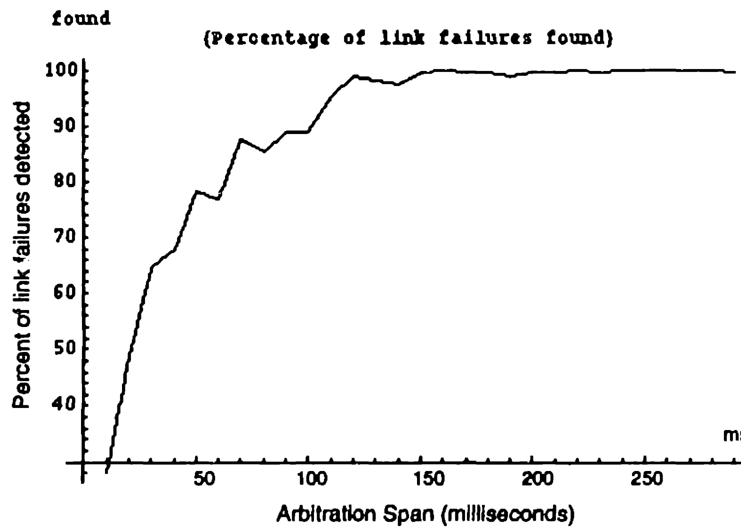

Figure 6.9. Accuracy of link failure detection as a function of Arbitration Span.

system.Therefore, if we vary Arbitration Span under these conditions, it will effect the accuracy of distinction between node and link failures and will have no effect on false

positive rate. As usual, we run the system on the battery of standard tests injecting 150 node and link failures into the system.

As expected, larger arbitration spans provide a better accuracy of link vs. node failure differentiations. The distinction becomes accurate at about 200 ms. Considering that the message round trip time in our system maybe as high as 140ms, this is a very good result for a protocol requiring three synchronous messages.

One reason why this is the case is that the majority of massages have RTT of about 10ms, rather than a 100ms. More importantly, we can show that when the message RTT may increase on a TCP connecting between two processes, it is likely that the other channels at the system will not slow down at the same time. This point is illustrated by the comparing the traces of message RTTs on two different connections during one of the test runs (Figure 6.10). One explanation is that when a number of TCP connections share the

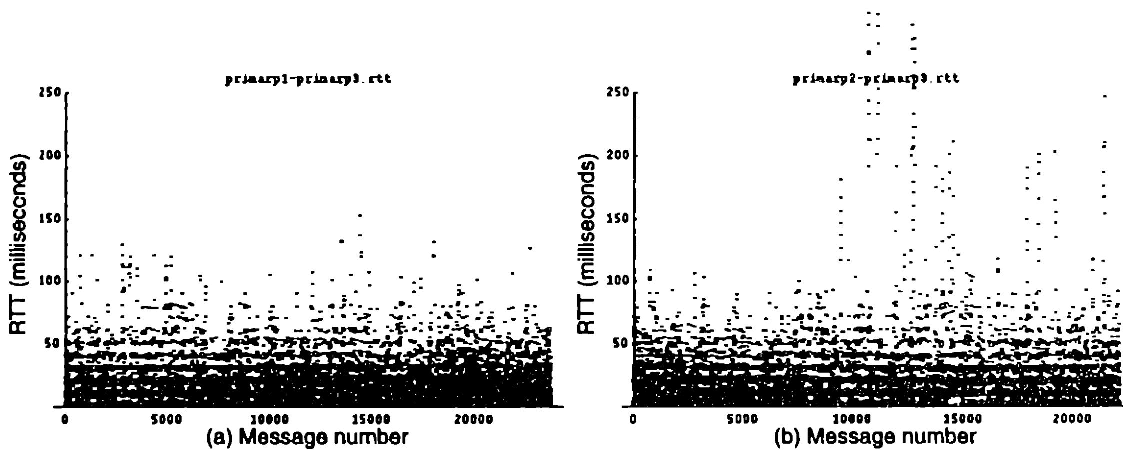

Figure 6.10. Scatter plots of message RTTs on two different connections in the same experiment.Peak latencies do not happen at the same time.

same physical link, exponential back-off on one of the TCP connections frees the physical bandwidth that can be used by the other TCP connections. Consequently, other connec-

tions are able to send more data contained in their mbufs, actually reducing their end-to-end RTT [21].

Performance with limited Total Duration

Finally, let us analyze the behavior of failure detection algorithms in a setting close to the constraints imposed by telecoms. From the previous experiments we know that if failure detection runs for longer than 200ms, the overall false positive rate is quite small. Additionally, accurate differentiation between the node and the link failures requires a Total Duration of 300ms. This is greater than the telecom requirement of 250 ms. However, in this study we are interested in examining the feasibility of the proposed algorithms, rather than building a production solution. We know that if the algorithms work well at 300ms, their performance can be scaled to meet the telecom timing requirements by using better network software (see Section 6.2.1). Let us assume an optimistic scenario that failure detection takes 300ms and that process recovery happens instantaneously.

In this last experiment we set the total duration of failure detection to be 300ms and vary the values of Fire Threshold and Arbitration Span.

When the Arbitration Span is small, there is not enough time to make the decision about the nature of the failure, thus the node failure is assumed. Therefore, at low arbitration spans failure detection is very inefficient at distinguishing between the link and the node failures. When the Arbitration span is large, say more than 250ms, in this experiment Fire Threshold is less than 50ms. This leads to a large number of failure detections being initiated. They generate a lot of traffic that overloads the system and increases the communication latency. As a result, there are many false positives and arbitration requires much more time than normal. Consequently, at high Arbitration Spans, differentiation between the node and the link failures is poor.

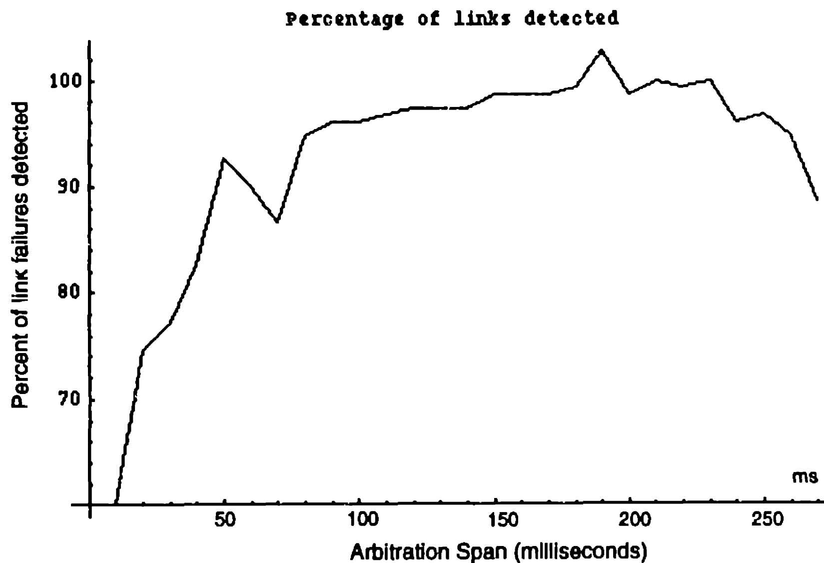

Figure 6. Percentage of link failures detected as a function of Arbitration Span varying from 10ms. to 270ms., while keeping the Total Duration constant at 300ms.

The optimal performance of foreground failure detection under the telco requirements is observed at an Arbitration Span of about 200ms, and a Fire threshold of 100ms. At this point the false positive rate is low and differentiation between the node and the link failures is almost 100% accurate.

6.3 Summary

This chapter presented an experimental system implementing the Foreground and the Background FD algorithms, and the tests performed using this system. The experimental data collected during the tests shows the following. Background FD works correctly, and Foreground FD has an acceptably low false positive rate and reliably distinguishes between the node and the link failures when it runs with Fire Threshold of 100 ms, and Total Duration of 300ms. These timings exceed the telecom requirement of detecting of and recovering from failures in 250ms. However, with an improved network software, or

by simply providing the FD system with access to the TCP acks, the timings of Foreground FD can be brought in line with the telecom requirements.

Chapter 7

Discussion

We have built a failure detector for Somersault distributed system. This failure detector is complete [8], moreover it guarantees a bounded failure detection latency; it is not accurate, but according to the result by Fischer, Lynch and Paterson [14], no failure detector can be accurate. Still, in our experiments, we demonstrated that this failure detector has a low false positive rate and is able to distinguish between the node and link failures in real time.

The contributions of this thesis are twofold. First, we employ a globalized approach to failure detection. It uses multiple failure indications collected from the different parts of the system to increase its' accuracy. Second, we separate failure detector into two functional parts: (1) a Foreground Failure Detector, which is responsible for quick, on-demand failure detection; (2) a Background Failure Detector, which is responsible for finding failures that do not immediately affect the system. Background Failure Detector significantly reduces the probability of multiple failures accumulating in the system, thus decreasing the probability of a total system failure, and increasing its MTTF. Separating failure detection into two distinct mechanisms allows us to address both tasks of doing fast failure detection and increasing long-term the availability of system, with minimal overhead.

Globalized approach is employed in both the Background and the Foreground failure detectors. Multiple failure indications from different sources are used to determine the cause of a failure. Background FD uses temporal distribution of failure suspicions to filter out transient failures and spatial distribution to distinguish between process and link failures. Foreground FD runs under much tighter timing constraints. Thus, it can not exploit

temporal distribution. However, it forces multiple tests to be performed simultaneously throughout the system in order to determine the nature of a suspected failure.

To verify correctness and evaluate performance of failure detection algorithms presented in this thesis, we constructed a distributed test-bed. It allowed us to inject failures into the system and observe the accuracy and latency of their detection. Our measurements showed that Background FD was correct, and accurately found the failures we injected. Experiments showed that Foreground FD was also correct. However, under the strict telecom timing requirements, it did not have enough time to make an accurate decision. Namely, Foreground FD always found a cause of failure, but occasionally it misinterpreted link failures as node faults. Fortunately, under the timing requirements relaxed by only 20% the accuracy of Foreground FD was close to 100%. We believe that the telecom timing requirements can be met by simply improving the performance of underlying protocols.

The ultimate goal of building Somersault's failure detector was to insure system's MTTF of approximately 20 years. Testing for the fulfillment of this requirement is beyond our capabilities. However, in order to explore the effects of failure detection performance on MTTF, we built a mathematical model of availability of Somersault. It assumed independence of failures and an optimal machine configuration. Our model showed that with Background FD in place, performance of components and their reliability dominate reliability of Somersault. In particular, improving network throughput was identified as the key factor for increasing availability of Somersault.

Scalability is an important factor of success of distributed systems. In this thesis, we paid particular attention to scalability of Somersault failure detection. Because Foreground FD requires only a small constant number of messages for each failure suspicion, it scales well. Background FD messaging overhead, however, grows quadratically with

the number of processes in the system. From reliability analysis we know that systems with several hundreds of processes, large enough to make Background FD overhead prohibitively high, can not be built due to component reliability constraints. Therefore, within the domain of realistic Somersault configurations, Background FD is also scalable.

7.1 Future work

The next logical step is replacing TCP that our system uses with some other reliable protocol. TCP in its current implementation seems to introduce a lot of delay into the system. For instance, on the FDDI ring we were using the Estimated Token Rotation Time was well under 8ms. However, it was not uncommon to see massage round trip times of over 100ms. Also, the variance of round trip times that reduced the accuracy of Foreground FD was very high. These behaviors are highly undesirable from the point of FD algorithms. Much of this undesirable behavior can be explained by interactions of TCP back-offs on different connections [21], and by inefficient interactions between the UNIX kernel and TCP [9]. So, an immediate way of improving accuracy and speed of failure detection would be using a different protocol or a different operating system. Perhaps, even a version of TCP without the slow start could improve performance of failure detection in the well controlled LAN environment of our system.

As stated above, using more reliable, faster, higher throughput networks may significantly improve scalability and reliability of Somersault. Implementing highly-available systems using such network technology, for instance ATM, is another promising topic for future research.

Finally, building a more accurate mathematical model of Somersault's reliability will produce more realistic availability estimates and will rectify the scalability limits of the system and performance requirements for failure detection.

References

[1] T. Anderson, et. al., *A Case for NOW (Networks of Workstations)*, UC Berkeley, December 1994. (Paper to appear in IEEE Micro)

[2] J.Bartlet, *A NonStop Kernel*, Proceedings of the Eights symposium on Operating System Principles, Pacific Grove, CA, 1981, pp 22-29.

[3] T. Becker, *Transparent Service Reconfiguration after Node Failures*, International Workshop on Configurable Distributed Systems, London, UK, 1992, pp. 212-223.

[4] K.Birman, T. Joseph, Reliable Communication in the Presence of Failures. ACM Transactions on Computer Systems, Vol. 5, No. 1, February 1987, pp. 47-76.

[5] A.Borg, et.al., *Fault Tolerance Under Unix*, ACM Trans on Computer Systems, Vol. 7, No. 1, February 1989, pp. 1-24.

[6] F.Brooks, *The Mythical Man-Month*, Addison-Wesley, 1982, p20.

[7] S.Bruso, *A Failure detection and Notification Protocol for Distributed Computing Systems*, Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado, May 1985, pp 116-23.

[8] T.Chandra and S.Toueg, *Unreliable Failure Detectors for Asynchronous Systems*, Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, August 1991, pp. 325-40.

[9] D.Clark, V.Jacobson, et.al, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June 1989, pp.23-29.

[10] F.Cristian, *Automatic Reconfiguration in the Presence of Failures*, International Workshop on Configurable Distributed Systems, 1992, pp. 4-17.

[11] F.Cristian, *Understanding Fault-Tolerant Distributed Systems*, Communications of the ACM, Vol. 34, No. 2, February 91.

[12] M. Dahlin, et. al., *Cooperative Caching: Using Remote Client Memory to Improve File System Performance*, First USENIX Symposium on Operating System Design and Implementation, Monterey, CA, November 1994.

[13] E.Elnozahy and W.Zwaenepoel, *Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit*, IEEE Transactions on Computers, Vol. 41, No. 5, May 1992.

[14] M. Fischer, N. Lynch and M. Paterson, *Impossibility of Distributed Consensus with one Faulty Process*, Journal of the ACM, Vol. 32, No. 2, April 1985.

[15] R. Fleming, *Consistent Process View Protocol: a Proposal*, Intelligent Networks Computing Laboratory, Hewlett Packard Laboratories, Bristol, 1994.

[16] R.Fleming, *Note on FT Terminology*, Intelligent Networks Computing Laboratory, Hewlett Packard Laboratories, Bristol, 1994.

[17] J.Gray, D.Sieworek. *High Availability Computer Systems*, Computer, 24, 9 (September, 1991) pp. 39-48.

[18] P. Harry, *The Design of Somersault*, Internal Memo, Intelligent Networks Computing Laboratory, Hewlett Packard Laboratories, Bristol, 1994.

[19] P. Harry and R.Fleming, *Somersault Membership Layers II*, Internal Memo, Intelligent Networks Computing Laboratory, Hewlett Packard Laboratories, Bristol, 1994.

[20] Personal commmunication with P.Harry.

[21] V.Jacobson, *Congestion Avoidance and Control*, in Partridge C., Innovations in Networking, Artech House Inc., Dedhem, MA, 1988, pp. 273-288.

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

[23] S.Leffler, et.al., *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addisson-Wesley, 1989.

[24] B.Liskov. et.al., *Replication in the Harp File System*, Proc. Thirteenth Symposium on Operating System Principles, pp.226-238, Pacific Grove, CA, Oct. 1991.

[25] O.Loques and J.Kramer, *Flexible Fault Tolerance for Distributed Computer Systems*, IEE Proceedings, Vol. 133, pt. E, No. 6, November 1986.

[26] T.Ng, *Design and Implementation of a reliable Distributed Operating System - ROSE*, Proceedings of the 9th Symposium on Reliable Distributed Systems, Huntsville, Alabama, Ocober 1990, pp2-11.

[27] A.Riccardi and K.Birman, *Process Group Approach to Reliable Distributed Computing*, Communications of the ACM, Vol. 36, No. 12, Dec 1993.

[28] R.E.Strom, S.Yemini, *Optimistic Recovery in Distributed Systems*, ACM Trans on Computer Systems, Vol. 3, No.3, August 1985, pp 204-226.

[29] W.R.Stevens, *UNIX Network Programming*, Prentice Hall Software Series, 1990.

[30] A.Thomas, *Failure Detection*, Internal Memo, Intelligent Networks Computing Laboratory, Hewlett Packard Laboratories, Bristol, 1994.