Implementing uniform treatment of hardware and software faults with distributed queues
by Phillip Louis Amicucci

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in
Computer Science
Montana State University
© Copyright by Phillip Louis Amicucci (1993)

Abstract:
This thesis explores the use of replicated queues with distributed execution of recovery blocks (DRB) as an approach for uniform treatment of hardware and software faults. An approach to incorporating the capabilities of DRB’s into a loadsharing multiprocessing scheme using replicated queues is discussed. A simulation aimed at testing the execution efficiency and fault tolerance of the scheme in a realtime environment has been conducted using IBM’s OS/2 2.0’s operating system. The simulated completely connected network was developed and successfully tested on a single 386DX/33 PC. The results indicate the feasibility of achieving tolerance to hardware and software faults in a real-time computer system. The effective use of the schemes distributed execution of recovery blocks with replicated queues deserves serious consideration.
IMPLEMENTING UNIFORM TREATMENT OF
HARDWARE AND SOFTWARE FAULTS
WITH DISTRIBUTED QUEUES

by
Phillip Louis Amicucci

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

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 1993
APPROVAL

of a thesis submitted by

Phillip Louis Amicucci

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

May 17, 1993
Date
Chairperson, Graduate Committee

Approved for the Major Department

May 17, 1993
Date
Head, Major Department

Approved for the College of Graduate Studies

5/21/93
Date
Graduate Dean
STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature

Date 17 May 93
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. FAULT TOLERANCE THROUGH REDUNDANCY</td>
<td>6</td>
</tr>
<tr>
<td>3. MODEL DESCRIPTION</td>
<td>10</td>
</tr>
<tr>
<td>4. MODEL VARIATION</td>
<td>15</td>
</tr>
<tr>
<td>Scope</td>
<td>15</td>
</tr>
<tr>
<td>Communications</td>
<td>16</td>
</tr>
<tr>
<td>Process Failure</td>
<td>17</td>
</tr>
<tr>
<td>5. DISTRIBUTED QUEUES</td>
<td>19</td>
</tr>
<tr>
<td>Queue Fault Tolerance</td>
<td>21</td>
</tr>
<tr>
<td>6. EVALUATION</td>
<td>25</td>
</tr>
<tr>
<td>Conclusion</td>
<td>34</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>36</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Queue States Specifically Guarded Against</td>
<td>23</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Basic Structure of the Distributed Recovery Block (DRB)</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>A Scheme for Distributed Execution of Recovery Blocks with Load Balancing</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Simulation run with no faults, the Baseline</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Primary Try Block Fault</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Delete Fault without Failure</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Insert Fault without Failure</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Delete Fault with Failure</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>Insert Fault with Failure</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>One Process Failure</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>Two Process Failures</td>
<td>33</td>
</tr>
</tbody>
</table>
This thesis explores the use of replicated queues with distributed execution of recovery blocks (DRB) as an approach for uniform treatment of hardware and software faults. An approach to incorporating the capabilities of DRB's into a load-sharing multiprocessing scheme using replicated queues is discussed. A simulation aimed at testing the execution efficiency and fault tolerance of the scheme in a real-time environment has been conducted using IBM's OS/2 2.0's operating system. The simulated completely connected network was developed and successfully tested on a single 386DX/33 PC. The results indicate the feasibility of achieving tolerance to hardware and software faults in a real-time computer system. The effective use of the schemes distributed execution of recovery blocks with replicated queues deserves serious consideration.
CHAPTER 1

INTRODUCTION

The rapid growth in on-line applications is largely responsible for the increased interest in system reliability. There are two approaches to improved reliability: fault avoidance (fault intolerance) and fault tolerance. Fault avoidance uses high-reliability components, conservative design practices, and extensive design reviews to eliminate design flaws [29] to ensure that the running system satisfies all reliability criteria before production begins. But for a typical system, current proof techniques and testing methods cannot guarantee the absence of residual design flaws, bugs, hardware malfunctions, or user errors; hence, pure fault avoidance is not realistic for critical applications.

Fault-tolerant computing [23][24][26][29][32]; however, has the ability to compute in the presence of errors. The increased reliability is achieved by designing the system around three key issues [15]:

1) the ability to detect errors before an intolerable degree of damage is incurred,
2) the ability to discard faulty information arising from the error and to retrieve a valid system state, and
3) the ability to continue, with the expectation that further useful work can be performed.
The concept of fault-tolerant computing has existed for some time. Reliability has concerned users since the early days of the relay and vacuum tube machines which were notoriously unreliable. These early systems generally employed rudimentary fault tolerance techniques such as error-detection circuits and repeated execution to ensure correct results [23]. Since then, hardware reliability has been dramatically improved by the introduction of solid-state devices and integrated circuits.

As hardware became more and more reliable, users began writing larger and more complex applications. These new powerful software systems, running on the desktop computer, have increased society’s overall dependence on computer systems beyond expectations. For all the benefits these modern computer systems offer, they often have an enormous cost associated with their failure. The decision to employ fault-tolerance in a computer system involves a trade off between the cost of failure and the cost of implementing the fault detection and recovery scheme. The need for computer systems to be fault-tolerant seems to have been recognized first by designers of real-time control systems, notably telephone switching systems [25].

Computer controlled electronic switching systems began to appear in central offices of the public telephone network in the U.S. and in France around 1965 [7]. Because of the real-time requirements of electronic switching systems, the computers incorporated in such switching complexes have extremely high reliability constraints.

For example, the reliability objectives of an electronic telephone switching office designed for use in telephone applications are (1) a total system downtime of no more than three minutes per year with a mean time to
repair (MTTR) of four hours; and (2) no more than 0.01 percent of the calls should be lost or handled incorrectly during the system’s operations [29].

Even with these reliability requirements designed into the system it was not immune to failure. In January 1990, large parts of AT&T’s long-distance telephone switching network failed due to a software problem. This mishap reminded us how important computer systems reliability has become and how hard it is to actually achieve.

Protecting systems from unexpected failure situations poses the principal challenge facing reliable fault-tolerant systems. That is, the design specification must correctly anticipate all possible situations; as a result, when a failure occurs, at least one situation was not accounted for causing the failure of the fault-tolerant mechanisms (assuming any existed). Put very simply, all software faults result from design errors [6].

For example, the latest generation of aircraft, both military and civil use ‘fly-by-wire’ technology [22]. ‘Fly-by-wire’ means the controls operated by the pilot are simply input devices attached to computers. That is, the pilot’s controls have no direct mechanical linkage to anything of physical significance (i.e., ailerons, rudder etc). Most computer hardware can be made much more reliable than traditional mechanical systems, which are more subject to wear and tear. Hardware is often much simpler in its design than software; as a result, hardware design faults are relatively rare once full production begins. But software, unlike hardware, does not ‘wear out’ or suffer damage through ill treatment (excluding deliberate alteration);
however, software reliability is much harder to measure and control since bugs (faults) in software are essentially errors in design.

Even with the computerized flight-critical control systems proposed failure probability of about $10^{-10}$ per hour [27], in February 1990, a fly-by-wire A320 Airbus crashed in Bangalore, India. The fact that the A320 employs a fully electronic cockpit with no mechanical backup for control has raised much speculation about the computers role in the crash [23].

The crash of the A320 Airbus and the failure of AT&T's long distance service emphasize the growing need for fault-tolerant computing systems; systems that respond correctly even in the presence of faults or other anomalies.

Under NASA sponsorship, two fault-tolerant computers for flight control systems have been designed; the Fault-Tolerant MicroProcessor (FTMP) by the C.S. Draper Laboratory; and the Software Implemented Fault Tolerance system (SIFT) by Stanford Research Institute (SRI) International.

Both systems design specifications require that the systems must experience less than $10^{-9}$ failures in a ten-hour mission time with an mean time to repair (MTTR) of ten hours. The availability factor is equal to 99.9999999% [29]. A third flight related fault-tolerant system I found personally impressive, is the Jet Propulsion Laboratory Self Testing and Repairing (JPL-STAR) computer. The JPL-STAR's reliability requirements are for 100,000 hours survival with a probability of 0.95, and with a maximum time requirement for recovery of 50 milliseconds [12].
The complexity and sophistication of computer based weapon systems is perhaps the best example of an ultra-reliable computing system. Typically, weapon systems remain dormant for months, activate on a moments notice, and are regularly operated in extremely harsh environments. These systems demand extreme design requirements in reliability, availability, cost, performance, and growth capabilities.

An ideal example of the complexity of weapon systems is the US Army Ballistic Missile Defence (BMD) Program [3][5]. This system has a very tight design requirement of around 30 seconds to detect, track, and destroy an incoming target.

A BMD computer system must be designed to produce extremely high confidence in all system components. It must also possess very high reliability: once up, there must be a high probability that the configuration will stay up long enough to conduct an entire engagement. Computer architectures for BMD must be able to tolerate hardware failures during mission initialization and operational modes. Should one component in the BMD computer architecture fail during an engagement, the remainder of the system should be capable of continuing execution, albeit at a slower rate and with accompanying reduced system effectiveness [3].

In systems designed to meet such an extreme level of dependability, fault tolerance is achieved through modular redundancy and voting from replicated tasks.
CHAPTER 2

FAULT TOLERANCE THROUGH REDUNDANCY

The effective implementation of redundancy techniques is the key to error detection, correction, and recovery from many types of hardware or software malfunctions. Redundancy allows computers to bypass errors so that the final results are correct. The amount and type of redundancy used is determined by a compromise between the cost of failures and the cost of the facilities necessary to cope with them [25]. In general, the stronger a specified failure semantics is, the more expensive and complex it is to implement.

There are four forms of redundancy which can be used to enhance system fault tolerance: redundant hardware, redundant software, redundant data, and redundant information about the system's behavior [25].

Hardware redundancy [12][31] consists of additional circuits that detect and correct errors. There are two types of hardware redundancy: static and dynamic. The static approach uses massive replication of permanent components, circuits, and subsystems; these redundant components remain in use, and in the same fixed relationship, whether or not any errors are detected [12]. The components correct or mask errors to prevent their propagating through the system and this masking occurs
automatically; corrective action is immediate and 'wired in'. Static types of redundancy have been used primarily in military applications that require high reliability for a short duration [29]. Examples of static redundancy include fault masking via coding, and fixed replication with voting as in Triple Modular Redundancy.

Triple Modular Redundancy (TMR) involves the use of three subcomponents (or modules) of identical design, and majority voting circuits which check the module outputs for exact equality. Thus, it is designed to mask the failure of any single module by accepting any output that at least two of the modules agree on.

The TMR organization was adopted in the SIFT computer where each task is executed by three (or more) processors and the results voted on by three subsequent processors. TMR schemes are based on the assumption that failures in the different modules are independent of each other. Thus, a (hardware or software) design fault which caused all three modules to produce identical but incorrect outputs would not be masked.

Dynamic redundancy requires additional parts or subsystems to serve as spares. Dynamic redundancy, also referred to as selective redundancy...in response to an error, the faulty unit is automatically (or manually) replaced with a good unit to correct the trouble [29]. A replaceable hardware unit means a physical unit of failure, that is, a unit which fails independently of other units which can be removed without affecting other units, and can be added to a system to augment its performance, capacity or availability. An example of hardware redundancy is the
Boeing 757/767 flight-control system. It comprises 80 distinct functional microprocessors, 300 when we account for redundancy [31].

If the hardware of a computer were fault free, some degree of software redundancy would still be necessary to ensure reliability [29]. Redundant software protects the system from both hardware and software faults at the macro and micro levels.

Again, there are two types of software redundancy [4][6][14][30]: static and dynamic. Static software redundancy employs extensive replication of algorithms. The replicated algorithm is, of course, not merely a copy of the main algorithm. Rather it is of independent design written to the same specifications. The idea behind an independent design is the hope that the replicated copy can cope with the circumstances that caused the main component to fail. There are two widely excepted techniques for tolerating software design faults. First, N-version programming (NVP), a static software redundancy scheme, and second, the recovery block (RB) scheme, a dynamic approach.

N-version programming [2][16][28][33] is an example of static software redundancy. Multiple versions of a program are executed and their outputs are compared. If the output is not identical, it is assumed that the majority is correct. This method depends, for its reliability, on the assumption that programs that have been developed independently will fail independently. This majority vote scheme by software provides the means for immediate error detection and correction since the
output from the erroneous algorithm is ignored; this is a software equivalent to the triple modular redundant (TMR) system [29].

The recovery block scheme [13][17][19][20][21] uses multiple versions of software modules too. There is a subtle difference between the recovery blocks use of 'multiple versions' (called try blocks) and NVP discussed above. In the recovery block scheme an alternate try block, say $M + 1$ where $M \geq 1$, is executed only after try block $M$ has failed. In NVP, all versions of the program are executed and the majority vote is assumed correct. It is possible however, that NVP could be used within try blocks to further improve each try blocks reliability by using an acceptance test that fails if the output from all 'N-versions' do not agree.
CHAPTER 3

MODEL DESCRIPTION

In an article by Kim and Welch [1], a scheme is developed which applies to both NVP and recovery blocks for uniform treatment of hardware and software faults in real-time applications. In this article, Kim and Welch describe the use of distributed recovery blocks (DRB) to improve the reliability in a load-sharing, real-time scheme. An improvement to this scheme is the focus of the remainder of this document.

Figure 1 depicts [1]'s DRB scheme which is based on replication of the entire recovery block of the computing station into the primary and backup nodes. The key points of this scheme are: 1) the acceptance tests, both logic and time tests are replicated in both the primary and backup nodes, and 2) the try blocks are fully replicated in both nodes. The roles of the two try blocks are not the same in the two nodes, it is important that the current backup node always use try blocks in the reverse order relative to the primary node. For example: if the primary node (say X) uses try block A as the primary try block then the backup node (say Y) should use try block B. With this arrangement, the same input data is processed by two different try
blocks; as a result, a degree of software fault-tolerance is achieved while the primary and backup nodes form the hardware fault-tolerance.

Under the DRB scheme, both hardware and software faults are treated in a uniform manner. Yet the scheme does not require any additional software design. It is just an efficient way of utilizing both hardware and software resources to maximally prolong the lifetime of a computing station. The recovery time is minimal since maximum concurrency is exploited in the redundant try block execution. In fact, the effect of forward recovery is achieved [1].

![Diagram of the Distributed Recovery Block (DRB)](image)

**Figure 1.** Basic Structure of the Distributed Recovery Block (DRB).
In [1]'s scheme, load balancing is achieved by the use of five queues that allow nodes to dynamically select tasks. Figure 2 depicts Kim and Welch's load balancing model, it is this specific model that will be altered, but first it is necessary to describe Kim and Welch's approach.

**Figure 2.** Scheme for Distributed Execution of Recovery Blocks with Load Balancing.
Kim and Welch's model uses five queues: the try results queue (TRQ), the input data queue (IDQ), the arrival time record queue (ATRQ), the retry data queue (RDQ), and the validated results queue (VRQ). As nodes find themselves idle and looking for work, the nodes access a queue and perform the tasks associated with that queue.

The basic flow of data is as follows: input data (say D) is placed into IDQ with the arrival time and the maximum expected processing time placed into ATRQ. An idle node then finds D and processes it using its primary try block and puts the result (say D1) into TRQ. Again, any idle node can access this queue and then apply an acceptance test to D1. An acceptance test is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. Therefore, the acceptance test can be viewed as a combination of both the logic acceptance test and the time acceptance test [1].

If the acceptance test is failed, then the node moves D and its arrival time into RDQ. An idle node discovers D in RDQ and now uses the alternate try block to process the input and places the results (say D2) into TRQ again. The next idle node then runs the acceptance test on D2, this time the test is passed and D2 is placed into VRQ to wait for the successor computing stations retrieval. A try not completed within the time limit fails the acceptance test and is discarded.

When Kim and Welch describe their load balancing queue arrangement, a serious flaw is evident. Kim and Welch use queues located in the global storage area
[1]. Using a common global storage area leaves the entire model with a single point of vulnerability...the queues.

This thesis implements, with minor variations, the distributed execution of recovery blocks with load balancing described by Kim and Welch [1]. The variations include: 1) using a distributed queuing scheme which places a copy of each queue in each node, 2) using only four queues instead of five, 3) instead of using N-nodes, the model will be based on TMR, and 4) the entire model will be simulated on a single multitasking node, running under IBM’s OS/2.0.

The rest of this thesis is organized as follows: 1) the motivation behind altering Kim and Welch's model and the scope of the alteration, 2) specifics regarding the implementation, and 3) an evaluation of the finished model.
CHAPTER 4

MODEL VARIATION

Scope

As stated earlier, Kim and Welch's [1] use of a common global storage area for their queues leaves the entire model with a single point of vulnerability. It can be successfully argued that the queues were not the intent of Kim and Welch's model but rather the distributed execution of the recovery blocks. Kim and Welch simply tacked on whatever was necessary to achieve their primary goal. The purpose of this model is to simulate Kim and Welch's basic scheme using replicated queues instead of a queue in a global storage area.

Like Kim and Welch, the intent is not to distinguish between hardware faults and software faults, that is costly even if it is logically feasible. Also, introducing different treatments for all possible different faults can quickly add up to an intolerable system complexity [1]. But rather to show that the approach described in [1], if expanded on, can be adapted to a distributed queuing scheme; thereby increasing the model's overall fault-tolerance.
Further, the intent of this uniform detection, treatment and recovery scheme is not, and should not be, to 'correct' the fault, but to prevent its propagation. The goal is to detect, isolate, and recover from specific hardware and software faults.

Communications

It is assumed an interprocess communications protocol exists. That is, the underlying network will handle routine communication errors: lost, duplicate, and garbled messages [18]. After the assumed network protocol successfully delivers a message, the model then takes responsibility for the handling of the messages.

This model has been tested with four separate processes, three being server processes and the fourth being the client process. Each server process creates a named pipe and listens to that pipe for the client. The client initiates the connection with each server process.

As each connection is established, the server process generates a thread to listen to that named pipe instance. When all processes are up and connected, a simulated completely connected network exists. It is assumed this simulation can be initialized; if not, the entire network shuts down.

After the network has completed its initialization, each process has multiple threads of which all but one, thread-1, are dedicated to listening to the named pipes. Each thread listens to one specific named pipe and handles all messages received on that named pipe instance. These listening threads will be referred to as thread-n (n > 1).
Messages come in two flavors: a command originating at the other end of the named pipe, or a response to a command returning to thread-1. Since only thread-1 originates messages, thread-n uses the Process Identification (PID) contained in the message header to distinguish between a command message and a response message.

If the message is a command, thread-n handles the entire transaction from beginning to end; thread-1 does not get involved with requests from other processes. If the message is a response, thread-n stores this message in a queue, posts an event semaphore, then returns to continue listening to its assigned named pipe.

When thread-1 generates and sends a message to another process it blocks on an event semaphore until either a response is received or a time limit is reached. After thread-n posts the event semaphore thread-1 is unblocked, retrieves the message from the queue, and proceeds according to the response received. If thread-1 times-out while blocked on that event semaphore the destination process is assumed to have failed.

**Process Failure**

An actual process failure is immediately recognized by thread-n. If a process fails a broken named pipe error is generated by OS/2 2.0. Since the model simulates a completely connected network, every process in the model receives the broken pipe error code. Thread-n then updates the site identifier lattice, which thread-1 uses as a mailing list, then thread-n terminates.
It is important to recognize that thread-1 does not update the site identifier lattice. If thread-1 has assumed a process failure, that assumption only lasts until the next time thread-1 needs to send a message to that process. Only when the named pipe is broken, is the site identifier lattice altered by thread-n prior to thread-n’s termination.

No attempt is ever made to recover or restart a failed process. The intent of a recovery algorithm is to restore the computing system to a consistent global state, recover lost data, rejoin the failed processor to the processor pool, and continue useful work [8][9][10][11]. The complexity of such an attempt goes well beyond the scope of this model. Therefore, if a process fails, this model makes no attempt to rejoin the failed processor to the processor pool.
CHAPTER 5

DISTRIBUTED QUEUES

Each process has its own copy of each of the four queues. For any process to remove or insert data from/to any queue it must put its own lock on each replicated queue header. After all the locks have been put in place, the process then commits, only then does the process receive or insert any data.

This simulations locking scheme is totally dependent on time, either the systems clock time or the data time. In the following paragraphs the data time refers to the clients system clock time when the client generated the data. The lock time is the system clock time when a queue lock was set.

All access to a queue is done through the queue header. The following queue locking rules were designed to avoid deadlocks and insure mutually exclusive access.

Locking rules:

1) a process always attempts the first lock of any lock cycle on a local queue replica,
2) a process cannot take-over a lock in a local queue replica,
3) a process will take-over a lock if its data time is less than the data time of the current lock owner and (2) holds,
4) a process will take-over any lock that exceeds an established maximum age limit,
5) if a process fails any lock attempt, that process stops attempting locks and waits for one of three events to occur: a) the desired local queue
replica header is unlocked, b) a process fails or c) the maximum allowable age of a lock is exceeded, and

6) for processes that wish to delete from a queue, the time used for lock priorities is the system clock time of the process when it originally began the current locking/commit cycle.

These rules allow a process that achieves all its locks the assurance that it owns all the locks.

For example, assume there are two processes, A and B competing for the same queue. Both processes have achieved their local locks and are now attempting to obtain external queue locks. At some point the two processes must find their competitor’s locks. The process with the earliest time stamp will take-over the lock. Let A have the earlier time stamp, then A will take-over B’s lock. B on the other hand, will fail in its attempt to take-over A’s lock and will block on an event semaphore described in rule 5.

After B has been released, B will begin the locking/commit cycle from the beginning with one exception, B will not update its time stamp. By retaining a time stamp until a successful commit can be achieved, B becomes more and more powerful with respect its ability to take-over a lock.

There is a subtle but important difference between the time stamp used when attempting an insert lock compared to a delete lock. When inserting, the time stamp used to obtain the lock is the data time. For deletion locks, the time stamp is the system time of the process when the request was originated.
Queue Fault Tolerance

This model specifically guards against three queue faults: the disappearance of a queue due to a process failure, a queue delete fault, and a queue insert fault. Recovering from a process failure has been discussed; however, a process failure can cause a queue delete fault or an insert fault.

A process failure is not the only source of an insert/delete fault. A process may also cause this queue inconsistency by failing to meet commit cycle time limitations. For example, let \( m \) be the number of replicated queues that are expecting a commit from process A. Further, let \( n \) represent the number of commits that A has successfully accomplished. If \( 0 < n < m \), a delete/insert fault has occurred and an inconsistent queue state now exists.

An inconsistent queue state is a replicated queue that has been divided into two subsets; one subset contains all the copies which received a correct delete/insert while the other subset did not. Recovery from the delete/insert fault involves the restoration of the replicated queues to a consistent state.

The delete/insert fault is not actually detected, only the inconsistent queue state left by the fault. For a delete fault the inconsistency is discovered and corrected during the next locking phase of a delete request. An insert fault causes an inconsistency at the bottom of the queue and is not discovered until a process attempts to delete the inconsistent queue elements from the top of the queue.

The algorithm that corrects an inconsistent queue state is totally dependent on the assumption that faults do not occur simultaneously to the same queue. That is, a
delete fault for any given queue must be repaired before another delete fault occurs to that queue. Since an insert fault takes time to arrive at the top of the queue, the rules are a bit different. If an insert fault occurs on a specific queue, that queue must not experience another insert fault until a successful insert has been achieved; unfortunately, there is no way of guaranteeing this and with this model, there is no algorithm to detect a double insert/delete fault (perhaps a subject worthy of further study).

Further, the appearance of a double delete fault must also be avoided. For example, assume an insert fault occurs, the inconsistent queue elements will not be discovered until those elements reach the top of their queue. If a delete fault occurred at the same time those elements involved with the insert fault arrived at the queue top, a fault resembling a double delete fault may occur. The realignment algorithm may survive a double delete fault, or the appearance of one, but this model was not designed nor tested for such an event; the algorithms reaction to this queue state is unpredictable.

Each time a delete is requested, the requesting process receives the time stamp of the top two elements from the requested queue replica. After completing the locking phase, the requesting process will have built a two dimensional array which contains the time stamp of the first two elements of each queue replica in the network.

Table 1 represents the two dimensional array if a delete/insert fault had occurred and in the case of the insert fault the fault has arrived at the top of the
queue. In Table 1 it is assumed that an odd number of server processes are still available. If not, a majority vote may require all processes to agree or the data will be considered invalid and discarded.

Table 1. Queue States Specifically Guarded Against.

<table>
<thead>
<tr>
<th>Majority of commits achieved prior to fault</th>
<th>Delete Fault</th>
<th>Insert Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>x y y</td>
<td>x x y</td>
<td>y y z</td>
</tr>
<tr>
<td>y z z</td>
<td></td>
<td>y y z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minority of commits achieved prior to fault</th>
<th>Delete Fault</th>
<th>Insert Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x y</td>
<td>x x y</td>
<td>y z z</td>
</tr>
<tr>
<td>y y z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

By examining these elements, the requesting process can determine if a fault has occurred during a previous delete/insert cycle. The algorithm assumes that at least one of the queue elements exists in all three queue replicas. Hence, the requirement that the queue state not resemble a double fault.

By selecting and removing the head element(s) that are not found in the second row, the queues are realigned after a previous delete/insert fault. If these newly removed elements form a majority, the data is considered valid and is processed normally. If the data does not make a majority, the data is considered invalid and discarded.

Lee, Chen, and Holdman [18] remind the reader that no algorithm can survive all possible failures, only those failures the algorithm is designed to survive. This
scheme does not overcome all possible faults; however, it is designed to survive some
making it better than [1]'s queuing scheme.
CHAPTER 6

EVALUATION

The following evaluation used three server processes forming a completely connected network with an additional process, the client, acting as the predecessor computing station and the successor computing station (see Figure 2).

Each test was run repeatedly and some test results were slower or faster than others. In some instances a failed process went unnoticed, while the same scenario run a second time would result in a relative disaster. A relative disaster is caused by unfortunate timing; such as, every process in the network waiting for a response from the failed process. The simulations that where chosen to plot reflect the most common results.

This simulations predecessor computing station or client process generates data and inserts this data into the IDQ as a job for the server processes to process. A job occupies one queue element and consists of an array of integers that need to be sorted. Each job has a fixed size of fifty integers and the client process generates and submits, to the IDQ, a total of fifty jobs. The point behind this configuration is to maximize the process interactivity and queue activities.
Figure 3 shows the simulation running with no faults occurring. Along the y-axis is the accumulated time a job from the client generated the data until the job passes the acceptance test and is routed to the Validated Results Queue (VRQ). The x-axis is the chronological order of arrivals into the VRQ. The plotted line itself reflects the accumulated time of each job. A distortion to this line requires a simulation slow down; a single job suffering a set back is not visually evident in these plots.

The plot of Figure 3 is the foundation to every graph in this evaluation, its purpose is to enhance the visual comparisons between a fault free environment and those that contain faults. In subsequent graphs, this fault free plot is referred to as
the TMR BaseLine and the faulted plot is called the fault line. Further, in the following discussion the term ‘dropped lock’ refers to a queue lock that has not been, for one reason or another, properly removed.

Figure 4 displays the model’s reaction to a software fault in the primary try block. In the graph, the effects of a single primary try block fault is hardly noticeable. The failure of the primary try block caused one job to be delayed for reprocessing by the backup try block. The additional overhead of routing and execution by the backup try block causes a slight rise in the fault line after the fault occurs. A second penalty paid is reflected in the ordering of the jobs coming into the
VRQ. In this evaluation a rerouted job enters the VRQ, at most, two jobs behind its original ordering.

Figures 5 and 6 show the effects of a software fault introduced during the commit phase of a delete and insert operation, respectively. There is an obvious time penalty suffered during an delete/insert fault without a process failure that is not experienced during a process failure (Figures 7 & 8). The abrupt rise in the fault line is the time required before a lock 'times out'.

![Graph showing VRQ Arrival Order](image)

**Figure 5.** Delete Fault at Job 17 without Failure.

The locking scheme described previously has the effect of prioritizing who obtains locks. Chronologically speaking, the older the time stamp of a lock requester,
the greater the ability to take-over existing locks. It is generally safe to say that once a lock is achieved, the time stamp of that lock is the oldest in the network. If that lock is not properly cleared (dropped), no process on the network can take-over that lock; hence, the abrupt rise in the fault line. All the processes in the network have younger time stamps than the dropped lock; therefore the lock must time out before any process can continue. The simulations overall delay distorts the fault line since the fault affects the accumulated time of all the jobs still present in the queues.

![Accumulated Time vs VRQ Arrival Order](image)

**Figure 6.** Insert Fault at Job 18 without Failure.

Figures 7 and 8 show the effects of a hardware failure introduced during the commit phase of a delete and insert operation, respectively. The hardware failure in
this scenario is a process crashing. Immediately after introduction of the fault, the fault line raises above the BaseLine. This rise is due to additional overhead brought on by the queue realignment algorithm and the process failure recovery algorithms, but notice afterwards the fault line dips and crosses under the TMR BaseLine.

The improved performance by the surviving processes is a result of reduced overhead during subsequent lock/commit cycles. An additional improvement to simulations performance was triggered by the process failure itself. In figures 5 and 6 all the processes were blocking on the dropped lock causing a very expensive delay; however in the case of a failed process, thread-n posts an event semaphore and releases all blocked processes immediately.

Figure 7. Delete Fault around Job 17 caused by Failure.
The apparent superior performance during the insert fault (Figure 8) is caused by two factors working in favor of an insert. The fault introduced for this particular test has the affect of a double fault; the queue inconsistency fault and the process failure. When this particular fault affects the top of the queue, both the queue inconsistency and the process failure occur simultaneously. With respect to an insert fault, the process failure is dealt with prior to the queue inconsistency. The queue inconsistency is not even detected until the inconsistent elements reach the top of the queue.

**Figure 8.** Insert Fault around Job 17 caused by Failure.
Another factor that improves the recovery time of the insert fault is a priority scheme that favors inserts relative to deletes. That is, an insert request during the lock/commit cycle runs at a higher priority than delete requests. The combination of the fault separation and the higher priorities result in an overall simulation delay less than that of a similar fault occurring during a delete operation.

The spike at the end in Figure 8 is due to the client. The client routinely checks the VRQ for any jobs, but will sleep when it finds the VRQ empty. If the client wakes to find jobs waiting in the VRQ, the client will remove jobs from the VRQ until the queue is once again empty, or until the CPU performs a context switch. In this case, the client was waiting for a response from the failed process and time out on an event semaphore. After the client timed out, it found several jobs queued up in the VRQ. The height of the spike is the amount of time required to empty the VRQ or the amount of time the CPU grants the client process.

Figure 9 displays the model’s reaction to a process failure that does not result in a queue inconsistency. The only difference between this graph and that of Figure 7, is the missing ‘penalty’ caused by the queue realignment algorithm.

Figure 10 also displays the model’s reaction to process failure that does not cause a queue inconsistency but with one difference, two processes fail; as close to simultaneously as can be coded. The likelihood that the surviving process was not attempting to communicate with one of the other processes in a TMR configuration is unlikely. The events causing this reaction have been discussed in one way or another in the previous scenarios and will not be rehashed here.
Figure 9. One Process Failure around Job 17.

Figure 10. Two Process Failures around Job 17.
Conclusion

This paper introduced a modification to Kim and Welch’s [1] model with the goal of increasing the original model’s reliability by replacing the global storage area queuing scheme with replicated queues. By adding queue redundancy the system will continue providing service in spite of the presence of some queue faults.

The queuing scheme presented above was specifically designed to accommodate small, short lived queue elements. Further development of this model with larger, longer lived queue elements would perhaps be better suited for the DRISP [18] queuing scheme. The DRISP scheme is considerably more complicated but offers better reliability by actually rebuilding a damaged queue from any undamaged queue replica.

There has been extensive work done in the field of fault tolerance, both hardware and software fault tolerance. Perhaps we are at a threshold where much of the divergent work can be consolidated and synthesized into a few highly modular architectures. Modular approaches could provide good fault coverage and allow cost-effective system configuration in a wide range of applications [24].
REFERENCES


(Interlibrary loan from UCLA)

(Univ of Washington Libraries)

(Oregon State Univ. @ Corvallis)

(Oregon State Univ. @ Corvallis)

(Oregon State Univ. @ Corvallis)


TK7885.A1I24

QA76.I15

TK7885.A1I2

TK7885.A1I24

QA76.I15

QA76A3 v.26

TK7885.A1I2

TK7885.A1I24

QA76.A76