Computer Systems Research at Rice University

Annual Report
1988–1989

John Bennett
Rick Bubenik
John B. Carter
Elmootazbellah Nabil Elnozahy
Jerry Fowler
David B. Johnson
Pete Keleher
Mark Mazina
Willy Zwaenepoel

This work was supported in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, by the Office of Naval Research under contract ONR N00014-89-K-0140, and by IBM Corporation under Research Agreement No. 14040016.

Contents

Optimistic Make

Rick Bubenik
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

This paper has been submitted to and is under revision for publication in *IEEE Transactions on Computers.*

An earlier version appeared in
Proceedings of 1989 ACM SIGMETRICS and PERFORMANCE '89.

Abstract

Optimistic make is a version of the *make* program that begins execution of the commands needed to update *makefile* targets before the user issues the *make* request. Outputs of these *optimistic computations* (such as file or screen updates) are concealed until the request is issued. If the inputs read by the optimistic computations have not been changed by the time of the *make* request, the results of the optimistic computations are used, leading to improved response time. Otherwise, the necessary computations are reexecuted.

We introduce the notion of *encapsulations* as the basic construct used to support optimistic *make*, and we describe the implementation of optimistic *make* in the V-System on a collection of SUN workstations. Statistics measured from this implementation are used to synthesize a workload for a discrete-event simulation, and to validate the simulation's results. The simulation shows a speedup distribution over pessimistic *make* with a median of 1.72 and a mean of 8.28. The speedup distribution is strongly dependent on the ratio between the target out-of-date times and the command execution times. With faster machines the median of the speedup distribution grows to 5.1, and then decreases again. Given the large idle times observed in many workstation environments, the extra machine resources used by optimistic *make* are well within the limit of available resources.

This research was supported in part by the National Science Foundation under Grants CDA-8619893 and CCR-8716914, and by IBM Corporation under Research Agreement No. 16140046.

1 Introduction

Make is a tool used primarily in software development environments for creating up-to-date executable programs from their source files [6]. Using a *makefile*, the user specifies a number of *targets*, the *sources* they depend on, and the commands necessary to construct the targets from the sources. A target is said to be *out-of-date* if one of its sources has a later timestamp than the target. When the user types *make*, out-of-date targets are reconstructed according to the *makefile*. If some of the commands are independent, they may be executed in parallel on separate machines.

Optimistic *make* is identical in functionality to *make*. However, unlike the conventional *pessimistic* implementation of *make*, it monitors the file system for out-of-date targets, executes the commands necessary to bring the targets up-to-date *before* the *make* request is issued, and conceals the outputs of the optimistically executed commands until the user types *make*. If the inputs read by the optimistic commands remain unchanged until the *make* request is issued, these optimistic results are used immediately. Otherwise, the necessary commands are reexecuted.

The operational differences between optimistic *make* and pessimistic *make*, and the potential performance benefits of optimistic *make* are shown in Figure 1. The top portion of the figure depicts a pessimistic distributed *make*, whereby the user edits and saves a number of files, and then issues a *make* request, at which time the commands necessary to bring the targets up-to-date are executed. The bottom part of Figure 1 depicts the operation of optimistic distributed *make*. Commands are started as soon as files are saved, when targets become out-of-date. The response time for the *make* request is significantly improved since most command execution occurs before the request is issued.

The outline of the rest of this paper is as follows. Section 2 discusses the notion of *encapsulations*, the primary mechanism used to support optimistic *make*. Section 3 discusses the implementation of encapsulations and optimistic *make* in the V-System. Section 4 presents the statistics collected

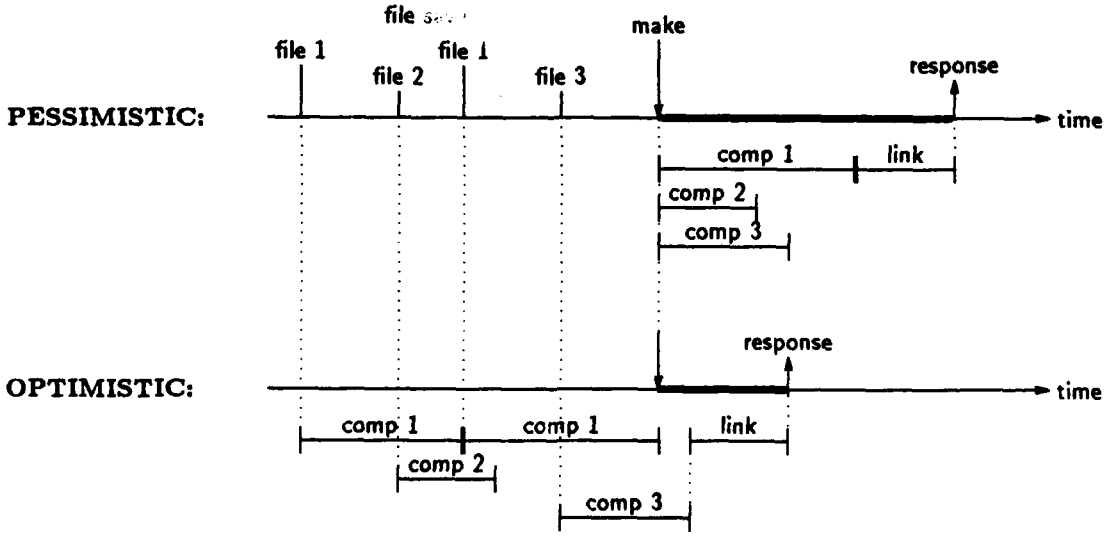

Figure 1 Optimistic vs. Pessimistic Distributed *Make*.

from our implementations of both pessimistic *make* and optimistic *make*. Section 5 describes the simulation model used to further evaluate the performance of optimistic *make*. Results from this simulation are presented in Section 6. Related work is covered in Section 7. Finally, conclusions are drawn and further work is discussed in Section 8.

2 Encapsulations

2.1 Definition

An *encapsulation* is a computation whose outputs are concealed until the computation is *mandated*. Once mandated, the outputs are made visible in an order consistent with the order in which they were produced during the execution of the encapsulation. The following three operations are defined on encapsulations:

eid = CreateEncapsulation() Create an encapsulation with unique identifier *eid*. Output produced by the encapsulation is not visible outside the encapsulation until it is mandated, with one exception: it is possible to allow an encapsulation to read the outputs of one or more *input encapsulations* by specifying these encapsulations as arguments to the **CreateEncapsulation()** call. The newly created encapsulation is then said to be *dependent* on its input encapsulations.

result = MandateEncapsulation(eid) If the inputs read by the encapsulation are unchanged, then reveal all outputs produced so far, do not conceal further output, and return *success*. Otherwise, abort the encapsulation and return *failure*.

AbortEncapsulation(eid) Abort the encapsulation and discard its concealed output.

Encapsulations are superficially similar to atomic transactions in that both mechanisms hide operations until a later time (commit time for atomic transactions, mandate time for encapsulations). However, the semantics of encapsulations differ considerably from those of transactions. Encapsulations can be mandated before the concealed computation completes, allowing an encapsulation to be converted into a normal computation at any point during execution. When an encapsulation is mandated, output is made visible in steps rather than atomically. This simplifies the implementation by avoiding atomicity concerns. Encapsulations may be destroyed at any time, even while concealed output is being made visible, allowing the user to abort unwanted computations before the remaining unwanted output has appeared.

2.2 Optimistic Make and Encapsulations

The optimistic *make* program reads an unmodified *makefile* and monitors the file system for modifications to the source files on which the *makefile* targets depend. File system monitoring is done efficiently by requesting notification from the file server when any file in a specified directory is modified. This results in shorter notification times and less overhead on the file server than polling,

while keeping the amount of state to be maintained at the file server for this purpose small. When optimistic *make* sees a target in the *makefile* that is out-of-date, it starts an encapsulation to update that target. If two (or more) dependent computations are necessary to update a target (for instance, a compilation and a linkage), the first computation is started as an encapsulation eid_1 without input encapsulations, and when it finishes, the second computation is started as an encapsulation eid_2 with the first encapsulation eid_1 as an input encapsulation. This allows the linker to read the output of the compiler. If a source file changes after an encapsulation has been started, the corresponding encapsulation is aborted, and a new one is started. If any encapsulation in a sequence of dependent encapsulations is aborted, all subsequent encapsulations in the sequence are also aborted. When more than one independent encapsulation is necessary to update a target, the independent encapsulations are started concurrently on separate machines (resources permitting). If an encapsulation is not mandated within a certain timeout interval, the encapsulation is automatically aborted to release resources.

3 Implementation

This section describes an implementation of encapsulations and optimistic *make* in the V-System [3]. The V-System follows the conventional client-server model of user programs executing as client processes and accessing most operating system services by sending messages to server processes. The V kernel provides efficient, location-independent message passing.

Encapsulations are transparent to the client programs. The same client program can be run either as a normal computation or as an encapsulation, with no need for recompilation or relinking. An *encapsulation server* process provides most of the support for encapsulations. Several encapsulation servers may be running at the same time, but a particular encapsulation and all its dependent encapsulations must be handled by the same encapsulation server. Minor modifications to the kernel and to some of the servers are also required. However, not all servers need to be modified.

3.1 Kernel Support

Two new fields are added to each kernel process descriptor in order to support encapsulations: the *eid* field, which contains the encapsulation identifier, and the *encapsulation flag* which indicates whether this process supports encapsulations. The *eid* field is zero, by default, for processes that do not run as encapsulations. A normal computation is converted into an encapsulation by instructing the kernel to set the *eid* field to a specified non-zero value for all processes of the computation. The *eid* value is also inherited by all processes created by the encapsulation. Servers that support encapsulations instruct the kernel to set the *encapsulation flag* field in their process descriptor.

All messages are tagged with the value of *eid* field of the sender. The kernel delivers messages sent by an encapsulation (that is, messages with a non-zero *eid* tag) only to other processes in the same encapsulation or to server processes who have indicated that they support encapsulations. All other messages sent by encapsulations are blocked, typically blocking the progress of the sending

process as well. This allows us to run encapsulations in environments where not all servers support encapsulations, accommodating servers whose code cannot be modified. Although optimistic computation cannot proceed if the encapsulation communicates with one of these servers, correctness is preserved.

In total, the kernel modifications for encapsulation support consist of an additional 65 bits in the process descriptor, plus approximately 120 lines of C-language code.

3.2 Running an Encapsulation

The create, mandate, and abort encapsulation requests are issued by optimistic *make* and serviced by an *encapsulation server* process. The encapsulation server allocates a unique *eid* identifier for each encapsulation, and keeps track of all input and output operations performed by an encapsulation. Servers that support encapsulations inform the encapsulation server of the input and output operations performed by an encapsulation in a server-specific manner. These servers know that a message comes from an encapsulation by checking the *eid* tag of the message.

In our current implementation, both the file server and the terminal server support encapsulations. When an encapsulation opens a file for read, the file's timestamp is recorded with the encapsulation server. When an encapsulation opens a file for write, the request is first recorded with the encapsulation server, then a *hidden file* is created, and all subsequent writes are redirected to that file. Hidden files do not appear in the file system directory structure and are only accessible through low level identifiers. Furthermore, the encapsulation server maintains a hidden file system directory tree for each encapsulation, recording the modifications made by that encapsulation to the directory structure. The hidden directory tree is also used to record the mapping between the names of files modified by the encapsulation and the low level identifiers of the corresponding hidden files. When an encapsulation writes to the terminal server, the data to be written is recorded with the encapsulation server. Any other operation sent to the terminal server (including a read) is blocked.

In summary, each server records the operations of an encapsulation in a server-specific manner with the encapsulation server. This allows us to take advantage of the semantics or common usage patterns of certain servers. For instance, the file server only records opens with the encapsulation server, and does not need to record individual reads and writes, an important optimization in our environment where multiple reads and writes are typically performed on each open file.

The encapsulation server transfers information between encapsulations during a create encapsulation request if input encapsulations are specified. When a single input encapsulation is specified, the hidden directory tree containing the modifications of the input encapsulation is inherited by the new encapsulation. When more than one input encapsulation is specified, the hidden directory trees of each input encapsulation are first merged, then passed on to the new encapsulation.

3.3 Mandating an Encapsulation

When an encapsulation is mandated, the encapsulation server inquires with the relevant servers (in our implementation, with the file server) whether the timestamps of the inputs that the encapsulation has read have remained unchanged. If so, it instructs the servers to make visible the outputs performed by the encapsulation in the order they were recorded. Servers *synchronously* record the output operations of encapsulations with the encapsulation server. Hence, the order in which the outputs are recorded, and thus made visible, is a serialization of the order in which they were created. After all outputs have been made visible, if the encapsulation is still running, the encapsulation server zeroes the *eid* field of the encapsulated process and all its descendents, converting the encapsulation into a normal computation. As a result, blocked messages to servers not supporting encapsulations are now delivered.

3.4 Encapsulation Performance

The overhead of executing an encapsulation compared to a normal computation is roughly proportional to the number of file *opens*, as opposed to the number of reads or writes. In our implementation, on SUN-3/50 workstations, the encapsulation overhead is 18 milliseconds per open for read, and 8 milliseconds per open for write, for the first open of each file. The encapsulation overhead is lower if the same file is opened again: 10 milliseconds per open for read and 4 milliseconds per open for write. The overhead is lower on subsequent opens because the hidden file system directory tree does not need to be updated. Most of the encapsulation overhead results from communication between the file server and the encapsulation server, and from the cost of maintaining the hidden file system directory tree. An implementation in which the encapsulation server is integrated with the file server might be more efficient, but we prefer the modularity of our approach.

At mandate time, overhead is minimized by obtaining a number of timestamps for examination in a single operation. We measured an overhead of 8 milliseconds per open for read, and 31 milliseconds per open for write. These times are limited by the time it takes our file server to find a file in the directory tree and to overwrite a file, respectively. When a computation is mandated while still executing, the mandate can proceed in parallel with the computation, so the overhead does not add to the computation's response time. For the types of computations considered in this paper (compilations and linkages), a conservative estimate for the encapsulation overhead is 2 seconds per computation during execution and 1 second per computation at mandate time.

4 Measurements

4.1 Measurement Environment

The system used for measurement consists of between 8 and 12 diskless SUN-2/50 and SUN-3/50 workstations, and a SUN-3/160 file server, connected by a 10 megabit Ethernet. All machines run the V-System [3]. Remote execution of programs is transparent and incurs a negligible performance penalty. File access is also transparent, and has equal cost from all diskless machines.

The availability of other machines on the network can be determined efficiently using the V group communication mechanism [4].

These machines are used for software development by our group, which consists of 8 graduate students and faculty members, and for projects in a graduate distributed systems course. Most of our *makefiles* involve C compilations and linkages, with a small number of Modula-2 compilations and some TEX text processing. There are typically 4 to 6 active users on the system during the day, although commonly only 2 or 3 of these are actually engaged in software development.

4.2 Method of Measurement

We have instrumented our *make* programs (both the pessimistic and optimistic versions) to collect the following statistics each time a *make* request is executed:

- The out-of-date time for all out-of-date targets: the difference between the time of the *make* request and the latest timestamp of any of the target's sources.
- Command execution time: the running time of each program executed as part of the *make*. All times are normalized to SUN-3 CPU speed.
- The shape of the dependency graph and the number of commands executed as part of the *make*.
- The number of encapsulations aborted as part of each optimistic *make*.

We gathered *make* statistics for more than 6 months, over which time we measured approximately 4,000 requests.

4.3 Measurement Results

Figure 2 shows the cumulative distribution of the target out-of-date times. The median and mean values of this distribution are 32 and 378 seconds, respectively. This implies that the *make* request for most targets is issued fairly soon after a change to the source files is made, but occasionally, users wait quite a while longer before issuing a *make* request. Figure 3 shows the cumulative distribution of the command execution times. The distribution varies with the number of commands per *make* request, where requests with a small number of commands have lower execution times for each command. We speculate that this is due to the fact that many *make* requests with a small number of commands (and especially those with one command) terminate quickly due to compilation errors.

Most of our *makefiles* have a similar dependency graph (see Figure 4): a number of independent commands (usually compilations) followed by a single command (usually a linkage). The distribution of the number of commands per *make* is given in Figure 5. The median number of commands per *make* request is 2, usually corresponding to a change to a single source file, resulting in a recompilation of that source file and a linkage. The mean number of commands is 4.39.

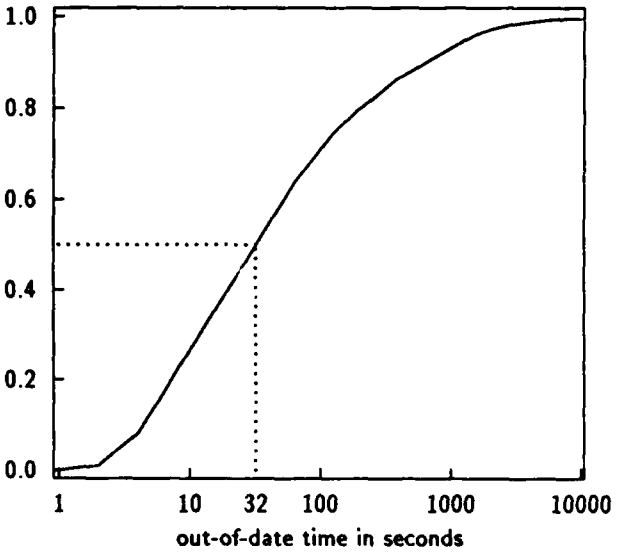

Figure 2 Cumulative Distribution of Target Out-of-date Times.

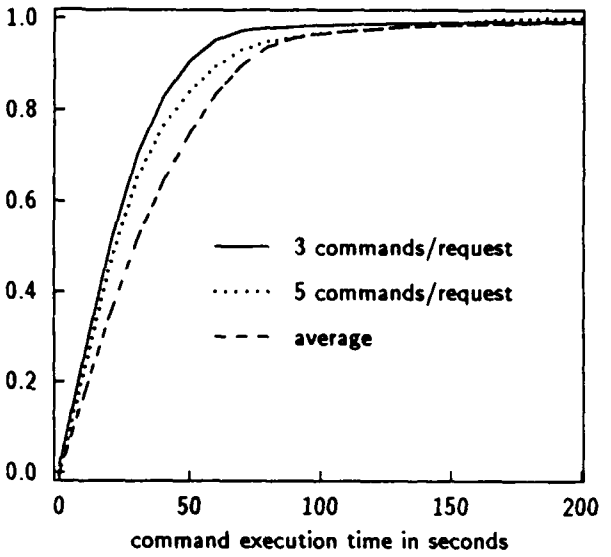

Figure 3 Cumulative Distribution of Command Execution Times.

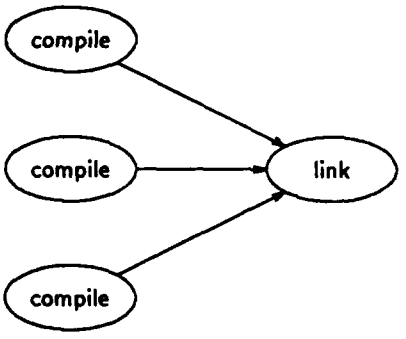

Figure 4 Typical *Makefile* Dependency Structure.

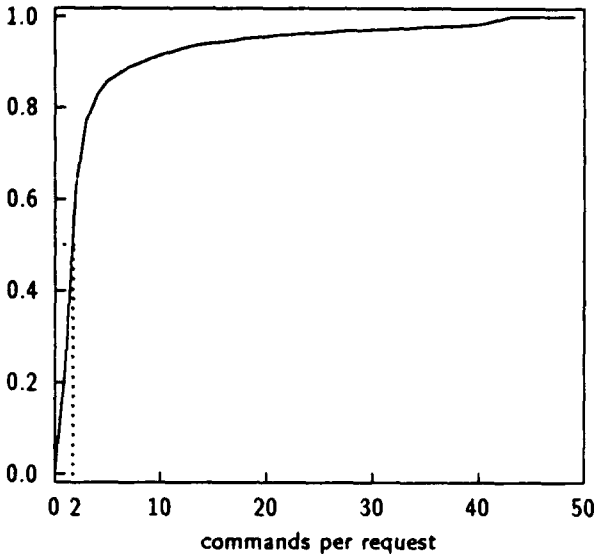

Figure 5 Cumulative Distribution of Number of Commands per Request.

4.4 Overhead Estimates

Optimistic *make* uses more system resources (CPU time, device input/output bandwidth, and memory space) than pessimistic *make* due to the presence of aborted optimistic commands and the encapsulation overhead. Table 1 shows the number of commands mandated and aborted with optimistic *make*. For each mandated command (that is, for every command also necessary in pessimistic *make*), an average of 1.39 optimistic commands are started. Hence, aborted commands impose an extra load of at most 39 percent. This is an upper limit on the extra load since many of the aborted commands do not run to completion, and thus use fewer resources. For the compilations and linkages considered here, encapsulations add less than 5 percent overhead on average (see Section 2.2). Hence, we estimate that the total extra load is at most 44 percent, and is in practice significantly lower. This extra load is small compared to the large idle times that have been observed in workstation environments, even during peak usage periods [12].

Encapsulations use additional disk space beyond that used by normal commands to store the hidden files. To estimate how much extra space might be used, we assume that each user has a completed optimistic *make* containing the measured average of 4.39 commands. These normally consist of a linkage (producing an executable file) and an average of 3.39 compilations (producing object modules). Using the average executable and object module sizes in our system, each of these optimistic *makes* requires a total of 81 kilobytes. If we assume the typical file server has at least 10 megabytes per client, this represents less than 1 percent of the client's disk allocation.

5 Simulation

To further evaluate the performance of optimistic *make*, we now use the measurements of Section 4 to parameterize a simulation of a software development environment. The purpose of the simulation is to determine the *response time improvement* of optimistic *make* over pessimistic *make*. *Response time* is the elapsed time from when the *make* request is issued to when all the commands corresponding to that *make* request are completed. *Response time improvement* is the ratio of response time in pessimistic *make* to response time in optimistic *make*.

The simulation model consists of N identical machines and M users. Each user issues *make* requests, with the think time between requests drawn from an exponential distribution. A command may use any of the N machines, although at any time we allow only a single command to execute on each machine. A centralized scheduler assigns commands to machines in FCFS order, preferring normal commands to optimistic ones. Once a command is started, it runs to completion

Commands	Number	Percent
mandated	16634	100%
aborted	6448	39%
total	23082	139%

Table 1 Mandated and Aborted Commands.

unless aborted, with no preemption. When all workstations are busy, requests are queued until one becomes available. Simulations with centralized, distributed, preemptive and non-preemptive schedulers show that in our environment, under normal load, the choice of scheduling algorithm has little effect on the response time improvement [1].

We simulate both pessimistic and optimistic *make* with identical arrivals of *make* requests. For each pessimistic *make* request, we draw the number of commands to be executed from the empirical distribution shown in Figure 5, and then select the command execution times from the distribution in Figure 3 for requests with that number of commands. The commands are started when the pessimistic *make* request arrives, subject to the dependencies in the *makefile*. Only dependencies of the form depicted in Figure 4 are considered. For optimistic *make*, we use the same request stream as used for pessimistic *make*, and for each request we draw the out-of-date times for each of the targets from the empirical distribution shown in Figure 2. The commands for the optimistic *make* are started at the time of the *make* minus the time drawn from the out-of-date time distribution. In order to simulate aborted commands in optimistic *make*, we introduce an extra command for P percent of the optimistic commands, where P is normally set to the measured value of 39 percent. We assume both pessimistic and optimistic *make* have negligible request processing overhead. In order to account for encapsulation overhead, each optimistic command is assessed an overhead of 2 seconds during execution and 1 second at mandate time.

Since the response time improvement is dependent on the particular *make* request and the out-of-date times, we provide as the main result of our simulations the *cumulative distribution* of the response time improvement of optimistic over pessimistic *make*. Additionally, we provide the median response times for both optimistic and pessimistic *make* as an indication of the absolute difference in response times.

We run a terminating (finite horizon) simulation for a period of 10 simulated hours. Pessimistic and optimistic results are compared by constructing a 95 percent confidence interval on the median response time improvement for each run with a relative precision of ± 3 percent.[1] This typically requires between 10 and 100 runs of the simulator.

6 Simulation Results

6.1 The Baseline System

Figure 6 shows cumulative distributions for the response time improvement in a system similar to the one we are using. All simulation inputs are drawn from the empirical distributions, the number of machines is set to 10, and the mean think time is set to 6 minutes. Results are shown for 1, 5, and 10 users. For the 5-user curve, the median response time improvement is 1.72, and the mean improvement is 8.28. The median improvement in the other curves is similar, but the mean improvement varies slightly. The shape of the curves reflects the fact that most *make* requests are

[1] In those experiments where the median pessimistic and optimistic response times are also recorded, the relative precision for each of the three statistics is set at ± 3 percent, resulting in a lower aggregate precision.

issued shortly after changes to the source files are made. Improvements are occasionally very high, when all optimistic commands have completed by the time of the *make* request, in which case the response time for the optimistic *make* is equal to the time necessary to mandate the commands. A small percentage of optimistic requests perform worse than the same request in the pessimistic simulation, particularly under high load.

Figure 7 shows cumulative distributions of response time improvement in the baseline system with varying mean think times. Decreasing the think time affects the improvement more than varying the number of users, in part because shorter think times imply that users do not wait as long before issuing a *make* request after modifying source files, leaving less time to complete the optimistic work. However, the effect of varying the think time on the response time improvement curves becomes minimal for think times larger than 6 minutes. Measurements indicate that the mean think time in our environment is at least 6 minutes. Hence, for the remaining experiments described in this paper, we fix the mean think time at 6 minutes.

Validation To validate the simulation model, we compare the measured cumulative response time distribution from our implementation to the value obtained from the simulation, for both optimistic and pessimistic *make* (see Figure 8). We compare response times rather than response time improvements since the improvement, as it is computed in the simulator, cannot be measured from the implementation: each real *make* request is either pessimistic or optimistic, but not both (as in the simulator).

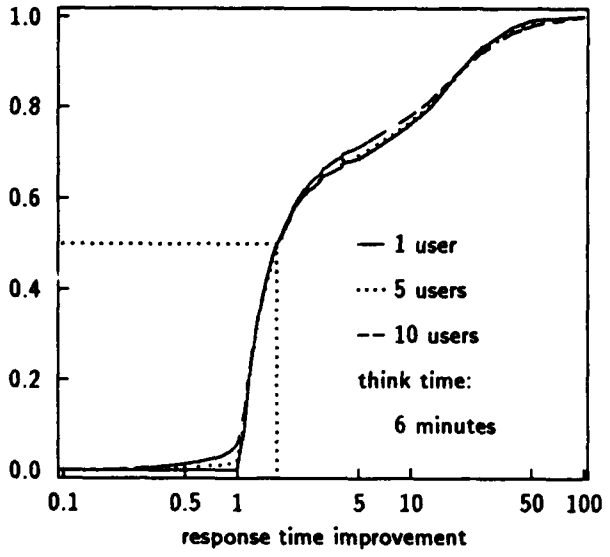

Figure 6 Cumulative Distribution of Response Time Improvement in Baseline System (Varying Users).

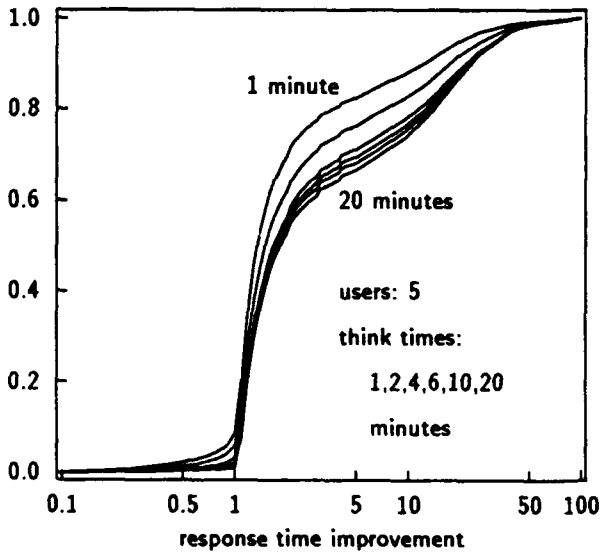

Figure 7 Cumulative Distribution of Response Time Improvement in Baseline System (Varying Think Time).

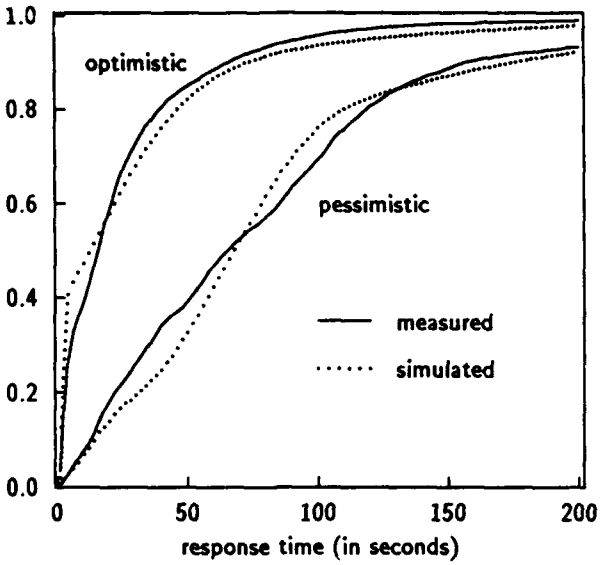

Figure 8 Cumulative Distributions of Simulated and Measured Response Times.

Correlations The response time improvement is somewhat correlated with the number of commands per *make* request and with the total CPU demand per request. Figure 9 shows the median improvement plotted as a function of the number of commands per *make* request, and Figure 10 shows the median improvement plotted as a function of the CPU demand per request. With one or two commands per request and with a total CPU demand less than 30 seconds, the median improvement is noticeably higher than with more commands and larger CPU demands. This is because commands are shorter in these cases and thus more likely to have completed optimistically when the *make* request is issued.

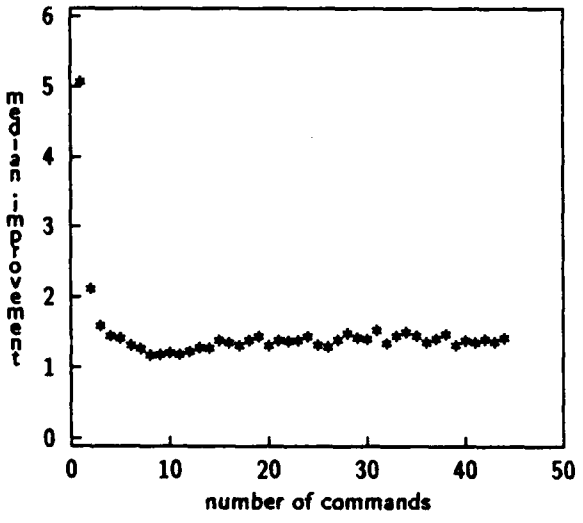

Figure 9 Median Improvement as a Function of the Number of Commands per Request.

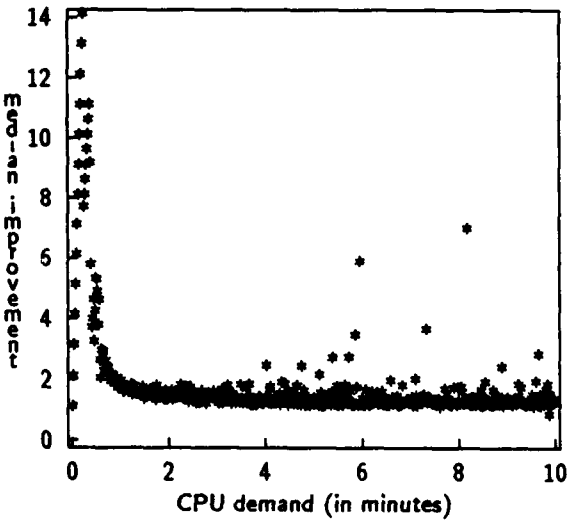

Figure 10 Median Improvement as a Function of the CPU Demand per Request.

Discussion Response time improvement is affected mainly by the *ratio* of target out-of-date times to command execution times, and also by the number of machines available for execution. The ratio of target out-of-date times to command execution times is important because it determines the amount of optimistic computation that can be executed before requested. To isolate the effect of changing this ratio from the effect of the number of machines available for execution, we initially assume an infinite number of machines and alternately vary the command execution and out-of-date times (Sections 6.2 and 6.3). In Section 6.4, we compare the machine utilization of pessimistic and optimistic *make*, then address the effect of limiting the number of machines in Section 6.5. Finally, the effect of heterogeneous machines on the response time improvement is studied in Section 6.6.

6.2 Varying Machine Speeds

To assess the effect of varying machines speeds, the number of machines is set to infinity, and the command execution times (from Figure 3) are divided by a scale factor. Encapsulation overhead is also reduced by the same factor. Other inputs to the simulation (out-of-date times, think time, and number of commands per *make* request) are as in the baseline model.[2]

Figure 11 shows the cumulative distribution of response time improvement for the original machine speed (labeled SUN-3), and for systems 8 and 16 times faster (labeled 8*SUN-3 and 16*SUN-3). Figure 12 shows the median response times for both pessimistic and optimistic *make* plotted side-by-side for several CPU speeds.[3] These figures show that as machine speed increases,

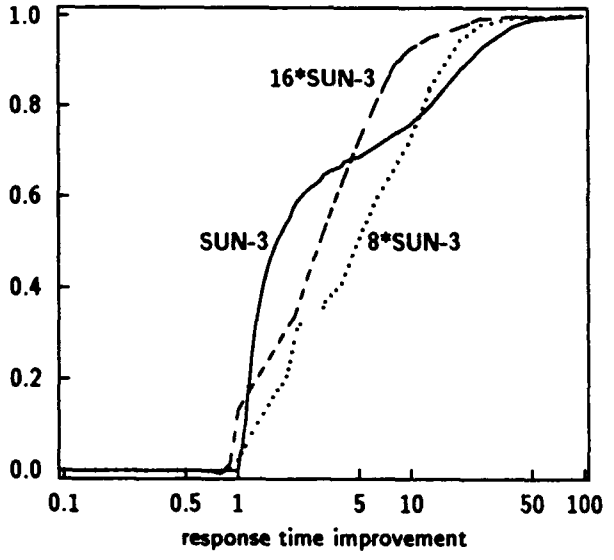

Figure 11 Cumulative Distribution of Response Time Improvement for Varying Machine Speeds.

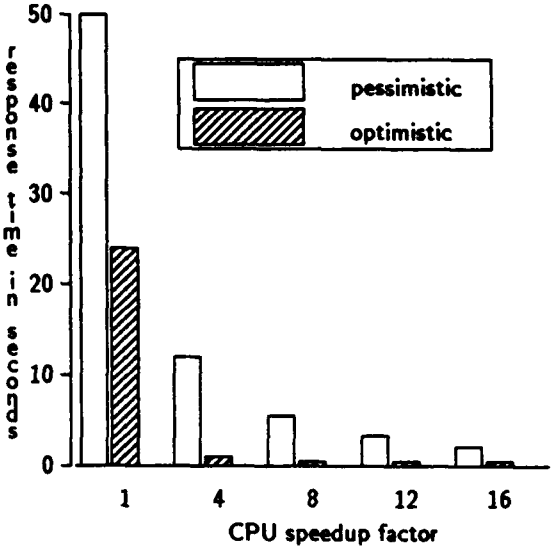

Figure 12 Median Response Times for Varying Machine Speeds.

[2]The number of users is irrelevant with an infinite number of machines.

[3]The ratio of the median response times is not the same statistic as the median response time ratio (the latter is computed by selecting the median of all individual improvements).

the difference between the response time of optimistic and pessimistic *make* decreases. The response time improvement, however, first grows and then decreases with faster machines, from a median of 1.7 in the SUN-3 curve, to a maximum median of 5.1 in the 8*SUN-3 curve, and then back down to a median of 3.3 in the 16*SUN-3 curve. As the machine speed goes from SUN-3 to 8*SUN-3, many more optimistic commands are completed or nearer to completion by the time the *make* request is issued. Hence, response time for optimistic *make* is greatly improved. Response time for pessimistic *make* does not improve as fast as for optimistic *make*, yielding a higher response time improvement. Beyond the CPU speed at which most optimistic commands are completed by the time of the *make* request, there is little additional improvement in optimistic *make's* response time. Pessimistic *make* continues to improve, though, decreasing the response time improvement.

6.3 Varying Out-of-Date Times

While measuring our system, we observed that the median and mean of the out-of-date time distribution changed slightly between different measurement periods. We simulate this effect by using values drawn from the empirical out-of-date time distribution multiplied by different scale factors. Other simulation inputs are as in the baseline system, with an infinite number of machines.

Figure 13 shows the cumulative distributions for scale factors of 0.25, 1, and 4. Figure 14 shows the median response times for both pessimistic and optimistic *make* for several scale factors between 0.25 and 8. Unlike with increasing machine speed (Section 6.2), larger out-of-date times increase both the response time improvement and the difference between median response times, until most optimistic commands are completed by mandate time. With even larger out-of-date times, both remain constant, again in contrast with Section 6.2.

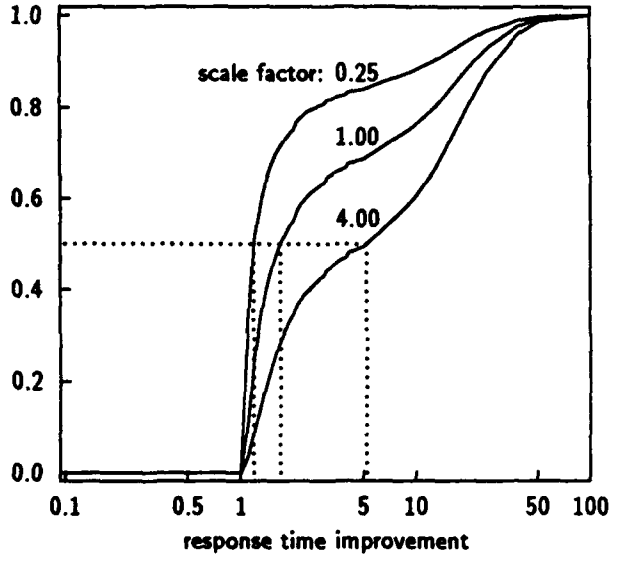

Figure 13 Cumulative Distribution of Response Time Improvement for Varying Out-of-date Scale Factors.

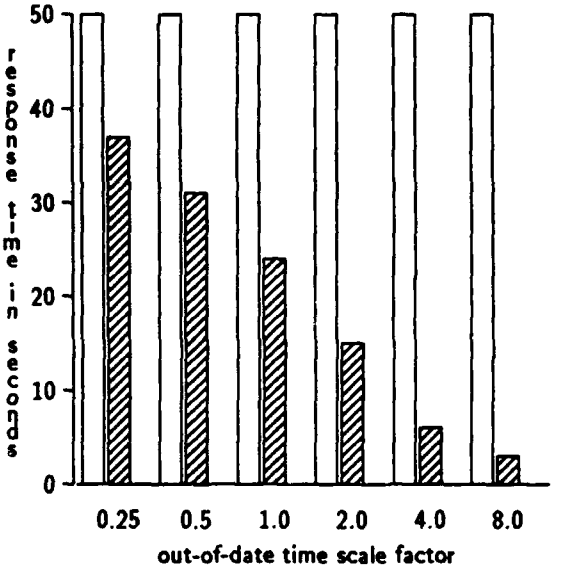

Figure 14 Median Pessimistic and Optimistic Response Times for Varying Out-of-date Scale Factors.

6.4 Machine Utilization

Figure 15 shows the probability distribution for the number of busy machines with optimistic *make* using 39 percent aborted commands (the percentage measured). This distribution is obtained by sampling the number of busy machines once a minute during the simulation. The number of users is varied between 1 and 16, the mean think time is kept constant at 6 minutes, other inputs are drawn from the empirical distributions, and an infinite number of machines are available. Simulations with a constant number of users and varying think times give similar results.

Figure 16 shows the probability distribution of the number of busy machines for 16 users with pessimistic *make*, optimistic *make* with no aborted commands, optimistic *make* with the measured 39 percent aborted commands, and optimistic *make* with 72 percent aborted commands (where all source node commands in the *makefile* dependency graph are aborted once). This figure shows that optimistic *make* distributes CPU load more evenly over time: it is less likely to use very few machines or very many machines. This arises because pessimistic *make* needs many machines when the *make* request arrives, while optimistic *make* spreads out machine use for each request by using machines as soon as files are modified. The aborted commands add to the overall machine utilization of optimistic *make*, but CPU use remains less variable.

6.5 Limiting the Number of Machines

We now limit the number of machines, while fixing the number of users at 16 and the mean think time at 6 minutes. All other simulation inputs are taken from the empirical distributions. Figure 17 shows the response time improvement distribution with 8, 16, and an infinite number of machines. Figure 18 shows the median response times for optimistic and pessimistic *make* for the

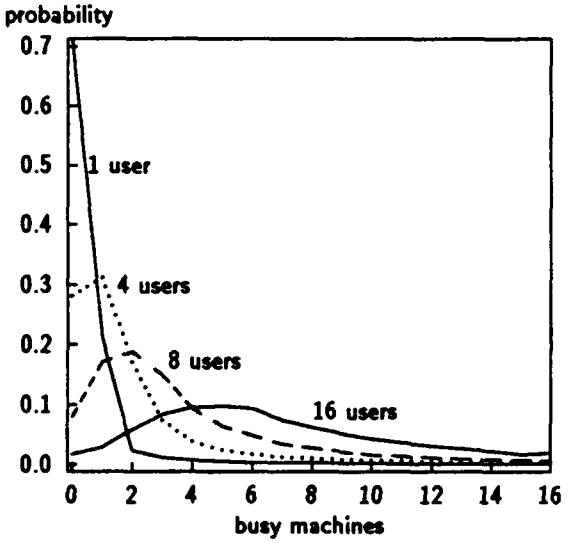

Figure 15 Probability Distribution of Busy Machines for Varying Numbers of Users.

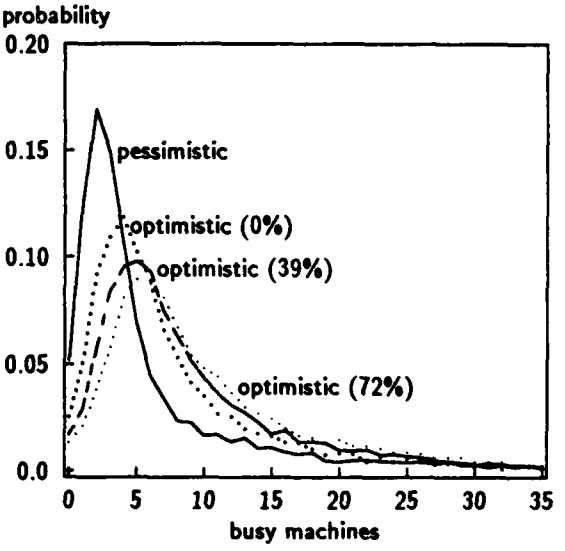

Figure 16 Probability Distribution of Busy Machines for Varying Percentages of Aborted Commands.

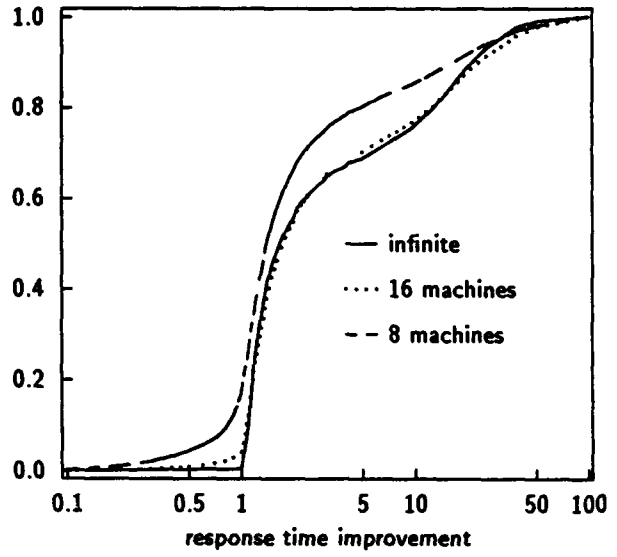

Figure 17 Cumulative Distributions of Response Time Improvement for Varying Numbers of Machines.

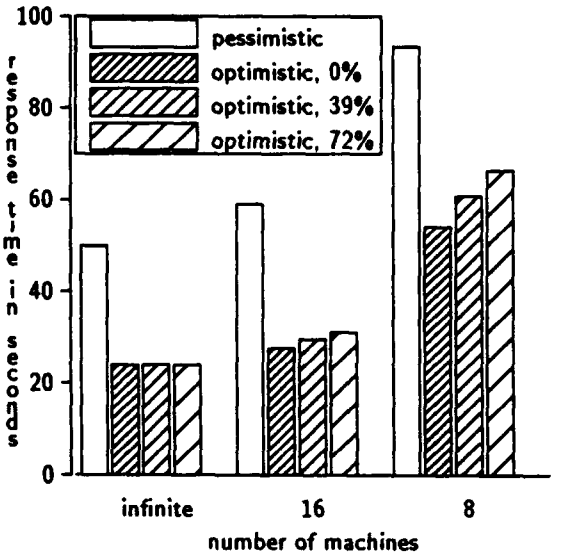

Figure 18 Median Response Times for Varying Numbers of Machines.

same numbers of machines using the three abort ratios from above.

In going from an infinite number of machines to 16, the improvement remains approximately constant, since neither optimistic nor pessimistic *make* are machine-limited. When further decreasing the number of machines to 8 (with 2 users per machine), the improvement declines because optimistic commands are frequently blocked while normal commands use all the resources. The roughly constant improvement down to 16 machines (one user per machine) indicates optimistic *make* provides significant benefits under normal circumstances. Even with unexpectedly high loads, optimistic *make* still provides some improvement.

6.6 Effects of Heterogeneity

In Section 6.2, we showed that increased machine speed improves the response time improvement of optimistic *make*. We now look at how this improvement is affected in a heterogeneous environment. We assume two types of machines: slow machines, at the speed used in our baseline system (SUN-3/50), and fast machines, at twice this speed. In a heterogeneous environment, the scheduler prefers fast machines and thus executes commands on fast machines whenever possible. The number of fast machines is varied from 1 to 10 for different numbers of users on a 10-machine system. The remaining parameters are set as in the baseline system with a 6 minute mean think time. The resulting median response time improvements are shown in Table 2. With 20% fast machines, the median improvement increases substantially, but with more fast machines, the improvement either levels off or increases a small amount before leveling off (depending on the number of users). The reason for such a large improvement with a small number of fast machines is twofold. First, with

	Number of fast machines					
users	0	2	4	6	8	10
1	1.72	2.88	2.81	2.79	2.76	2.73
2	1.71	2.74	2.70	2.66	2.65	2.61
4	1.68	2.62	2.69	2.67	2.66	2.64
6	1.70	2.50	2.65	2.68	2.67	2.66
8	1.71	2.38	2.62	2.69	2.71	2.70
10	1.69	2.24	2.52	2.66	2.70	2.69

Table 2 Median Response Time Improvements for a Varying Number of Fast Machines in a 10-Machine System.

faster machines, optimistic commands are more likely to be completely executed when requested. Second, with optimistic *make*, executions are distributed over time, allowing a larger percentage of commands to be executed on fast machines. For example, when a user types *make* using pessimistic *make*, all commands are started at once (limited by available resources). If the total number of commands is large, several of these are likely to run on slow machines. With optimistic *make*, the user modifies files over time, allowing commands to start at different times. It is likely that a later modification will occur after a previous optimistic command has already completed execution. Thus, a fast machine will be available where a slow one would have been used with pessimistic *make*.

7 Related Work

Optimistic computations have been incorporated into the Integral C programming environment developed at Tektronix [13]. Unlike our implementation, which allows optimistic execution of arbitrary programs, their system only allows a small set of tools to be executed optimistically. No performance evaluation of their system is given, and there is no evidence that Integral C conceals the output of optimistic computations, something we consider to be essential.

The work on eager evaluation in functional programming languages is, to a lesser extent, related to our work [2, 8, 9, 10]. Here, function arguments with call-by-need semantics are evaluated before they are known to be needed. The functional nature of the language obviates the need for explicit concealment of side effects. Our work differs in that we explicitly deal with outputs, and in that the grain of computation we consider is much larger. We believe that with a large grain of computation, the potential for optimistic computations increases significantly, since the overhead involved in concealing outputs becomes relatively less important.

There are also similarities between our work and load sharing [5, 7]. Load sharing attempts to improve throughput by spreading out the workload over different machines. Optimistic execution attempts to improve response time by spreading out the workload over time.

8 Conclusions and Future Work

Optimistic *make* offers significant response time improvement under a wide variety of circumstances. The probability distribution of the response time improvement typically peaks early and then has a long tail, reflected in a small median and a large mean. In our current environment, the median improvement is 1.72 and the mean improvement is 8.28. With faster machines, the median improvement grows significantly, until all optimistic commands are completed by the time the user types *make*. The amount of extra resource use resulting from optimistic *make* is small. Given the increased availability of machines and the observed large idle time percentages in many workstation environments, the extra utilization does not adversely affect performance.

We have introduced the notion of encapsulations as the basic construct used in our implementation of optimistic *make*. Encapsulations appear to be useful for other applications as well. For instance, optimistic fault tolerance methods [11, 14] require that outputs be concealed until it is guaranteed that the states from which these outputs are performed will never be rolled back. Optimistic fault tolerance requires a more incremental notion of encapsulations that allows partial mandates and aborts.

We are also interested in investigating the performance of optimistic *make* with different workloads, for instance workloads measured at other sites or workloads in which program development no longer predominates. Also of interest are the performance implications of different implementations of optimistic *make*, for instance in a system where modification of the kernel and servers is not possible and encapsulation support has to be provided through a library that is linked in with user programs.

References

[1] R. G. Bubenik. *Optimistic Computations.* PhD thesis, Rice University, 1989. In preparation.

[2] F. W. Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, C-34(12):1190–1193, December 1985.

[3] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[4] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[5] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load balancing in homogenous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[6] S. Feldman. Make—a computer program for maintaining computer programs. *Software Practice and Experience*, 9(4):255–265, April 1979.

[7] R. Hagmann. Process server: Sharing processing power in a workstation environment. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 260–267, May 1986.

[8] R. H. Halstead. Parallel symbolic computing. *IEEE Computer*, 19(8):35–43, August 1986.

[9] D. A. Hornig. *Automatic Partitioning and Scheduling on a Network of Personal Computers.* PhD thesis, Carnegie-Mellon University, November 1984.

[10] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proceedings of the Thirteenth Annual Symposium on Principles of Programming Languages*, pages 243–254, January 1986.

[11] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, August 1988.

[12] M. W. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processor bank network. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 2–9, September 1987.

[13] G. Ross. A practical environment for C programming. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, January 1987. Also available as SIGPLAN Notices 22(1), January 1987.

[14] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

Optimistic Implementation of Bulk Data Transfer Protocols

John B. Carter
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

This paper appeared in
Proceedings of 1989 ACM SIGMETRICS and PERFORMANCE '89.

Abstract

During a bulk data transfer over a high speed network, there is a high probability that the next packet received from the network by the destination host is the next packet in the transfer. An *optimistic* implementation of a bulk data transfer protocol takes advantage of this observation by instructing the network interface on the destination host to deposit the data of the next packet immediately into its anticipated final location. No copying of the data is required in the common case, and overhead is greatly reduced.

Our optimistic implementation of the V kernel bulk data transfer protocols on SUN-3/50 workstations connected by a 10 megabit Ethernet achieves peak *process-to-process* data rates of 8.3 megabits per second for 1-megabyte transfers, and 6.8 megabits per second for 8-kilobyte transfers, compared to 6.1 and 5.0 megabits per second for the pessimistic implementation. When the reception of a bulk data transfer is interrupted by the arrival of unexpected packets at the destination, the *worst-case* performance of the optimistic implementation is only 15 percent less than that of the pessimistic implementation. Measurements and simulation indicate that for a wide range of load conditions the optimistic implementation outperforms the pessimistic implementation.

This work was supported in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, and by a National Science Foundation Fellowship.

1 Introduction

In an *optimistic* implementation of a bulk data transfer protocol, the destination host assumes that the next packet to be received from the network is the next packet in the transfer. The destination host instructs its network interface to deposit the data of the next packet received immediately into its anticipated final location. The packet header is deposited in a reserved buffer area, and is later inspected to confirm that the packet is indeed the next packet in the transfer. If so, the protocol state is updated, but the packet data need not be copied, and the code that handles arbitrary packets is bypassed. If the assumption turns out to be wrong, the data is copied to its correct destination. With a more conventional *pessimistic* protocol implementation, the entire packet is first deposited into a packet buffer. The header is then inspected to decide if this is a packet in a bulk data transfer and to determine the address of the data portion of the packet. Finally, the data portion is copied to its final destination. An optimistic implementation takes full advantage of the *scatter-gather* capabilities of network interfaces such as the AMD LANCE [6] and the Intel i82856 [5] present on various models of SUN workstations. These interfaces allow portions of an incoming packet to be delivered to noncontiguous areas of memory.

The bulk data transfer protocol studied in this paper is a *blast* protocol [11]. The data to be transferred is divided into one or more blasts of fixed maximum size. For each blast, the sender transmits the required number of packets, and then waits for an acknowledgement. The receiver sends back an acknowledgement only after it has received the last packet in a blast. There is no per-packet acknowledgement. Figure 1 presents an example with 2 blasts of 4 packets each. If no acknowledgement is received from the destination, the last packet in the blast is retransmitted until either an acknowledgement is received or the destination is deemed to have failed. Selective retransmission is used to deal with missed packets. Flow control measures may be needed to reduce packet loss when going from a fast to a slow machine [2]. The advantages of a blast protocol over more conventional protocols such as stop-and-wait and sliding window derive from the reduced number of acknowledgements, and from the fact that protocol and transmission overhead on the sender and the receiver occur in parallel [11].

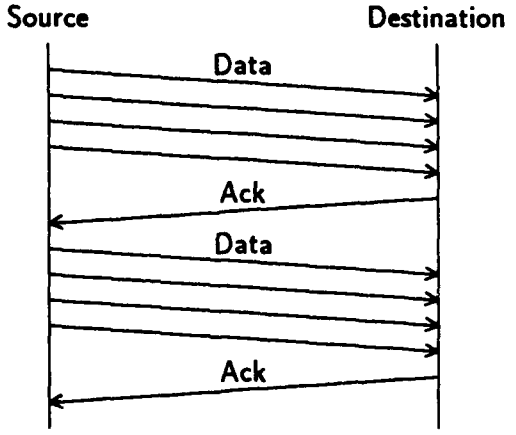

Figure 1 Blast Protocols

The ideas presented in this paper are not specific to blast protocols, and can be applied to any bulk data transfer protocol in which the following two properties hold:

1. Consecutive packets in a bulk data transfer are likely to arrive from the network at the destination machine uninterrupted by other traffic.

2. The final destination of the data in bulk data packets is known prior to the packets' arrival.

The first property assumes that the average network I/O rate into a machine is low, and that bulk data transfer packets are delivered in short bursts. The second property depends on the protocol's interface with user processes. In a request-response protocol, the destination address for the response data is commonly specified at the time of the request, and hence known before the response arrives. This property also holds for the arrival of the request packet, if the server is ready to receive the packet before the request arrives. In stream protocols like TCP, this property holds if the user *read* operation corresponding to the incoming data is already pending when the data arrives.

Our optimistic implementation of the blast protocols in the V kernel on SUN-3/50s connected by a 10 megabit Ethernet achieves peak *process-to-process* data rates of 8.3 megabits per second for 1-megabyte transfers, and 6.8 megabits per second for 8-kilobyte transfers, compared to 6.1 and 5.0 megabits per second for the original pessimistic implementation. These peak data rates occur when, during a data transfer, consecutive incoming packets are also consecutive packets in the transfer. This property can be disturbed by errors, out-of-sequence packets, and other incoming network traffic. When a blast is interrupted by intervening packets, the worst-case performance of the optimistic implementation is only 15 percent less than the performance of the pessimistic implementation. Measurements on our network indicate that on average 0.8 percent of the blasts directed at workstations and 4.0 percent of blasts directed at the file server are interrupted. Additional experiments show that even under heavier loads at the file server, the percentage of blasts that are interrupted remains low enough that on average the optimistic implementation outperforms the pessimistic implementation.

The outline of the rest of this paper is as follows. Section 2 discusses the implementation of optimistic blast protocols. Section 3 describes an experiment to determine the throughput available through the network interface. In Section 4 we present best-case and worst-case data rates for our optimistic implementation, and compare them to the data rates achieved by a pessimistic implementation. Section 5 describes a series of simulations which predict the performance of an optimistic protocol implementation under various system conditions. In Section 6 we report on the percentage of interrupted blasts observed on our network, and we also discuss the effect of artificially putting a higher load on a shared file server. We discuss related work on locality in network traffic and bulk data transfer protocols in Section 7. In Section 8 we draw conclusions and explore avenues for further work.

2 Implementation

An optimistic implementation of bulk data transfer protocols requires a scatter-gather network interface such as the AMD 7990 LANCE Ethernet interface used on the SUN-3/50 and SUN-3/60. Scatter-gather interfaces allow a single packet to be received in (transmitted from) several non-contiguous locations in memory, thereby avoiding intermediate copies during the reception (transmission) of packets. A pessimistic implementation cannot avoid making a copy at the receiving side. Typically, one or more fields in the packet header indicate where the packet data is to go. Thus, while a pessimistic implementation can receive the packet into noncontiguous locations in memory, it is not possible to determine the final destination of the data without examining the header. This results in a copy of the data, which is usually the largest part of the packet. An optimistic implementation avoids this extra copy except when the bulk data transfer is interrupted or an error occurs.

These interfaces also allow multiple buffers to be queued for reception by means of a *receive buffer descriptor ring.* Each receive buffer descriptor contains an address and a length. The interface deposits incoming packets at the addresses in consecutive buffer descriptors, spreading the packet over multiple receive buffers if the length of the packet exceeds the length indicated in the buffer descriptor. A new packet always starts in a new buffer.

In our implementation, all even-numbered buffer descriptors point to areas in the kernel of length equal to the size of the packet header, and all odd-numbered buffer descriptors point to areas of length equal to the maximum Ethernet packet size minus the size of the header. When no blast reception is in progress, the odd-numbered descriptors point to areas in the kernel such that buffer descriptor $i + 1$ points to the area in memory following that pointed to by buffer descriptor i (See Figure 2). The kernel then operates almost identically to the way it operates without the optimistic implementation, except that it sometimes needs to wait for the data part of a packet to arrive in the second buffer before it has a complete packet. The small packets used for 32-byte message transactions in the V interkernel protocol fit entirely into the header [4].

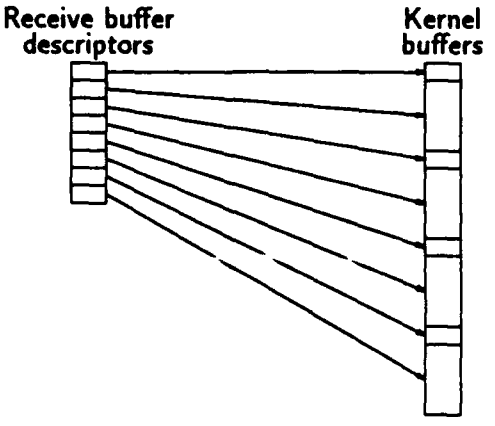

Figure 2 Receive Buffer Descriptors in Default Mode

When the first packet of a blast arrives, the kernel redirects the odd-numbered buffer descriptors to point to where consecutive blast packets should deposit their data in user memory.[1] The even-numbered buffer descriptors for the headers are not changed (See Figure 3). When handling the interrupt for a header buffer, we check if we received the expected packet, and if so, we simply note its receipt. Interrupts for data buffers require no further action.

When an unexpected packet interrupts a sequence of blast packets, further packets in the blast are deposited at incorrect locations in user memory. We do not attempt to dynamically rearrange the receive buffer descriptors, because this would lead to a complicated race condition between the interface and the kernel. After the last packet of the blast arrives, we send an acknowledgement and copy the blast packets to their correct locations. Thus, a worst-case scenario occurs when an unexpected packet is received immediately after redirection, since this causes all further blast packets to be copied. Because we only redirect buffers for at most 32 kilobytes at a time (the maximum blast size), the effects of an unexpected packet are limited to a single blast and do not spread throughout an entire large transfer.

Packets are also deposited at incorrect locations when one or more packets in the blast are lost. After receiving the final packet in the blast, we copy the data of the erroneously located packets to their intended locations, and request retransmission of the missing packets from the sender, in a manner similar to the selective retransmission strategy used in the pessimistic implementation.

The gather capability of a scatter-gather interface allows a packet to be transmitted from non-contiguous locations in memory. We use this capability to construct the packet at the transmitting side without an intermediate copy of the user data into the kernel. A transmit buffer descriptor ring, similar to the receive descriptor ring, is used to queue several packets for transmission. Both our pessimistic and optimistic implementation use the gather capability of the interface, since its use is independent of the optimistic assumption.

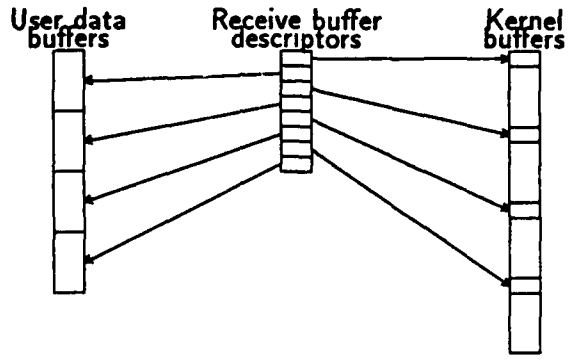

Figure 3 Receive Buffer Descriptors during Blasts

[1]Our current implementation redirects buffers starting at the third packet in the blast, because the hardware is not fast enough to allow us to redirect the second packet in time. The first two packets in a blast are copied to user address space. Also, the kernel must double-map the necessary pages in the receiver's address space into kernel DMA space, because SUN's memory management allows the interface access only to a limited portion of the virtual address space.

3 Hardware Performance

In order to measure the hardware data rate achievable through the LANCE Ethernet interface on the SUN-3/50, we run standalone programs *A* and *B* on two machines connected to the network. *A* sends data to *B* (possibly in multiple packets), and then *B* sends the same amount of data to *A*. The elapsed time of the data transfer is half of the measured time between the first packet leaving *A* and the last packet arriving at *A*. The transfers are implemented at the data link layer and at the device level so no protocol or process switching overhead appears in the hardware performance results. In particular, no header other than the Ethernet data link header is added to the data, and no provisions are made for demultiplexing packets, or for retransmission. When a transmission error occurs, the experiment is halted and restarted. Both programs busy-wait on the completion of their current operation, thereby avoiding interrupt handling overhead.

Our measurements show that the hardware can sustain an average raw data rate of 9.4 megabits per second. The actual data rate is somewhat susceptible to other traffic on the network, presumably as a result of the Ethernet's exponential backoff. Packet loss is commonly zero, except when the interface is configured with only a single receive buffer, in which case packet loss is around 40 percent, or with two receive buffers, in which case packet loss averages 0.1 percent. Performance is slightly worse if only a single transmit buffer is used. Otherwise, performance is relatively independent of the number of transmit and receive buffers. Performance is also relatively independent of the number of bytes transferred, as long as the amount transferred is larger than 8 kilobytes.

4 Protocol Performance

Next, we report the *process-to-process* data rates achievable by both the optimistic and the pessimistic implementations of the V kernel bulk data transfer protocol. All measurements are taken by user level processes running on top the V kernel. The data rate reported is the user observed data rate, calculated by dividing the total number of user data bits transferred by the elapsed time. Protocol header bits are not taken into account. Each operation is repeated 1,000 times, and both the average elapsed time and the standard deviation are reported. Time is measured using a software clock accurate to 10 milliseconds.[2]

Table 1 reports measurements of process-to-process data rates observed using our optimistic implementation, when the blast is not interrupted at the destination by other packets and when no errors occur. Under these conditions, the optimistic implementation achieves 88 percent of the hardware data rate for 1-megabyte transfers. The difference between the maximum data rate observed in this experiment and the hardware data rate results from a number of factors, including:

1. The headers (94 bytes long) and the acknowledgements (also 94 bytes long) must be transmitted and consume extra bandwidth. Taking this into account, the total data rate (including both user and header data) for 1-megabyte transfers is 9.1 megabits per second, which is within 5 percent of the hardware data rate.

[2]We repeat small transfers 20 times to guarantee that each measurement is much larger than the clock period.

Size (Kbytes)	Elapsed Time (msec.)		Rate (Mbps)	% of Hardware Data Rate
	Mean	Dev.		
4	6.2	0.2	5.3	56%
8	9.7	0.5	6.8	72%
32	33.0	1.0	8.0	85%
1024	1015	19	8.3	88%

Table 1 Best-Case Optimistic Implementation

2. There is a fixed cost for each data transfer, consisting of entry into and exit from the kernel, permission checking, interrupt handling, page map manipulation, and provisions for selective retransmission.

Worst-case behavior occurs when the blast is interrupted immediately after the buffers are redirected. We measure this case by modifying the reception routine so that the check to verify that the correct packet is received always fails. This causes the reception routine to always invoke its error recovery routine (including making a copy of the data in the packet). Table 2 indicates the performance of the optimistic blast protocol implementation under these circumstances. Note that there are two lines in this table corresponding to a 1-megabyte transfer. The first line refers to the extremely unlikely case that every 32-kilobyte blast during the transfer is interrupted in a worst-case manner. The second line refers to the case where only a single one of the 32-kilobyte blasts is interrupted in a worst-case manner and the others are not interrupted.

In Table 3 we provide the data rates achieved by a pessimistic implementation of the same protocol.[5] Comparison of Tables 1 and 3 shows the benefits of the optimistic implementation under favorable circumstances. These benefits derive mainly from avoiding copying the data except for the first two packets in the blast.[6] Permission checking and packet handling are also greatly simplified in the case when the expected packet arrives. Comparison of Tables 2 and 3 shows that

Size (Kbytes)	Elapsed Time (msec.)		Rate (Mbps)	% of Hardware Data Rate
	Mean	Dev.		
4	7.8	0.4	4.2	45%
8	14.4	0.6	4.6	49%
32	52.3	0.9	5.0	53%
1024 [3]	1667	22	5.0	53%
1024 [4]	1035	—	8.1	86%

Table 2 Worst-Case Optimistic Implementation

[3] Each 32-kilobyte blast is interrupted.

[4] Only one 32-kilobyte blast is interrupted.

[5] These measurements are better than those reported in [3], because the latter implementation did not use the gather capability of the interface during transmission, while ours does.

[6] A memory-to-memory copy takes 0.37 milliseconds per kilobyte.

the performance loss caused by an erroneous guess is relatively minor, even in the worst case. We only need to copy the data from where we assumed it would land to its correct destination. In a pessimistic implementation, the data must always be copied from the kernel receive buffer area to its destination. The extra overhead results from setting up the appropriate structures for the optimistic implementation and releasing them. Figure 4 provides a graphical comparison of the throughput achieved by the various implementations.

When two 1-megabyte data transfers are taking place simultaneously between two pairs of machines, each transfer receives roughly half the total available network throughput (See also [1]). When 1-megabyte data transfers from two different machines are directed simultaneously at the same destination machine, the transfers achieve an average total throughput of 6.5 megabits per second. 35 percent of these blasts are interrupted by a blast packet from a different blast.

Packet loss is uncommon in our implementation. A large percentage of the packet loss on an Ethernet occurs as a result of receive buffer overflow. Our implementation is configured with 64 receive buffers, enough to avoid noticeable packet loss. We intend to study the packet loss issue further in connection with flow control for transfers from a fast to a slow machine.

Size (Kbytes)	Elapsed Time (msec.)		Rate (Mbps)	% of Hardware Data Rate
	Mean	Dev.		
4	7.8	0.4	4.2	45%
8	13.1	0.5	5.0	53%
32	44.4	1.0	5.9	63%
1024	1376	20	6.1	65%

Table 3 Pessimistic Implementation

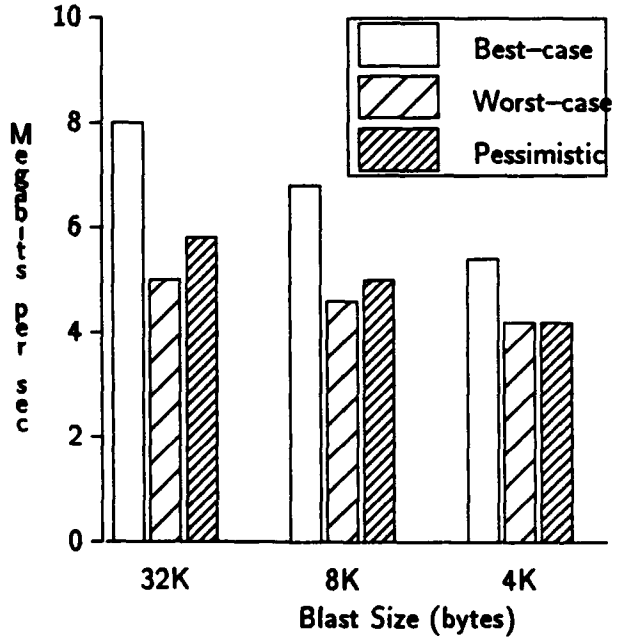

Figure 4 Throughput vs Transfer Size

5 Simulated Performance

We use simulation to study the performance of the optimistic blast protocol implementation over a wide range of system conditions. We assume that the interarrival times for blasts and for non-blast packets are exponentially distributed, and that the non-blast packets are independent of the blast transfers. The experiments consist of a series of terminating (finite horizon) simulations, each with a period of 500 simulated seconds. The experiments are repeated a sufficient number of times to construct a confidence interval on the percentage of blasts interrupted with a 95 percent approximate confidence and a relative precision of 5 percent. The resulting performance predictions match reasonably well with observed performance (See Section 6).

Figures 5 and 6 present the results of a series of experiments designed to study the performance of an optimistic blast protocol implementation on a workstation. We assume that the blasts are transmitted from a single source (i.e., the file server), and thus cannot interrupt one another. The blast arrival rate is chosen so that an average of 8 kilobytes of blast data arrive per second. Figure 5 shows the probability that a blast of a given size is interrupted as a function of the arrival rate of non-blast packets, for blast sizes of 4, 8, and 32 kilobytes. Figure 6 gives the resulting throughput, which is calculated in the following way. For uninterrupted blasts, we take the best-case elapsed time from Table 1. For interrupted blasts, we calculate the elapsed time as the worst-case elapsed time from Table 2 minus the time to copy half of the number of redirected buffers,[7] since on average the blast is interrupted halfway through, after a number of packets have been received without a copy.

Figure 7 presents the results of a series of experiments designed to study the performance of an optimistic blast protocol implementation on a file server. Here, we assume that the blasts are transmitted from multiple sources, so that blasts can interrupt one another. We choose a constant

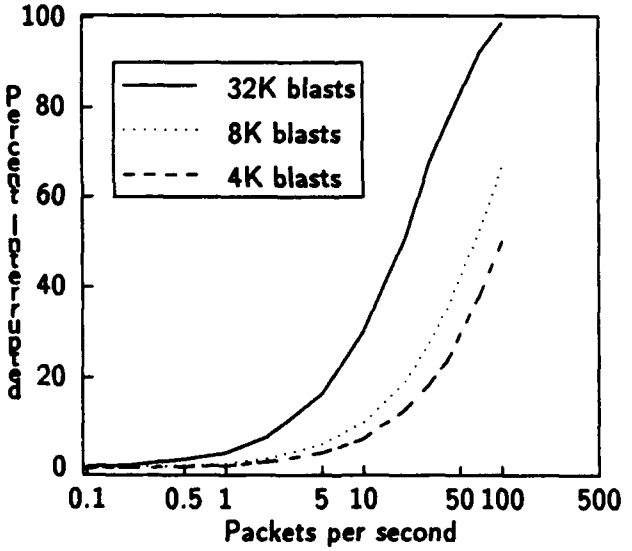

Figure 5 Percentage of Blasts Interrupted at a Workstation

[7] Redirection in the simulation begins with the third packet.

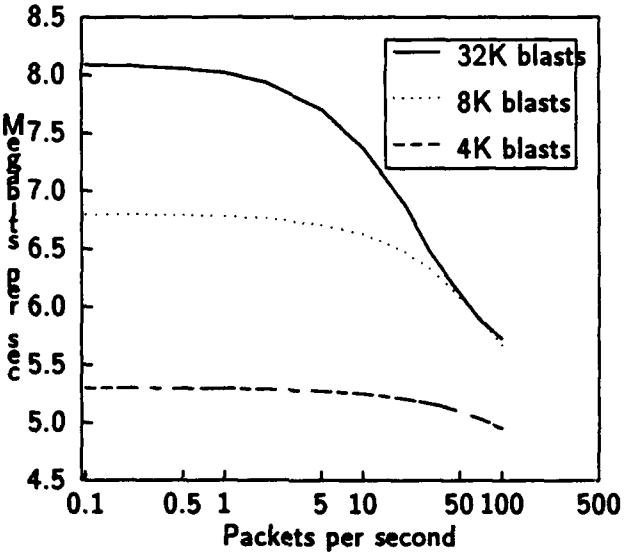

Figure 6 Throughput at a Workstation

blast size of 8 kilobytes, which is the predominant size of blasts arriving at our file server. Figure 7 shows the percentage of interrupted blasts as a function of the arrival rate of non-blast packets for blast traffic rates of 0.5, 2, 8, and 32 kilobytes per second.

6 Performance in an Operating Environment

6.1 Observed Environment

Our Ethernet connects over 60 machines, mostly diskless SUN workstations, 8 file servers, and a few large machines. Virtually all of the machines are used for research, software development, and text processing. Additionally, the network has gateways to the ARPANET, the NSF regional

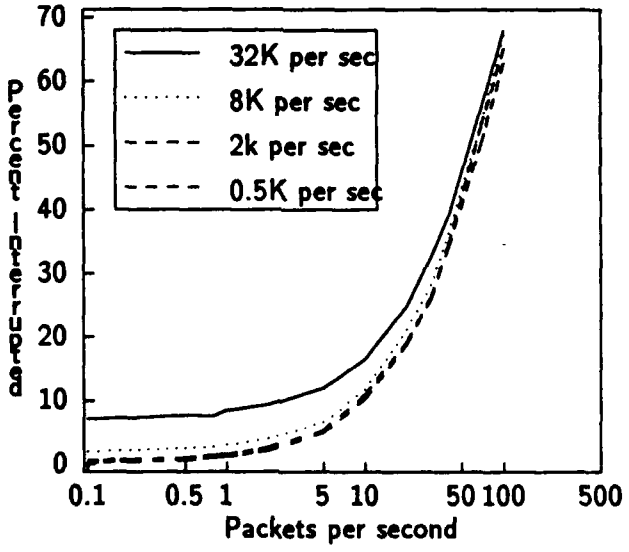

Figure 7 Percentage of Blasts Interrupted at a File Server (8K blasts)

and backbone networks, and the campus backbone network. These gateways are responsible for a large part of the broadcast and multicast traffic on the network (approximately 0.8 packets per second). The network is heavily loaded. The network load averaged over 1-minute intervals fluctuates between 5 and 30 percent.

Between 8 and 12 diskless SUN workstations run the V-System. They are supported by a V-based SUN-3 file server. Most V network traffic is between workstations and the file server, although there is some workstation-to-workstation traffic.[8] The file server uses an 8-kilobyte page size. The following statistics were gathered for each machine:

- The total number of blasts received.
- The number of interrupted blasts, and the type of packet that caused the blast to be interrupted.
- The average blast size.
- The number of non-blast packets received, including non-blast V, other unicast, and broadcast packets.

6.2 Normal Operation

To assess performance during normal operation, we have collected statistics for different periods of 10,000 seconds each. While the exact numbers differ somewhat from one session to another, the overall shape of the figures remains the same. Table 4 contains the results of a representative monitoring session. The column labeled WS represents measurements taken on workstations, and the column marked FS represents measurements taken on the file server.

The percentage of blasts interrupted matches quite well with the predictions from the simulation. At a workstation, an average of 1.4 unrelated packets arrive per second, and the average incoming blast size is 7.0 kilobytes. The simulation predicts that for 8-kilobyte blasts and an unrelated packet arrival rate of 1.4 packets per second, 1.2 percent of the blasts should get interrupted, vs. 0.8 percent

	WS	FS
Number of Blasts	3237	789
Number of Interrupted Blasts	27	32
Percentage of Blasts Interrupted	0.8%	4.0%
Interruptions by Broadcasts	4	1
Interruptions by V Blasts	0	8
Interruptions by Other V packets	23	23
Blasts per Second	0.3	0.1
Other Packets per Second	1.4	3.7
Average Blast Size (kilobytes)	7.0	6.9

Table 4 Normal Load

[8] For a detailed account of V network traffic see [3].

as observed. Similarly, for the observed file server traffic, the simulation predicts that 3.7 percent of the blasts will get interrupted, vs. 4.0 percent as observed.

We conclude that under normal operation, the precentage of interrupted blasts directed at workstations is extremely low. The file server is the target of more unrelated traffic from different sources, and hence the percentage of interrupted blasts is somewhat higher, but it is still low enough that the actual performance approximates the best-case performance.

6.3 High Load Experiment

A high load situation on the file server is created by running a 15-minute sequence of compilations on N diskless SUN-3/50s ($N = 1, ..., 5$). Tables 5 and 6 present the results of these experiments. In Table 5 we indicate the number of incoming blasts, the number of incoming non-blast packets, and the percentage of blasts interrupted, both per workstation and for the file server. Table 6 presents some cumulative statistics for all of the experiments.

We compare these results with those predicted by the simulation. For workstations, the correspondence is still quite good. For example, for the experiment with 5 workstations, the measurements indicate 2.0 percent vs. 2.4 percent predicted by the simulation. The prediction for the file server is less accurate: 11.4 percent measured vs. 16.2 percent predicted. In reality, blasts arriving at the file server are not totally independent, and blasts and non-blast packets are somewhat correlated as well. This correlation seems to reduce the percentage of blasts interrupted from what the simulation predicts.

N	Blasts (per sec.)		Non-Blast Packets (per sec.)		Percentage of Blasts Interrupted	
	WS	FS	WS	FS	WS	FS
1	2.52	0.60	2.21	5.67	2.3%	4.1%
2	2.33	1.03	2.45	7.21	2.1%	6.3%
3	1.98	1.21	2.69	10.13	2.2%	6.8%
4	2.05	1.73	2.98	13.10	2.2%	9.2%
5	2.17	2.14	3.06	16.23	2.0%	11.4%

Table 5 High Load Experiment

	WS	FS
Number of Blasts	28953	6039
Number of Blasts Interrupted	637	477
Percentage of Blasts Interrupted	2.2%	7.9%
Interruptions by Broadcasts	180	24
Interruptions by V Blasts	0	36
Interruptions by Other V packets	457	417

Table 6 High Load Experiment Cumulative Statistics

Even for the largest configuration considered, the "average" performance of our optimistic implementation is still significantly better than the pessimistic implementation. For instance, using the figure in Table 5 of 11.4 percent interruptions on the file server, assuming 8-kilobyte transfers, and assuming a worst-case scenario for each interrupted blast, the average data rate becomes 7.6 megabits per second (from Tables 1 and 2), which is significantly better than the data rate of 5.0 megabits per second achieved by the pessimistic implementation (See Table 3). The numbers reported here are conservative estimates for blast interruptions at a file server because we do not use any caching on the workstations. This significantly increases the traffic to the file server.

7 Related Work

Locality in network traffic has been noted earlier [3, 8, 10]. However, to the best of our knowledge, no protocol implementation has taken advantage of this phenomenon to the extent described here. In his efforts to improve TCP performance, Jacobson predicts that the next incoming TCP packet on a given connection will be the next packet on that connection [7]. This prediction is less aggressive than ours, which assumes that the next packet that is received at a host is the next in the current bulk data transfer. In the expected case, Jacobson's TCP implementation avoids invoking the general purpose packet reception code, but the data must still be copied to user space.

The blast protocols discussed here were developed as part of the V interkernel protocol [4]. Similar protocols are now part of the VMTP protocol definition [3]. VMTP claims 4.5 megabits per second process-to-process data rates on SUN-3/75s (slightly faster than our SUN-3/50s), with no performance improvements when increasing the segment size beyond 16 kilobytes. The current VTMP implementation uses neither the scatter nor the gather feature of the network interfaces on the SUN.

Sprite uses blast protocols (called implicit acknowledgement and fragmentation) for multi-packet RPCs [9]. Sprite performance measurements, also on SUN-3/75s, indicate a kernel-to-kernel data rate of approximately 6 megabits per second, and a process-to-process data rate of 3.8 megabits per second. We speculate that the difference between the kernel-to-kernel and process-to-process data rates is largely due to an extra copy between kernel and user address space, further emphasizing the need for avoiding this copy as in our optimistic implementation.

8 Conclusions

We have described an optimistic implementation of a bulk data transfer protocol, which predicts that during receipt of a blast the next packet that comes in from the network is the next packet in the blast. The implementation instructs the network interface to deposit the data of the next packet in the location for the data in the next bulk data transfer packet. If the prediction is correct, an optimistic implementation avoids an extra data copy.

We have presented both best-case and worst-case performance, and compared them with the performance of a pessimistic implementation. Both simulation and experience indicate that the actual performance of the optimistic implementation is significantly better than the performance

of the pessimistic implementation, even with a relatively high network load on a shared server machine.

More work is required on network interfaces that facilitate optimistic implementation of bulk data transfers. Also, flow control is necessary when transmitting from a fast to a slow machine. We intend to experiment with adapting the blast size as a function of the speed of the receiving machine, while still transmitting at full speed. Hopefully, this will allow uninterrupted packet arrival without overrunning the receiver's buffers. Finally, we want to extend our optimistic blast implementation to allow multicast delivery of bulk data.

References

[1] D.R. Boggs, J.C. Mogul, and C.A. Kent. Measured capacity of an Ethernet: Myths and reality. In *Proceedings of the 1988 Sigcomm Symposium*, pages 222–234, August 1988.

[2] D.R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the 1986 Sigcomm Symposium*, pages 406–415, August 1986.

[3] D.R. Cheriton and C.L. Williamson. Network measurement of the VMTP request-response protocol in the V distributed system. In *Proceedings of the 1987 ACM Sigmetrics Conference*, pages 128–140, October 1987.

[4] D.R. Cheriton and W. Zwaenepoel. The distributed V operating system and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 128–140, October 1983.

[5] Intel Corporation. Intel i82856 interface reference guide.

[6] Advanced Micro Devices. Am 7990: Local area network controller for Ethernet (LANCE).

[7] V. Jacobson. Note on TCP/IP mailing list, tcp-ip@SRI-NIC.ARPA, March 1988.

[8] R. Jain and S. Ruthier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communication*, SAC-4(6):986–995, September 1986.

[9] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[10] J.F. Shoch and J.A. Hupp. Measured performance of an Ethernet local network. *Communications of the ACM*, 23(12):711–721, December 1980.

[11] W. Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32, September 1985.

Causal Distributed Breakpoints

Jerry Fowler
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

Abstract

Causal distributed breakpoint is a new notion of a distributed breakpoint for systems of deterministic processes that communicate solely via messages. Causal distributed breakpoint is the natural extension for distributed programs of the conventional notion of a breakpoints in sequential programs.

The partial order of events in a distributed system defines a lattice of consistent system states that contain a given target process state. The causal distributed breakpoint is the greatest lower bound of this lattice, that is, the system state in which each process reflects the state from which it had the most recent causal effect on the target process. We present an algorithm for finding the causal distributed breakpoint given the breakpoint state of the breakpoint process. The dual of this algorithm finds the least upper bound of the consistent lattice of the target process state.

This work was supported in part by the National Science Foundation under Grants DCR-8716914 and CDA-8619893, and by the Office of Naval Research under Contract N00014-88-K-0140.

1 Introduction

1.1 Problem Statement

We define *causal distributed breakpoints*, a new notion of distributed breakpoints for systems of deterministic processes that communicate solely via messages. The execution of processes is deterministic in the sense that if two processes start in the same state, and both receive an identical sequence of input messages, they send an identical sequence of output messages and finish in the same state. The implementation of a causal distributed breakpoint requires a facility for rollback and replay of individual processes by means of message logging (and checkpointing), and also requires some amount of causal dependency tracking: each message must carry with it a state identifier for the sending process at the time the message was sent.

1.2 Results Summary

We define the notion of a *causal distributed breakpoint*, and describe an algorithm for finding it. A causal distributed breakpoint is a global system state (that is, a set of process states, one for each participating process) such that

1. It contains the state of one process that has been halted under programmer control or due to a software error. We refer to this process as the *breakpoint process*, and to its state as the *breakpoint state.*

2. For all other processes it contains the *largest* state that "happened before" the breakpoint state occurred in the breakpoint process. The "happened before" relationship is Lamport's partial order of events in a distributed system [Lamport78].

A causal distributed breakpoint is a consistent system state in the sense that all messages received have been sent. We present an algorithm for finding the causal distributed breakpoint given the breakpoint state of the breakpoint process.

1.3 Significance

Causal distributed breakpoints are the natural extension for distributed programs of the conventional notion of breakpoints in sequential programs. At a breakpoint in a sequential program, the program can be observed in a state that reflects all events prior to the breakpoint, and no later events. At a causal distributed breakpoint, a distributed program can be observed in a state that reflects all events that "happened before" the breakpoint, according to Lamport's partial order, and no other events. The key distinction derives from the fact that in a distributed system only a partial order of events is observable. Another way of looking at the definition of causal distributed breakpoints is that they reveal each individual process of the computation in the largest state from which it had a causal effect on the breakpoint state.

Previous approaches to distributed breakpointing either broadcast a halt request out-of-band [Cooper87], which does not guarantee a consistent system state in a system under replay

from logs, or propagate a halt request in-band [Haban88, Miller88], similar to the markers in the Chandy-Lamport snapshot algorithm [Chandy85]. The latter method guarantees a consistent state, but may leave other processes in states beyond those that must have happened before the breakpoint state. In other words, this method finds *any* consistent global system state including the breakpoint state, while the causal distributed breakpoint is the *minimum* such consistent state.

1.4 Overview of the Paper

We describe our model of distributed computation in Section 2. We define causal distributed breakpoints in this model and motivate their definition in Section 3. Section 4 contains the algorithm for finding a causal distributed breakpoint. In Section 5 we explain the relationship between causal distributed breakpoints and previously published definitions of distributed breakpoints. Finally, in Section 6 we survey related work, and in Section 7 we conclude and identify avenues for further work.

2 The Model

We consider a collection of deterministic processes communicating solely via messages. The execution of the individual processes is deterministic in the sense that if two processes start from the same state, and receive the same sequence of input messages, they terminate in the same state and send the same sequence of output messages. We define an *event* to be the sending or the receipt of a message. We define a *process state interval* as the execution sequence of a process between two consecutive events. Within a particular process, a process state interval is uniquely identified by a *state interval index*, which is incremented by one during the occurrence of an event. There is a one-to-one relationship between a process state interval and the event that initiates it, so we use the two terms interchangeably.

We use Lamport's "happened before" partial order between events, denoted $\rightarrow$, and defined:

1. If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$.
2. If a is the sending of a message and b is the receipt of the same message, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are *concurrent* if and only if $a \not\rightarrow b$ and $b \not\rightarrow a$.

A *system state* is a set of process state intervals, one for each process. A system state is *consistent* if and only if all messages received by all processes have been sent. That is, for each process i, if process i exhibits state interval σ_i in the system state, then all process state intervals of other processes that happened before state interval σ_i must also be included. Only consistent system states can occur during normal execution. Figure 1 shows examples of consistent and inconsistent states among processes exchanging messages.

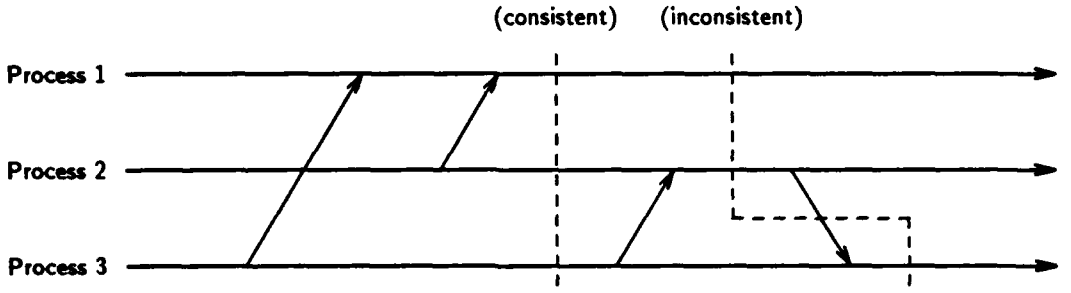

FIGURE 1: Consistent and Inconsistent System States

3 Causal Distributed Breakpoints

We assume that a distributed breakpoint is originated by the setting of a (sequential) breakpoint in a single process, either during normal execution or during replay from message logs. In either case, we call this process the *breakpoint process*, and we call the process state interval during which the breakpoint occurs the *breakpoint state interval.*

We define a *causal distributed breakpoint* as a system state consisting of

1. The breakpoint state interval, and
2. For all other processes, the largest state interval that must have preceded the breakpoint state interval according to Lamport's "happened before" partial order.

A causal distributed breakpoint is a consistent system state (See Figure 2).

With a sequential breakpoint, the state of a sequential program reflects all events that happened in physical time before the breakpoint. The naive extension of this notion to a distributed program (i.e., the state of all processes such that they reflect all events that happened in physical time before the breakpoint) is not achievable, because it is not possible to take an instantaneous snapshot of the entire system. Causal distributed breakpoints extend the notion of a sequential breakpoints so that the system state reflects all events that must have happened before the breakpoint according to Lamport's "happened before" partial order, which is an observable ordering. Another way of

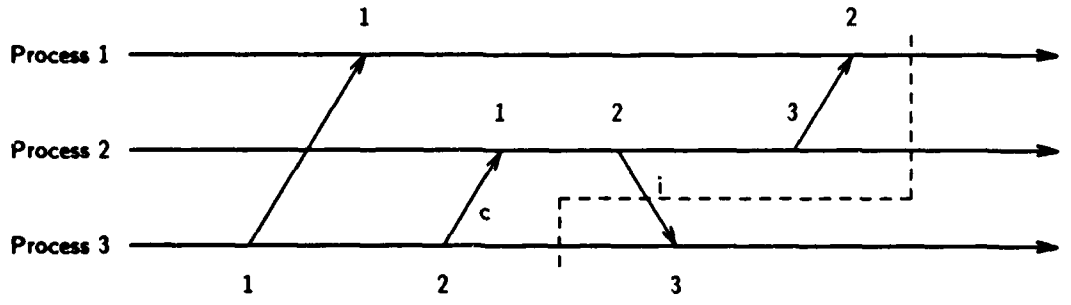

FIGURE 2: Causal Distributed Breakpoint for State Interval 2 of Process 1.

looking at this definition is that it shows the processes other than the breakpoint process in the largest state interval from which they have had a causal effect on the breakpoint state interval.

Alternative definitions of distributed breakpoints accept as a distributed breakpoint any consistent system state that includes the breakpoint state interval [Cooper87, Haban88, LeBlanc87, Miller88]. In an arbitrary consistent state, some process may reflect events that did not have a causal effect on the breakpoint process because they are concurrent with the breakpoint state interval. These events may have changed the state of the other process in such a way that all evidence of the causal effects it had on the breakpoint state is obscured or removed. For instance, in Figure 2, Process 3 receives message i *after* it has sent message c that affects the breakpoint state interval. The arrival of message i, and any changes in state that result from receiving message i, can destroy information that might help explain why c was sent.

4 The Causal Distributed Breakpoint Algorithm

In order to find the causal distributed breakpoint given a particular breakpoint state interval, each message must carry with it the process state interval index of its sender at the time the message was sent. This information, which encodes the "happened before" partial order, is recorded with the message when it is logged.

The algorithm relies on the observation that if state interval τ_j in process j must happen before state interval σ_i in process i, then all state intervals from 0 to $\tau_j - 1$ in process j must also happen before state interval σ_i in process i. Hence, it suffices to know for each process state interval σ_i of i the largest process state interval of all other processes that must have happened before σ_i of i. We define the *dependency vector* [Johnson88] of state interval σ_i of process i, $\mathbf{DV}_i^\sigma$, as a vector of length equal to n

$$\mathbf{DV}_i^\sigma = \langle \delta_* \rangle = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle \ ,$$

where n is the total number of processes in the system. Component j of process i's dependency vector, δ_j, gives the maximum state interval index in process j that happened before process i's current state interval. Component i of process i's dependency vector is always set to σ_i, the index of process i's current state interval. If process i has no dependencies on some process j, then δ_j is set to $\perp$, which is less than all possible state interval indices. The dependency vector of any state interval can easily be computed using the message log for that process, given that for each message the sender's state interval index is included with the message in the log.

Figure 3 shows the algorithm for finding the causal distributed breakpoint for state interval σ of process i. Procedure **CausalBkpt** starts by initializing the causal distributed breakpoint, **CDB**, to contain the breakpoint state interval for the breakpoint process, and the state interval 0 for all processes. The algorithm then examines the dependency vector of each state interval it includes in **CDB** by recursively invoking **Meet**. When **Meet** finds a state interval index that is larger than the index of the corresponding process already included in **CDB**, **CDB** is altered to include the new, larger state interval index, and the dependency vector of that state interval is also examined. Thus, **Meet** recursively includes all state intervals that happened before the breakpoint

state interval. Hence, when the recursion terminates, **CDB** holds the state interval indices of the causal distributed breakpoint.

Suppose that in Figure 2 we wish to find the causal distributed breakpoint for state interval 2 of Process 1. **CDB** is initialized to $\langle 2,0,0\rangle$, and **Meet** is invoked for process state interval 2 of Process 1. The dependency vector $\mathbf{DV}_1^2$ for that state interval is $\langle 2,3,1\rangle$. The state interval for Process 2 in this dependency vector is 3, and is not included yet in **CDB**. Thus, **CDB** is set to $\langle 2,3,0\rangle$ and **Meet** is invoked with state interval 3 of Process 2. The dependency vector $\mathbf{DV}_2^3$ for that state interval is $\langle \perp,3,2\rangle$. The state interval for Process 3 in this dependency vector is 2 and is not included yet in **CDB**. Thus, **CDB** is set to $\langle 2,3,2\rangle$ and **Meet** is invoked with state interval 2 of Process 3. The dependency vector $\mathbf{DV}_3^2$ for that state interval is $\langle \perp,\perp,2\rangle$. All of its state intervals have been included in **CDB** and hence this invocation of **Meet** returns without making changes to **CDB**. The invocation of **Meet** with state interval 3 of Process 2 also returns without further changes **CDB**, as does the initial invocation of **Meet** with the breakpoint state interval. The final value of **CDB** is $\langle 2,3,2\rangle$, the causal distributed breakpoint for state interval 2 of Process 1.

The algorithm can easily be distributed by making **Meet** a remote procedure call to the process named in the arguments and sending the current value of **CDB** with the remote procedure call.

```
CausalBkpt (i : process, σ : state interval)
   /* Compute the causal distributed breakpoint for state interval σ of
   process i and store it in CDB */
   for all k ≠ i
      CDB[k] = 0
   end for.
   CDB[i] = σ
   Meet( i, σ )
end CausalBkpt.

Meet (j : process, τ : state interval)
   /* Include the dependencies of state interval τ of process j in CDB */
   for all k ≠ j
      α = DV_j^τ[k]
      if α > CDB[k]
         CDB[k] = α
         Meet( k, CDB[k] )
      endif
   end for.
end Meet.
```

FIGURE 3: Causal Distributed Breakpoint Algorithm.

5 Relationship to Other Breakpoint Definitions

Miller and Choi, and Haban and Weigel define a distributed breakpoint as any consistent system including the breakpoint state interval [Miller88, Haban88]. In this section, we formally characterize the difference between their definition and causal distributed breakpoint. We define a partial order on *system* states, and we show that the set of all consistent system states that includes a given state interval of a given process forms a lattice under this partial order. The causal distributed breakpoint is the greatest lower bound of this lattice.

The set of consistent system states during any single execution is partially ordered by the relation that one system state precedes another if and only if it *must* occur first during the execution. Let a system state be described by a vector consisting of the state interval index of each process in the system state. If $\mathbf{A} = [\alpha_*]$ and $\mathbf{B} = [\beta_*]$ describe system states from the same execution, then

$$\mathbf{A} \preceq \mathbf{B} \iff \forall i \, [\alpha_i \leq \beta_i] \, .$$

This partial order differs from "happened before" in that it orders the system states that result from events rather than the events themselves. Using this partial order, we can define the upper and the lower bound of two system states.

Definition 1 *The* greatest lower bound *(GLB) of two system states* $\mathbf{A}$ *and* $\mathbf{B}$ *is* $\mathbf{L} = [\lambda_*]$ *such that* $\lambda_i = min(\alpha_i, \beta_i)$. *The* least upper bound *(LUB) of two system states* $\mathbf{A}$ *and* $\mathbf{B}$ *is* $\mathbf{U} = [\upsilon_*]$ *such that* $\upsilon_i = max(\alpha_i, \beta_i)$.

Theorem 1 *Let* σ_i *be an arbitrary process state interval in a computation. Let* $\mathcal{L}_{\sigma_i}$ *be the set of consistent system states of that computation that contain* σ_i. *Then* $\mathcal{L}_{\sigma_i}$, *ordered by* $\preceq$, *is a lattice, called the* consistent lattice *of* σ_i.

Proof Define the dependency matrix of a system state [Johnson88] as an $n \times n$ matrix

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} \delta_{11} & \delta_{12} & \delta_{13} & \cdots & \delta_{1n} \\ \delta_{21} & \delta_{22} & \delta_{23} & \cdots & \delta_{2n} \\ \delta_{31} & \delta_{32} & \delta_{33} & \cdots & \delta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n1} & \delta_{n2} & \delta_{n3} & \cdots & \delta_{nn} \end{bmatrix} ,$$

where row i, δ_{ij}, $1 \leq j \leq n$, is the dependency vector for the state interval of process i in this system state.

If a system state is consistent, then for each process i, there is no state interval of process i beyond process i's current state interval that happened before the state interval of any other process j in the system state. In the dependency matrix, for each column i, no element in column i may be larger than the element in that column on the diagonal of the matrix, which is process i's current state interval index. In other words, if $\mathbf{D} = [\delta_{**}]$ is the dependency matrix of a system state, then that system state is consistent if and only if

$$\forall i, j \, [\delta_{ji} \leq \delta_{ii}] \, .$$

We need to show that any two consistent states in $\mathcal{L}_{\sigma_i}$ have a GLB and an LUB that are consistent and contain σ_i. Let **A** and **B** be arbitrary consistent system states that contain σ_i. Let $\mathbf{D}_A = [\alpha_{**}]$ and $\mathbf{D}_B = [\beta_{**}]$ be the dependency matrices corresponding to **A** and **B**.

Let **U** be the LUB of **A** and **B**, and let $\mathbf{D}_U = [\upsilon_{**}]$ be the corresponding dependency matrix. Since **A** and **B** are consistent, $\alpha_{ji} \leq \alpha_{ii}$ and $\beta_{ji} \leq \beta_{ii}$ for all i and j. Since $\upsilon_{ji} = \alpha_{ji}$ or $\upsilon_{ji} = \beta_{ji}$, and $\upsilon_{ii} = \max(\alpha_{ii}, \beta_{ii})$, $\upsilon_{ji} \leq \upsilon_{ii}$ for all i and j as well. Therefore, **U** is consistent, and since it also must include state interval σ_i for process i, **U** must be a member of $\mathcal{L}_{\sigma_i}$.

Let **L** be the GLB of **A** and **B**, and let $\mathbf{D}_L = [\lambda_{**}]$ be the corresponding dependency matrix. By the definition of GLB, and since no element in the dependency vector for any process ever decreases as the process executes, $\lambda_{ji} = \min(\alpha_{ji}, \beta_{ji})$, for all i and j. This implies that $\lambda_{ji} \leq \alpha_{ji}$ and $\lambda_{ji} \leq \beta_{ji}$. Since **A** and **B** are consistent, $\alpha_{ji} \leq \alpha_{ii}$ and $\beta_{ji} \leq \beta_{ii}$. Combining this with the previous result yields $\lambda_{ji} \leq \alpha_{ii}$ and $\lambda_{ji} \leq \beta_{ii}$. This implies that $\lambda_{ji} \leq \min(\alpha_{ii}, \beta_{ii})$, and thus $\lambda_{ji} \leq \lambda_{ii}$, for all i and j. Therefore, **L** is consistent, and since it also must include state interval σ_i for process i, **L** must be a member of $\mathcal{L}_{\sigma_i}$. □

The greatest lower bound of the consistent lattice of a state interval σ_i is the causal distributed breakpoint of σ_i. Since the causal distributed breakpoint for σ_i includes σ_i, and since it is consistent, it must be a member of $\mathcal{L}_{\sigma_i}$. Furthermore, it precedes every member of $\mathcal{L}_{\sigma_i}$ because any smaller system state containing the breakpoint state interval must necessarily lack at least one of σ_i's state intervals and therefore be inconsistent. Therefore, the causal distributed breakpoint for σ_i cannot be larger than $GLB(\mathcal{L}_{\sigma_i})$. Hence, they are equal.

The LUB of the consistent lattice of σ_i is the state beyond which execution cannot progress without process i first advancing to $\sigma_i + 1$. This knowledge can also be of value in distributed debugging. The algorithm to find the LUB of the consistent lattice is the dual of the algorithm in Figure 3, and is shown in Figure 4.

Consider Figure 2 for an example of the way LeastUpperBound works. If **LeastUpperBound** is invoked on Process 3 in state interval 1, **LUB** is initially set to $\langle 2, 3, 1 \rangle$, and **Join** is invoked on Process 3, state interval 1. For Process 1, the greatest α such that $\mathbf{DV}_1^{\alpha}[3] \leq 1$ is 2, so no change is made. For Process 2, the greatest α such that $\mathbf{DV}_2^{\alpha}[3] \leq 1$ is 0, because message c is sent from a state interval that Process 3 has not reached. Consequently, $\alpha <$ **LUB**[2], so **LUB** becomes $\langle 2, 0, 1 \rangle$, and **Join** is invoked on Process 2, state interval 0. For Process 1, the greatest α such that $\mathbf{DV}_1^{\alpha}[2] \leq 0$ is 1, which is less than **LUB**[1]; then **LUB** becomes $\langle 1, 0, 1 \rangle$, and **Join** is invoked on Process 1, state interval 1 with no further effect because all dependencies on Process 1 are $\perp$. For Process 3, the greatest α such that $\mathbf{DV}_3^{\alpha}[2] \leq 0$ is 2, which exceeds **LUB**[3], so no change is made, and **Join** for Process 2 returns without altering **LUB**. The initial invocation of **Join** on Process 3 returns with **LUB** $= \langle 1, 0, 1 \rangle$, and the algorithm terminates.

Knowing the bounds of the consistent lattice of a process state interval σ_i makes possible a simple test for parallellism between another process state interval τ_j and σ_i. If **CDB** and **LUB** are the bounds of $\mathcal{L}_{\sigma_i}$, and **CDB**$[j] \leq \tau_j \leq$ **LUB**$[j]$, then τ_j can run concurrently with σ_i during this particular execution history of the computation.

```
LeastUpperBound (i : process, σ : state interval)
    /* Compute the least upper bound of the consistent lattice for state
    interval σ of process j and store it in LUB */
    for all k ≠ i
        LUB[k] = highest state interval index for process k in the message
        log
    end for.
    LUB[i] = σ
    Join( i, σ )
end LeastUpperBound.

Join (j : process, τ : state interval)
    /* Include the dependencies of state interval τ of process j in LUB */
    for all k ≠ j
        α = largest state interval index of process k such that DV_k^α[j] ≤ τ
        if α < LUB[k]
            LUB[k] = α
            Join( k, LUB[k] )
        endif
    end for.
end Join.
```

FIGURE 4: Least Upper Bound Algorithm.

6 Related Work

Our work is based on Lamport's happened-before relation for describing the partial ordering of distributed events [Lamport78].

Miller and Choi [Miller88] adapt Chandy and Lamport's distributed snapshots [Chandy85] to distributed breakpointing. Haban and Weigel [Haban88] use a similar approach to that of Miller and Choi for generating interactive breakpoints. Both approaches lend themselves to static communication channels. Haban and Weigel's message passing mechanism carries the same causal information with each message as does our algorithm.

Cooper's method of broadcasting a halt request out-of-band [Cooper87] produces a consistent state when used during normal execution. However, when replaying from a log, it may result in an inconsistent state.

The size of the logs necessary to support causal distributed breakpoints may be a concern. LeBlanc and Mellor-Crummey observe in their Instant Replay mechanism, that it is not necessary to store the data passed in messages, if all message communication can be described as reads and writes of shared objects (under certain restrictions for concurrent access of those objects) [LeBlanc87].

In those circumstances, their methods can be used in conjunction with ours without the need for logging the data passed in messages.

Causal dependency tracking by including the process state interval index of the sender in each message, and the notion of a dependency vector and a dependency matrix are due to the work of Johnson and Zwaenepoel on optimistic message logging [Johnson88].

7 Conclusion

We have defined the notion of causal distributed breakpoints. When one process of a distributed computation is halted in a particular state, causal distributed breakpoints show the other processes in the computation in their largest state that has happened before the state of the breakpoint process. We have argued that causal distributed breakpoints naturally extend the notion of a sequential breakpoint to distributed systems, where only a partial ordering between events can be observed. Previous work on distributed debugging has defined a distributed breakpoint as any consistent state including the breakpoint state. We have shown that the set of consistent system states that include a particular process state interval forms a lattice, and that the greatest lower bound of this lattice is equal to the causal distributed breakpoint for that particular state interval. Future work includes examining the use of the lattice to derive alternate valid executions of a computation via permutation of event queues in individual processes.

References

[Chandy85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Cooper87] Robert Cooper. Pilgrim: a debugger for distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 458–465, IEEE, September 1987.

[Haban88] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Hawaii Intl Conf, Vol II, Software*, pages 166–175. IEEE Computer Society, January 1988.

[Hewitt77] Carl Hewitt and Henry Baker, Jr. Actors and continous functionals. Technical Report MIT/LCS/TR-194, Massachusetts Institute of Technology, Cambridge, MA, 02139, December 1977.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. IEEE Computer Society, August 1988. Also available as Rice University Technical Report COMP TR88-68, May 1988.

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LeBlanc87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987. Also available as a Technical Report, Sept. 1986, Department of Computer Science, University of Rochester,Rochester, NY 14627.

[Miller88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. Technical Report TR 648, University of Wisconsin, Madison, WI, February 1988. (Revised).

Distributed System Fault Tolerance Using Sender-Based Message Logging

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

This paper is being submitted to
ACM Transactions on Computer Systems.

An earlier paper appeared in *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers.*

Abstract

Sender-based message logging is a new transparent method of providing fault tolerance in distributed systems. It uses a *pessimistic* message logging protocol designed to reduce the overhead on the system. It differs from other pessimistic message logging protocols in that the message log is stored in the *volatile* memory on the node from which the message was *sent*. When a process receives a message, it returns to the sender a *receive sequence number*, or *RSN*, which indicates to the sender the order in which the message was *received* relative to other messages sent to the same process.

Sender-based message logging has been implemented under the V-System on a network of SUN-3/60 workstations. The measured overhead on V-System communication operations is about 25 percent. The overhead experienced by distributed applications programs using sender-based message logging is affected most by the amount of communication performed during execution. For the most communication-intensive program measured, this overhead ranged from about 16 percent to about 3 percent, for different problem sizes. For all other programs measured, overhead ranged from about 2 percent to much less than 1 percent.

This work was supported in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under contract ONR N00014-88-K-0140.

1 Introduction

Sender-based message logging is a new transparent method of providing fault tolerance in distributed systems. It needs no specialized hardware and requires little additional communication in the system. It uses *message logging and checkpointing* to record information for recovering a consistent system state following a failure. All messages received by each process are logged, and occasionally, each process is independently checkpointed. When a process fails, it is recovered by restoring it from its checkpoint and replaying to it from the log the sequence of messages it received after the checkpoint before the failure. Previous systems using message logging and checkpointing to provide fault tolerance include Auros and TARGON/32 [Borg83, Borg89], PUBLISHING [Powell83], and Strom and Yemini's Optimistic Recovery [Strom85].

Sender-based message logging differs from these earlier methods in that each message is logged in the local *volatile* memory of the node from which it is *sent*, as illustrated in Figure 1. Other message logging protocols send a copy of each message either to stable storage [Lampson79, Bernstein87] on disk or to some special backup process for logging. Instead, since both the sender and receiver of a message either get or already have a copy of it, it is less expensive to save one of these in the local volatile memory to serve as a log. Since the purpose of the logging is to recover the receiver if it fails, a volatile copy at the receiver cannot be used as the log, but the sender can easily save a copy of each message sent. It is this idea that forms the basis of sender-based message logging.

The sender-based message logging protocol is a *pessimistic* protocol designed to reduce the overhead on the system for the provision of fault tolerance. A pessimistic message logging protocol prevents the system from entering a state in which a failure could force any process other than those that failed to be rolled back during recovery. Any failed process can be recovered using only its most recent checkpoint and the log of messages received since that checkpoint. In any system using message logging and checkpointing to provide fault tolerance, the cost of message logging should dominate the overhead of fault tolerance. The frequency of checkpointing can be tuned to balance its expense against the time needed for recovery or the space needed to store the message log, and the cost of failure recovery should be less important if failures are infrequent. However, the cost of message logging places a continuous burden on the system even when no failures occur. Sender-based message logging concentrates on reducing the cost of message logging in a pessimistic logging protocol.

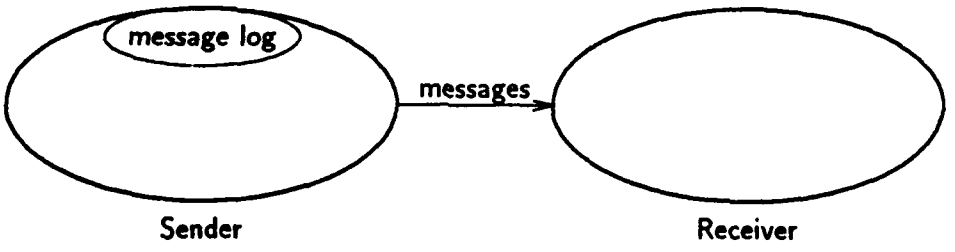

Figure 1 Sender-based message logging configuration

Since messages are logged in volatile memory, sender-based message logging can guarantee recovery from only a single failure at a time within the system. That is, after one process fails, no other process may fail until the recovery of the first is completed. If another process fails during this time, some logged messages required for recovery from the first failure may be lost, and recovery of a consistent system state may not be possible from the available logged messages. Sender-based message logging detects this error, allowing the system to notify the user or abort the computation if desired. Extensions to the basic sender-based message logging protocol are also possible to support recovery from any number of failures at once.

This paper examines the design of sender-based message logging, describes a complete implementation of it, and presents an analysis of its performance. Section 2 describes the model of a distributed system assumed in this work. The specification of the sender-based message logging protocol is presented in Section 3. Section 4 describes an implementation of it in the V-System [Cheriton83, Cheriton88], and Section 5 analyzes its performance in this implementation. Extensions to the basic sender-based message logging protocol for recovery from multiple failures are discussed in Section 6. Related work is covered in Section 7, and Section 8 presents some conclusions.

2 Distributed System Model

Sender-based message logging is designed to work in existing distributed systems without the addition of specialized hardware to the system or specialized programming to applications. The following assumptions about the underlying distributed system are made:

- The system is composed of a network of fail-stop processors [Schlichting83].
- Processes communicate with each other only through messages.
- The execution of each process in the system is *deterministic* between received input messages. That is, if two processes start in the same state and receive the same sequence of inputs messages, they produce the same sequence of output messages and must finish in the same state. The state of a process is thus completely determined by its starting state and the sequence of messages received.
- The network includes a shared stable storage service [Lampson79, Bernstein87] that is always accessible to all active nodes in the system.
- Packet delivery on the network need not be guaranteed, but reliable delivery can be implemented by retransmitting a packet a limited number of times until an acknowledgement arrives from the destination.
- The network protocol supports broadcast communication. All active nodes can be reached by a broadcast through a limited number of retransmissions of the packet.
- Processes use a *send sequence number* (*SSN*) for duplicate message detection. For each message sent, the SSN is incremented, and the new value is used to tag the message. Each

process also maintains a table recording the highest SSN value tagging a message received from each other process. If the SSN tagging a new message received is not greater than the current table entry for its sender, the message is considered to be a duplicate.

3 Protocol Specification

3.1 Overview

Each process participating in sender-based message logging maintains a *receive sequence number* (*RSN*), which counts messages *received* by the process. When a process receives a new message, it increments its RSN, and returns this *new* value to the sender. The RSN indicates to the sender the order in which that message was received relative to other messages sent to the same process, possibly by other senders. This ordering information is not otherwise available to the sender, but is required for recovery because the messages must be replayed to the recovering process from the log in the same order as they were received before the failure. When the RSN arrives at the sender, it is saved in the local volatile log with the message.

The log of messages received by a process is distributed among the processes that sent these messages, such that each sender has in its local volatile log only those messages that it sent. Figure 2 shows an example of such a distributed log resulting from sender-based message logging. In this example, process Y initially had an RSN value of 6. Process Y received two messages from process X_1, followed by two messages from process X_2, and finally another message from X_1. For each message received, Y incremented its current RSN and returned the new RSN value to the sender. As each RSN arrived at the sender, it was added to the sender's local volatile log with the message. After receiving these five messages, the current RSN value of process Y is 11.

In addition to returning the RSN to the sender when a message is received, each message *sent* by a process is also tagged with the current RSN of the sender. Since the execution of each process

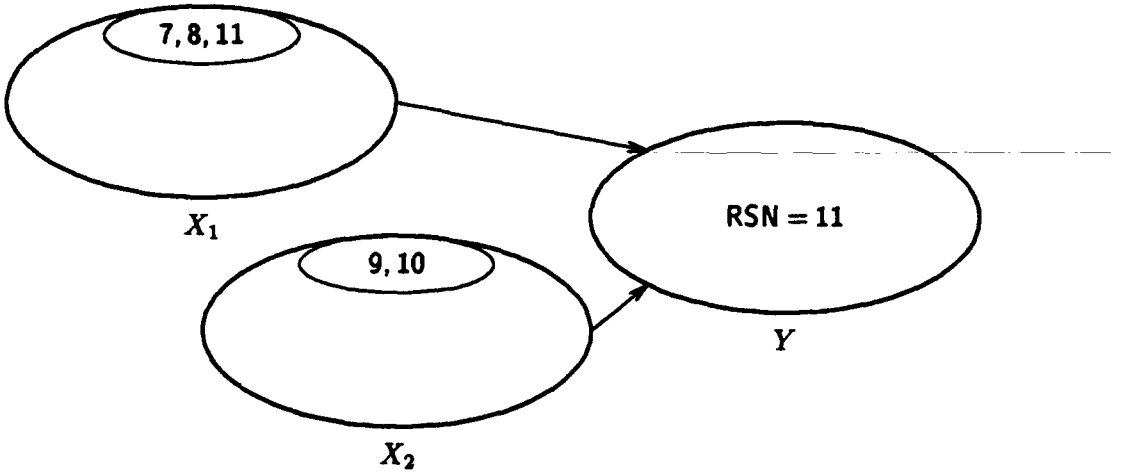

Figure 2 An example message log

between received messages is deterministic, this RSN value uniquely identifies to the receiver the state of the sender from which the message was sent. When a message is received, the state of the receiver then *depends on* this state of the sender, since any part of the sender's state may have been included in the message. Each process records these dependencies locally in a *dependency vector* [Johnson88]. For each process from which this process has received messages, the dependency vector records the *maximum* RSN value tagging a message received from that process. Only the maximum state currently depended on of each other process is recorded, since each state of the sender naturally also depends on all previous states of the same process.

After any failure, the system must be recovered to a consistent state. The state of the system is *consistent* if no process X depends on a state of some other process Y later than Y's maximum state that can be reached through its deterministic execution [Johnson88]. During failure-free operation, the system state is always consistent. During recovery, a process deterministically executes from its checkpointed state, based on the sequence of logged messages that are replayed to it. Thus, the state of the system that can be recovered is consistent if no process X depends on a state of some failed process Y that resulted from Y receiving a message beyond the sequence of messages logged for Y before the failure that are available for replay during recovery. Sender-based message logging uses the dependency vector during recovery to verify that the resulting system state is consistent.

3.2 Data Structures

Sender-based message logging requires the maintenance of the following new data structures for each participating process:

- A *receive sequence number* (*RSN*), numbering messages *received* by the process. This indicates to the sender of a message the order in which that message was received relative to other messages sent to the same process, possibly by other senders. The RSN is incremented each time a new message is received. The *new* value is assigned as the RSN for this message, and is returned to the sender. Each message sent by a process is tagged with the current RSN value of the sending process.

- A *message log* of messages *sent* by the process. For each message sent, this includes the message data, the identification of the destination process, the SSN and RSN used to send the message, and the RSN returned by the receiver. After a process is checkpointed, all messages sent to that process and received before the checkpoint can be removed from the logs in the sending processes.

- A *dependency vector*, recording the maximum state of each other process on which this process currently depends. For each other process from which this process has received messages, the dependency vector stores the maximum RSN value tagging a message *received* from that process.

- An *RSN history list*, recording the RSN value *returned* for each message received by this process since its last checkpoint. For each message received, this includes the identification of the sending process, the SSN used to send the message, and the RSN returned when the

message was received. This list is used when a duplicate message is received, and may be purged when the process is checkpointed.

Each of these data items except the RSN history list must be included when the process is checkpointed. Also, the existing data structures used by a process for duplicate message detection (Section 2) must be included in the checkpoint. When a process is restarted from its checkpoint, the value of each of these data structures in the checkpoint is restored along with the rest of the process state. The RSN history list need not be checkpointed, since messages received before this checkpoint will never be needed for recovery. Only the most recent checkpoint of each process must be saved, and only messages received by a process since its most recent checkpoint must be saved in the message log.

3.3 Message Transmission

Sender-based message logging operates with any existing non-fault-tolerant message transmission protocol of the underlying system. The following steps are required by sender-based message logging to send a message M from process X to process Y:

1. Process X copies message M into its local volatile message log before transmitting M to process Y across the network. The message sent is tagged with the current RSN and SSN values of process X.

2. Process Y receives the message, increments its own RSN value, and assigns this new value as the RSN for M. The entry for process X in process Y's dependency vector is set to the maximum of its current value and the RSN tagging message M, and an entry in process Y's RSN history list is created to record that this new RSN has been assigned to message M. Finally, process Y returns to process X a packet containing the RSN value assigned to message M.

3. Process X adds the RSN for message M to its volatile log, and sends back to process Y a packet containing an acknowledgement for the RSN.

Process Y must periodically retransmit the RSN until its acknowledgement is received, or until process X is determined to have failed. After returning the RSN, process Y may continue execution without waiting for the RSN acknowledgement, but it must not send any messages (including output to the "outside world") until the RSNs of all messages that it has received have been acknowledged. It may or may not experience some delay in execution, depending on whether it tries to send messages immediately after receipt of message M. Process X does not experience any extra delay, but does incur the overhead of copying the message and the RSN to the local volatile log. The operation of this protocol in the absence of retransmissions is illustrated in Figure 3.

This message logging is not an atomic operation. The message data is entered into the log when it is sent, but the RSN can only be recorded after it is received from the target process. Since the sender and the receiver execute on separate nodes in the distributed system, this cannot be completely synchronized with the actual receipt of message. A message is called *partially logged* until the RSN has been recorded in the log. It is then called *fully logged*, or just *logged*.

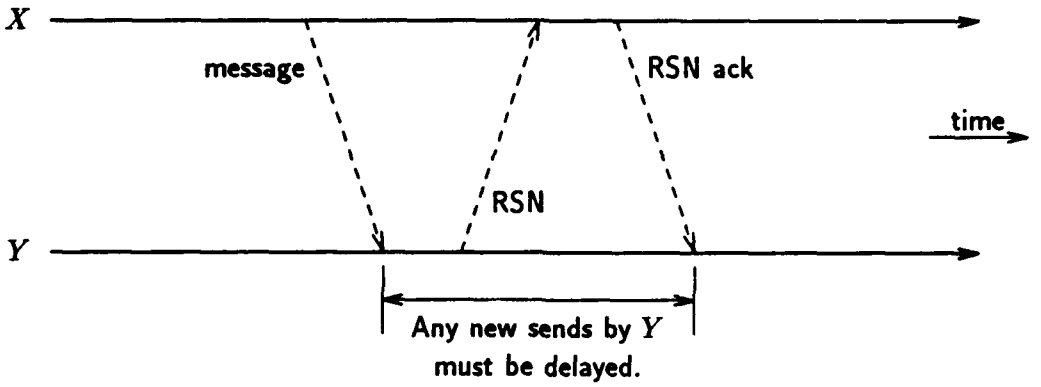

Figure 3 Operation of the message logging protocol in the absence of retransmissions

Sender-based message logging is a *pessimistic* logging protocol, since it prevents the system from entering any state in which a failure could force any process other than those that failed to be rolled back during recovery to achieve a consistent system state. During recovery, a failed process can only be recovered up to its state resulting from its receipt of the sequence of *fully logged* messages sent to it before the failure. Until all RSNs returned by a process have been acknowledged, the process cannot know that its state can be recovered if it should fail. By preventing the process from sending new messages in such a state, no other process can receive a message sent from an unrecoverable state of its sender, and thus, no other process can be forced to roll back during recovery.

Depending on the protocol and network used by the underlying system, a process may receive duplicate messages during failure-free operation. Processes are assumed to detect any duplicate messages on receipt using the SSN tag sent with each message (Section 2). When a duplicate message is received, no new RSN is assigned to the message, and the original RSN assigned when the message was first received is returned to the sender, instead. This RSN is found by searching the RSN history list for an entry with the SSN tag and sending process identification of the message. If the receiver has been checkpointed since originally receiving this message, however, the RSN history list entry for this message will have been purged. In this case, the message cannot be needed in any future recovery of this receiver, since the later checkpoint can always be used. In this case, the receiver instead returns to the sender an indication that this message need not be logged.

3.4 Failure Recovery

To recover a failed process, it is first reloaded on some available processor from its most recent checkpoint. This restores the state of the process to the value it had at the time that the checkpoint was written. Since the data structures used by sender-based message logging (Section 3.2) are included in the checkpoint, they are restored as well.

Next, all fully logged messages originally received by the process after this checkpoint are retrieved from the message logs, beginning with the first message following the current RSN recorded in the checkpoint. Execution of the recovering process is resumed, and these messages are replayed

to it in ascending order of their logged RSNs. The process is not allowed to receive other messages until this sequence has been completely received. Since process execution is deterministic, the process reaches the same state it had after receipt of these messages before the failure.

If only one process has failed, the state of the system must be consistent after the process has received this sequence of fully logged messages during recovery. Since the volatile message log at the sender survives the failure of the receiver, all fully logged messages received by the recovering process before the failure must be available. Since processes are restricted from sending new messages until all messages they have received are fully logged, the recovering process sent no messages before the failure after having received any message beyond this fully logged sequence. Thus, no other process depends on a state of the recovering process beyond its state that can be restored using the available fully logged messages.

If more than one process has failed at a time, though, recovery of a consistent system state may not be possible, since some messages needed for recovery may not be available. Each process can only be recovered up to its state resulting from its receipt of the last message in the fully logged sequence that is still available. This sequence must begin with the first message following the RSN recorded in the checkpoint and must not skip any messages. If any process depends on a state of some failed process beyond this state, the state of the system is not consistent. Sender-based message logging determines this using the dependency vector maintained by each process. For each failed process, if any other process has an entry in its dependency vector that names an RSN of this process greater than the RSN of the last available message in the fully logged sequence, then the state of the system that can be recovered is not consistent In this case, the system may issue a warning announcing this fact, or may abort the application.

As the recovering process reexecutes from its checkpointed state using this sequence of fully logged messages, it resends any messages that it sent before the failure after this checkpoint. Since the current SSN for a process is included in its checkpoint, the SSNs used by this process during recovery are the same as those used when these messages were originally sent. Such duplicates are detected and handled by the same method used during failure-free operation, using the SSN tag in each message (Section 2). For each duplicate received, either the original RSN or an indication that the message need not be logged is returned to the recovering process.

After this sequence of fully logged messages has been received by the recovering process, any partially logged messages destined for it may be resent to it. Also, any new messages that other processes may need to send to it may be sent at this time. These messages may be sent and received in any order after the sequence of fully logged messages has been received. Again, since processes are restricted from sending new messages until all messages they have received are fully logged, the failed process sent no messages before the failure after having received any of these messages. Thus, any effect of their earlier receipt is not visible to any other process, and the order of their receipt now is unimportant.

The system data structures necessary for further participation in sender-based message logging, including any future recovery of other failed processes, are correctly recovered. They are restored from the checkpoint, and then modified as a result of sending and receiving the same sequence of

messages as before the failure. In particular, the volatile log of messages sent by the failed process is recreated as each duplicate message is sent during reexecution.

To guarantee progress in the system in spite of failures, any fault-tolerance method must avoid the *domino effect* [Randell75, Russell80], an uncontrolled propagation of process rollbacks necessary to restore the system to a consistent state following a failure. Sender-based message logging avoids the domino effect, since any failed process can always be recovered from its most recent checkpoint, and no other process must be rolled back during recovery.

3.5 Protocol Optimizations

The protocol described above contains the basic steps necessary for the correct operation of sender-based message logging. However, two optimizations to this basic protocol are possible to reduce the number of packets transmitted. These optimizations combine more information into each packet than would normally be present. They do not alter the logical operation of the protocol as described above, and their inclusion in an implementation of the protocol is optional.

The first of these optimizations is to encode more than one RSN or RSN acknowledgement in a single packet. This is effective when an uninterrupted stream of packets is received from a single sender. For example, when receiving a *blast* bulk data transfer [Zwaenepoel85], the RSN for every data packet of the blast can be returned to the sender in a single packet. This optimization is limited by the distribution of RSNs and RSN acknowledgements that can be encoded in a single packet, but encoding a single contiguous range of each handles the most common case, as in the example above.

The second optimization to the basic sender-based message logging protocol, which further reduces the number of packets necessary, is to *piggyback* RSNs and RSN acknowledgements onto existing message packets, rather than sending them in additional special packets. For example, RSNs can be piggybacked on existing acknowledgment packets used by the underlying system for reliable message delivery. If a message is received that requests the application program to produce some user-level reply to the original sender, the RSN for the request message can be included in the same packet that carries this reply. Likewise, if the sending application program sends a new request to this same receiver shortly after the reply is received, the acknowledgement of this RSN and the RSN for the reply itself can be included in the same packet as this new request. As long as messages are exchanged between the same two processes in this way, no new packets are necessary to return RSNs or their acknowledgements. When this message sequence terminates, one additional packet is needed in each direction to return the RSN and its acknowledgement for the last reply message. The use of this piggybacking optimization is illustrated in Figure 4 for a sequence of these request-reply exchanges. This optimization is particularly useful in systems using *remote procedure call* (RPC) [Birrell84], or other request-response protocols [Cheriton88, Cheriton86], since all communication takes place as a sequence of message exchanges.

This piggybacking optimization may only be used when all unacknowledged RSNs for messages received by the sending process are destined for the same process as the new message packet, and can be included in that packet. When such a packet is received, the piggybacked RSNs and RSN acknowledgements must be handled before the message carried by the packet. When these RSNs

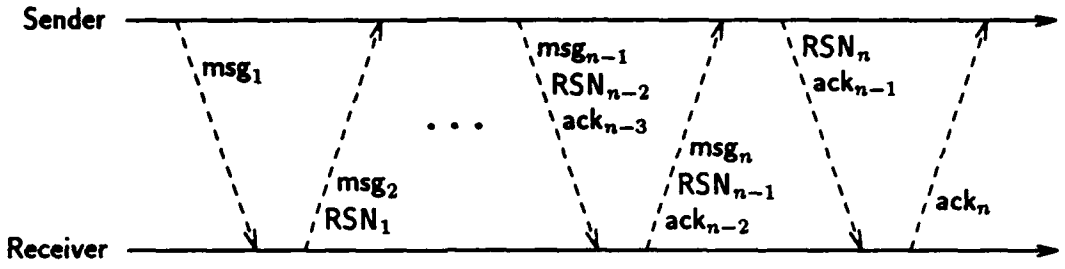

Figure 4 Piggybacking RSNs and RSN acknowledgements on existing message packets

are entered in the message log, the messages for which they were returned become fully logged. Since this packet carries all unacknowledged RSNs from the sender, all messages received by that sender become fully logged before this new message is seen. If the RSNs are not received because the packet is lost in delivery on the network, the new message cannot be received either. The correctness of the protocol is thus preserved by this optimization, since all messages received earlier by the sender are guaranteed to be fully logged before the message in the new packet is seen. Therefore, the state of the sender from which the message was sent must be recoverable, and the recover cannot be forced to roll back during recovery.

These two protocol optimizations can be combined. Continuing the blast protocol example above, the RSN for every data packet of the blast can be encoded together and piggybacked on the reply packet acknowledging the receipt of the blast. If there are n packets of the blast, the unoptimized sender-based message logging protocol requires an additional $2n$ packets to exchange their RSNs and RSN acknowledgements. If both protocol optimizations are combined, instead only one additional packet is required in each direction, to exchange the RSN and RSN acknowledgement for the packet acknowledging the blast. This is illustrated in Figure 5.

4 Implementation

Sender-based message logging has been implemented under the V-System [Cheriton83, Cheriton88] on a collection of diskless Sun workstations connected by an Ethernet to a shared network file

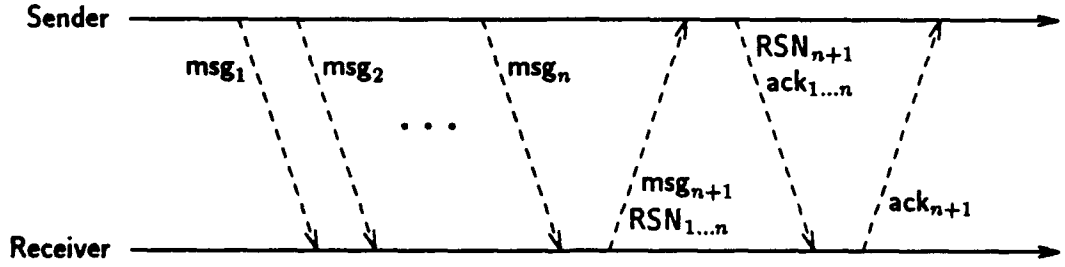

Figure 5 A blast protocol with sender-based message logging using both optimizations

server. This implementation supports the full protocol specified in Section 3, including both protocol optimizations, and supports all V-System message passing operations.

In the V-System, multiple processes may share a single address space to form a *team*, and multiple teams executing together may form a *logical host*. There may be multiple logical hosts executing on each physical node in the system. Since all processes executing as part of the same logical host must be located at the same physical network address, they must remain together through failure and recovery. Therefore, this implementation treats each logical host as a single process in terms of the protocol specification. For example, each logical host is checkpointed as a unit, rather than checkpointing each process individually, and all execution within each logical host is assumed to be deterministic. The implementation is currently limited to a single logical host using sender-based message logging per network node.

4.1 Division of Labor

The implementation is divided between a *logging server* process and a *checkpoint server* process running on each node in the system, and a small collection of support routines in the V-System kernel. The kernel records messages in the log in memory as they are sent, and handles the exchange of RSNs and RSN acknowledgements. This information is carried in normal V kernel packets, and is handled directly by the sending and receiving kernels. This reduces the overhead involved in these exchanges, eliminating any process scheduling delays. All other details of message logging and retrieving logged messages during recovery are handled by the logging server process. The checkpoint server process manages recording checkpoints and restoring them during recovery.

This use of server processes limits the increase in complexity and size of the kernel. In total, only three new primitives to support checkpointing and four new primitives to support message logging were added to the kernel. Also some changes were made to the internal operation of several existing primitives. Table 1 summarizes the amount of executable instructions and data added to the kernel to support checkpointing and message logging for the SUN-3/60 configuration. The percentages given are relative to the size of that part of the base kernel without checkpointing or message logging.

Table 1

Size of kernel additions to support sender-based message logging

	Checkpointing		Message Logging		Total	
	Bytes	Percent	Bytes	Percent	Bytes	Percent
Instructions	3632	4.5	12248	15.1	15880	19.5
Data	0	0.0	36854	18.5	36854	18.5
Total	3632	1.3	49102	17.5	52734	18.8

Within the kernel, a layered implementation is used. The sender-based message logging module acts as a filter on all packets sent and received by the kernel. In a standard V kernel without message logging, packets are composed by one of several routines responsible for network communication and are then passed to the Ethernet device driver for transmission. Incoming packets are passed by the Ethernet interrupt handler to the correct routine based on a type field in each packet. The sender-based message logging module is placed between these upper-level routines and the Ethernet device driver. Packets may be modified or held by this module before transmission, and received packets are interpreted before passing them on. The messages logged by the implementation are actually the contents of each packet passing out through the sender-based message logging module. This organization separates the sender-based message logging protocol processing from the rest of the kernel, largely insulating it from changes in the kernel or its communication protocol. Instead of this layered kernel implementation, integrating the kernel message logging with the existing V kernel protocol handling functions was considered, but was determined to be unnecessary for efficiency.

4.2 Message Log Format

On each node, a single message log is used to store all messages sent by any process executing on that node. The log is organized as a list of fixed-size blocks of message logging data that are sequentially filled as needed by the kernel, and are written to disk by the logging server during a checkpoint. The message log is stored in the volatile address space of the local logging server process. This allows much of the message log management to be performed by the server outside the kernel. Since only one user address space at a time can be accessible, the message log block currently being filled is always double-mapped through the hardware page tables into the kernel address space. This allows new records to be added to the log without switching accessible address spaces, although some address space switches are still necessary to access records in earlier blocks of the log.

Each message log block is 8 kilobytes long, the same size as data blocks in the file system and as hardware memory pages. Each block begins with a 20-byte header, which describes the extent of the space used within the block. The following two types of records are used to describe the logging data in these message log blocks:

`LoggedMessage`: This type of record saves the data of the message, and when the RSN from the receiver arrives, the RSN value, the logical host identifier of the receiver, and the SSN value of the receiver when the message was received are added. It varies in size from 92 to 1116 bytes, depending on the size of any appended data segment that is part of the message.

`AdditionalRsn`: This type of record saves an additional RSN returned for a message logged in an earlier `LoggedMessage` record. It contains the SSN of the message sent, the new RSN value, the logical host identifier of the receiver, and the SSN value of the receiver when the message was received. It is 12 bytes long.

For most messages sent, only the `LoggedMessage` record type is used. However, for messages sent to a process group [Cheriton85], only the first RSN returned from any receiver is stored in

the `LoggedMessage` record, and an `AdditionalRsn` record is created to store each other new RSN returned for this same message. Group messages are delivered reliably to only one receiver, but unreliably to other members of the group. When the message is sent, it is not known how many receivers there will be. Therefore, space for only one RSN is reserved in the `LoggedMessage` record, and new `AdditionalRsn` records are created to save the remainder. Likewise, since messages sent as a datagram are not reliably delivered, it cannot be known when the message is sent if it will be received. Thus, the kernel cannot wait for the RSN to be returned. Instead, the RSN field in the `LoggedMessage` record is not used, and an `AdditionalRsn` record is created to hold the RSN when it arrives, if the message was received.

The standard V kernel uses retransmissions for reliable message delivery, but these retransmissions should not appear as separate messages for message logging. Also, it is important that retransmissions continue during failure recovery, rather than possibly timing out after some maximum number of attempts. Finally, with multiple processes sharing a single team address space, the appended data segment of a message could be changed between retransmissions, which can interfere with successful failure recovery. For these reasons, sender-based message logging performs all necessary retransmissions from the saved copy of the message in the `LoggedMessage` record in the log.

4.3 The Logging Server

At times, the logging server writes modified blocks of the message log from volatile memory to a file on the network file server. A separate file is maintained for the log of messages sent by each logical host. When the kernel adds a new record to a message log block, or modifies an existing record, it flags this mer. .ry block as being modified. These flags are used by the logging server to control the writing of message log blocks from memory to the logging file. The writing of modified blocks is further simplified since blocks in the message log are the same size as blocks in the file system.

This file is updated during a checkpoint of the logical host, so that the log of messages sent by this host can be recovered when the checkpoint is recovered. This is necessary to be able to recover from additional failures after recovery of this process is completed. As such, the file serves as part of the host's checkpoint.

This file may also be written to at other times to reclaim space in the volatile memory of the log. The file may be larger than the space in volatile memory allocated to the message log. When a block has been written from volatile memory to the file, that space in memory may be reused for other logging data if the kernel is no longer retransmitting messages from that block or adding new records to it. When the contents of such a block is again needed in memory, it is read from the file into an unused block in the volatile memory copy of the log. The logging file thus serves as a type of "virtual memory" extension to the message log, and allows more messages to be logged than can fit in the available volatile memory of the node. The kernel automatically requests this when the amount of memory used by the log approaches the allocated limit.

During recovery, the logging server on the node on which recovery is taking place coordinates the replay of logged messages to the recovering logical host. It must collect each message from the

log on the node from which it was sent. This is done by requesting each message from the group of logging server processes. For each message, in ascending order by RSN, a request is sent to the logging server process group, naming this RSN as the next message needed. The server that has this message logged returns it, and replies with the RSN of the next message that it also has logged for the same logical host. When that RSN is needed next during the collection, the request is sent directly to that server rather than to the process group. All servers that do not have the named message logged ignore the request. The sequence of messages to be retrieved is complete when no reply is received to a request sent to the group after the kernel has retransmitted it several times. Then, as the logical host reexecutes from its checkpointed state, the kernel simulates the arrival from the network of each message in this sequence collected.

4.4 The Checkpoint Server

Checkpointing is initiated by sending a request to the local checkpoint server process. This request may be sent by the kernel when the logical host has received a given number of messages or has consumed a given amount of processor time since its last checkpoint. Any process may also request a checkpoint at any time, but this is never necessary. Likewise, failure recovery is initiated by sending a request to the checkpoint server on the node on which the failed logical host is to be recovered. Normally, this request would be sent by the process that detected the failure. However, no failure detection is currently implemented in this system, and the request instead comes from the user.

The checkpoint is written as a file on the shared network file server. On each checkpoint, only the modified pages of the user address space to the checkpoint file, overwriting their previous values in the file. The checkpoint also includes all kernel data used by the logical host, the state of the local *team server* for that logical host, and the state of the local logging server. This data is entirely rewritten on each checkpoint, since it is small and modified portions of it is difficult to detect. Since the file server supports atomic commit of modified versions of files, the most recent complete checkpoint of a logical host is always available, even if a failure occurs while a checkpoint is being written. To limit any interference with the normal execution of the logical host during checkpointing, the bulk of the checkpoint data is written while the logical host continues to execute. The logical host is then *frozen* while the remainder of the data is written to the file [Theimer85].

A failed logical host may be restarted on any node in the network. Other processes sending messages to the recovered logical host determine its new physical network address using the existing V-System mechanism. The kernel maintains a cache recording the network address for each logical host from which it has received packets. In sending packets, after a small number of retransmissions to the old network address, future retransmissions use a dedicated Ethernet multicast address to which all V-System kernels respond. All processes are restored with the same process identifiers that they had before the failure.

5 Performance

The performance of this implementation of sender-based message logging has been measured on a network of diskless SUN-3/60 workstations. The workstations each use a 20-megahertz Motorola MC68020 processor, and are connected by a 10 megabit per second Ethernet network to a single shared network file server. The file server runs on a SUN-3/160, with a 16-megahertz MC68020 processor, and uses a Fujitsu Eagle disk. This section presents an analysis of the individual costs involved in communication, checkpointing, and recovery in this implementation, and an evaluation of the performance of several distributed application programs using sender-based message logging.

5.1 Communication Costs

Table 2 presents the average time in milliseconds required for common communication operations in the V-System. This table shows the times for a **Send-Receive-Reply** exchange with no appended data and with a 1-kilobyte appended data segment, for a **Send** as a datagram, and for **MoveTo** and **MoveFrom** of both 1 and 64 kilobytes of data. For each operation, the times are given for a V kernel without sender-based message logging and for the same operation using this implementation. The overhead of using sender-based message logging is given for each operation as the difference between these two times, and as a percentage increase over the standard time. These times are measured in the initiating user process, and indicate the elapsed time from invoking the operation to its completion. The overhead for most operations is approximately 25 percent.

This measured overhead is caused entirely by the time necessary to execute the instructions of the sender-based message logging protocol implementation. Because of the request-response nature of the V-System communication operations, and due to the presence of the protocol optimizations described in Section 3.5, no extra packets were required for each operation, and no delays were incurred in the transmission of any packet while waiting for an RSN acknowledgement to arrive.

Table 2

Performance of common V-System communication operations with sender-based message logging (milliseconds)

Operation	Message Logging		Overhead	
	With	Without	Time	Percent
Send-Receive-Reply	1.9	1.4	.5	36
Send(1K)-**Receive-Reply**	3.4	2.7	.7	26
Datagram **Send**	.5	.4	.1	25
MoveTo(1K)	3.5	2.8	.7	25
MoveTo(64K)	107.0	88.0	19.0	22
MoveFrom(1K)	3.4	2.7	.7	26
MoveFrom(64K)	106.0	87.0	19.0	22

Two extra packets were required to exchange the final RSN and RSN acknowledgement for each test sequence, but this occurred asynchronously within the kernel, after the user process had completed the timing.

To better understand how this execution time is spent, the execution times for a number of components of the implementation were measured. This was done by independently executing each component in a loop a large number of times and averaging the results. The time for a single execution could not be measured directly, since the SUN lacks a hardware clock of sufficient resolution. The packet transmission portion of the implementation requires approximately 126 microseconds for packets of minimum size, including 27 microseconds to copy the packet into the log. For sending a packet with a 1-kilobyte appended data segment, this time is increased by 151 microseconds for the additional time required to copy the segment into the log. Within this sending time, 38 microseconds occurs after the packet is transmitted on the Ethernet, and executes concurrently with the packet reception on the remote node. The packet reception portion of the implementation requires approximately 142 microseconds. Of this time, about 39 microseconds is spent processing the RSN piggybacked on the packet, and about 45 microseconds is spent processing the RSN acknowledgement.

These measurements agree well with the overhead times shown in Table 2 required to execute the protocol implementation for each operation. For example, for a **Send-Receive-Reply** with no appended data segment, one minimum-sized packet is sent by each process. The sending protocol executes concurrently with the receiving protocol for each packet after its transmission on the network. The total sender-based message logging overhead for this operation is calculated as

$$2\,((126 - 38) + 142) = 460 \text{ microseconds.}$$

This closely matches the measured value of 500 microseconds given in the table. The time beyond this required to execute the protocol for a 1-kilobyte appended segment **Send-Receive-Reply** is simply the extra 151 microseconds needed to copy the segment into the message log, and is close to the difference for these two operations of 200 microseconds given in the table. As a final example, for a 64-kilobyte **MoveTo** operation, 64 packets with 1 kilobyte of data each are sent, followed by a reply packet of minimum size. No concurrency is possible in the sending of the first 63 data packets, but they are each received concurrently with the following send. After the last data packet is transmitted, and after the reply packet is transmitted, execution of the protocol proceeds concurrently between the sender and receiver. The total calculated overhead for this operation is 18.062 milliseconds, which agrees well with the measured overhead of 19 milliseconds given in the table.

In less controlled environments and with more than two processes communicating, additional overhead may be caused by any time spent idle waiting for an RSN acknowledgement to arrive before a new message can be sent. To understand the effect of this delay on the communication overhead, the average round-trip time required to send an RSN and receive its acknowledgement was measured. Without transmission errors, this delay should not exceed the round-trip time, but may be less if the RSN has already been sent when the new packet transmission is first attempted. The average round-trip time required in this environment is 550 microseconds. Although the same

amount of data is transmitted across the network, this time is significantly less than the 1.4 milliseconds shown in Table 2 for **Send-Receive-Reply**, since the RSN exchange takes place directly between the two kernels rather than between two processes at the user level.

To put this measured average communication overhead of 25 percent with sender-based message logging into perspective, the performance of several other pessimistic message logging techniques may be considered. If messages are logged on some separate logging node on the network, without using special network hardware, the overhead should be approximately 100 percent, since all packets must be sent and received one extra time. The TARGON/32 system, using special networking hardware assistance and available idle time on a dedicated processor of each multiprocessor node, claims a total system overhead for the provision of fault tolerance of approximately 10 percent [Borg89]. If each message is synchronously written to disk as it is received, the overhead should be several orders of magnitude higher, due to the relative speed of the disk. This approach does allow for recovery from multiple failures at once, though.

5.2 Checkpointing Costs

Table 3 shows the measured elapsed time for performing checkpoints in this implementation, based on the size of the address space portion that was written to the checkpoint file. During a checkpoint, each contiguous range of modified pages is written to the file in a separate operation. The time required to write the address space thus increases as the number of noncontiguous ranges of modified pages increases. For these measurements, the modified pages were each separated by one unmodified page, resulting in the most separate write operations. The hardware page size on the SUN-3 is 8 kilobytes.

These measurements illustrate that the cost of checkpointing is dominated by the cost of writing the address space to the checkpoint file. The elapsed time required to complete the checkpoint grows roughly linearly with the size of the address space portion written to the file. Other costs involved in checkpointing are minor. A total of approximately 17 milliseconds is required to open

Table 3

Sender-based message logging checkpointing time by size of address space portion written (milliseconds)

Kilobytes	Pages	Time
8	1	140
16	2	170
32	4	220
64	8	300
128	16	500
256	32	880
512	64	1570
1024	128	2980

the checkpoint file and later close it. Checkpointing the state of the kernel requires .8 milliseconds, and checkpointing the team server requires 1.3 milliseconds. The time required to checkpoint the logging server varies with the number of message log blocks to be written to the logging file, with a minimum of 18 milliseconds, and increasing approximately 25 milliseconds per block.

5.3 Recovery Costs

The costs involved in recovery are similar to those involved in checkpointing. The address space of the logical host being recovered must be read from the checkpoint file into memory. The state of the kernel must be restored, as well as the states of the team server and the logging server. In addition, the sequence of messages received by this logical host after the checkpoint was written must be retrieved, and the logical host must complete any reexecution based on these logged messages necessary to bring its state up to the value that it had at the time of the failure.

Table 4 shows the measured times required for recovery in this implementation based on the size of the address space of the logical host being recovered. These measurements do not include any time required for the logical host to reexecute from the checkpointed state, since this time is specific to the particular application being recovered. In general, this reexecution time is bounded by the interval at which checkpoints are recorded. As with the cost of checkpointing, these measured recovery times given in Table 4 vary approximately linearly with the size of the address space being read. There is also a large fixed cost included in each of these times, due to the necessary timeout of the last group send of the request to collect the next logged message. In the current implementation, this timeout is 2.5 seconds.

5.4 Application Program Performance

The preceding three sections have examined the three sources of overhead caused by the operation of sender-based message logging. Distributed application programs, though, spend only a portion of their execution time on communication, and checkpointing and failure recovery occur only infre-

Table 4

Sender-based message logging recovery time by address space size (milliseconds)

Kilobytes	Pages	Time
8	1	2580
16	2	2600
32	4	2620
64	8	2670
128	16	2760
256	32	2950
512	64	3320
1024	128	4080

quently. To analyze the overhead of sender-based message logging in a more realistic environment, the performance of three distributed application programs was measured using this implementation. The following three application programs were used in this study:

nqueens: This program counts the number of solutions to the *n-queens problem* for a given number of queens n. The problem is distributed among multiple processes by assigning each a range of subproblems to solve resulting from an equal division of the possible placements of the first two queens. When each process finishes all allocated subproblems, it reports the number of solutions found to the main process.

tsp: This program finds the minimum solution to the *traveling salesman problem* for a given map of n cities. The problem is distributed among multiple processes by giving each a different initial edge to include in all paths. A branch-and-bound technique is used. As each new possible solution is found, it is reported to the main process, which records the minimum solution and returns the length of the currently known minimum solution. When a process finishes its assigned search, it requests a new graph edge to search from.

gauss: This program performs *Gaussian elimination with partial pivoting* on a given $n \times n$ matrix of real numbers. The problem is distributed among multiple processes by giving each a subset of the matrix rows to operate on. At each step of the reduction, the processes send their possible pivot row number and value to the main process, which determines the row to be used. The current contents of the pivot row is sent to all other processes, and each process performs the reduction on its rows. When the last reduction step completes, each process returns its rows to the main process.

These three programs were chosen because of their dissimilar communication rates and patterns. With **nqueens**, the main process exchanges a message with each other process at the start of execution and again at completion, but there is no other communication during execution. The subordinate processes do not communicate with one another, and the total amount of communication is constant for all problem sizes. With **tsp**, the map is initially distributed to the subordinate processes, which then communicate with the main process to requests new subproblems and to report any new results found during each search. Since the number of subproblems is bounded by the number of cities in the map, the total amount of communication performed is $O(n)$ for a map of n cities, but due to the branch-and-bound algorithm used, the running time is highly dependent on the map input. Again, there is no communication between subordinate processes during execution. The **gauss** program performs the most communication of the three programs, including communication between all processes during execution. The matrix rows are distributed to the subordinate processes and collected at completion, each pivot row is decided by the main process, and the contents of the pivot row is distributed to all other processes. The total amount of communication performed is $O(n^2)$ for an $n \times n$ matrix.

These three application programs were used to solve a set of problems. Each problem was solved multiple times, both with and without using sender-based message logging. In each case, the same set of problems was solved. The maps used for **tsp** and the matrices used for **gauss** were randomly generated, but were saved for use on all executions. For each program, the problem

was distributed among 8 processes, each executing on a separate node of the system. When using sender-based message logging, all messages sent between application processes were logged, but no checkpointing was performed.

The overhead of using sender-based message logging ranged from about 2 percent to much less than 1 percent for most problems in this set. For the **gauss** program, which performs more communication than the other two programs, the overhead was slightly higher, ranging from about 16 percent to about 3 percent. As the problem size increases for each program, the overhead decreases because the average amount of computation between messages sent increases. Table 5 summarizes the performance of these application programs for all problems in this set. Each entry in this table shows the application program name and the problem size n. The running times in seconds required to solve each problem are given, both with and without using sender-based message logging. The overhead of using sender-based message logging for each problem is shown in seconds as the difference between its two running times, and as a percentage increase in the running time without logging.

Table 6 shows the average message log sizes per node that result from solving each of these problems using sender-based message logging. For each problem, the average total number of messages logged and the resulting message log size in kilobytes is shown. These figures are also shown averaged over the elapsed execution time in seconds to solve the problem. These message log sizes are all reasonable, and are well within the limits of available memory on the workstations used in these tests and on other similar contemporary machines.

The effectiveness of the piggybacking protocol optimization (Section 3.5) depends on the communication pattern used during execution. To assess the effectiveness of piggybacking with these application programs, its utilization was counted within each process for each message sent. This

Table 5

Performance of the distributed application programs using sender-based message logging (seconds)

Program	Size	Message Logging		Overhead	
		With	Without	Time	Percent
nqueens	12	5.99	5.98	.01	.17
	13	34.61	34.60	.01	.03
	14	208.99	208.98	.01	.01
tsp	12	5.30	5.19	.11	2.12
	14	16.40	16.13	.27	1.67
	16	844.10	841.57	2.53	.30
gauss	100	12.41	10.74	1.67	15.55
	200	71.10	66.40	4.70	7.08
	300	224.06	217.01	7.05	3.25

Table 6

Message log sizes for the distributed application programs using sender-based message logging (average per node)

Program	Size	Total		Per Second	
		Messages	Kilobytes	Messages	Kilobytes
nqueens	12	8	1.9	1.30	.32
	13	8	1.9	.23	.06
	14	8	1.9	.04	.01
tsp	12	43	5.5	8.09	1.04
	14	48	6.1	2.91	.37
	16	59	7.3	.07	.01
gauss	100	514	95.4	41.44	7.69
	200	1113	292.8	15.66	4.12
	300	1802	593.7	8.04	2.65

piggybacking utilization is summarized in Table 7 by the percentage of messages sent, averaged over all processes. The three possible cases encountered when sending a message are shown individually in the table. If no unacknowledged RSNs are pending, the message is sent immediately with no piggybacked RSNs. If all unacknowledged RSNs can be included in the same packet, the are piggybacked in it, and the message is sent immediately. Otherwise, the packet cannot be sent now and must wait for the acknowledgment of previous RSNs. In all applications, more than half the messages could be sent immediately. Without piggybacking, about half of these would have been forced to wait for the separate acknowledgment of the RSNs piggybacked in the same packet as the message. The utilization of piggybacking was lowest in the **gauss** program, since its communication pattern allowed all processes to communicate with one another. This reduces the chances that a message being sent is destined for the same process as the pending RSNs, which is required to be able to piggyback those RSNs on the message.

To evaluate the effect of checkpointing on the overhead of sender-based message logging, each application program was executed again with checkpointing enabled. Each program was used to solve its largest problem, with checkpoints being written by each process every 15 seconds. A high checkpointing frequency was used to generate a significant amount of checkpointing activity to be measured. For the **nqueens** and **tsp** programs, the additional overhead from this level of checkpointing was less than .5 percent of the required running time for that application with sender-based message logging. For the **gauss** program, checkpointing overhead was about 2 percent. This is higher than for the other two programs because **gauss** uses more data during execution, which must be written to the checkpoint. In all cases, the average additional time required for each checkpoint was much less than the elapsed time required for checkpointing alone, reported in

Table 7

Application program piggybacking utilization (percentage of messages sent)

Program	Size	None Pending	Piggybacked	Wait For RSN Ack
nqueens	12	42.9	44.4	12.7
	13	41.3	46.0	12.7
	14	41.3	46.0	12.7
tsp	12	19.9	62.4	17.6
	14	22.3	63.5	14.2
	16	33.0	58.3	8.7
gauss	100	20.7	30.0	49.2
	200	29.4	28.6	42.0
	300	29.0	31.0	40.0

Table 3 because the time spent in checkpointing waiting for writes to the disk to complete could be overlapped with execution of the application.

6 Multiple Failure Recovery

Sender-based message logging is designed to recover from only a single failure at a time within the system, since messages are logged in volatile memory. That is, after one process (or logical host in the V-System) has failed, no other process may fail until the recovery from the first is completed. Sender-based message logging can be extended, though, to recover from multiple failures at the same time.

Sender-based message logging uses the dependency vector of each process to verify that the resulting system state after recovery is consistent. If a process X depends on a state of some failed process Y that resulted from Y receiving a message beyond those messages logged for Y that are available for replay during recovery, the system state that can be recovered by the basic sender-based message logging protocol is not consistent. To recover a consistent system state in this situation would require each such process X to be rolled back to a state before it received the message from Y that caused this dependency.

With the existing sender-based message logging protocol, this consistent system state can be recovered only if the current checkpoint for each process X that must be rolled back was written before this message from Y was received. In this case, each process X can be rolled back by forcing it to fail and recovering it using this checkpoint. If the current checkpoint was written after the message from Y was received, process X cannot roll back far enough to remove the dependency.

To preserve as much of the existing volatile message log as possible, each of these processes must be rolled back one at a time. As the original failed processes reexecute based on the sequences of messages that could be replayed, they will resend any messages they sent before the failure, and

thus recreate much of their original volatile message log that was lost from the failure. Then, as each of these additional processes is forced to fail and recovered from its earlier checkpoint, it will recreate its volatile message log during its reexecution as well. By rolling them back one at a time, no additional logged messages needed for their reexecution from their checkpointed states will be lost.

If the checkpoint for some process X that must be rolled back was not written early enough to allow the process to roll back to before the dependency on process Y was created, recovery of a consistent system state using the basic sender-based message logging protocol is not possible. To guarantee the recovery of a consistent system state even in this case, the sender-based message logging can be modified to retain on stable storage all checkpoints for all processes, rather than just saving the most recent one for each process. Then, the existence of a usable checkpoint for each process X that must be rolled back is ensured. Although not all checkpoints must be retained to guarantee recovery, the existing sender-based message logging protocol does not maintain sufficient information to determine during failure-free execution when each can safely be released. The domino effect is still avoided by this extension since the data in the checkpoints is not volatile.

7 Related Work

Many fault-tolerance systems require application programs to be written according to specific computational models to simplify the provision of fault tolerance. For example, the ARGUS [Liskov88, Liskov87] and Camelot [Spector87] systems require applications to be structured as a (possibly nested) set of atomic actions on abstract data types. Likewise, some systems, such as the Tandem NonStop system [Bartlett81], require the programmer to embed fault-tolerance provision into each application. Since sender-based message logging is a transparent mechanism, it does not impose such restrictions on the applications.

The sender-based message logging protocol differs in design from other message logging protocols primarily in that messages are logged in the local *volatile memory* of the *sender*. Also, sender-based message logging requires no specialized hardware to assist the logging. The TARGON/32 system [Borg89] (and its predecessor, Auros [Borg83]) log messages at a backup node for the receiver, using specialized networking hardware that provides three-way atomic broadcast of each message to the backup process of the sender and the primary and backup processes of the receiver. The PUBLISHING mechanism uses a centralized logging node for all messages, which must reliably receive every network packet. Although this logging node avoids the need to send an additional copy of each message over the network, providing the guarantee of reliably receiving each message seems to be impractical without additional protocol complexity [Saltzer84]. Strom and Yemini's Optimistic Recovery mechanism logs all messages on stable storage on disk. With sender-based message logging, logging the messages directly at the sender avoids the expense of sending an extra copy of each for logging, and since sender-based message logging requires no such specialized hardware, it can be used over a broader class of existing systems without loss of efficiency.

Another difference of sender-based message logging from previous pessimistic logging protocols, which is not related to the logging of messages at the sender, is in the way that the requirements of

a pessimistic logging protocol are enforced. All pessimistic logging protocols must guarantee that during recovery processes other than those that failed need not be rolled back to achieve a consistent system state. Previous pessimistic logging protocols [Borg83, Powell83, Borg89] have required each message to be logged *before it is received* by the destination process, blocking the receiver while the logging takes place. Sender-based message logging allows the message to be received before it is logged, but prevents the receiver from *sending new messages* until all messages it has received are logged. This allows the receiver to execute based on the message data while the logging takes place asynchronously. For example, if the message requests some service of the receiver, this service can begin while the message is being logged.

Optimistic message logging methods [Strom85, Johnson88] have the potential to outperform pessimistic methods, since message logging proceeds asynchronously, without delaying either the sender or the receiver for message logging to complete. However, these methods require significantly more complex protocols for logging, since each process must be notified of the progress of the logging of messages received by each other process. Also, failure recovery in these systems is more complex and may take longer to complete, since processes other than those that failed may need to be rolled back to recover a consistent system state. Finally, optimistic message logging systems may require significantly more storage during failure-free operation, since logged messages may need to be retained longer, and processes may be required to save additional checkpoints earlier than their most recent. Sender-based message logging achieves some of the advantage of asynchronous logging more simply by allowing messages to be received before they are fully logged.

Some of the simplicity of the sender-based message logging protocol results from the concentration on recovering from only a single failure at a time. This allows the messages to be logging in volatile memory, significantly reducing the overhead of logging. The addition of the extensions of Section 6 to handle multiple failures causes no additional overhead during failure-free operation, although to guarantee recovery using the second extension requires that all checkpoints be retained on stable storage. Also, the recovery from multiple failures at once using these extensions may require longer to complete than with other methods, since any processes other than those that failed that must be rolled back must do so one at a time.

This work improves on our earlier work with sender-based message logging, which was reported before the system had been implemented [Johnson87]. The protocol optimizations of Section 3.5 result in a significant reduction in the number of extra network packets required for message logging. Sender-based message logging now also detects all cases when the system cannot be recovered to a consistent state following a failure. Through the addition of the dependency vector in each process, any multiple failure that causes the loss of portions of the volatile message log needed for recovery is detected, allowing the system to notify the user or abort the computation if desired. Furthermore, the extensions of Section 6 allow the system to be recovered in cases of multiple failures in which recovery previously would not have been possible using sender-based message logging.

8 Conclusion

Sender-based message logging is a new transparent method of providing fault tolerance in distributed systems, which uses *pessimistic* message logging and checkpointing to record information for recovering a consistent system state following a failure. It differs from other pessimistic message logging protocols in that the message log is stored in the *volatile* memory on the node from which the message was *sent*. The order in which the message was *received* relative to other messages sent to the same receiver is required for recovery, but this information is not usually available to the message sender. With sender-based message logging, when a process receives a message, it returns to the sender a *receive sequence number*, or *RSN*, to indicate this ordering information. When the RSN arrives at the sender, it is added to the local volatile log with the message. To recover a failed process, it is restarted from its most recent checkpoint, and the sequence of messages received by it after this checkpoint are replayed to it in ascending order of their logged RSNs.

Sender-based message logging concentrates on reducing the overhead placed on the system for the provision of fault tolerance by a pessimistic logging protocol. The cost of message logging is the most important factor in this system overhead. The checkpointing frequency can be tuned to balance its expense against the time needed for recovery or the space needed to store the message log, and the cost of failure recovery should be less important if failures in the system are infrequent.

Keeping the message log in the sender's volatile memory avoids the expense of synchronously writing each message to disk or sending an extra copy over the network to some special logging process. Since the message log is volatile, though, sender-based message logging supports recovery from only a single failure at a time within the system. Extensions to the basic sender-based message logging protocol are also possible to handle multiple failures.

Performance measurements from a full implementation of sender-based message logging under the V-System verify the efficient nature of this protocol. Measured on a network of SUN-3/60 workstations, the overhead on V-System communication operations is approximately 25 percent. The overhead experienced by distributed applications programs using sender-based message logging is affected most by the amount of communication performed during execution. For Gaussian elimination, the most communication-intensive program measured, this overhead ranged from about 16 percent to about 3 percent, for different problem sizes. For all other programs measured, overhead ranged from about 2 percent to much less than 1 percent.

Acknowledgements

We would like to thank Ken Birman, Rick Bubenik, John Carter, David Cheriton, Gerald Fowler, Ed Lazowska, and Rick Schlichting for their helpful comments on earlier drafts of this material.

References

[Bartlett81] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 22–29. ACM, December 1981.

[Bernstein87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Borg83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99. ACM, October 1983.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[Cheriton83] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[Cheriton85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[Cheriton86] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the 1986 SigComm Symposium*, pages 406–415. ACM, August 1986.

[Cheriton88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[Johnson88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[Lampson79] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

[Liskov87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122. ACM, November 1987.

[Liskov88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Powell83] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[Randell75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Russell80] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[Schlichting83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[Spector87] Alfred Z. Spector. Distributed transaction processing and the Camelot system. In *Distributed Operating Systems: Theory and Practice*, edited by Yakup Paker, Jean-Pierre Banatre, and Müslim Bozyiğit, volume 28 of *NATO Advanced Science Institute Series F: Computer and Systems Sciences*, pages 331–353. Springer-Verlag, Berlin, 1987.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[Theimer85] Marvin N. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12. ACM, December 1985.

[Zwaenepoel85] Willy Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32. IEEE Computer Society, September 1985.

Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

This paper has been submitted to and is under revision for publication in *Journal of Algorithms.*

An earlier version appeared in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing.*

Abstract

In a distributed system using message logging and checkpointing to provide fault tolerance, there is always a unique maximum recoverable system state, called the *current recovery state.* This paper presents an efficient algorithm for determining the current recovery state at any time, and proves its correctness. The algorithm is based on a general model for reasoning about these recovery methods. This model allows us to show that the set of system states that have occurred during any single execution of a system forms a lattice, with the sets of consistent and recoverable system states as sublattices. The current recovery state is the greatest upper bound of all elements of the recoverable sublattice. This work unifies existing approaches to fault tolerance using message logging and checkpointing, and improves on existing methods for optimistic recovery in distributed systems.

This work was supported in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under contract ONR N00014-88-K-0140.

1 Introduction

Message logging and checkpointing can be used to provide fault tolerance in a distributed system in which all process communication is through messages. Each message received by a process is logged on stable storage [Lampson79], and each process is occasionally checkpointed to stable storage, but no coordination is required between the checkpoints of different processes. Between received messages, the execution of each process is assumed to be deterministic.

The protocols used for message logging are typically *pessimistic*. With these protocols, each message is synchronously logged as it is received, either by blocking the receiver until the message is logged [Borg83, Powell83], or by blocking the receiver if it attempts to send a new message before all received messages are logged [Johnson87]. Recovery based on pessimistic message logging is straightforward. A failed process is restarted from its last checkpoint, and all messages received by this process after the checkpoint are replayed to it from the log, in the same order as they were received before the failure. The process reexecutes based on these messages to its state at the time of the failure. Messages sent by the process during recovery are ignored since they are duplicates of those sent before the failure.

On the other hand, *optimistic* message logging protocols operate asynchronously [Strom85]. The receiver continues to execute, and received messages are logged later, for example by grouping several messages and writing them to stable storage in a single operation. The state of a process can only be recovered, however, if all messages that it has received since its last checkpoint have been logged. When a process receives a message, the state of the receiver becomes dependent on the state of the sender at the time that the message was sent. If the sender fails and can only be recovered to an earlier state, the receiver process becomes an *orphan* and must be rolled back during recovery to a point before the message that created this dependency was received. Rolling back this process may cause other processes to become orphans, which must also be rolled back during recovery. The *domino effect* [Randell75, Russell80] is an uncontrolled propagation of such rollbacks, and must be avoided to guarantee progress in the system in spite of failures. Recovery based on optimistic message logging must, in effect, construct the "most recent" combination of process states such that no process is an orphan.

Optimistic message logging protocols are desirable in systems in which failures are rare and failure-free performance is of primary concern. Since they avoid synchronization delays during message logging, performance in the absence of failures is improved. Although the required recovery procedure is more complicated, it is only used when a failure occurs.

This paper presents an algorithm for determining the maximum recoverable state at any time in a system using message logging and checkpointing. Our algorithm can be used with any message logging protocol, whether pessimistic or optimistic, but its full generality is only required with optimistic logging protocols. Section 2 presents a general model for reasoning about these recovery methods. With it, we prove that the set of recoverable system states forms a lattice, and that there is thus always a unique maximum recoverable system state, which never decreases. Based on this model, Section 3 describes our algorithm for finding the maximum recoverable system state, and proves its correctness. Section 4 relates this work to existing message logging and checkpointing fault-tolerance methods. Finally, Section 5 summarizes the contributions of this work.

2 The Model

This section presents a general model for reasoning about the behavior and correctness of recovery methods using message logging and checkpointing. The model does not assume the use of any particular message logging protocol, and is based on the notion of *dependency* between the states of processes as a result of communication in the system.

2.1 Process States

Each time a process receives an input message, it begins a new *state interval*, a deterministic sequence of execution based only on the state of the process at the time that the message is received and on the contents of the message itself. Within each process, each state interval is uniquely identified by a sequential *state interval index*, which is simply a count of the number of input messages that the process has received. The creation of a process is modeled as its receipt of message number 0, and process termination is modeled as its receipt of one final message following the sequence of real input messages to be received by the process.

When a process i receives a message sent by some process j, the state of process i becomes dependent on the state of process j at the time the message was sent. All dependencies of process i on any state of process j can be encoded as the maximum index of any state interval of process j on which process i depends. This encoding is possible since the execution of a process within each state interval is deterministic and since any state interval in a process naturally depends on all previous intervals of the same process.

The dependencies of process i on *all* processes can be represented by a *dependency vector*

$$\langle \delta_* \rangle = \langle \delta_1, \delta_2, \delta_3, \ldots, \delta_n \rangle \ ,$$

where n is the total number of processes in the system. Component j of process i's dependency vector, δ_j, gives the maximum index of any state interval of process j on which process i currently depends. If process i has no dependency on any state interval of some process j, then δ_j is set to $\perp$, which is less than all possible state interval indices. Component i of process i's own dependency vector is always set to the index of process i's current state interval.

Processes cooperate to maintain their dependency vectors by tagging all messages sent with their current state interval index, and by remembering in each process the maximum index tagging any message received from each other process. During any single execution of the system, the current dependency vector for any process is uniquely determined by the state interval index of the process. No component of the dependency vector of any process can decrease through failure-free execution of the process.

2.2 System States

A system state is a collection of process states, one for each process in the system. These process states need not all have existed in the system at the same time. A system state is said to have *occurred* during some execution of the system if all component process states have each individually occurred during this execution. A system state is represented by an $n \times n$ *dependency matrix*

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} \delta_{11} & \delta_{12} & \delta_{13} & \cdots & \delta_{1n} \\ \delta_{21} & \delta_{22} & \delta_{23} & \cdots & \delta_{2n} \\ \delta_{31} & \delta_{32} & \delta_{33} & \cdots & \delta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \delta_{n1} & \delta_{n2} & \delta_{n3} & \cdots & \delta_{nn} \end{bmatrix} ,$$

where row i, δ_{ij}, $1 \leq j \leq n$, is the dependency vector for the state of process i included in this system state. Since component i of process i's dependency vector is always the index of process i's current state interval, the diagonal of the dependency matrix, δ_{ii}, $1 \leq i \leq n$, shows the current state interval index of each process contained in the system state.

Let $\mathcal{S}$ be the set of all system states that have occurred during any *single* execution of some system. The *system history relation* defines a partial order on the set $\mathcal{S}$ such that one system state precedes another in this relation if and only if it *must* have occurred first during this execution. This relation can be expressed in terms of the state interval index of each process as shown in the dependency matrices representing these system states.

Definition 1 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then

$$\mathbf{A} \preceq \mathbf{B} \iff \forall i \, [\alpha_{ii} \leq \beta_{ii}] .$$

The system history relation differs from Lamport's *happened before* relation [Lamport78] in that it orders the system states that result from events rather than the events themselves, and that only state intervals (started by the receipt of a message) constitute events.

For example, Figure 1 shows a system of four communicating processes. The horizontal lines represent the execution of each process, each arrow represents a message from one process to another, and the numbers give the index of the state interval started by the receipt of each message. Consider the two possible system states **A** and **B**, where in state **A**, message a has been received

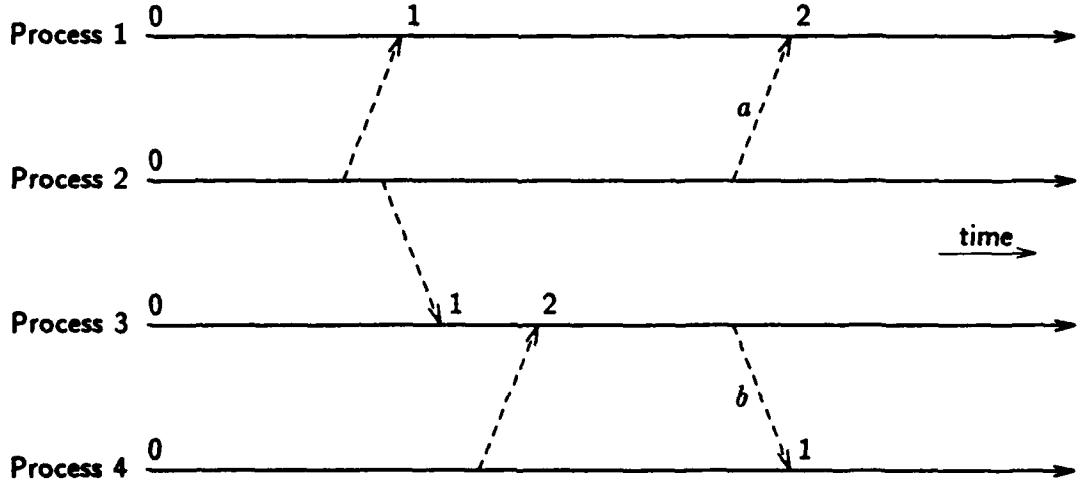

Figure 1 Neither message a nor message b must have been received first.

but message b has not, and in state **B**, message b has been received but message a has not. These system states can be expressed by the dependency matrices

$$\mathbf{A} = \begin{bmatrix} \textcircled{2} & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & \bot & \textcircled{0} \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} \textcircled{1} & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & 2 & \textcircled{1} \end{bmatrix}.$$

States **A** and **B** are incomparable under the system history relation. This can be seen by comparing the circled values on the diagonals of these two dependency matrices.

2.3 The System History Lattice

A system state describes the set of messages that have been received by each process. Two system states in $\mathcal{S}$ can be combined to form their *union* such that each process has received all messages that it has in either of the two original system states. This can be expressed in terms of the dependency matrices describing these system states by choosing, for each process, the row that has the *largest* state interval index of the corresponding rows in the original matrices.

Definition 2 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then the *union* of **A** and **B** is $\mathbf{A} \cup \mathbf{B} = [\gamma_{**}]$, such that

$$\forall i \left[\gamma_{i*} = \begin{cases} \alpha_{i*} & \text{if } \alpha_{ii} \geq \beta_{ii} \\ \beta_{i*} & \text{otherwise} \end{cases} \right].$$

Likewise, the *intersection* of two system states in $\mathcal{S}$ can be formed such that each process has received only those messages that it has in both of the two original system states. This can be expressed in terms of the dependency matrices describing these system states by choosing, for each process, the row that has the *smallest* state interval index of the corresponding rows in the original matrices.

Definition 3 If $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$ are system states in $\mathcal{S}$, then the *intersection* of **A** and **B** is $\mathbf{A} \cap \mathbf{B} = [\delta_{**}]$, such that

$$\forall i \left[\delta_{i*} = \begin{cases} \alpha_{i*} & \text{if } \alpha_{ii} \leq \beta_{ii} \\ \beta_{i*} & \text{otherwise} \end{cases} \right].$$

Continuing the example of Section 2.2 illustrated in Figure 1, the union and intersection of states **A** and **B** are

$$\mathbf{A} \cup \mathbf{B} = \begin{bmatrix} 2 & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & 2 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{A} \cap \mathbf{B} = \begin{bmatrix} 1 & 0 & \bot & \bot \\ \bot & 0 & \bot & \bot \\ \bot & 0 & 2 & 0 \\ \bot & \bot & \bot & 0 \end{bmatrix}.$$

The following theorem introduces the *system history lattice* formed by the set of system states that have occurred during any *single* execution of some system, ordered by the system history relation.

Theorem 1 The set $\mathcal{S}$, ordered by the system history relation, forms a lattice. For any $\mathbf{A}, \mathbf{B} \in \mathcal{S}$, the *least upper bound* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cup \mathbf{B}$, and the *greatest lower bound* of $\mathbf{A}$ and $\mathbf{B}$ is $\mathbf{A} \cap \mathbf{B}$.

Proof Follows directly from the construction of system state union and intersection in Definitions 2 and 3. □

2.4 Consistent System States

A system state is called *consistent* if and only if no component process depends on a state of any other process beyond the end of its state interval contained in this system state. Thus, no process has received a message that has not already been sent in this system state and cannot be sent within the deterministic execution of the sender. Any message shown by the system state to have been sent but not yet received does not cause the system state to be inconsistent, and can be handled by the normal mechanism for reliable message delivery, if any, used by the underlying system. If the system as a whole could be observed instantaneously, only consistent system states could be seen during failure-free execution of the system from its initial state, regardless of the relative speeds of the component processes [Chandy85]. By always recovering the system to a consistent state following a failure, the total execution of the system is equivalent to *some* possible failure-free execution.

The definition of a consistent system state can be expressed in terms of the dependency matrix representing this system state. If a system state is consistent, then for each process i, no other process j depends on a state interval of process i beyond process i's current state interval. In the dependency matrix, for each column i, no element in column i may be larger than the element in that column on the diagonal of the matrix, which is process i's current state interval index.

Definition 4 If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *consistent* if and only if

$$\forall\, i, j\, [\delta_{ji} \leq \delta_{ii}] \,.$$

Let the set $\mathcal{C} \subseteq \mathcal{S}$ be the set of *consistent* system states that have occurred during any single execution of some system. That is,

$$\mathcal{C} = \{\, \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is consistent} \,\} \,.$$

For example, consider the system of three processes whose execution is shown in Figure 2. The state of each process here is observed where the curve crosses the line representing the execution of that process, and the resulting system state is represented by the dependency matrix

$$\mathbf{D} = [\delta_{**}] = \begin{bmatrix} 1 & \textcircled{4} & \bot \\ 0 & \textcircled{2} & 0 \\ \bot & 2 & 1 \end{bmatrix} \,.$$

This system state is not consistent since process 1 has received a message (to begin state interval 1) from process 2 that was sent beyond the end of process 2's current state interval. This message has

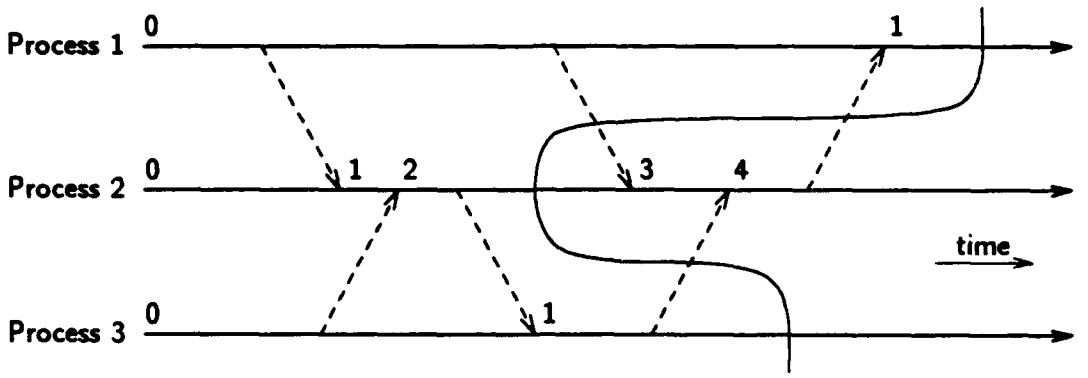

Figure 2 An inconsistent system state.

not been sent yet by process 2 and cannot be sent by process 2 during its deterministic execution. This inconsistency is shown in the dependency matrix since δ_{12} is greater than δ_{22}.

> **Lemma 1** The set $\mathcal{C}$, ordered by the system history relation, forms a sublattice of the system history lattice.

Proof It suffices to show that for any $\mathbf{A}, \mathbf{B} \in \mathcal{C}$, $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$. Let $\mathbf{A} = [\alpha_{**}]$ and $\mathbf{B} = [\beta_{**}]$.

($\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$): Let $\mathbf{C} = [\gamma_{**}] = \mathbf{A} \cup \mathbf{B}$. Since $\mathbf{A} \in \mathcal{C}$ and $\mathbf{B} \in \mathcal{C}$, $\alpha_{ji} \leq \alpha_{ii}$ and $\beta_{ji} \leq \beta_{ii}$ for all i and j. Since $\gamma_{ji} = \alpha_{ji}$ or $\gamma_{ji} = \beta_{ji}$, and $\gamma_{ii} = \max(\alpha_{ii}, \beta_{ii})$, $\gamma_{ji} \leq \gamma_{ii}$ for all i and j as well. Therefore, $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$.

($\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$): Let $\mathbf{D} = [\delta_{**}] = \mathbf{A} \cap \mathbf{B}$. By Definition 3, and since no element in the dependency vector for any process ever *decreases* as the process executes, then $\delta_{ji} = \min(\alpha_{ji}, \beta_{ji})$, for all i and j. This implies that $\delta_{ji} \leq \alpha_{ji}$ and $\delta_{ji} \leq \beta_{ji}$. Since $\mathbf{A}$ and $\mathbf{B}$ are consistent, $\alpha_{ji} \leq \alpha_{ii}$ and $\beta_{ji} \leq \beta_{ii}$. Combining this with the previous result yields $\delta_{ji} \leq \alpha_{ii}$ and $\delta_{ji} \leq \beta_{ii}$. This implies that $\delta_{ji} \leq \min(\alpha_{ii}, \beta_{ii})$, and thus $\delta_{ji} \leq \delta_{ii}$, for all i and j. Therefore, $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$. □

2.5 Message Logging and Checkpointing

A message is called *logged* if and only if its data and the index of the state interval that it started in its receiver process are *both* recorded on stable storage. The predicate *logged*(i, σ) is true if and only if the message that started state interval σ of process i is logged.

When a process is created, it is immediately checkpointed (in state interval 0) before it begins execution. For every state interval σ of each process, there must then be *some* checkpoint for that process on stable storage with a state interval index no larger than σ.

> **Definition 5** The *effective checkpoint* for a state interval σ of some process i is the checkpoint on stable storage for process i with the largest state interval index ϵ such that $\epsilon \leq \sigma$.

A state interval of a process is called *stable* if and only if all messages received by the process to start state intervals after its effective checkpoint are logged. The predicate $stable(i, \sigma)$ is true if and only if state interval σ of process i is stable.

Definition 6 If σ is a state interval index of some process i, and the effective checkpoint of this state interval has index ϵ, then state interval σ of process i is *stable* if and only if

$$\forall\, \alpha, \epsilon < \alpha \leq \sigma\, [\,\text{logged}(i, \alpha)\,] \,.$$

Any stable state interval σ for a process can be recreated by restoring the process from the effective checkpoint (with state interval index ϵ) and replaying to it in order any logged messages to begin state intervals $\epsilon+1$ through σ.

The checkpoint of a process includes the complete current dependency vector for the process. Each logged message only contains the state interval index of the sending process at the time that the message was sent, but the complete dependency vector for any stable state interval of some process can always be obtained, since all messages that started state intervals since the effective checkpoint must be logged.

2.6 Recoverable System States

A system state is called *recoverable* if and only if all component process states are *stable* and the resulting system state is *consistent.* To recover the state of the system, it must be possible to recover the states of the component processes, and for this system state to be meaningful, it must be possible to have occurred through failure-free execution of the system from its initial state.

Definition 7 If $\mathbf{D} = [\delta_{**}]$ is some system state in $\mathcal{S}$, $\mathbf{D}$ is *recoverable* if and only if

$$\mathbf{D} \in \mathcal{C} \;\wedge\; \forall\, i\, [\,\text{stable}(i, \delta_{ii})\,] \,.$$

Let the set $\mathcal{R} \subseteq \mathcal{S}$ be the set of *recoverable* system states that have occurred during any single execution of some system. That is,

$$\mathcal{R} = \{\, \mathbf{D} \in \mathcal{S} \mid \mathbf{D} \text{ is recoverable} \,\} \,.$$

Since only consistent system states can be recoverable, $\mathcal{R} \subseteq \mathcal{C} \subseteq \mathcal{S}$.

Lemma 2 The set $\mathcal{R}$, ordered by the system history relation, forms a sublattice of the system history lattice.

Proof For any $\mathbf{A}, \mathbf{B} \in \mathcal{R}$, $\mathbf{A} \cup \mathbf{B} \in \mathcal{C}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{C}$, by Lemma 1. Since the state of each process in $\mathbf{A}$ and $\mathbf{B}$ is stable, all process states in $\mathbf{A} \cup \mathbf{B}$ and $\mathbf{A} \cap \mathbf{B}$ are stable as well. Thus, $\mathbf{A} \cup \mathbf{B} \in \mathcal{R}$ and $\mathbf{A} \cap \mathbf{B} \in \mathcal{R}$, and $\mathcal{R}$ forms a sublattice. □

2.7 The Current Recovery State

In recovering after a failure, we wish to restore the state of the system to the "most recent" recoverable state that is possible from the information available, in order to minimize the amount of reexecution necessary to complete the recovery. The *system history lattice corresponds to this* notion of time, and the following theorem establishes the existence of a *single* maximum recoverable system state under this ordering.

Theorem 2 There is always a unique maximum recoverable system state in $\mathcal{S}$.

Proof The unique maximum in $\mathcal{S}$ is simply

$$\bigcup_{\mathbf{D} \in \mathcal{R}} \mathbf{D},$$

which must be unique since $\mathcal{R}$ forms a sublattice of the system history lattice. □

This unique maximum recoverable system state is called the *current recovery state* of the system.

Lemma 3 During any single execution of some system, the current recovery state of the system never decreases.

Proof Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state of the system at some time. $\mathbf{R}$ will always remain consistent, and for each process i, state interval ρ_{ii} will always remain stable. Since $\mathcal{R}$ forms a sublattice, any new current recovery state established after $\mathbf{R}$ must be greater than $\mathbf{R}$ in the lattice. □

Corollary 1 If the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then the system can always be recovered without rolling back any state interval $\sigma \leq \rho_{ii}$, for any process i.

Proof Each process i can always be recovered to its state interval ρ_{ii} in the current recovery state of the system. By Lemma 3, the current recovery state never decreases, and thus, by Definition 1, the state interval index of each process in any new current recovery state must be greater than or equal to ρ_{ii}. Therefore, for each process i, no state interval $\sigma \leq \rho_{ii}$ need ever be rolled back. □

Corollary 2 If all messages received by executing processes are *eventually* logged, there is no possibility of the domino effect in the system.

Proof If all messages are eventually logged, all state intervals of all processes eventually become stable by Definition 6, and thus new recoverable states must become possible through Definition 7. By Corollary 1, these states will never need to be rolled back. □

2.8 Committing Output

If some state interval of a process must be rolled back to recover a consistent system state, any output messages sent while that state interval is being reexecuted after recovery may not be the same as those originally sent. Any processes that received such messages will be orphans and must be rolled back to a point before these messages were received.

Messages sent to the *outside world*, such as those to the user's display terminal, cannot be treated in the same way, however. Since the outside world generally cannot be rolled back, any messages sent to the outside world must be delayed until it is known that the state interval from which they were sent will never need to be rolled back. They can then be *committed* by releasing them. This lemma establishes when it is safe to commit an output message sent to the outside world.

> **Lemma 4** If the current recovery state of the system is $\mathbf{R} = [\rho_{**}]$, then any message sent by a process i from a state $\sigma \leq \rho_{ii}$ may be committed.

Proof Follows directly from Corollary 1. □

2.9 Garbage Collection

During operation of the system, checkpoints and logged messages accumulate on stable storage in case they are needed for some future recovery. Each of these may be removed from stable storage whenever doing so will not interfere with the ability of the system to recover. The following two lemmas establish when this can safely be done.

> **Lemma 5** Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for its state interval ρ_{ii}, then any checkpoint for process i with state interval index $\sigma < \epsilon_i$ may be released from stable storage.

Proof Follows directly from Corollary 1 and Definition 5. □

> **Lemma 6** Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state. For each process i, if ϵ_i is the state interval index of the effective checkpoint for its state interval ρ_{ii}, then any message that begins a state interval in process i with index $\sigma \leq \epsilon_i$ may be released from stable storage.

Proof Follows directly from Corollary 1 and Definition 5. □

3 The Recovery State Algorithm

Theorem 2 shows that there is always a unique maximum recoverable system state, called the current recovery state. Conceptually, this system state can always be found by an exhaustive

search of all combinations of currently stable process state intervals for the maximum combination. However, such a search would be too expensive in practice. Our *recovery state algorithm* finds the current recovery state more efficiently by limiting this search. The algorithm is invoked each time a process state interval becomes stable, either from a checkpoint or when a message is logged. It is incremental in that it starts with the existing current recovery state, and attempts to advance it based on the fact that a single new process state interval has become stable. Since it only uses information on stable storage, it is restartable and can handle any number of concurrent process failures.

Each time some new state interval σ of some process k becomes stable, the algorithm determines if a new current recovery state exists. It first attempts to find *some* new recoverable system state in which the state of process k is advanced to state interval σ. If no such system state can be found, the current recovery state remains unchanged. If such a recoverable system state is found, the algorithm looks for other greater recoverable system states. The new current recovery state of the system is the maximum recoverable system state found in this search.

3.1 Finding a New Recoverable System State

The heart of the recovery state algorithm is the procedure *FIND_REC*. Given *any* recoverable system state $\mathbf{R} = [\rho_{**}]$, and some stable state interval σ of some process k with $\sigma > \rho_{kk}$, *FIND_REC* attempts to find a new recoverable system state in which the state of process k is advanced *at least* to state interval σ. It does so by including any stable state intervals from other processes that are necessary to make this new system state consistent, by a direct application of the definition of system state consistency in Definition 4. The procedure succeeds if such a consistent system state can be composed from the set of process state intervals that are currently stable. Since the state of process k has advanced, the new recoverable system state found must be greater than $\mathbf{R}$ in the system history lattice.

The inputs to procedure *FIND_REC* are the given recoverable system state $\mathbf{R} = [\rho_{**}]$, a stable state interval of process k with index $\sigma > \rho_{kk}$, and the dependency vector for each stable process state interval of each process x with index $\theta > \rho_{xx}$. Conceptually, *FIND_REC* performs the following steps:

1. Make a new dependency matrix $\mathbf{D} = [\delta_{**}]$ from $\mathbf{R}$, with row k replaced by the dependency vector for state interval σ of process k.

2. Loop on step 2 while $\mathbf{D}$ is not consistent. That is, loop while there exists some i and j for which $\delta_{ji} > \delta_{ii}$. This shows that state interval δ_{jj} of process j depends on a state interval δ_{ji} of process i that is greater than process i's current state interval δ_{ii} in $\mathbf{D}$.

   Find the minimum $\alpha \geq \delta_{ji}$ such that stable(i, α):

   (a) If no such α exists, return **false**.

   (b) Otherwise, replace row i of $\mathbf{D}$ with the dependency vector for this state interval α of process i.

3. The system state represented by **D** is now consistent and is composed entirely of stable process state intervals. It is thus recoverable and greater than **R**. Return **true**.

An efficient implementation of procedure *FIND_REC* is shown in Figure 3. This implementation operates on a vector, *RV*, rather than the full dependency matrix. For all i, $RV[i]$ is diagonal element i of the corresponding dependency matrix. When *FIND_REC* is called, $RV[i]$ is the state interval index of process i in the given recoverable system state. The dependency vector of each stable state interval θ of process x is represented by DV_x^θ. As each row is replaced, the corresponding element of *RV* is changed. Also, the maximum element in each column of the matrix is kept in a vector *MAX*, such that $MAX[i]$ is the maximum element in each column i.

Lemma 7 If function *FIND_REC* is called with a known recoverable system state $\mathbf{R} = [\rho_{**}]$ and state interval σ of process k such that $\sigma > \rho_{kk}$, *FIND_REC* returns **true** if and only if there exists *some* recoverable system state $\mathbf{R}' = [\rho'_{**}]$, such that $\mathbf{R} \prec \mathbf{R}'$ and $\rho'_{kk} \geq \sigma$. If *FIND_REC* returns **true**, then on return, $RV[i] = \rho'_{ii}$, for all i.

Proof The predicate of the **while** loop determines whether the dependency matrix represented by *RV* and *MAX* is consistent, by application of Definition 4. When the condition becomes false and the loop exits, this matrix must be consistent since, in each column i, no element is larger than the diagonal element in that column. Thus, if *FIND_REC* returns **true**, the system state returned in *RV* must be consistent. This system state must also be recoverable since all process state intervals initially in it are stable, and only stable process state intervals are used to replace entries in it during the execution of *FIND_REC*.

The following loop invariant is maintained by function *FIND_REC* on each iteration at the head of the **while** loop:

If a recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists such that, for all i, $\rho'_{ii} \geq RV[i]$, then $\rho'_{ii} \geq MAX[i]$.

---

function *FIND_REC*(RV, k, σ)
    $RV[k] \leftarrow \sigma$;
    for $i \leftarrow 1$ **to** n **do** $MAX[i] \leftarrow \max(RV[i], DV_k^\sigma[i])$;
    while $\exists\, i$ such that $MAX[i] > RV[i]$ **do**
        $\alpha \leftarrow$ minimum such that $\alpha \geq MAX[i] \wedge$ stable(i, α);
        if no such α exists **then return false**;
        $RV[i] \leftarrow \alpha$;
        for $j \leftarrow 1$ **to** n **do** $MAX[j] \leftarrow \max(MAX[j], DV_i^\alpha[j])$;
    return true;

Figure 3 Procedure to find a new recoverable state.

The invariant holds initially since any consistent state must have $RV[i] \geq MAX[i]$ for all i. Thus, any state **R'** found such that $\rho'_{i,i} \geq RV[i]$, must have $\rho'_{i,i} \geq RV[i] \geq MAX[i]$. At each subsequent iteration of the loop, the invariant is maintained by choosing the *smallest* $\alpha \geq MAX[i]$ such that stable(i, α). To make the matrix consistent, α must not be less than $MAX[i]$. By choosing the minimum such α, the components of DV_i^α are also minimized. Thus, after replacing row i of the matrix with DV_i^α, the components of MAX are minimized, and any recoverable state **R'** that exists must still have $\rho'_{i,i} \geq MAX[i]$, for all i.

If no such $\alpha \geq MAX[i]$ exists, then no recoverable system state **R'** can exist, since any such **R'** must have $\rho'_{i,i} \geq MAX[i]$. This is exactly the condition under which the procedure *FIND_REC* returns **false**. □

The state intervals of each process i that are currently stable partition the set of *all* state intervals of process i into a collection of disjoint subsets known as *cover sets*.

Definition 8 The *cover set* for a stable state interval σ of process i is the set

$$\text{cover}(i, \sigma) = \{ \alpha \mid \beta < \alpha \leq \sigma \} ,$$

where β is the largest state interval index for process i such that $\beta < \sigma$ and stable(i, β).

That is, cover(i, σ) contains state interval σ and all preceding state intervals of process i that do not also precede any other earlier stable state interval β of process i. These sets thus form a partition of the state intervals of each process. In the **while** loop of function *FIND_REC*, if $MAX[i] = \delta$ and $\delta \in$ cover(i, σ), then the state interval index found for α will be σ.

These cover sets define a new dependency relation called *stable dependency*, based on which process state intervals are currently stable.

Definition 9 A state interval σ of some process k, with dependency vector $\langle \delta_* \rangle$, has a *stable dependency* on some state interval α of some process i if and only if $\delta_i \in$ cover(i, α).

The stable dependency relation describes the state interval α that must be chosen by *FIND_REC* during each iteration of the **while** loop. The transitive closure of the stable dependency relation for state interval σ of process k describes the set of state intervals α for each process i that can be used during *any* iteration of this **while** loop, although the subset of these that are actually used in some execution depends on the order in which the loop finds the next i that meets the required condition.

3.2 The Complete Algorithm

Using function *FIND_REC*, the complete recovery state algorithm can now be stated. The algorithm, shown in Figure 4, uses a vector *CRS* to represent the current recovery state of the system. When a process is created, its entry in *CRS* is initialized to 0. Then, when some state interval σ of some process k becomes stable, if it is in advance of the known current recovery state in *CRS*, the algorithm searches for the new current recovery state. During this search, the vector *NEWCRS* stores the maximum known recoverable system state.

```
if σ ≤ CRS[k] then exit;

NEWCRS ← CRS;

if ¬FIND_REC(NEWCRS, k, σ) then
   for i ← 1 to n do if i ≠ k then
      β ← DV_k^σ[i];
      if β > CRS[i] then DEFER_i^β ← DEFER_i^β ∪ {(k, σ)};
   exit;

WORK ← ∅;
for β ∈ cover(k, σ) do WORK ← WORK ∪ DEFER_k^β;
while WORK ≠ ∅ do
      remove some (x, θ) from WORK;
      if θ > NEWCRS[x] then
         RV ← NEWCRS;
         if FIND_REC(RV, x, θ) then NEWCRS ← RV;
      if θ ≤ NEWCRS[x] then
         for β ∈ cover(x, θ) do WORK ← WORK ∪ DEFER_x^β;

CRS ← NEWCRS;
```

Figure 4 The recovery state algorithm.

The algorithm first calls *FIND_REC* with the known current recovery state and state interval σ of process k. If *FIND_REC* returns **false**, then no greater recoverable system state exists in which the state of process k has advanced at least to state interval σ. Thus, the current recovery state has not changed, as shown in the following lemma and its corollary.

Lemma 8 If the current recovery state changes from $\mathbf{R} = [\rho_{**}]$ to $\mathbf{R}' = [\rho'_{**}]$, $\mathbf{R} \neq \mathbf{R}'$, when state interval σ of process k becomes stable, then $\rho'_{kk} = \sigma$.

Proof By contradiction. Suppose the new current recovery state has $\rho'_{kk} \neq \sigma$. Since state interval σ of process k is not part of system state $\mathbf{R}'$, all process state intervals in $\mathbf{R}'$ must have been stable before state interval σ of process k became stable. Thus, system state $\mathbf{R}'$ must have been recoverable before state interval σ of process k became stable. By Lemma 3, the current recovery state of the system never decreases, which leads to a contradiction, since $\mathbf{R} \neq \mathbf{R}'$ was the current recovery state of the system before state interval σ of process k became stable. Thus, either $\mathbf{R} = \mathbf{R}'$ or $\rho'_{kk} = \sigma$. □

Corollary 3 When state interval σ of process k becomes stable, if the initial call to *FIND_REC* by the recovery state algorithm returns **false**, then the current recovery state of the system has not changed.

Proof By Lemma 8, if the current recovery state changes to $\mathbf{R}' = [\rho'_{**}]$, then $\rho'_{kk} = \sigma$, but this **false** return from *FIND_REC* shows that *no* recoverable system state $\mathbf{R}'$ exists with $\rho'_{kk} \geq \sigma$, such that $\mathbf{R} \prec \mathbf{R}'$. Thus, the current recovery state has not changed. □

Associated with each state interval β of each process i that is in advance of the known current recovery state is a set $DEFER_i^\beta$, which records those stable process state intervals that have β in component i of their dependency vector. All *DEFER* sets are initialized to empty when the system begins execution. Each process state interval for which *FIND_REC* returns **false** when it becomes stable is entered into at least one *DEFER* set. The algorithm uses these sets to limit its search space for the new current recovery state.

If the initial call to *FIND_REC* by the recovery state algorithm returns **true**, however, a new greater recoverable system state has been found. Additional calls to *FIND_REC* are used to search for any other recoverable system states that exist and are greater than the one returned by this first call to *FIND_REC*. The algorithm uses a result of the following lemma to limit this search.

Lemma 9 Let $\mathbf{R} = [\rho_{**}]$ be the current recovery state before state interval σ of process k becomes stable. Then for any stable state interval θ of any process x such that $\theta > \rho_{kk}$, no recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists with $\rho'_{xx} \geq \theta$ if state interval θ of process x is not related to state interval σ of process k by the transitive closure of the stable dependency relation.

Proof Since state interval θ of process x is in advance of the old current recovery state, it could not be made part of any recoverable system state $\mathbf{R}'$ before state interval σ of process k became stable. If it is not related to state interval σ of process k by the transitive closure of the stable dependency relation, then the fact that state interval σ has become stable cannot affect this.

Let δ be the maximum state interval index of process k that state interval θ of process x is related to by this transitive closure. Clearly, any recoverable system state $\mathbf{R}' = [\rho'_{**}]$ that now exists with $\rho'_{xx} \geq \theta$ must have $\rho'_{kk} \geq \delta$, by Definitions 9 and 4, and since no component of any dependency vector decreases through execution of the process. If $\delta > \sigma$, then system state $\mathbf{R}'$ was recoverable before state interval σ became stable, which contradicts the assumption that $\theta > \rho_{kk}$. Likewise, if $\delta < \sigma$, then $\mathbf{R}'$ cannot exist now if it did not exist before state interval σ of process k became stable, since state interval δ was stable before state interval σ became stable. Since both cases lead to a contradiction, no such recoverable system state $\mathbf{R}'$ can now exists without this relation through the transitive closure. □

The **while** loop of the recovery state algorithm uses the *DEFER* sets to traverse the transitive closure of the stable dependency relation backward from state interval σ of process k. Each state interval θ of some process x encountered on this traversal is related to state interval σ of process k by this transitive closure. That is, either state interval θ of process x has a stable dependency

on state interval σ of process k, or it has a stable dependency on some other process state interval that is related to state interval σ of process k by this transitive closure. The traversal uses the set *WORK* to record those process state intervals from which the traversal must still be performed. When *WORK* has been emptied, the new current recovery state has been found and is copied back to *CRS*.

During this traversal, any branch along which no more successful results from *FIND_REC* can be obtained is not traversed further. If the state interval θ of process x that is being considered is in advance of the maximum known recoverable system state, *FIND_REC* is used to find a new recoverable system state in which process x has advanced at least to state interval θ. If no recoverable state exists that meets this requirement, the traversal from this state interval is not continued.

Lemma 10 If state interval β of process i is related to state interval θ of process x by the transitive closure of the stable dependency relation, and if no recoverable system state $\mathbf{R} = [\rho_{**}]$ exists with $\rho_{xx} \geq \theta$, then no recoverable system state $\mathbf{R}' = [\rho'_{**}]$ exists with $\rho'_{ii} \geq \beta$.

Proof This follows directly from the definition of a stable dependency in Definition 9. If such a recoverable system state $\mathbf{R}$ does not exist, no recoverable system state $\mathbf{R}'$ can exist, since either state interval β of process i has a stable dependency on state interval θ of process x, or it has a stable dependency on some other process state interval that is related to state interval θ of process x by this transitive closure. Thus, any such recoverable system state $\mathbf{R}'$ that exists must also have $\rho'_{xx} \geq \theta$. Therefore, no such recoverable system state $\mathbf{R}'$ can exist. □

Theorem 3 If the recovery state algorithm is called each time some state interval σ of some process k becomes stable, it will always return with $CRS[i] = \rho'_{ii}$, for all i, where $\mathbf{R}' = [\rho'_{**}]$ is the new current recovery state of the system.

Proof The theorem holds before any process state interval has become stable, since $CRS[i]$ is always initialized to 0 when process i is created. Likewise, when any new process i is created, it is correctly added to the current recovery state by setting $CRS[i] = 0$.

When some state interval σ of some process k becomes stable, if the initial call to *FIND_REC* returns **false**, the current recovery state remains unchanged, by Corollary 3. In this case, the recovery state algorithm leaves *CRS* unchanged.

If this call to *FIND_REC* returns **true**, the current recovery state has advanced as a result of this new state interval becoming stable. Let $\mathbf{R} = [\rho_{**}]$ be the old current recovery state of the system before state interval σ of process k became stable, and let $\mathbf{D} = [\delta_{**}]$ be the system state returned by this call to *FIND_REC*. Then $\mathbf{R} \prec \mathbf{D}$, by Lemma 7. $\mathbf{D}$ may be less than the new current recovery state $\mathbf{R}'$, but since the set of recoverable system states forms a lattice, $\mathbf{D} \preceq \mathbf{R}'$. The **while** loop of the recovery state algorithm thus searches forward in the lattice for the new current recovery state, without backtracking.

The **while** loop performs the backward traversal of the transitive closure of the stable dependency relation, through the information in the *DEFER* sets. For each state interval θ of each

process x examined by this loop, if no recoverable system state exists in which the state of process x has advanced at least to interval θ, the loop does not traverse further from this state interval. By Lemmas 9 and 10, the loop must consider all stable process state intervals for which a new recoverable system state exists. Thus, when the traversal is complete and the loop exits, the last recoverable system state found must be new current recovery state. The algorithm then copies this to *CRS*. □

3.3 An Example

Figure 5 shows the execution of a system of three processes. Each process has been checkpointed in its state interval 0, but no other checkpoints have been written. The three processes have received a total of four input messages, but none of these has been logged yet. Thus, only state interval 0 for each process is stable, and the current recovery state of the system is composed of state interval 0 of each process. In the recovery state algorithm, $CRS = \langle 0, 0, 0 \rangle$, and all *DEFER* sets are empty.

If message a now becomes logged, state interval 1 of process 1 becomes stable, with a dependency vector of $\langle 1, 1, \perp \rangle$. The recovery state algorithm is executed and calls *FIND_REC* with $k = 1$ and $\sigma = 1$. *FIND_REC* sets *RV* to $\langle 1, 0, 0 \rangle$ and *MAX* to $\langle 1, 1, 0 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is required, but does not exist. *FIND_REC* thus returns **false**. The recovery state algorithm changes $DEFER_2^1$ to $\{ (1, 1) \}$, and exits, leaving $CRS = \langle 0, 0, 0 \rangle$.

Now, if process 2 is checkpointed in state interval 2, this state interval becomes stable. Its dependency vector is $\langle 0, 2, 1 \rangle$. *FIND_REC* sets *RV* to $\langle 0, 2, 0 \rangle$ and *MAX* to $\langle 0, 2, 1 \rangle$. Since no state interval $\alpha \geq 1$ of process 3 is stable, *FIND_REC* returns **false**. The recovery state algorithm sets $DEFER_3^1$ to $\{ (2, 2) \}$, and exits, leaving *CRS* unchanged.

Finally, if message b becomes logged, state interval 1 of process 3 becomes stable, with dependency vector $\langle \perp, 1, 1 \rangle$. *FIND_REC* is called, and sets *RV* to $\langle 0, 0, 1 \rangle$ and *MAX* to $\langle 0, 1, 1 \rangle$. Since $MAX[2] > RV[2]$, a stable state interval $\alpha \geq 1$ of process 2 is required. Now, state interval 2 of process 2 is stable and satisfies this. *RV* and *MAX* are then updated, yielding $\langle 0, 2, 1 \rangle$ for both. This system state is thus consistent, and *FIND_REC* returns **true**. The maximum known recoverable system state in *NEWCRS* is then increased to $\langle 0, 2, 1 \rangle$, and the *WORK* set is initialized for

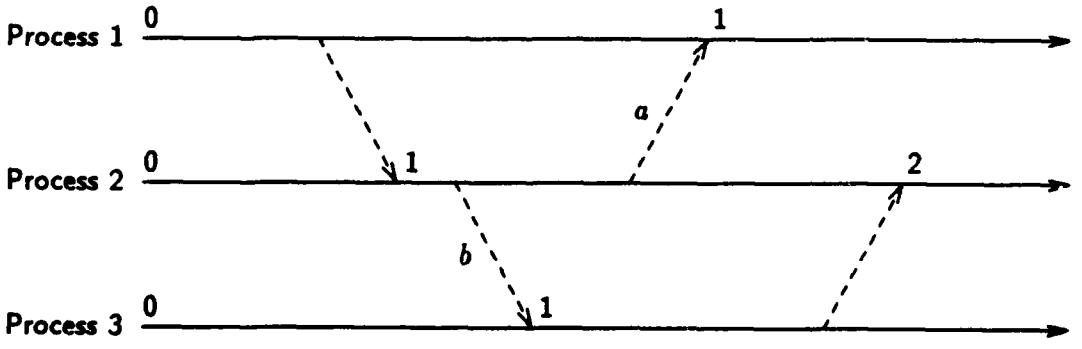

Figure 5 An example system execution.

the loop. Since cover$(3,1) = \{1\}$, *WORK* is set to $DEFER_3^1 = \{(2,2)\}$. When state interval 2 of process 2 is checked, it is not in advance of *NEWCRS*, and the call to *FIND_REC* is skipped. The *DEFER* sets identified in cover$(2,2) = \{2,1\}$ are added to *WORK*, giving $\{(1,1)\}$. State interval 1 of process 1 is then checked, and *FIND_REC* is called. *RV* and *MAX* are both set to $\langle 1,2,1 \rangle$, and *FIND_REC* returns **true**. $DEFER_1^1$ is added to *WORK*, since cover$(1,1) = \{1\}$, but this leaves $WORK = \emptyset$, and the **while** loop thus terminates. The value left in $NEWCRS = \langle 1,2,1 \rangle$ is the new current recovery state and is copied back to *CRS*.

This example illustrates a unique feature of our recovery state algorithm. Our algorithm uses both logged messages and checkpoints in its search for the maximum recoverable system state. Although only two of the four messages received during this execution of the system have been logged, the current recovery state has advanced due to the checkpoint of process 2. In fact, these two remaining messages need never be logged, since the current recovery state has advanced beyond their receipt.

4 Related Work

A number of fault-tolerance methods using message logging and checkpointing have been published in the literature. This includes ones using pessimistic logging protocols such as Auros [Borg83], Publishing [Powell83], and sender-based message logging [Johnson87], as well as optimistic methods [Strom85]. The model and recovery state algorithm presented in Sections 2 and 3 can be applied to each of these and used to reason about their correctness.

Our model is more general than is required by recovery methods using pessimistic message logging, but the definitions of consistency, stability, and recoverability still apply, and the recovery state algorithm still computes the correct current recovery state. In this case, the current recovery state is identical to the state of the system at the time the failure occurred, since orphan processes are not possible. Since message logging is synchronous, however, a simpler recovery state algorithm is possible that takes advantage of the order that information arrives on stable storage. In particular, checkpoints never add new information for the algorithm, since messages are always logged in ascending order by the index of the state interval that they start in their receivers, and all messages received before a checkpoint have already been logged before the checkpoint can be recorded.

Recovery using optimistic logging protocols requires the full generality of our model, however. Since orphan processes are possible when using optimistic logging, recovery from a failure is more difficult. Any orphan processes must be rolled back during recovery to achieve a consistent state. Since there is no synchronization between message logging, checkpointing, and computation, information for the recovery state algorithm may arrive on stable storage at any time and in any order. Thus, the algorithm must be able to make use of all this information in order to find the maximum possible recoverable system state at any time.

Our model is more general than that used for optimistic recovery by Strom and Yemini [Strom85], since they require reliable delivery of messages between processes. As a result, their definition of consistency differs from ours by requiring all messages sent to have been received. Our model does not require reliable delivery, but this can be incorporated easily by inserting a return acknowledge-

ment message immediately following each message receipt. Our definition of consistency and our algorithm then remain unchanged.

Their algorithm requires each process to maintain a vector of its *transitive* dependencies, and requires each message to be tagged with this vector, which has size proportional to the number of processes. As each message is received, the process merges the dependency vector from the message with its own local vector. Our algorithm only requires each process to maintain a vector of its *direct* dependencies, and only requires the the current state interval index of the sending process to be carried in each message. Their algorithm also requires each process to maintain a *log vector* to record its knowledge of which messages in the system have been logged. This is achieved either by having each process periodically broadcast its vector, or by appending it to each message sent. This requirement adds additional communication and complexity to the system that is not required by our algorithm, although this does allow control of recovery in their system to be more decentralized than in ours. Also, although their system checkpoints processes in order to shorten recovery times and release old logged messages from stable storage, they do not take advantage of these checkpoints in computing the current maximum recoverable system state. Our algorithm uses both checkpoints and logged messages to compute the maximum recoverable system state, and thus may find recoverable states that their algorithm does not.

5 Conclusion

Recovery using optimistic message logging protocols constitutes a beneficial performance tradeoff in operating environments where failures are rare and failure-free performance is of primary concern. The recovery state algorithm of Section 3 represents an improvement on earlier work with recovery using optimistic message logging by Strom and Yemini [Strom85]. Although their algorithm eventually achieves a recoverable system state, this state may be less than the maximum possible. Furthermore, their method requires reliable communication and seems more complex than the method presented here.

This work unifies existing approaches to fault tolerance using message logging and checkpointing published in the literature, including those using pessimistic message logging [Borg83, Powell83, Johnson87] and those using optimistic methods [Strom85]. By using this model to reason about these types of fault-tolerance methods, properties that are independent of the message logging protocol used can be deduced and proven. We have shown that the set of system states that have occurred during any single execution of a system forms a lattice, with the sets of consistent and recoverable system states as sublattices. There is thus always a unique maximum recoverable system state, called the current recovery state. The current recovery state never decreases, and if all all messages received by processes in the system are eventually logged, the domino effect cannot occur.

Acknowledgements

We would like to thank Rick Bubenik, John Carter, Matthias Felleisen, and Gerald Fowler for many helpful discussions on this material and for their comments on earlier drafts of this paper.

References

[Borg83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99. ACM, October 1983.

[Chandy85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson79] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

[Powell83] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[Randell75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Russell80] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.