FIMD-MPI: A Tool for Injecting Faults into MPI Applications*

Douglas M. Blough
School of Electrical & Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332–0250 USA

doug.blough@ece.gatech.edu

Peng Liu
Dept. of Electrical & Computer Engineering
University of California
Irvine, CA 92697–2625 USA

pengl@ece.uci.edu

Abstract

Parallel computing is seeing increasing use in critical applications. The need therefore arises to test the robustness of parallel applications in the presence of exceptional conditions, or faults. Communication-software-based fault injection is an extremely flexible approach to robustness testing in message-passing parallel computers. A fault injection methodology and tool that use this approach are presented. The tool, known as FIMD-MPI, allows injection of faults into MPI-based applications. The structure and operation of FIMD-MPI are described and the use of the tool is illustrated on an example fault-tolerant MPI application.

1 Introduction

An often-cited benefit of distributed-memory parallel computers is that they possess inherent hardware redundancy that can be exploited to provide software-implemented fault tolerance. Several important projects such as SIFT [17], MAFT [9], and MAX/COSMOS [11, 12] have demonstrated the feasibility of this software-implemented fault tolerance concept. Fueled in part by these projects, researchers have developed literally hundreds of algorithms to solve problems in distributed-memory systems such as fault-tolerant unicast communication, fault-tolerant broadcast, fault diagnosis, checkpoint/rollback, various consensus problems such as Byzantine agreement, approximate agreement, clock synchronization, and error detection/correction in numeric applications. Unfortunately, the vast majority of fault-tolerant parallel algorithms remain untested with their correctness demonstrated solely through mathematical proofs. In part, this is due to the lack of general-purpose tools for injection of faults in parallel computers.

Exacerbating the situation is the fact that correctness proofs for fault-tolerant parallel algorithms tend to be quite complex and are, therefore, error-prone. Formal verification of a number of fault-tolerant algorithms has been performed and, not surprisingly, errors have been found in published algorithms [13]. While formal methods are a promising approach, verification of many algorithms is still beyond the capabilities of state-of-the-art tools, particularly in cases where temporal behavior is an important characteristic of the algorithm. In addition, formal methods verify an abstract algorithm description rather than its implementation. Even if an algorithm has been formally verified, there is still a significant chance of errors being introduced in the implementation. Hence, fault-injection-based testing will always remain an extremely important component of an overall verification process.

In this paper, we review our communication-based fault injection methodology and describe a new software tool for fault injection in message-passing parallel computers. Our fault injection tool is called FIMD-MPI[1] (Fault Injection for Message-passing Distributed-memory systems - MPI version) and is designed to run on any system that has an implementation of MPI. FIMD-MPI has been used to test a fault-tolerant clock synchronization algorithm that was implemented on networks of Solaris and Linux workstations running MPICH. A series of fault injection tests of this algorithm demonstrated not only its robustness but also the capability of the tool to inject a wide variety of faults of different specifications and timings.

2 Fault Injection Methodology

We are interested in injecting faults according to fault models used in the design and analysis of fault-tolerant algorithms for multicomputer systems. These include, among others, symmetric and asymmetric faults [16]; delay faults and other timing faults [5]; omission faults [5]; and fail-silent, fail-stop, and crash faults [4, 14, 15].

Our approach is to emulate these models through modification of internode communications. Communications are modified by intercepting message-passing calls that are made by an application and changing them in various ways. Figure 1 illustrates how this is carried out in FIMD-MPI.

The fault models that we support with this injection mechanism are: timing accelerated, timing delay, omission, message corruption datatype, message corruption length,

*This research was supported by the National Science Foundation under Grant CCR-9803741.

[1] FIMD-MPI is open-source software that can be downloaded from http://www.eng.uci.edu/ftmp/fimd.html

0-7695-0574-0/2000 $10.00 © 2000 IEEE

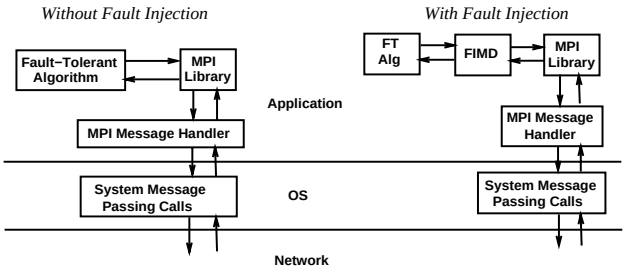

Figure 1. Fault Injection in FIMD-MPI

message corruption destination, message corruption tag, message corruption data, spurious generation, spurious replication, symmetric multicast, and asymmetric multicast. For a detailed discussion of these fault models and how they are implemented, the interested reader is referred to [3].

3 Fault Injection Tool

This section describes the implementation of our fault injection tool, FIMD-MPI.

3.1 Overview

FIMD-MPI includes two parts, the Graphical User Interface (GUI) and the Fault Injection Library (FIL). The GUI is written with Tcl/Tk. With the GUI, the user can define the faults which are saved in a basic fault specification file (BFSF) and then injected into the application. FIL is used to perform the fault injection. FIL is C code that provides the user with a set of fault-injection versions of MPI message passing calls and a run-time fault injector called the event manager (EM). MPI library calls in the application source code are replaced by their fault-injection versions. When a fault injection message passing call is made, the message is passed to the event manager (EM) which could possibly change the message before sending it. When and how the messages are sent out are decided by the EM according to the faults specified in the BFSF.

Since FIMD-MPI is intended for use during the application testing phase, its design assumed that the source code of the target application is available. Even though an application must be initially compiled with FIL, re-compilation is not required upon a change in fault specification.

3.2 Faults and their Parameters

Faults are injected into individual nodes and affect only the messages generated by the node (not its internal operation). Specifying a fault requires selecting from a number of options and setting various parameters.

Options that must be selected by the user for a specific fault instance are:

- the node on which to inject the fault
- the fault model
- the fault type – permanent or transient/intermittent
- the fault injection method – deterministic or random and time-based or message-count-based

The user must also specify parameters for each fault instance. Several parameters are common to all fault instances while others depend on the fault model and injection method. Parameters that are common to all faults are the affected destinations, the start time, and the duration. A fault can be specified to affect only messages being sent to a specific destination node, or it can affect messages being sent to all destinations. For faults of transient/intermittent type, messages are affected by the fault only from the specified start time and lasting for the specified duration. For permanent faults, messages are affected from the specified start time and lasting until the end of the experiment.

The injection methods are important for understanding how faults are injected by FIMD-MPI. In the deterministic methods, the user provides specific and fixed information about which messages should be affected by a fault. In the random methods, the user specifies only some general parameters and allows the tool to randomly decide, based on those parameters, which messages to affect. For each of these methods, some timing information is required. For the deterministic methods, the only parameters required are the aforementioned start time and duration. These parameters can be specified either in terms of time or by giving specific message counts. Both of these are relative to the experiment start. Additional timing parameters are required to specify the random injection methods. These methods and their parameters are described in detail in Section 3.5.6.

3.3 Running Fault Injection Experiments

The FIMD-MPI GUI provides a complete environment for specifying and running fault injection experiments.

Figure 2 shows the “Model Definition” window in which fault specification is done. In this window, a start time and duration can be specified for the entire experiment. “-1” is used to denote forever (as long as the application runs). The experiment start time is relative to the start of the application. Start times of individual faults are relative to the experiment start time. The buttons at the top of this window allow the user to move between multiple fault instances defined for the same experiment, to read a previously-generated set of fault instances, clear the current set of faults, and exit the “Model Definition” window (“OK” button).

0-7695-0574-0/2000 $10.00 © 2000 IEEE

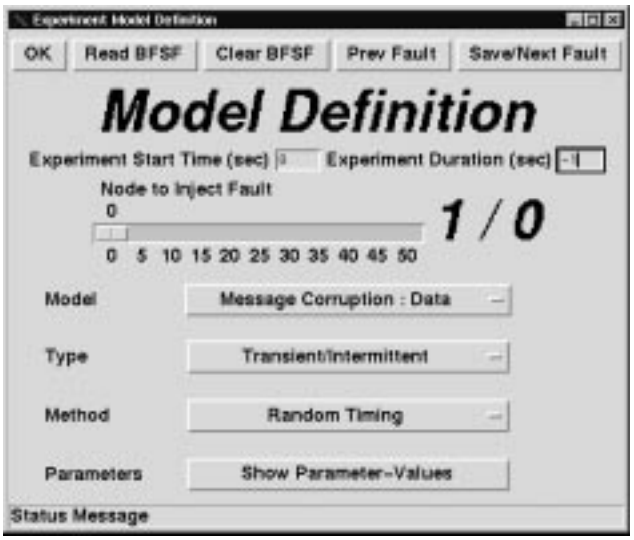

Figure 2. Model Definition Window in GUI

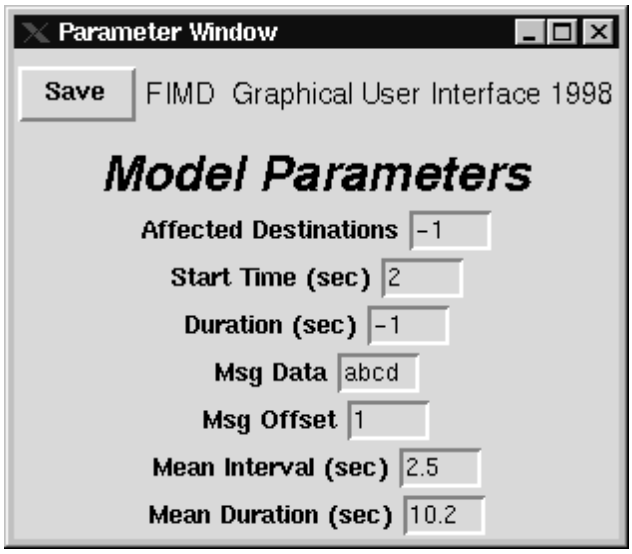

Figure 3. Model Parameters Window in GUI

The horizontal scroll bar defines in which node the fault being specified will be injected. The bold numbers that appear to the right of the scroll bar signify the number of the fault being specified and the total number of faults that have been saved to this point. The three sections with pull-down menus below the scroll bar – "Model", "Type", and "Method" – represent the fault options described earlier. The parameters for the fault being specified can be edited by clicking the button in the "Parameters" section.

For the options shown in Figure 2, the "Model Parameters" window is shown in Figure 3. This window shows the parameters that are common to all faults as well as parameters that are specific to the fault model, type, and method selected in the "Model Definition" window. "-1" is used to indicate that all destinations are affected by the fault and also to indicate that the duration is for the entire experiment. The corrupt data to be overwritten in the message is specified along with an offset that indicates the position within the message at which to write the data. The mean interval and mean duration are explained in Section 3.5.6.

When fault specification is complete, the fault instances are compiled into a basic fault specification file (BFSF), which is in the the low-level format that can be read by the event manager. This is the final step before running a fault injection experiment.

A fault injection experiment is started from the GUI's "Execute Experiment" window, an example of which is shown in Figure 4. In this example, the experiment is run with four processes, the set of machines on which processes should be run is in the file "config", and output is recorded in the file "output.txt". The command line that is executed when the "Execute" button is clicked is shown in the "Command" field. This command can be edited inside the window or can be copied for execution in a shell window.

3.4 Compiling the Application

To inject faults into an MPI application, it must be compiled with the fault injection library. The following changes to the application's source code are necessary:

- the function call "initEM()" must be added at the beginning of the source code to initialize the EM
- the names of any message-passing calls that should be subject to fault injection must be preceded by "FI_" but their parameters remain unchanged
- the function call "dumpFILogfile()" should be added as the last line of the application to produce the log file

The user can choose to not precede certain message-passing calls by "FI_" and then those calls will not be subject to faults. This allows the user to keep part of the application fault-free which is useful in many situations. Log file data is written only after the application completes so that log file I/O does not interfere with execution of the application.

3.5 Fault Injection Library

The fault injection library (FIL) is the core component of FIMD-MPI and is responsible for actually injecting faults into an application.

3.5.1 Overview

FIL consists of fault-injection versions of the MPI message-passing calls and the Event Manager (EM), which are compiled with an application so that they reside in every process

0-7695-0574-0/2000 $10.00 © 2000 IEEE

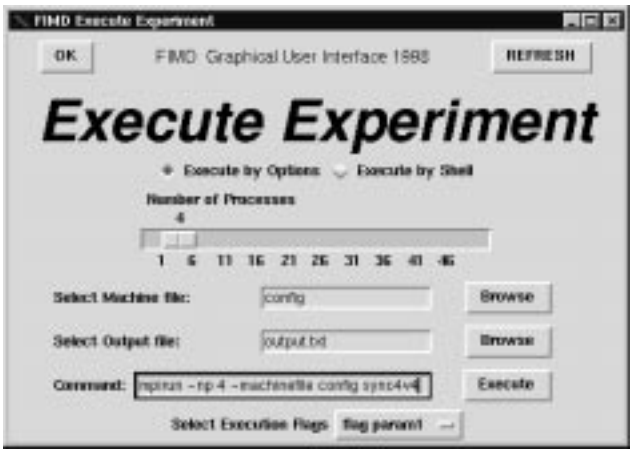

Figure 4. Execute Experiment Window in GUI

that is part of the application. The EM coordinates the fault injections instructed by the BFSF. Every time a fault is injected, it is recorded to a fault log file along with certain run-time parameter values of the fault.

3.5.2 FIL Data Structures

The primary data structures used by FIL are event queues, a timer queue, and look-up tables. An entry in any of these data structures corresponds to a specific fault occurrence that will be injected into the system. The queues and tables are maintained as singly-linked lists sorted according to the active start point of each fault. Expired entries are removed from the lists. The structures of the entries in these queues and tables are almost the same with only minor differences. To conserve memory, all entries are allocated dynamically, and are freed as soon as they expire. The use of these data structures is described in subsequent sections.

3.5.3 Fault Injection Experiment Initialization

At the beginning of application execution, the FIL data structures are initialized and the fault specification is read. The BFSF is read by each process and the faults relevant to a specific process are put into either its event queues or its timer queue depending on the fault model.

3.5.4 Event Manager

The EM, as its name implies, is in charge of the application's events. Events are calls to one of FIL's message-passing routines, which are made whenever the application attempts to send a message. The FIL routine prepares a new message parameter data structure which is initialized with the parameters of the application's call and is then passed to the EM. Based on the faults in the event queues, the EM may choose to modify these message parameters which are then returned to the FIL message-passing routine. A value of NO_SEND for the destination field of the parameter structure indicates that the message should not be sent at this time and causes the FIL routine to omit the MPI message-passing call. Otherwise, the FIL routine simply issues the MPI message-passing call with the (possibly modified) parameters.

3.5.5 Timer Handler

The EM is invoked only when a message-passing call is made. However, injection of certain fault models is time-triggered and, therefore, does not correspond to any message-passing event in the original application. These faults are put into the timer queue instead of the event queues during initialization. Certain fault models, e.g. delay faults, use both the event queue and the timer queue. The original message that should be delayed corresponds to a message-passing event in the application and is therefore specified in the event queue. However, after the event occurs, the message should be sent out at a later time not corresponding to any event. Hence, after the original message passing call, the EM places an entry containing the message parameters in the timer queue that will trigger the later send. The EM then causes the FIL message-passing routine to omit the current MPI message-passing call.

3.5.6 Random Fault Injection Methods

The random injection methods produce faults which are alternately active and inactive, where the durations of active and inactive sections are chosen randomly according to specified parameters.

For random message-count-based injection, the user specifies the maximum fault interarrival time and the maximum fault duration, both in terms of message counts. The first active interval is selected at the fault's specified start time by choosing with equal likelihood any interarrival count and duration between one and the specified maxima. When the current active section ends, the parameters for the next active section are chosen in a similar manner. This process continues until the total duration of the random fault has passed.

Random time-based fault injection is more complicated. In this method, the user specifies the mean interarrival time and mean duration of the fault, which are used to produce a fault with Poisson arrivals. If we calculate the next active interval at the end of the current active interval, as done in random message-count-based injection, we must invoke the timerhandler at the end of each interval. This is necessary because the end of the interval does not necessarily coincide

0-7695-0574-0/2000 $10.00 © 2000 IEEE

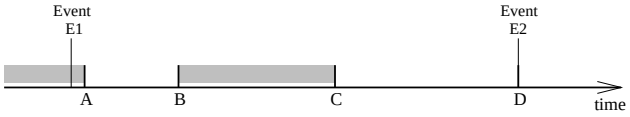

Figure 5. Random Time-Based Injection

with an event. This solution is not ideal because it could result in frequent interrupts of the application.

Instead, we would like to calculate the active intervals on an event-driven basis. Consider the situation illustrated in Figure 5. Suppose event E2 has just occurred and the previous event was E1. The only information we have is that the last active interval ended at time A. We can then generate the next active interval, e.g. from time B to time C, according to the Poisson arrival distribution. However, if event E2 is much later than E1, this could result in the generation of a long sequence of active intervals before we are able to know whether or not time D is covered by an active interval.

To eliminate the need for calculation of long sequences of intervals, we handle this calculation as follows. If point D is not far from point A compared to the mean fault period, then we repeatedly calculate active and inactive intervals until we have reached time D and can decide whether to inject a fault or not. If the distance from point A to point D is longer than a specific value (a predefined multiple of the mean fault period), we do not use the repeated calculation method. Instead, we assume that the end point of an active interval prior to time D is randomly and uniformly distributed between D and D minus the mean fault period. Once we have randomly placed the end of this active interval, we calculate active and inactive intervals from that point until we have reached time D. For events that are far enough apart in time, this method is statistically indistinguishable from the brute force method, but it requires significantly less computation time.

4 Fault Injection Experiments

In the experiments described herein, a fault-tolerant clock synchronization application was developed using MPI and tested with FIMD-MPI.

4.1 The Algorithm and its Implementation

The purpose of the fault-tolerant clock synchronization algorithm is to synchronize the clocks of the running processes, despite the presence of some processes that are influenced by arbitrary faults. Synchronization algorithms typically execute in a sequence of synchronization rounds. For most synchronization algorithms, each process receives, once per round, messages containing clock information from a subset of the other processes. Based on the clock information of other processes and its own local clock value, a process determines in each round a clock offset and adjusts its clock by this amount. In our experiments, we implemented a hierarchically-partitioned multistep interactive convergence (ICV) clock synchronization algorithm, referred to as m-ICV [6]. m-ICV uses an m-dimensional array as a logical model for communication between processes and exchanges clock information multiple times per round. Each synchronization round is divided into m steps. During step i, processes that share a row in the i^{th} dimension of the array exchange clock values. During each step of one round, a clock offset is calculated using only the clock values received during that step. After the last step of the round, all the clock offsets on a node are combined to produce the final clock adjustment for that node. For more detail on this algorithm the interested reader is referred to [6].

Our implementation of m-ICV involves 16 processes running on a network of SunOS hosts and uses MPI to exchange messages between the processes. The machines used in the experiments cover SunOS versions 5.5.1, 5.6 and 5.7. For this experiment, we executed the program on 8 machines with each machine running two processes. Although two processes share the same machine, they each run their own separate logical clock.

The logical communication model we adopted for the implementation is a 4×4 array. Within each round, there are two steps. In the first step, the processes that share a row in the array exchange clock values with each other and each calculates a row offset. The second step works similarly among processes that share a column in the array. After the second step, the row and column offsets are combined to produce the final adjustment for each process. This implementation can tolerate one fault of arbitrary type per row and per column of the array. Hence, from 1 to 4 faults of arbitrary type can be tolerated depending on their location.

4.2 Experimental Results

Our m-ICV implementation was tested by running a wide variety of fault injection experiments on it using FIMD-MPI.

4.2.1 Experiment Set-Up and Results Overview

The experiments were run with a synchronization round duration of 20 seconds. Experiments lasted for 400 synchronization rounds, which corresponds to more than two hours of real time. During this time, faults were injected continuously into different nodes. Two classes of fault injection experiments were performed, deterministic and random.

Extensive deterministic fault injections were carried out to verify that our m-ICV implementation tolerates the ap-

0-7695-0574-0/2000 $10.00 © 2000 IEEE

propriate arbitrary fault models. Single faults of all unicast models supported by FIMD-MPI were injected continuously during an experiment.[2] The identity of the faulty node, the affected destinations, and the fault timing were all varied for each of these fault models. A separate set of deterministic experiments was also run in which four simultaneous faults, all in different rows and different columns of the array, were specified. The fault models and timing were again varied for these multiple-fault specifications. For every deterministic fault injection experiment performed, the algorithm maintained synchronization continuously for the entire duration of 400 synchronization rounds.

The random injection experiments were used to test the random timing methods of FIMD-MPI, and also to investigate the capability of m-ICV to handle temporary fault bursts, in which the fault limit of the algorithm was temporarily exceeded. The m-ICV implementation was shown to be robust in the sense that, during these experiments, nodes would temporarily lose synchronization during a fault burst but would then regain synchronization as soon as the exceptional fault condition terminated.

To illustrate the type of experiment that can be carried out with FIMD-MPI, the next section describes in detail one set of deterministic fault injections that we performed.

4.2.2 Details of One Experiment

In this experiment, four permanent delay faults were injected simultaneously. All messages from node 3 to node 1, node 6 to node 10, node 9 to node 5, and node 12 to node 15 were delayed by 3.5 seconds, 2.7 seconds, 1.2 seconds, and 5.0 seconds, respectively. We show results from an abbreviated 10 round version of the experiment.

We focus on the messages from node 6 to node 10 to illustrate the results of this experiment. A portion of the log file for the experiment is shown in Figure 6. Note that each fault injected into node 6 is recorded in the log with two entries. An entry is recorded when the message would have been sent if no delay were introduced. A second entry is recorded when the delayed message is sent and this entry includes the *actual* amount of delay that occurred. The two entries allow the experimenter to easily see when the message should have been sent, what the intended delay was, and what the actual delay incurred by the message was.

Figure 7 shows the output of node 10 from Round 5 of the experiment. The figure shows that a clock value of 13.21124 was received from node 6 at local time 15.94990. Thus, the clock difference calculated by node 10 for node 6 is approximately 2.7 seconds (-2.73866 in Figure 7), which is significantly larger than it should be under fault-free operation. The function which calculates the clock offsets discards the smallest and largest differences and averages the remaining two values. Hence, the function filters the large clock difference and enables node 10 to calculate a small adjustment at this round. Similar results were seen on the outputs of nodes 1, 5, 10, and 15 at every round, demonstrating that the implementation worked correctly despite the occurrence of the delay faults.

[2] Since our m-ICV implementation used solely unicast communication, multicast fault models could not be injected.

```
--- Log for node 5 ---
No fault is injected on node 5
Message count: 60

--- Log for node 6 ---
[model][submod][dest][meth][delay][time][count][ref]
              .
              .
              .
-- The following entries are put by the event manager
Timing Delay 10 detm time 2.700000 133.543807 41 1

-- The following entries are put by the timerHandler
Timing Delay 10 detm time 2.707530 136.251417 41 1

-- The following entries are put by the event manager
Timing Delay 10 detm time 2.700000 153.534832 47 1

-- The following entries are put by the timerHandler
Timing Delay 10 detm time 2.708683 156.243597 47 1
              .
              .
              .
Message count: 60

--- Log for node 7 ---
No fault is injected on node 7
Message count: 60
```

Figure 6. A Portion of the Log File for Delay Fault Experiment

```
Proc 10, Round 5: Omissions= 0 Started= 180.48
                  offset= 44.25432675 adjust= -0.04064008
Values Received:
recv[0]:  4.60285 at time:  4.62354    diff[0]: -0.02069
recv[1]:  4.86609 at time:  4.86267    diff[1]: 0.00343
recv[2]:  0.00000 at time:  0.00000    diff[2]: 0.00000
recv[3]:  5.31610 at time:  5.25245    diff[3]: 0.06365
recv[4]: 12.45368 at time: 12.53839    diff[4]: -0.08471
recv[5]: 13.21124 at time: 15.94990    diff[5]: -2.73866
recv[6]:  0.00000 at time:  0.00000    diff[6]: 0.00000
recv[7]: 14.86700 at time: 14.65165    diff[7]: 0.21535
```

Figure 7. A Portion of the Output File for Delay Fault Experiment

5 Discussion

FIMD-MPI is a tool for injecting faults into MPI applications. It can inject a wide variety of faults including omission, delay, corruption of both message headers and data, spurious and replicated messages, and both symmetric and asymmetric multicast. FIMD-MPI has been used to successfully test several fault-tolerant applications that use

0-7695-0574-0/2000 $10.00 © 2000 IEEE

MPI to pass messages including a fault-tolerant clock synchronization algorithm described in the paper.

One interesting observation that was made early in the testing process is that applications that use blocking synchronous communication, e.g. MPISSend() with MPIRecv(), are inherently susceptible to faults. In these applications, a single send omission will cause the receiving process to block forever which can in turn cause other processes to block. In effect, blocking synchronous communication is a rapid error-propagation mechanism that can permit a single send omission to deadlock an entire application. Programmers who are designing fault-tolerant applications should therefore be careful to avoid using this communication mode in their programs.

Even when using non-blocking communication, omissions can eventually create deadlocks if messages that can not be delivered fill or reserve all the buffer space. It, therefore, becomes necessary for a fault-tolerant application to detect when non-blocking communications have failed and cancel them in order to prevent eventual deadlock.

A final observation is that the application is only as fault-tolerant as the underlying system on which it runs. We noted during our experiments that the MPICH implementation we used is not very robust. For example, we could simulate a crash fault by omitting all messages coming out of a process and our application worked perfectly. However, if we actually killed one of the application's processes, MPICH panicked due to the occurrence of socket-level errors and it aborted the application. Development of a robust MPI implementation would be an interesting and extremely worthwhile research project.

References

[1] M. Barborak, M. Malek, and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, Vol. 25, pp. 171–220, June 1993.

[2] D. Blough and H. Brown, "The Broadcast Comparison Model for On-Line Fault Diagnosis in Multicomputer Systems: Theory and Implementation," *IEEE Transactions on Computers*, vol. 48, pp. 470–493, May 1999.

[3] D. Blough and T. Torii, "Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message-Passing Parallel Computers," *Digest of the 27th International Symposium on Fault-Tolerant Computing*, pp. 258–267, 1997.

[4] F. Cristian, H. Aghili, and R. Strong, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement" *Digest of the 15th International Symposium on Fault-Tolerant Computing*, pp. 200–206, 1985.

[5] F. Cristian, H. Aghili, and R. Strong, "Clock Synchronization in the Presence of Omissions and Performance Faults and Processor Joins," *Digest of the 16th International Symposium on Fault-Tolerant Computing*, pp. 218–223, 1986.

[6] M.M. de Azevedo and D. Blough, "Multistep Interactive Convergence: An Efficient Approach to the Fault-Tolerant Clock Synchronization of Large Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, pp. 1195–1212, December 1998.

[7] D. Dolev, "The Byzantine Generals Strike Again," *Journal of Algorithms*, vol. 3, pp. 14–20, 1982.

[8] S.L. Johnsson and C.-T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercubes," *IEEE Transactions on Computers*, vol. 38, pp. 1249–1268, September 1989.

[9] R. Kieckhafer, C. Walter, A. Finn, and P. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers*, Vol. 37, pp. 398–405, April 1988.

[10] L. Lamport and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults," *Journal of the ACM*, vol. 32, pp. 52–78, January 1985.

[11] B.F. Lewis, *et al.*, "COSMOS Multicomputer Operating System and Development Environment Functional Specification", *NASA Technical Memorandum*, Caltech, JPL, Aug. 1992.

[12] B.F. Lewis and R.L. Bunker, "MAX: An Advanced Parallel Computer for Space Applications", *Proc. of the 2nd Int. Symp. on Space Info. Systems*, Sep. 1990.

[13] P. Lincoln and J. Rushby, "A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model," *Digest of the 23rd International Symposium on Fault-Tolerant Computing*, pp. 402–411, 1993.

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

[15] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, vol. 1, pp. 222–238, August 1983.

[16] P. Thambidurai and Y-K Park, "Interactive Consistency with Multiple Failure Modes," *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pp. 93–100, 1988.

[17] J.H. Wensley, *et al.*, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, pp. 1240–1255, October 1978.

0-7695-0574-0/2000 $10.00 © 2000 IEEE