Low-cost Fault-tolerance in Barrier Synchronizations

Sandeep S. Kulkarni *Anish Arora*

Department of Computer and Information Science [1]
The Ohio State University
Columbus, OH 43210 USA

Abstract

In this paper, we show how fault-tolerance can be effectively added to several types of faults in program computations that use barrier synchronization. We divide the faults that occur in practice into two classes, *detectable* and *undetectable*, and design a fully distributed program that tolerates the faults in both classes. Our program guarantees that every barrier is executed correctly even if detectable faults occur, and that eventually every barrier is executed correctly even if undetectable faults occur. Via analytical as well as simulation results we show that the cost of adding fault-tolerance is low, in part by comparing the times required by our program with that required by the corresponding fault-intolerant counterpart.

Keywords: fault-tolerance, multitolerance, detectable and undetectable faults, synchronization, concurrency.

---

[1] Email: {kulkarni,anish}@cis.ohio-state.edu; Web: http://www.cis.ohio-state.edu/{~kulkarni,~anish }. Research supported in part by NSF Grant CCR-93-08640, OSU Grant 221506, and NSA MDA904-96-1-1011.

1 Introduction

In this paper, we show how to effectively add tolerance to several types of faults in program computations that use barrier synchronization. Barrier synchronization involves a collective communication between processes, in order to establish that all processes have reached a barrier before any process can compute beyond that barrier. We focus our attention on this form of synchronization because it is frequently used in implementing parallel algorithms on message passing, shared memory, and network of workstation systems.

Each of the above-mentioned systems is subject to various types of faults in practice. Examples of standard fault types include:

- *Communication faults* such as loss, corruption, duplication, reorder, and unexpected reception of messages; and failure and repair of channels.
- *Processor faults* such as fail-stop, repair, and rebooting of a processor.
- *Process faults* such as internal/design faults and "hanging" processes.
- *System faults* such as system reconfiguration, memory leaks, memory corruption, I/O faults, deadlocks, and non-availability of buffers or other resources.
- *Performance faults* such as network congestion and floating point errors.

Despite the existence of these multiple types of faults in practice and an extensive literature on barrier synchronization [1-9, to cite but a few], we are aware of little research that has considered fault-tolerance in this context. Among the exceptions are [6], which deals with message loss and corruption only, and [9], which relies upon high atomicity shared memory access. We are therefore led to designing and analyzing a fully distributed program for barrier synchronization computations that provides tolerance to most of the faults discussed above.

Issues in dealing with multiple types of faults. When multiple types of faults are considered, it is possible that some "high" type of tolerance cannot be achieved with respect to each type of faults. For example, in the presence of all of the faults mentioned above, it is clrearly impossible to guarantee that each barrier is executed correctly. On the other hand, even if some "low" type of tolerance can be achieved with respect to each fault type, this may yield unacceptable functionality or performance with respect to some fault type. Consider, for instance, the tolerance guarantee that if any fault occurs the program will be eventually restored to a state from where every subsequent barrier is executed correctly. This type of tolerance may not be appropriate for faults such as message loss, in whose presence it is possible to easily and efficiently guarantee that every barrier executes correctly. It follows that one issue is to identify the potentially different types of tolerances that are respectively appropriate for the different fault types.

With this issue in mind, we classify the faults mentioned above into two: (i) the class of faults that can be masked, i.e., the faults in whose presence, it is possible to ensure that each barrier is executed correctly, and (ii) the class of faults that cannot be masked, i.e., the faults in whose presence, it is impossible to ensure that each barrier is executed correctly. We call these classes of faults *detectable* and *undetectable* respectively. While we give the formal description of what we mean by detectable and undetectable in the next section, we note that the former fault-class includes faults such as message loss, unexpected message reception, fail-stop, repair, and rebooting of a processor; and the latter includes faults such as internal/design errors, hanging processes, memory corruption, and memory leaks. (The interested reader is referred to [10] for heuristics for the selection of fault-classes.)

A second issue is whether to differentiate between the mechanisms used to achieve the same type of tolerance for the different types of faults within the same fault-class. On one hand, choosing a uniform mechanism for dealing with all faults in a fault-class yields a simpler design. On the other hand, choosing a potentially different mechanism for dealing with different faults in a fault-class may enable better optimization of the cost of achieving tolerance.

For the problem at hand, we prefer the uniform mechanism approach, based on the following arguments. Simpler designs are easier to implement, they are less prone to interference between the fault-tolerance mechanisms and are, hence, more reliable. In addition, if the overhead of adding fault-tolerance is small, then the payoff in differentiating the mechanisms is not significant.

Goals of our design. Tolerance properties. As motivated above, we seek to provide tolerance to the class of detectable faults as well as the class of undetectable faults. In the presence of the former fault-class, we will ensure that each barrier is executed correctly, i.e., *masking* tolerance will be exhibited with respect to the detectable fault-class. And, in the presence of the latter fault-class, we will ensure that even if the program reaches an arbitrary state, it eventually recovers to a state from where each barrier is executed correctly, i.e., *stabilizing* tolerance will be exhibited with respect to the undetectable fault-class [11]. Moreover, until the program recovers to such a state, the number of barriers executed incorrectly will be kept to a minimum. (We choose to design stabilizing tolerance to undetectable faults as it accommodates a large set of faults, given that the program recovers from an arbitrary state.)

Overhead of tolerances. We aim to keep the overhead due to the tolerances as small as possible, since parallel algorithm designers are unlikely to use barrier synchronization primitives where fault-tolerance comes at a significant cost.

Hardware implementation. Barrier synchronization programs are sometimes implemented in hardware, for reasons of efficiency. One of our goals, therefore, is to design a program that is simple and that uses small data structures, to enable easy implementation in hardware.

MPI implementation. Currently, MPI [12] provides users with two alternatives for dealing with faults: (i) to abort the program in the event of a fault, and (ii) to return an error code in the event of a fault, so that the user may be able to effect a recovery. Another of our goals is to provide a third alternative to users of barrier synchronizations in MPI: the guarantee of an appropriate type of tolerance to each fault-class.

Stepwise design. We will develop our program via a sequence of refinement steps. More specifically, we will first start with a shared memory program wherein process actions can instantaneously access the state of all other processes. With this assumption, the program design and its proof of correctness can be simplified. Subsequently, we will relax this assumption in steps until the process actions can be implemented on a message passing system. In each step, we will verify that the program is a refinement of the program in the previous step, enabling a simple proof of correctness for the final program.

Outline of the paper. The rest of the paper is organized as follows. In Section 2, we formally specify the problem and the faults-classes. In Section 3, we present a solution for the problem which is coarse-grain in the sense that each process action can instantaneously communicate with all other processes and update its own state. In Section 4, we distribute this solution so that each process action can instantaneously communicate with only its neighboring processes and also update its own state. In Section 5, we further refine the granularity of process actions so that each action can instantaneously either communicate with one neighboring process or update its own state, but not both. In Section 6, we present analytical and simulation results for the performance of our program in the presence of detectable and undetectable faults, and quantify the overhead of fault-tolerance. Finally, we discuss extensions of our program and its fault-tolerances in Section 7 and make concluding remarks in Section 8.

2 Problem Specification and Fault Model

Specification of barrier synchronization. Given is a system of processes, each of which consists of a cyclic sequence of n terminating phases, $[phase.0 \; ; \; phase.1 \; ; \; \ldots \; ; \; phase.(n{-}1)]$. The following two properties are required for each i, $0 \leq i < n$:

(Safety) Execution of $phase.(i{+}1)$ begins only after *phase.i is executed successfully.*

(Progress) Eventually *phase.i* is executed successfully,

where $phase.n = phase.0$. Initially, $phase.(n-1)$ has executed successfully and each process is thus ready to execute *phase.*0. "*Phase.i* is executed successfully" is defined in the following terms.

Definition. An *instance of phase.i is executed* iff some process starts executing *phase.i* and each process executes *phase.i* at most once.

Note that when an instance of *phase.i* is executed, some processes may execute *phase.i* partially or not at all.

Definition. An *instance of phase.i is executed successfully* iff all processes execute *phase.i* fully in that instance.

Definition. *Phase.i is executed successfully* iff one or more instances of *phase.i* are executed in sequence, the last instance of which is executed successfully. □

From this specification, it follows that in a barrier synchronization program the processes execute the next phase only after all processes have completed execution of the previous phase. As described next, in the presence of faults, to execute *phase.i*, successfully, multiple instances of *phase.i* may be executed (successfully or unsuccessfully) in sequence. Of course, in the absence of faults, any reasonable implementation should execute *phase.i* exactly once.

Faults. As discussed in the introduction, we classify the faults into two classes: detectable and undetectable.

Detectable faults. A fault is detectable if the state of the process where the fault occurs can be reset before any process accesses it. Faults such as message loss, detectable corruption, duplication, reorder; processor fail-stop, repair or reboot; I/O errors; exceptions such as floating point errors and access violations; and system reconfiguration are detectable faults. By way of explanation, a fault that reboots a processor is detectable because the state of the processes executing on that processor can be reset before restarting the processes.

Note that a reset after a fault may yield a state that is different from that before the fault. Therefore, in the presence of detectable faults, the local state of a process may be lost. Thus, information regarding the current phase being executed by that process may be lost. It follows that, in the presence of detectable faults, we cannot ensure that each process executes its current phase exactly once. However, we can still satisfy the specification of barrier synchronization by executing a new instance of the current phase successfully.[2] Note that, in order to satisfy Safety, this new instance must begin when no process is executing in the current instance.

Undetectable faults. A fault is undetectable if the state of the process where the fault occurs cannot be reset before any process accesses it. Faults such as internal/design errors, hanging processes, undetectable message corruption or reorder, memory leaks and transient state corruptions are undetectable faults. By way of explanation, a transient state corruption is undetectable as some processes may inadvertently access the corrupted state without detecting that it is in error. Sometimes, even if faults are detectable in principle, there may be factors that limit the ability of systems to detect them, e.g., the cost of detection [12]; such faults may instead be classified as undetectable.

In the presence of undetectable faults, the program may be perturbed to a state where processes are executing in different phases. It follows that the specification of barrier synchronization cannot be satisfied in the presence of such faults. Therefore, we ensure that even if the program is perturbed to an arbitrary state, it will eventually recover to a state from where the subsequent execution will satisfy the specification of barrier synchronization, and, the number of phases executed unsuccessfully in the interim is kept to a minimum.

Fault Assumptions. The following two assumptions apply to all faults:

1. Faults are eventually correctable, i.e., no part of the program is permanently affected by them. For example, if a processor fail-stops, one way to eventually correct is to restart all fail-stopped processes of that processor on some other processor—albeit with different states. (We will discuss how to relax the assumption of eventual correctability in Section 7.)

[2]This assumes that after a detectable fault occurs the identity of the current phase can be accessed from the state of some process. We will therefore treat the fault that corrupts the state of all processes detectably as an undetectable fault.

2. Faults can occur at any time in any order and at any process. However, eventually they stop occurring. (Alternatively, they stop occurring for at least a sufficient duration so that the program makes progress.)

Programming notation. We write programs using a guarded command notation: Each process consists of a finite set of variables and a finite set of actions. Each action consists of two parts: a guard and a statement. For convenience, a unique name is associated with each action. Thus, each action has the following form:

$$\langle name \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$$

The guard is a boolean expression over the variables of that and possibly other processes, and the statement updates zero or more variables of that process. An action is executed only if it is enabled, i.e., if its guard evaluates to true. To execute the action, its statement is executed atomically.

Each computation of the program is a fair interleaving of steps: In every step, some action that is enabled in the current state is chosen and its statement is executed atomically. Fairness of the interleaving means that each action that is continuously enabled is eventually chosen for execution in some step.

We represent each fault by an action. An undetectable fault assigns to variables of a process nondeterministically chosen values from their domains. A detectable fault assigns to variables of a process "reset" values from their domains. (Note that this reset would in practice be implemented in the process, but since the corrupted state of the process cannot be accessed by any process until the reset is complete, it is convenient to specify the reset as part of the fault action.)

3 Coarse-Grain Solution

In this section, we present a solution for the case where the graph of the processes is fully connected and the process actions can instantaneously communicate with the neighboring processes and update their own state. The resulting program recovers from detectable faults with no phase executed incorrectly and from undetectable faults that perturb processes into, say, m distinct phases with at most m phases executed incorrectly. For ease of exposition, we first assume that the cyclic sequence of phases consists of at least two phases. We later remark on how the resulting program can be used for the case where the cyclic sequence of phases consists of a single phase.

To synchronize the underlying computation, each process j maintains a control state, which we represent by the variable $cp.j$ (for control position of j). Clearly, there exists a $cp.j$ value, say *execute*, which denotes that j is executing its phase, and a value, say *success*, which denotes that j has completed its phase. We consider two additional control position values: *ready*, which denotes that j is ready to execute its phase, and *error*, which denotes that the control position of j is detectably corrupted. Also, to distinguish between the phases, each process j maintains a variable $ph.j$, whose value denotes the number of the phase that j is currently in.

Informally, the program works as follows. In a start state, all processes are in the control position $ready$ and in the same phase. In the absence of faults, processes first change their control position to *execute*, execute the current phase, and change their control position to $success$. They then increment their phase and change their control position to $ready$, thereby returning to a start state, from where the cycle repeats. In the presence of a detectable fault at a process, the control position of that process is changed to $error$. And if some process is in control position $error$, all processes are pulled back into the start state of the current phase, from where a new instance of the current phase is executed. In the presence of undetectable faults, the program recovers to some start state, from where the subsequent execution satisfies Safety and Progress.

Formally, the state transitions between the control position values of j are shown in Figure 1. We describe the conditions under which j executes these transitions next.

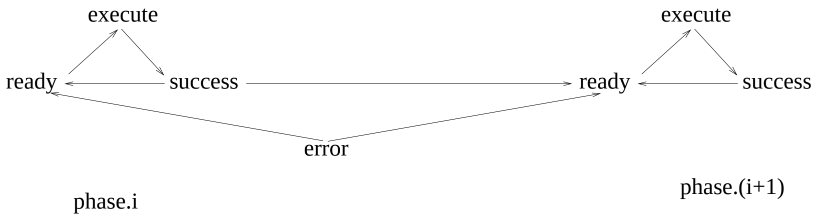

Figure 1: State transitions for program CB

The transition from $ready$ to *execute* occurs only after some process checks that all processes are in control position $ready$. After the first process changes its control position thus from $ready$ to *execute*, the remaining processes can likewise change their control position without checking the state of all processes; they need only check that some process is in control position *execute*.

The transition from *execute* to $success$ occurs only after all processes start execution of their phases. This restriction is introduced to tolerate detectable faults, as motivated by the following scenario: Consider a system of two processes, say j and k, in a start state. Let j change its control position to *execute*, execute its phase, and then change its control position to $success$, in which case a state is reached where the control position of j is $success$ and that of k is $ready$. The same state is also reached when k is subject to a detectable fault and it then changes its control position to $ready$ in order to execute a new instance of the current phase. In the first case, k should change its control position to *execute* and execute its phase, and in the second case, j should change its control position to $ready$ in order to execute a new instance of the current phase. By introducing this restriction, we prevent j from changing its control position to $success$ in the second case, which in turn prevents the violation of *Progress* that would occur were j to subsequently change its control position to $ready$ in order to execute a new instance of the current phase.

The transition from $success$ to $ready$ occurs only when no process is in the control position *execute*. This restriction is introduced to tolerate undetectable faults: It prevents the system from remaining forever in states where the control positions *ready*, *execute*, and *success* all co-exist. More specifically, as we prove later in this section, it enables the system to recover from such states to start states, from where the computation resatisfies the specification of barrier synchronization. The transition from *success* to *ready* also chooses the phase in which the process will execute next. For the first process that executes this transition in the current phase, if the control position of all processes is *success* (i.e., all processes have completed the current phase successfully), its phase is incremented, thereby leading to the execution of the next phase; if, however, the control position of some process is *error* (i.e., the instance of the current phase is not executed successfully), the phase is unchanged, thereby leading to a execution of a new instance of the current phase. After the first process executes transition to *ready*, other processes obtain their phase from any process whose control position is *ready*.

The transition from *error* to *ready* also occurs only when no process is in the control position *execute*. This restriction ensures that if any detectable corruption occurs during execution of a phase, execution of a new instance of that phase does not begin as long as some process is executing in that instance. This transition also obtains the phase in which the process will execute next. If there exists a process that is in control position *ready*, the process obtains its control position from that process. Otherwise, it obtains the phase from some process that is control position *success*. If, however, there is no process in control position *ready*, i.e., the phase of all processes is corrupted, the phase is chosen arbitrarily.

Our barrier synchronization program, CB, consists of four actions, one for each transition discussed above:

$$
\begin{array}{llllll}
CB1 :: & cp.j = ready & \wedge & ((\forall k :: cp.k = ready) \vee (\exists k :: cp.k = execute)) & & \longrightarrow \; cp.j := execute \\[1ex]
CB2 :: & cp.j = execute & \wedge & ((\forall k :: cp.k \neq ready) \vee (\exists k :: cp.k = success)) & & \longrightarrow \; cp.j := success \\[1ex]
CB3 :: & cp.j = success & \wedge & (\forall k :: cp.k \neq execute) & & \longrightarrow \\
 & & \text{if} & (\exists k :: cp.k = ready) & \text{then} & ph.j := (\underline{any}\ k : cp.k = ready : ph.k) \\
 & & \text{elseif} & (\forall k :: cp.k = success) & \text{then} & ph.j := ph.j + 1; \\
 & & cp.j := ready & & & \\[1ex]
CB4 :: & cp.j = error & \wedge & (\forall k :: cp.k \neq execute) & & \longrightarrow \\
 & & \text{if} & (\exists k :: cp.k = ready) & \text{then} & ph.j := (\underline{any}\ k : cp.k = ready : ph.k) \\
 & & \text{elseif} & (\forall k :: cp.k = success) & \text{then} & ph.j := (\underline{any}\ k : cp.k = success : ph.k); \\
 & & cp.j := ready & & &
\end{array}
$$

$$
\text{where} \quad (\underline{any}\ k : k \in X : ph.k) \;\; \begin{array}{lll} = & \text{the phase of any process in the set } X & \text{if } X \neq \{\} \\ = & \text{an arbitrary number in the set } \{0..n-1\} & \text{otherwise} \end{array}
$$

Faults. The detectable and undetectable faults that affect the program are represented, respectively, as follows:

$$true \longrightarrow ph.j, cp.j := ?, error$$

$$true \longrightarrow ph.j, cp.j := ?, ?, \qquad \text{where ? is any value from the domain (of } cp.j \text{ or } ph.j\text{)}$$

Proof of correctness: To prove Safety, we show (i) no two instances of *phase.i* overlap, and (ii) execution of an instance of *phase.*$(i{+}1)$ is executed only after a successful instance of *phase.i*. (Observe that (i) is trivial in the absence of faults as only one instance is executed.)

To prove Progress, we show (i) after *phase.*$(i-1)$ is executed successfully, in the absence of faults, any instance of *phase.i* is executed successfully, and (ii) in the presence of faults, eventually a new instance of *phase.i* is executed.

Lemma 3.1 Program CB satisfies the specification of barrier synchronization in the absence of faults.

Proof. *(Safety)* In the absence of faults, when the first process increments its phase from i to $i+1$, all processes are in control position *success*, i.e., all processes have executed *phase.i* successfully. It follows that a process executes *phase.*$(i+1)$ only after all processes have executed *phase.i* successfully . Thus, Safety is satisfied.

(Progress) In a state where *phase.*$(i-1)$ has been executed successfully, i.e., in a state where the phase of all processes is $i-1$ and all are in control position *success*, action $CB3$ is enabled at all processes. Thus, each process executes $CB3$ and increments its *phase* to i and changes its control position to *ready*. After all processes change their control position to *ready*, they all can execute $CB1$ to begin executing *phase.i*. Finally, after all processes begin executing *phase.i*, they all can execute $CB2$ to execute *phase.i* successfully. Thus, Progress is satisfied. □

Lemma 3.2 Program CB is masking tolerant to the class of detectable faults.

Proof. *(Safety)* In the presence of detectable faults, a process that is subject to the fault changes its control position to *error*. Since the first process to execute a new instance of *phase.i* checks that all processes are in control position *ready* and since a process changes its control position from *error* to *ready* only when no process is in control position *execute*, it follows that a new instance of *phase.i* does not overlap with the previous instance. Moreover, as shown in Lemma 3.1, execution of *phase.*$(i+1)$ begins only after a successful instance of *phase.i*. Thus, Safety is satisfied.

(Progress) In the presence of detectable faults, if some process is in control position *error* and the current phase is i, the first process to execute $CB3$ or $CB4$ sets its phase to i. Later, the remaining processes follow likewise. From the resulting start state, a new instance of *phase.i* is executed. Moreover, as shown in Lemma 3.1, the new instance of *phase.i* is executed successfully. Thus, Progress is satisfied. □

Lemma 3.3 Program CB is stabilizing tolerant to the class of undetectable faults.

Proof. In the presence of undetectable faults, CB eventually reaches a state where the control position of all processes is either *success* or *error*. To see this, observe that if the program is not in such a desired state, some processes are in control position *ready* or *execute*. Among such processes, we consider the following three cases: (1) all are in control position *execute*, (2) all are in control position *ready*, and (3) some are in *ready* and some in *execute*.

In the first case, all processes in control position *execute* change their control position to *success*, thus reaching the desired state. In the second case, eventually all processes change their control position to *ready*. Thereafter, all processes change their control positions to *execute*, after which all change their control position to *success*, thus reaching the desired state. In the third case, eventually processes in control position *ready* change their control position to *execute*, those in *execute* change their control position to *success*, and those in *success* or *error* remain in *success* or *error*, thus reaching a state that satisfies the first or the second case and, hence, eventually reaching a desired state.

When all processes are in control position *success* or *error*, some process changes its control position to *ready*. The remaining processes follow, change their control position to *ready*, and copy the phase of this first process. From the resulting state, the specification of barrier synchronization is satisfied for all phases. □

Lemma 3.4 In the presence of undetectable faults, the number of phases executed unsuccessfully is kept to a minimum.

Proof. If the system is perturbed to a state where processes are in m distinct phases, each of these m phases may execute incorrectly with respect to the barrier synchronization specification. But correct execution resumes before any more phases execute incorrectly. To see this, observe that if a process executes $CB3$ to enter a new phase, that phase is executed correctly. Thus, at most m phases are executed incorrectly. □

Remark. If the cyclic sequence of phases consists of a single phase, program CB is modified to drop the variable ph: When a process executes $CB3$ or $CB4$, depending upon whether there was a fault in the execution of the current phase, it decides whether a new instance of the current phase is to be executed or the next phase is to be executed. An alternative solution is to map the single phase case onto the multiple phase case, without loss of generality, by replicating the single phase.

4 Refinement 1 : Accessing Neighbors Only

We now refine program CB so that instead of communicating with all processes, each process communicates with only one neighboring processes at a time. The refinement is based on the observation that in every phase execution the first process to change its control position to *ready* or to *execute* or to *success* necessarily detects the state of all other processes. So to refine CB, we let a distinguished process, say process 0, bear the responsibility of all of these detections. After 0 has detected the appropriate global condition, it changes its phase and control position, and the other processes then follow 0 one after another to change their phase and control position accordingly.

First, in Section 4.1, we refine CB for the case where the processes are organized in a ring. Then, in Section 4.2, we refine it for other topologies.

4.1 Ring Topology

We organize the processes, say $0..N$, in a ring and circulate a token around the ring. When 0 gets the token, it locally detects the global condition involving all processes, changes its phase and control position accordingly, and then forwards the token to its successor, process 1. Upon receiving the token, each non-0 process updates its phase and control position accordingly and then forwards the token to its successor. When 0 receives the token again, from N, it can locally detect the next global condition on the state of the processes, and the cycle repeats.

It follows that we will superpose the refined barrier synchronization upon an underlying multitolerant token ring program. Next, we briefly recall such a token ring program, which we have formally derived and proven correct elsewhere [10].

Underlying token ring program. Each process j maintains a sequence number, $sn.j$, which is in the domain $\{0..K-1\}$ for some $K > N$ in the absence of detectable faults. To handle detectable faults, two special values $\bot$ and $\top$ are added to the domain of the sequence number: when the sequence number of a process is corrupted, it is set to $\bot$, and the sequence number $\top$ is used to detect whether a detectable fault has occurred at that process. Process $j, j \neq N$, has the token iff $sn.j \neq sn.(j+1)$ and both $sn.j$ and $sn.(j+1)$ are different from $\bot$ and $\top$. Process N has the token iff $sn.N = sn.0$ and $sn.N$ and $sn.0$ are both different from $\bot$ and $\top$.

Notational remark. In the rest of the paper, our use of $+$ and $-$ is context sensitive: when used in connection with the process numbers, these operations are in modulo $N+1$ arithmetic, when used in connection with the sequence numbers, they are in modulo K arithmetic, and when used in connection with the phases of processes, they are in modulo n arithmetic. (End of remark.)

The token ring program has five actions. The first action lets process 0 receive the token, and the second action lets process j, $j \neq 0$, receive the token. The remaining three actions are used to detect if the state of all processes is corrupted. Formally, the actions are as follows:

$$
\begin{array}{lll}
T1 :: j=0 \ \wedge\ sn.N \neq \bot \ \wedge\ sn.N \neq \top \ \wedge\ (sn.j = sn.N \ \vee\ sn.j = \bot \ \vee\ sn.j = \top) & \longrightarrow & sn.j := sn.N + 1 \\
T2 :: j \neq 0 \ \wedge\ sn.(j-1) \neq \bot \ \wedge\ sn.(j-1) \neq \top \ \wedge\ sn.j \neq sn.(j-1) & \longrightarrow & sn.j := sn.(j-1) \\
T3 :: sn.N = \bot & \longrightarrow & sn.N := \top \\
T4 :: j \neq N \ \wedge\ sn.j = \bot \ \wedge\ sn.(j+1) = \top & \longrightarrow & sn.j := \top \\
T5 :: sn.0 = \top & \longrightarrow & sn.0 := 0
\end{array}
$$

This program has the following properties: In the absence of faults, it repeatedly circulates exactly one token around the ring. In the presence of detectable faults, (a) the ring contains at most one token but the program eventually reaches a state where the ring contains exactly one token; (b) each process can detect if it has been detectably corrupted by checking whether its sequence number is $\bot$ or $\top$; and (c) 0 never executes actions $T4$ and $T5$ (these are executed only in the case of undetectable faults, including the scenario when all processes are corrupted concurrently). Finally, in the presence of undetectable faults, the ring may contain any number of tokens but the program eventually reaches a state where the ring contains exactly one token.

Superposed program for updating cp and ph variables. Informally, the refined program works as follows. Each process updates its phase and control position whenever it receives a token: specifically, process 0 updates whenever it executes action $T1$ and the other processes update whenever they execute action $T2$. In a start state, all processes are in the same phase and in control position $ready$, and action $T1$ is enabled at process 0. Process 0 executes $T1$, increments its sequence number, and changes its control position to $execute$. This enables action $T2$ at process 1, which then updates $cp.1$ and $ph.1$ likewise, and so on until N updates $cp.N$ and $ph.N$ likewise. Subsequently, 0 changes its control position to $success$ and in the next circulation of the token around the ring, all processes change their control position to $success$. With this circulation of the token, 0 can detect whether some process had a detectable fault, by using a new control position, say $repeat$, as follows: If in this circulation the token reaches a process that had a detectable fault, that process changes its control position to $repeat$ (instead of $success$). The control position $repeat$ is propagated along with the token to process N. Therefore, if the state of some process is detectably corrupted, N will change its control position to $repeat$. In this situation, 0 decides to execute a new instance of the current phase. Else, if the control position of N is $success$ and $ph.0 = ph.N$, 0 detects that all processes have executed $ph.0$ and, hence, 0 decides to increment its phase.

Formally, the phase and the control position of j are updated as described below.

Updating $ph.0$ and $cp.0$ in process 0. When 0 receives the token (i.e., it executes action $T1$), it also executes the following statement in parallel with that of $T1$:

$$\begin{array}{lll}
\textsf{if} \;\; cp.0 = ready \wedge cp.0 = cp.N \wedge ph.0 = ph.N & \textsf{then} & cp.0 := execute \\
\text{elseif} \;\; cp.0 = execute & \textsf{then} & cp.0 := success \\
\text{elseif} \;\; cp.0 = success & \textsf{then} & \textsf{if } cp.0 = cp.N \wedge ph.0 = ph.N \textsf{ then } ph.0 := ph.0+1; \\
 & & cp.0 := ready \\
\text{elseif} \;\; cp.0 = error & \textsf{then} & ph.0 := ph.N \;;\; cp.0 := ready
\end{array}$$

Updating $ph.j$ and $cp.j$ in process j, $j \neq 0$. When j receives the token (i.e., it executes action $T2$), it also executes the following statement in parallel with that of $T2$:

$$\begin{array}{llcllll}
\multicolumn{7}{l}{ph.j := ph.(j-1) \;;} \\
\textsf{if} & cp.j = ready & \wedge & cp.(j-1) = execute & \textsf{then} & cp.j := execute \\
\textsf{elseif} & cp.j = execute & \wedge & cp.(j-1) = success & \textsf{then} & cp.j := success \\
\textsf{elseif} & cp.j \neq execute & \wedge & cp.(j-1) = ready & \textsf{then} & cp.j := ready \\
\textsf{elseif} & cp.j = error & \vee & cp.(j-1) \neq cp.j & \textsf{then} & cp.j := repeat
\end{array}$$

Faults. The detectable and undetectable faults that affect the refined program are represented, respectively, as follows:

$$true \longrightarrow ph.j, cp.j, sn.j := ?, error, \perp$$

$$true \longrightarrow ph.j, cp.j, sn.j := ?, ?, ?$$

Proof of correctness.

Lemma 4.1.1 Program RB satisfies the specification of barrier synchronization in the absence of faults.

Proof. *(Safety)* In the absence of faults, if process j has the token and its control position is $success$, then all processes in the set $\{0..j\}$ have executed their current phase. Thus, when N has the token and $cp.N$ is $success$, all processes have executed their current phase. It follows that when 0 increments its phase from i to $i+1$, $phase.i$ is executed successfully. Hence, Safety is satisfied.

(Progress) In the absence of faults, the proof of progress is identical to that of program CB. □

Lemma 4.1.2 Program RB is masking tolerant to the class of detectable faults.

Proof. *(Safety)* In the presence of detectable faults, recalling properties (b) and (c) of the underlying token ring program, we can observe that the control position of a process is $error$ iff its sequence number is either $\perp$ or $\top$. From this, we can observe that if j has the token and its control position is $ready$, then all non-corrupted processes in the set $\{0..j\}$ are in the same phase provided process 0 has not been detectably corrupted in the interim. (If 0 was corrupted in the interim, it could have changed its control position to $ready$ and copied a different phase number from process N.) Thus, when N has the token and $cp.N = ready \wedge ph.0 = ph.N$, all non-corrupted processes are in the same phase.

Also, if j has a token and its control position is $ready$, the control position of all processes in the set $\{0..j\}$ is neither $execute$ nor $success$. When 0 changes its control position from $ready$ to $execute$, process N has a token and the control position of N is $ready$. It follows that, whenever 0 changes its control position to $execute$, all processes are in the same phase and no other process is in control position $execute$. It follows that when 0 starts executing a new instance of the phase, no process is in control position $execute$, i.e., two instances of $phase.i$ do not overlap.

Finally, as shown in Lemma 4.1.1, execution of $phase.(i+1)$ begins only after a successful instance of $phase.i$. Thus, Safety is satisfied.

(Progress) In the presence of detectable faults, if some process changes its control position to $error$ in the execution of $phase.i$, eventually process N changes its control position to $repeat$. Then, 0 changes its control position to $ready$ in order to execute a new instance of $phase.i$. In the next circulation of the token, all processes change their phase to i and their control position to $ready$. In the following token circulation, a new instance of $phase.i$ is executed.

Moreover, as shown in Lemma 4.1.1, the new instance of $phase.i$ is executed successfully. Thus, Progress is satisfied. □

Lemma 4.1.3 Program RB is stabilizing tolerant to the class of undetectable faults.

In the presence of undetectable faults, eventually the token ring reaches a state where there is exactly one token and action $T1$ at 0 is enabled. When 0 executes $T1$ to receive the token, its control position is either $ready$, $execute$, or $success$. If control position of 0 is $success$, after the token is circulated once around the ring, 0 can change its control position to *ready*. If the control position of 0 is *execute*, after the token is circulated once around the ring, 0 can change its control position to *success* and then, after one more circulation of the token, to *ready*. Thus, eventually the program reaches a state where process 0 has the unique token and its control position is *ready*.

Once 0 has a token and its control position is *ready*, the following rounds of token circulation satisfy the specification of barrier synchronization. To see this, we consider two cases: (1) there does not exist a process in control position *execute*, and (2) there exists a process in control position *execute*. In the first case, after the token is circulated once around the ring, all processes are in a start state. From any start state, every computation satisfies *Safety* and *Progress*. In the second case, processes in control position *execute* change their control position to *repeat*, thus reaching a state satisfying the first case and, hence, after the token is circulated around the ring once more, the program reaches a start state from where every computation satisfies *Safety* and *Progress*. □

Lemma 4.1.4 In the presence of undetectable faults, the number of phases executed unsuccessfully is kept to a minimum.

Just as in program CB, if the system is perturbed to a state where processes are in m distinct phases, each of these m phases may execute incorrectly with respect to the barrier synchronization specification. But correct execution resumes before any more phases execute incorrectly. To see this, observe that if process 0 enters into a phase other than these m by incrementing $ph.0$, that phase is executed correctly. Thus, at most m phases are executed incorrectly. □

4.2 Alternative Topologies

Due to the ring topology of program RB (see Figure 2 (a)), $O(N)$ time is required to detect that all processes have executed their phase successfully and to inform them to begin executing the next phase. We show how this detection and information dissemination can be reduced by parallelizing RB, next.

One alternative to a ring topology is to use two rings that intersect at process $0..j$, $j \geq 0$ (see Figure 2 (b)). The refined version of program CB for the two ring topology, RB' is obtained by changing RB as follows:

1. Process 0 checks that the sequence number 0 is the same as that of both N_1 and N_2 before executing $T1$. To update ph and cp, the condition $cp.0 = cp.N$ is replaced by $cp.0 = cp.N_1 = cp.N_2$ (and likewise for $ph.0 = ph.N$), and the assignment $ph.0 := ph.N$ is replaced by choosing $ph.N_1$ or $ph.N_2$.

2. Action $T3$ is executed by both N_1 and N_2.

3. Action $T4$ is executed by all processes other than N_1 and N_2: before executing this action, process j checks that the sequence numbers of all its successors are $\top$.

4. Actions T_2 and T_5 remain unchanged.

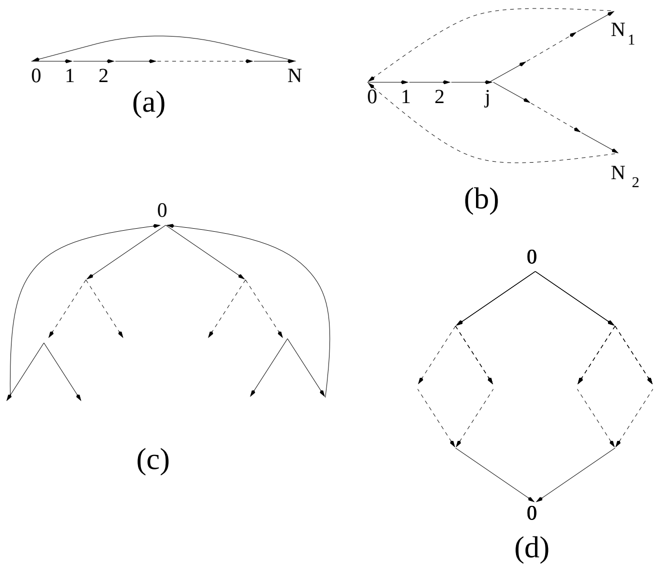

Figure 2: Refining RB for various topologies

Lemma 4.2.1 Program RB' is masking tolerant to detectable faults and stabilizing tolerant to undetectable faults. Moreover, in the presence of undetectable faults, the number of phases executed unsuccessfully by program RB' is kept to a minimum. □

By repetitively using Lemma 4.2.1, program CB can be refined for the topologies shown in Figure 2(c) and 2(d): in topology 2(c), the processes are organized in a tree and all the leaves are connected to the root, and topology 2(d), the processes are arranged in two trees such that every leaf is connected to at least one leaf in another tree. Note that in both these topologies, a process may occur more than once: for example, in 2(d), process 0 is the root of both trees. Thus, we have:

Proposition 4.2.2 If processes are organized in a tree as shown in Figure 2(c) (2(d) respectively), program CB can be refined while preserving its tolerance properties.

Observe that using the topology shown in Figure 2(c) (2(d) respectively), the time required to detect that all processes have executed their phase successfully and to inform all processors to begin executing the next phase is $O(h)$, where h is the height of the tree. It follows that if binary trees are used, the required time is $O(log\ N)$.

Finally, we note that the topology in Figure 2(d) can be embedded in any connected graph: embed a tree in that graph and use the same tree twice, once as the top tree and once as the bottom tree. Hence, CB can be refined for an arbitrary topology while preserving its tolerance properties.

5 Refinement 2 : Message Passing

The actions of program RB allow j to instantaneously access the state of one of its neighbors, $j-1$ or $j+1$, as well as update its own state. In this section, we refine the actions further so that they instantaneously either access the state of one of its neighbors or update its own state, but not both. As shown below, this program can be implemented on message passing systems.

To this end, the message passing program MB is obtained from RB as follows:

- The domain of all sn variables in augmented from $\{0..K-1\}$ where $K>N$ to $\{0..L-1\}$ where $L>2N+1$.
- Process j additionally maintains a local copy of each of the variables $sn.(j-1), cp.(j-1)$, $ph.(j-1)$ and $sn.(j+1)$.
- Actions $T1$, $T2$, and $T4$ are replaced with actions that instead of accessing the variables of the neighbors access the corresponding local copies.
- The local copy of $sn.(j-1)$ in j is updated only if $sn.(j-1)$ is different from $\perp$ and $\top$. Whenever this copy is updated, the local copies of $cp.(j-1)$ and $ph.(j-1)$ in j are also updated (from $cp.(j-1)$ and $ph.(j-1)$) by executing the same statement with which a non-0 process updates its phase and control position. Thus, the resulting local copy update action is identical to the superposed action $T2$ at a non-0 process.
- The local copy of $sn.(j+1)$ in j is updated only if $sn.(j+1)$ is $\top$.

Moreover, the fault actions are modified as follows:

- A detectable fault at j additionally sets the local copies of $sn.(j-1)$ and $sn.(j+1)$ to $\perp$, the local copy of $cp.(j-1)$ to $error$, and sets the local copy of $ph.(j-1)$ to an arbitrary value.
- An undetectable fault at j sets all the variables at j (including the local copies) to arbitrary values from the respective domains.

Observe that in the resulting program no process other than j accesses the local copy variables in j. Also, j accesses only its local state when it updates a variable that is accessed by the other processes. It follows that the actions of j that update its local copy variables from the variables of a neighboring process need not be executed instantaneously and, hence, all program actions can be implemented using messages. It also follows that the concurrent execution of actions of neighboring processes can be equivalently represented by an interleaving of the action executions. Hence, for the purposes of the proof of correctness, we can "pretend" that the actions that update the local copies execute instantaneously.

Proof of correctness. The proof consists of three parts: (1) eventually actions $T3, T4$ and $T5$ are disabled, (2) thereafter, the computations of MB are equivalent to that of RB where the ring consists of $2(N+1)$ processes, and (3) in the absence of undetectable faults process 0 never executes action $T5$. The detailed proof is relegated to the appendix.

6 Performance Analysis

In this section, we evaluate our barrier synchronization program on a tree of height h (as shown in Figure 2(c)). Specifically, we evaluate: (i) the effect of communication latency, (ii) the effect of frequency of faults, and (iii) the overhead of fault-tolerance. In subsections 6.1 and 6.2, we present the analytical results and the simulation results respectively: in each subsection, detectable faults are considered first followed by undetectable faults.

In both analytical and simulation results, we model true concurrency among processes via the maximum parallel semantics, i.e., time is computed in terms of steps, where in each step every process executes one of its enabled actions unless all its actions are disabled.

Let the time required to execute a phase be the unit time. With respect to this unit time, let the communication latency be c, and the fault frequency be f. Thus, if the time required to execute one phase is $1ms$ and the time required to communicate a message from a source to a destination is $10\mu s$, c will be 0.01.

6.1 Analytical Results

We begin by considering detectable faults. As shown in Figure 1, to execute a phase successfully in the absence of faults, a process changes its control position thrice. During each such change of control position, a process receives a message from its predecessor. Thus, the communication time for barrier synchronization is $3hc$. Hence,

the maximum time required for executing a phase successfully in the absence of faults	$=$	$1+3hc$
the probability that a fault does not occur during a phase	$=$	$(1-f)^{(1+3hc)}$
the probability that a fault occurs during a phase	$=$	$1-(1-f)^{(1+3hc)}$

To execute a phase successfully in the presence of detectable faults, exactly k instances of that phase are executed if detectable faults occur in each the first $k-1$ instances and no fault occurs in the k^{th} instance. Therefore, the probability that a phase is executed exactly k times is $(f_{freq})^{k-1}(1-f_{freq})$, where $f_{freq}=1-(1-f)^{(1+3hc)}$.

Hence, the number of instances executed to execute a phase successfully in the presence of detectable faults is

$$
\begin{aligned}
& \textstyle\sum_{k=1}^{\infty} \quad k(f_{freq})^{k-1}(1-f_{freq}) \\
= \;& (1-f_{freq})\textstyle\sum_{k=1}^{\infty} \quad k(f_{freq})^{k-1} \\
= \;& (1-f_{freq})\frac{1}{(1-f_{freq})^2} && \text{(by differentiating } \textstyle\sum_{k=1}^{\infty}(f_{freq})^k = \frac{f_{freq}}{1-f_{freq}} \text{ wrt } k\text{)} \\
= \;& \frac{1}{(1-f_{freq})} \\
= \;& \frac{1}{(1-f)^{(1+3hc)}}
\end{aligned}
$$

It follows that the time required to execute a phase successfully in the presence of detectable faults is $\frac{1+3hc}{(1-f)^{(1+3hc)}}$.

Effect of frequency of faults. In Figure 3, we compare the effect of the fault-frequency while keeping the number of processors fixed at 32 (so h is 5). We vary the frequency of faults from $f=0$ to $f=0.1$. As expected, the number of executions of a phase increases as the frequency of the faults increases.

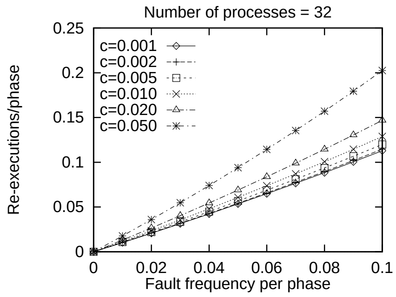

Figure 3: Effect of frequency of faults

To give a more concrete interpretation to this evaluation, we will consider the following scenario: let the phase execution time be $1ms$. Then, $f=0.001$ implies that a fault occurs once every second, and $c=0.01$ implies that the communication latency is $10\mu s$. We conclude that when the frequency of faults is small ($f \leq 0.01$), the percentage of phases executed incorrectly is lower than 1.6%.

Based on the claim that the phase execution time dominates the synchronization time, we focus our attention on programs where the barrier synchronization time is at most one-half of the time required to execute a phase. Since in the absence of faults, barrier synchronization can be achieved in time $1 + 2hc$ —one communication over the tree suffices to detect that all processes have completed execution of their phase and another to inform them to start the next phase— we assume is that $2hc \leq 0.5$. (In this scenario, since h is 5, we will therefore let the values of c range from 0 to 0.05.) And, for subsequent results, we will assume that the phase execution time is $1ms$.

Effect of communication latency. Figure 3 also illustrates the effect of communication latency on the number of instances executed in a phase in the presence of detectable faults. For a given fault-frequency, as the communication latency increases, there is a corresponding increase in the number of instances executed for a successful execution of phase. Observe that, even at high communication latency, $c=0.05$, when the frequency of faults is low, say $f=0.01$ (i.e., 10 faults per second), the probability that a phase is re-executed is as low as 1.7%. If the frequency of a fault is even less, the number of re-executions is reduced further.

Overhead of fault-tolerance. As mentioned above, if fault-tolerance is not an issue, barrier synchronization can be achieved in time $(1 + 2hc)$, whereas our program takes $\frac{(1+3hc)}{(1-f)^{(}1+3hc)}$ time. Figure 4 illustrates the overhead required for fault-tolerance. Again, we consider 32 processes. If no fault occurs, our program incurs 4.5% overhead compared to the fault-intolerant program. If $f = 0.01$ (10 faults in a second), the overhead is 5.7%, while if $f = 0.05$ (50 faults in a second), the overhead is bounded by 10.8%. We conclude that the overhead for fault-tolerance is reasonably low. Moreover, the cost of fault-tolerance is directly proportional to the frequency of faults.

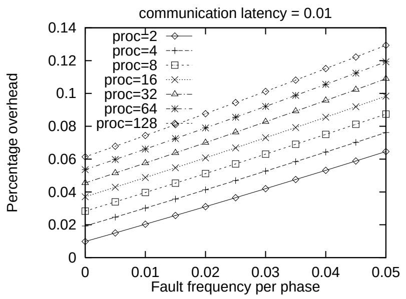

Figure 4: Overhead of fault-tolerance

Recovery from undetectable faults. We conclude our analysis by considering undetectable faults. The key issue in the case of undetectable faults is the time it takes for the program to recover to a state from where further computation satisfies the specification of barrier synchronization.

The program recovers from an arbitrary state in two stages: (i) the *sn* values of all processes are corrected, and (ii) subsequently the *ph* and *cp* values of all processes are corrected. Under the maximum parallel semantics, it is easy to show that the *sn* values are corrected in at most h communications steps. Thus, the first stage takes at most hc time.

After the *sn* values are corrected, 0 will get the token within hc time. Now, consider two cases: (a) 0 is in control position *ready*, and (b) 0 is not in control position *ready*. In case (a), as shown in (the proof of) Lemma 4.1.3, it takes at most $2hc$ time for the program to reach a state from where further computation satisfies the specification of barrier synchronization. In case (b), as shown in Lemma 4.1.3, it takes at most $2hc$ time for the program to reach a state where the control position of 0 is *ready*. Moreover, when 0 changes its control position to *ready*, no process is in control position *execute*. (This follows from the fact that if 0 is in control position *success* when it sends the token to 1, each process will change its control position to either *success* or *repeat* in the circulation of that token.) Therefore, as shown in Lemma 4.1.3, it takes at most hc time for the program to reach a state from where further computation satisfies the specification of barrier synchronization. Thus, the maximum time required in cases (a) and (b) is at most $3hc$ and, hence, the second stage takes at most $4hc$ time. It follows that within $5hc$ time the program recovers to a state from where further computation satisfies the specification of barrier synchronization. Moreover, under our assumption that $2hc$ is less than 0.5, the program recovers in at most 1.25 time.

6.2 Simulation Results

We obtained simulation results using SIEFAST, a *Simulation and Implementation Environment for FAult-tolerant, Secure, and real-Time protocols* which we have developed at Ohio State (cf. http://www.cis.ohio-state.edu/~anish). SIEFAST allows the modeling of a program and its environments. The program is modeled in the guarded command notation discussed in Section 2. A real-time value is associated with each action to model the time required to execute that action. The environments of the program consist of faults, security intrusions, input traffics, etc. One advantage of using SIEFAST is that it uses the exact program discussed in this paper, and requires no further translation into another language such as C or C++.

We used program RB as the input to SIEFAST and selected the option that simulated the program under maximal parallelism semantics. For the purposes of the barrier synchronization simulation, we only modeled the fault environment. This fault environment specified the frequency of fault occurrence and the actions that capture how faults perturb the program. Our simulation results are presented below, beginning with the case for detectable faults.

Effect of frequency of faults. Figure 5 summarizes our simulation results for the effect of the frequency of faults. We performed these simulations for various values of fault-frequencies in the range $f = 0$ to $f = 0.1$. As predicted by the analytical results, the number of instances executed for a successful completion of a phase increases as the fault-frequency increases. Moreover, the number of re-executions is the same as those predicted analytically (cf. Figures 3 and 5).

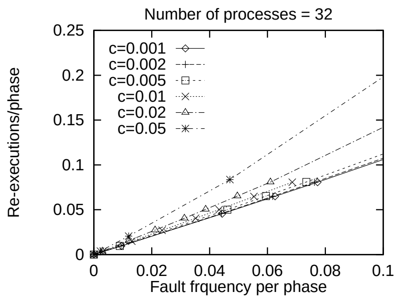

Figure 5: Effect of frequency of faults

Effect of communication latency. Figure 5 also summarizes our simulation results for the effect of communication latency on the number of instances executed for a successful completion of a phase. Again, comparing Figures 3 and 5, we observe that the number of re-executions in the simulated program is as predicated by analytical results.

Overhead of fault-tolerance. Figure 6 shows the overhead of fault-tolerance. Comparing Figures 4 and 6, we observe that the overhead in the simulated program is less than that predicated by analytical results. This is due to worst-case assumptions made in the analytical model. In particular, in the analytical model, we assume that if a detectable fault occurs during an instance of a phase, the time required for execution of that instance is still $1 + 3hc$.

This is not the case if the fault occurs early on in the phase, in which case processes may complete an unsuccessful instance of the phase quickly.

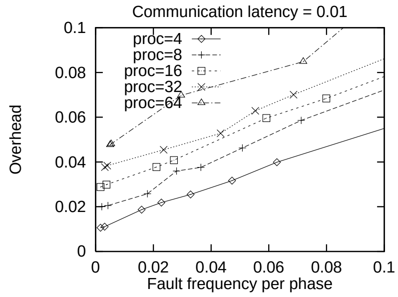

Figure 6: Overhead of fault-tolerance

Recovery from undetectable faults. We conclude this section by presenting the simulation results for the time the program takes in order to recover from an arbitrary state (see Figure 7).

We performed these simulations for various values of communication latency, $c = 0$ to $c = 0.05$, and various numbers of processes, $h=1$ to $h=7$. As shown in Figure 7, the recovery time increases with the communication latency and the number of processes. However, the time required for recovery is small: for example, if the number of processors is 32 and c is 0.01 (10 μs), the recovery time is only 0.56 time units (560 ms). Also, if the c is 0.05 and the number of processes is 128, the recovery time is less than one time unit. Note that due to the worst case assumptions made in the analytical results, the recovery time in the simulations is lower than that predicated by the analytical results.

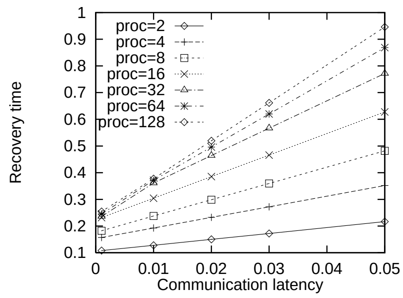

Figure 7: Recovery from undetectable faults

7 Discussion

Extensions to tolerate other fault-classes. We have thus far assumed that faults are eventually correctable. While this assumption is true of many types of faults, it is clearly not true of all types. Table 1 gives a more general classification of faults and sets the stage for a discussion of how to extend our program to provide appropriate tolerances for faults that are not eventually correctable.

	Detectable	Undetectable
Immediately correctable	Trivially masking	
Eventually correctable	Masking[3]	Stabilizing
Uncorrectable	Fail-safe	Intolerant

Table 1 : Classification of faults and appropriate tolerances to them in barrier synchronization

If a fault is immediately correctable[3] i.e., it is possible to pretend that an occurrence of that fault can be recovered from instantaneously, we may model the correction simultaneously with fault occurrence and, hence, pretend that the fault does not exist. As an example, consider message corruptions in a setting where messages contain enough redundancy bits not only to detect but also to correct the corrupted message. Tolerance to such faults can be added to our program in a straightforward manner.

If a fault is uncorrectable, it may be impossible to guarantee that Progress is satisfied. Still, if the fault is at least immediately detectable, it is possible to ensure that Safety is always satisfied. Thus, an appropriate tolerance to faults in this class is fail-safe tolerance [10], whereby the program guarantees that it never reports a completion of a barrier incorrectly. But the program may not always report (a successful) completion in the presence of faults. Note that if an uncorrectable fault is also undetectable, it may not be possible to guarantee Safety (and thus any tolerance whatsoever).

Uncorrectable faults raise an issue that we did not explicitly address thus far in this paper. A fault such as a permanent crash of a processor or a fault that causes a process to become Byzantine seems to corrupt actions —as opposed to variables— in the program. It is, however, possible to represent the corruption of actions by faults that corrupt variables, by introducing so-called "auxiliary" variables. For example, the crash of a process can be captured by introducing an auxiliary variable *up* for that process, as follows. Each action of that process is to be executed only if *up* is true. The crash itself is modeled as the occurrence of a fault that corrupts *up*, by setting it to false. Similarly, the Byzantine behavior of a process can be captured by introducing an auxiliary variable *good*, as follows: If the variable *good* is true, then the process executes its normal actions. When a fault action corrupts *good* to false, the process executes actions whose behavior is nondeterministic.

[3] A further distinction can be made between faults that are detectable "immediately" versus those that are detectable "eventually" but not "immediately". Masking fault-tolerance is appropriate for the former but not the latter, assuming eventual correctability. Likewise, fail-safe tolerance is appropriate for the former but not the latter, assuming uncorrectability.

The faults that we considered earlier in this paper belong to the middle row of Table 1. As mentioned above, the faults in the top row can be trivially handled by our program. Our program can also be extended to provide tolerance to faults in the bottom row: whenever a process detects a fault in the bottom row, it reports a fatal error to the application and stops further execution. As such, our program can tolerate all the above fault-classes in a manner that is appropriate with respect to each fault-class.

Instantiations to solve other problems. Our barrier synchronization program can be instantiated to obtain fault-tolerant programs for other problems such as atomic commitment [13, 14], clock unison [15, 16] and phase synchronization [17].

Atomic commitment. In atomic commitment, a program executes one or more transactions such that each transaction completes successfully only if all of its subtransactions complete successfully and, ideally, transaction $j+1$ is executed only after transaction j completes successfully.

Traditionally, the faults considered in the atomic commit problem make processes crash, restart, or exhibit Byzantine behavior. As discussed above, these faults can be expressed by introducing auxiliary variables.

To obtain an atomic commitment program, we allow each subtransaction to change its control position from *execute* to *success* if that subtransaction has completed successfully. Otherwise, it changes its control position to *error*.

Clock Unison. In the clock unison problem, every process maintains a bounded-value counter (clock) such that, at all times, the counter at two processes differs by at most one and that, infinitely often, the counter is incremented.

Traditionally, the faults considered in the clock unison problem corrupt the counters of processes undetectably. The desired tolerance to these faults is that upon starting from a state where the counters have arbitrary values, the program should eventually reach a state where the specification of clock unison is satisfied.

The barrier synchronization problem generalizes the clock unison problem, in the sense that phase i of the computation may be mapped onto the i-th value of the counter (chosen from its bounded domain). Note that in the absence of undetectable faults, the phases of all processes in the barrier synchronization differ from each other by at most one. Since our solution is stabilizing tolerant in the presence of undetectable faults, it meets the tolerance requirements of clock unison.

Phase Synchronization. In the phase synchronization problem, each process executes a (potentially infinite) sequence of phases. A process executes a phase only when all processes have completed the previous phase.

Traditionally, the faults considered in the phase synchronization problem corrupt the phase of processes initially in (and not during) the computation. The desired tolerance to these faults is that each phase be executed correctly, without making any assumptions about the speed of other processes. This implies that no process should read any shared variable (such as the phase variable) unless that variable has been previously initialized.

The barrier synchronization problem generalizes the phase synchronization problem, in the sense that each phase in the latter can be uniquely mapped onto an instance of a phase in the former. Our solution tolerates the detectable corruption of variables without executing any phase incorrectly, but it assumes that each process corrects its detectably corrupted shared variables before any process action accesses these variables. Our solution can, however, be extended so that each process corrects the shared variables of all process, thereby meeting the tolerance requirements of phase synchronization.

8 Concluding Remarks

In this paper, we presented barrier synchronization programs that tolerated two classes of faults, detectable and undetectable. We ensured that in the presence of detectable faults alone, the specification of barrier synchronization was always satisfied, and in the presence of undetectable faults, the program eventually reached a state from where the specification of barrier synchronization was (re)satisfied. Thus, our program was multitolerant [10] in that it provided appropriate tolerances with respect to multiple classes of faults. (The interested reader is referred to [10] for a formal method for the design of multitolerant programs.) We are not aware of work by others that has addressed multitolerance of barrier synchronization.

We also presented analytical and simulation results for the overhead incurred by our program in tolerating faults. As shown in Section 6, this overhead was low. In particular, the overhead was merely 3 to 4 percent when the frequency of faults was low (i.e., about 1 fault per second). Since faults occur relatively infrequently in modern day parallel systems, the program was optimized for the common case. Even if the frequency of faults were high (say, about 50 faults per second), the overhead was approximately 10%.

As mentioned in the introduction, one of our goals was to be able to implement this program in hardware. Since each process in RB maintains a sequence number, a phase, and a control position, the state maintained at each process is at most $O(logN)$ where N is the number of processes. Moreover, our program is concise and can be implemented as a simple table lookup. Therefore, it can be implemented in the hardware.

Our program, if embedded in the barrier synchronization primitive provided in MPI, provides MPI users with the option of tolerating various types of faults in a way that is suitable to each type of fault. Based on the discussion in Section 7, the user could specify that the program should recover from correctable faults in an appropriate manner and abort in case of uncorrectable faults.

Finally, we note that our program can be systematically extended to deal with fuzzy barriers [2]. In particular, the transition from $execute$ to $success$ is the same as entering the barrier, and the transition from $ready$ to $execute$ is the same as leaving the barrier. It is therefore possible to allow a process can perform some useful work between these two state transitions, which captures the requirement of fuzzy barriers.

References

[1] M. Scott and J. Mellor-Crummey. Fast, contention-free combining barriers. *International Journal of Parallel Programming*, August 1994.

[2] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3), 1989.

[3] D. N. Jayasimha. Distributed synchronizers. *International Conference on Parallel Processing*, pages I 23–27, 1988.

[4] S. K. S. Gupta and D. N. Jayasimha. A generalized dynamic barrier synchronization scheme. *International Conference on Parallel Processing*, August 1992.

[5] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[6] R. Sivaram, C. B. Stunkel, and D. K. Panda. A reliable hardware barrier synchronization scheme. *International Parallel Processing Symposium*, pages 274–280, 1997.

[7] H. Xu, P.K. McKinley, and L. Ni. Efficient implementation of barrier synchronization in wormhole route hypercube multicomputers. *Journal of Parallel and Distributed Computing*, pages 172–184, 1992.

[8] D. Johnson, D. Lilja, J. Riedl, and J. Anderson. Low-cost, high-performance barrier synchronization on networks of workstations. *Journal of Parallel and Distributed Computing*, 40(1):131–137, January 1997.

[9] S. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, October 1997.

[10] A. Arora and S. S. Kulkarni. Component based design of multitolerance. *IEEE Transactions on Software Engineering*, January 1998.

[11] M. G. Gouda. The triumph and tribulation of system stabilization. In Hélary and Raynal [18], pages 1–18.

[12] MPI: A message-passing interface standard, June 1995. Version 1.1.

[13] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[14] Y. Zhao and F. B. Bastani. A self-adjusting algorithm for Byzantine agreement. *Distributed Computing*, 5:219–226, 1992.

[15] T. Herman and S. Ghosh. Stabilizing phase-clocks. *Information Processing Letters*, 54:259–265, 1995.

[16] J. Couvreur, N. Francez, and M. Gouda. Asynchronous unison. *Proceedings of the Twelveth International Conference on Distributed Computing Systems, Tokyo*, 1992.

[17] J. Misra. Phase synchronization. *Information Processing Letters*, 38:101–105, April 1991.

[18] Jean-Michel Hélary and Michel Raynal, editors. *Distributed Algorithms, 9th International Workshop, WDAG '95*, volume 972 of *Lecture Notes in Computer Science*, Le Mont-Saint-Michel, France,, 13–15 September 1995. Springer.

Appendix

Proof of correctness of program MB. We show that the MB program satisfies the property ($\star$): Upon starting from an arbitrary state, eventually actions $T3, T4$, and $T5$, and the action by which j updates the copy of $sn.(j+1)$ at j are disabled, and, hence, eventually the computations of MB are equivalent to the computations of MB where the ring consists of $2(N+1)$ processes.

To prove property ($\star$), we first show that the program eventually reaches a state where the sequence numbers of all processes are different from $\perp$ and $\top$. Let us consider two cases: (a) there exists a process j such that the $sn.j$ or the local copy of $sn.(j-1)$ at j is different from both $\perp$ and $\top$; and (b) no such process exists. In case (a), the program eventually reaches a state where $sn.j$ is different from both $\perp$ and $\top$. Then, the local copy of $sn.j$ at $j+1$ is set to a value that is different from $\perp$ and $\top$. Subsequently, $sn.(j+1)$ is set to a value that is different from both $\perp$ and $\top$, and so on. Thus, the program reaches a state where, for any process j, $sn.j$ and the local copy of $sn.(j-1)$ at j are different from $\perp$ and $\top$. In case (b), eventually N sets $sn.N$ to $\top$. Then, $N-1$ sets the copy of $sn.N$ at $N-1$ to $\top$, and subsequently sets $sn.(N-1)$ to $\top$, and so on. Thus, eventually 0 sets $sn.0$ to $\top$, executes action $T5$, and thereby satisfies case (a).

It now suffices for us to observe that in a state where the sequence numbers of all processes are different from $\perp$ and $\top$, actions $T3$, $T4$, and $T5$, and the action by which j updates the local copy of $sn.(j+1)$ at j are disabled. Recalling that the action that updates the local copy of $sn.(j-1)$ at j is identical to the superposed $T2$ action, it follows that subsequent computations of MB are equivalent to the computations of the distributed token ring program where the ring consists of $2(N+1)$ processes.

In the absence of faults, we observe that actions $T3, T4$, and $T5$ and the action by which j updates the local copy of $sn.(j+1)$ are always disabled, hence, from property ($\star$), *Safety* and *Progress* are satisfied.

In the presence of detectable faults (by the fault model of Section 2), there exists a process j whose state is not corrupted. In this case, the sequence numbers of all processes in the set $\{0..j\}$ are different from $\top$. It follows that action $T5$ is never executed. When actions $T3$ or $T4$ execute, no process receives a token and the control position of that process remains in *error*. It follows that there is at most one token in the ring and the control position of a process is *error* iff its sequence number is either $\perp$ or $\top$. From this, just as we showed in the previous section, we can show that when 0 changes its control position from *ready* to *execute*, to re-execute a phase, all non-corrupted processes are in the same phase and no non-0 process is control position *execute*. Thus, in the presence of detectable faults, the re-execution of a phase does not interfere with its current execution, and hence, *Safety* is satisfied. Also, from property ($\star$), eventually the computation of MB satisfies *Progress*.

In the presence of undetectable faults, from property ($\star$), the tolerance to undetectable faults is preserved (recall $L > 2N+1$). Moreover, as in the case of the distributed program, if the network of processes is perturbed to a state where there are in m distinct phases in the phase variables and their local copies, each of these m phases may execute incorrectly with respect to the barrier synchronization specification. However, if process 0 enters into a phase other than these m by incrementing $ph.0$, that phase is executed correctly. Thus, at most m phases are executed incorrectly. □