Fine-Grain Multitolerant Barrier Synchronization

Sandeep S. Kulkarni *Anish Arora*

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

Abstract

We design a multitolerant program for synchronizing the phases of concurrent processes. The tolerances of the program enable processes to (i) compute all phases correctly in the presence of faults that corrupt process state in a detectable manner, and (ii) compute only a minimum possible number of phases incorrectly before resuming correct computation in the presence of faults that corrupt process state in an undetectable manner. The program is fine-grain in the sense that each process action either updates the state of that process or involves communication with one of two neighboring processes.

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

1 Motivation

Barrier synchronization in its general form requires that a set of processes execute a cyclic sequence of phases so that a phase is executed by each process only after all processes have completed the previous phase. This form of synchronization generalizes a variety of others, such as clock unison, phase synchronization and atomic commitment, and appears frequently in parallel, distributed, and scientific computation applications.

Often the design of barrier synchronization has to accommodate the occurrence of faults. Commonly considered examples of faults include incorrect initializations; corruption, loss, reordering, and duplication of messages; processor restarts; and performance and timing violations. While some of these fault-classes can be masked (i.e., even in their presence each phase is executed correctly), others cannot.

To accommodate multiple fault-classes, not all of which can be masked, it is convenient to view the effect of faults in each fault-class as a "corruption" of the state of some process, i.e., the state of that process prior to the fault is lost and is replaced with some other value. The state corruption view suggests that one way to accommodate all of the fault-classes is to design the set of processes to be stabilizing [2], i.e., to recover from an arbitrarily corrupted state to one from where the specification of barrier synchronization is (re)satisfied. Unfortunately, a stabilizing design allows incorrect execution of (a finite number of) phases in the presence of each fault-class before the recovery is complete, and it is thus not ideal for the fault-classes that can be masked.

We therefore present in this paper barrier synchronization designs that offer multiple levels of tolerance corresponding to multiple fault-classes, a notion which we refer to as multitolerance [3].

[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. This paper is a substantially revised version of [1].

Specifically, our designs are able to tolerate all state corruptions, while keeping for each fault-class the number of phases executed incorrectly to be the minimum that is possible for that fault-class.

The rest of this paper is organized as follows. In Section 2, we state the problem formally. In Section 3, we present a solution for the problem which is coarse-grain in the sense that each process action can instantaeously communicate with all other processes and update its own state. In Section 4, we distribute this solution so that so that each process action can instantaneously communicate with only one of two 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 only of two neighboring processes or update its own state, but not both. Finally, we present concluding remarks in Section 6.

2 Problem Statement

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

- ***(Safety)*** No process executes $phase.(i+1)$ until all processes have successfully executed $phase.i$
- ***(Progress)*** After all processes have successfully executed $phase.i$, they each eventually execute $phase.(i+1)$,

where $phase.n = phase.0$. Initially, each process has successfully executed $phase.(n-1)$ and is thus ready to execute $phase.0$.

Fault-classes. We classify the faults that the barrier synchronization is subject to into two fault-classes: "detectable" and "undetectable". Detectable faults are those that corrupt the state of a process in such a way that the corrupted state can be "reset" before it is accessed by any process, whereas undetectable faults are those that corrupt the state of a process in such a way that the corrupted state need not be reset before it is accessed by any process. We assume that the state of a process before its corruption by a fault is lost; hence, a reset after a detectable fault may yield a state that differs from that before the fault, and a fault may cause the processes to execute the current phase unsuccessfully.

We assume that the execution of each $phase.i$ is idempotent in the sense that any number of (possibly unsuccessful) executions of $phase.i$ followed by the successful execution of $phase.i$ is equivalent to the successful execution of $phase.i$. Hence, detectable faults can be masked by requiring all processes to re-execute the current phase when a detectable fault occurs. Since the underlying computation of the processes in any phase may depend on each other, for the re-execution of the current phase to be correct, the failed execution and the re-execution of the current phase should not overlap. Since each phase execution of a process is terminating, this overlap can be avoided this by starting the re-execution of the current phase only after no process is executing in that phase. As an example, consider a network of two processes, j and k, both in $phase.0$. If j is subject to a detectable fault, then k is unaware whether j has executed $phase.0$. Hence, $phase.0$ has to be re-executed. But if k is currently executing $phase.0$ then the re-execution can be started after k finishes executing $phase.0$.

Undetectable faults cannot be masked since they allow processes to access corrupted state. We may however ensure that after undetectable faults occur, the system eventually reaches a state from where the specification of barrier synchronization is satisfied. Until such a state is reached, execution of some phases may violate Safety or Progress of barrier synchronization. (Note that if all processes are corrupted detectably, the resulting state cannot be reset without violating Safety and Progress, and hence, this fault is considered to be an undetectable fault.)

The problem. Our goal is to design a multitolerant barrier synchronization that tolerates both detectable and undetectable faults in a manner best suited to each fault-class. Thus, the problem is:

> *Design for the process network a barrier synchronization program that, when subject to a finite number of detectable or undetectable faults, (re)satisfies the barrier synchronization specification with minimum number of phases executed incorrectly.*

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 the 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 the current phase is re-executed. In the presence of undetectable faults, the program recovers to some start state, from where subsequent execution of each phase 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.

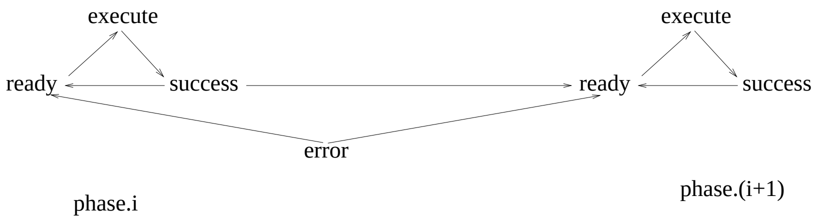

Figure 1: State transitions for program CB

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 network 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 re-execute 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 re-execute 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 re-execute 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 network 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 network 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., some processes have not completed the current phase successfully), the phase is unchanged, thereby leading to a re-execution 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, re-execution of that phase does not begin as long as some process is executing in that phase. 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 multiphase barrier synchronization program, CB, consists of four actions, one for each transition discussed above:

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

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

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

$$
\begin{array}{llll}
true & \longrightarrow & ph.j, cp.j := ?, error & \\
true & \longrightarrow & ph.j, cp.j := ?, ?, & \text{where ? is any value from the domain (of } cp.j \text{ or } ph.j)
\end{array}
$$

Proof of correctness: *Safety.* In the absence of faults, when a process increments its phase from i to $i+1$, all processes have successfully completed execution of *phase.i*, i.e., a process executes *phase*.$(i+1)$ only after all processes have successfully executed *phase.i*.

In the presence of detectable faults, a process that is subject to state corruption changes its control position to *error*. Since the first process to re-execute *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 the re-execution of *phase.i* does not interfere with its current execution.

Progress. In the absence of faults, once all processes have successfully completed *phase.i*, i.e., once the phase of all processes is i and their control position is *success*, action $CB3$ is enabled at all processes. Each process executes $CB3$ and changes its phase to $i+1$. Thus, the program reaches a state where all processes are in *phase.*$(i+1)$ and in control position *ready*. Starting from this state, each process executes *phase.*$(i+1)$.

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, each process re-executes *phase.i*.

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.

If the network of processes 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. $\square$

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 the current phase is to be re-executed or the next instance of the 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 Medium-Grain Solution

We now refine program CB so that instead of communicating with all processes, each process communicates with only one of two 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.

To this end, 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 [3].

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 $\perp$ and $\top$ are added to the domain of the sequence number: when the sequence number of a process is corrupted, it is set to $\perp$, and the sequence number $\top$ is used to detect whether the state of all processes is detectably corrupted. 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 $\perp$ and $\top$. Process N has the token iff $sn.N = sn.0$ and $sn.N$ and $sn.0$ are both different from $\perp$ 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:

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

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 $\perp$ 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 was detectably corrupted, by using a new control position, say *repeat*, as follows: If in this circulation the token reaches a detectably corrupted process, 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 re-execute 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 $T1$:

if	$cp.0 = ready \wedge cp.0 = cp.N \wedge ph.0 = ph.N$	then	$cp.0 := execute$
elseif	$cp.0 = execute$	then	$cp.0 := success$
elseif	$cp.0 = success$	then	if $cp.0 = cp.N \wedge ph.0 = ph.N$ then $ph.0 := ph.0+1$; $cp.0 := ready$
elseif	$cp.0 = error$	then	$ph.0 := ph.N$; $cp.0 := ready$

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 $T2$:

$ph.j := ph.(j-1)$;

if	$cp.j = ready$	$\wedge$	$cp.(j-1) = execute$	then	$cp.j := execute$
elseif	$cp.j = execute$	$\wedge$	$cp.(j-1) = success$	then	$cp.j := success$
elseif	$cp.j \neq execute$	$\wedge$	$cp.(j-1) = ready$	then	$cp.j := ready$
elseif	$cp.j = error$	$\vee$	$cp.(j-1) \neq cp.j$	then	$cp.j := repeat$

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. *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 $i+1$, all processes have successfully executed *phase.i*. Hence, a process executes *phase*.$(i+1)$ only after all processes have successfully executed *phase.i*.

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 $\bot$ 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, in which case 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*. Consequently, 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 re-executing the phase, no process is in control position *execute*, i.e., the re-execution of *phase.i* does not interfere with its current execution.

Progress. In the absence of faults, the proof of progress is identical to that of program CB: after all processes execute *phase.i* successfully, they change their control position to *success*, then change their control position to *ready* and increment their phase. From the latter state, each process executes *phase*.$(i+1)$.

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 re-execute *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, each process re-executes *phase.i*.

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*.

Just as in program CB, if the network of processes 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. $\square$

5 Fine-Grain Solution

The actions of the medium-grain solution 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.

To this end, the medium-grain solution is modified 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 are fine-grain. 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. We show that the the fine-grain 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 the fine-grain program are equivalent to the computations of the distributed token ring program 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 the fine-grain program 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 the fine-grain program 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. $\square$

6 Concluding Remarks

We presented a fine-grain barrier synchronization program that is multitolerant to state corruptions. Our presentation assumed that a ring could be embedded without edge dilation on an arbitrary process network. This assumption is justifiable for our program since a spanning tree can always be embedded without edge dilation on an arbitrary network and the ring can then be embedded via a tree traversal. (Note that repetitions of processes in the ring induced by the tree traversal are acceptable since our program allows a process to execute only once per token circulation, even if the token revisits it in that circulation.)

To the best of our knowledge, we are not aware of work by others that has addressed the multitolerance of barrier synchronization. Our work has focused on two tolerances: masking tolerance to detectable faults, where the program continues to satisfy its specification when detectable state corruptions occur, and nonmasking tolerant to undetectable state corruptions, where the program eventually resatisfies the specification when undetectable faults stop. This multitolerance is useful since instantiations of our multitolerant solution satisfy the specification and typical tolerance requirements of a variety of synchronization problems, such as clock unison [4], phase synchronization [5], and atomic commitment [6].

The last of these problems, atomic commitment, raises the issue that we did not explicitly consider faults that make processes crash, restart, or exhibit Byzantine behavior. Although these faults seem to corrupt actions —as opposed to variables— in processes, it is possible to represent the corruption of actions by faults that corrupt the auxiliary variables by introducing 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.

Our multitolerant solution can be readily extended to handle faults that effectively corrupt the actions of the program. Consider, for instance, fail-stop faults. To enable non-faulty processes to continue to satisfy the specification of barrier synchronization in the presence of fail-stop faults, we may ensure that the ring is (re)constructed and that a process 0 starts execution in a new phase only after it detects that all non-failed processes executed in the current phase.

In conclusion, we note that while the per-process space-complexity of our coarse-grain program is $O(1)$, that of the medium- and fine-grain programs is $O(logN)$, where N is the number of processes in the network. It is an open problem whether there is fine-grain program for this problem that has constant per-process space-complexity.

References

[1] S. S. Kulkarni and A. Arora. Stepwise design of tolerances in barrier computations. Technical Report OSU-CISRC TR-22, Ohio State University, 1997.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[3] A. Arora and S. S. Kulkarni. Component-based design of multitolerance. Technical Report OSU-CISRC TR37, Ohio State University, 1996. Revised for *IEEE Transactions on Software Engineering*.

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

[5] R. M. Lipton and A. Park. The processor identity problem. *Information Processing Letters*, 36(2):91–94, 1990.

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