Compositional Design of Multitolerant Repetitive Byzantine Agreement[1]

Sandeep S. Kulkarni *Anish Arora*

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

Abstract

We illustrate in this paper a compositional and stepwise method for designing programs that offer a potentially unique tolerance to each of their fault-classes. More specifically, our illustration is a design of a repetitive agreement program that offers two tolerances: (a) it masks the effects of Byzantine failures and (b) it is stabilizing in the presence of transient and Byzantine failures.

1 Introduction

Dependable systems are required to acceptably satisfy their specification in the presence of faults, security intrusions, safety hazards, configuration changes, load variations, etc. Designing dependable systems is difficult, essentially because some desired dependability property, say, availability in the presence of faults, may interfere with some other desired dependability property, say, security in presence of intrusions. As an example, in electronic commerce systems, design by replication facilitates availability but complicates security—replicas can be used to deal with faults which lose electronic currency, but they can also make it easier for intruders to double spend the money.

To effectively formulate multiple dependability properties, we have proposed the concept of multitolerance [1]: Each source of undependability is formulated as a "fault-class" and each dependability property in the presence of that fault-class is formulated as a type of "tolerance". Thus, multitolerance refers to the ability of a system to tolerate multiple classes of faults, each in a possibly different way.

To design multitolerant systems, we recommend a component-based method [1]: Our method, explained briefly, starts with an intolerant system and adds a set of components, one for each desired type of tolerance. Thus, the complexity of multitolerant system design is reduced to that of designing the components and of correctly adding them to the intolerant system. And, the complexity of reasoning about the interference between different tolerance properties is often reduced to considering only the relevant components, as opposed to involving the whole system.

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

To further simplify the complexity of adding multiple components to the system, the method observes the principle of stepwise refinement: in the first step an intolerant program is designed; in each successive step, a component is added to the system resulting from the previous step, to offer a desired tolerance to a previously unconsidered fault-class, while preserving the tolerances to the previously considered fault-classes.

The basic idea of transforming an intolerant program into one that possesses the required tolerance properties is indeed well understood and practiced, see for example [2, 3]. In designing multitolerant programs, however, we have to deal with the additional complexity of multiple fault-classes: with respect to each fault-class, each component needs to behave in some desirable manner. To handle this complication in our compositional and stepwise method, while designing and adding a component in each step, we focus attention on the following strategy.

(1) How to ensure that the added component will offer a desired tolerance to the system *in the presence of faults in the fault-class being considered* ?

(2) How to ensure that execution of the component will not interfere with the correct execution of the system *in the absence of faults in all fault-classes being considered* ?

(3) How to ensure that execution of the component will not interfere with the tolerance of the system corresponding to a previously considered fault-class *in the presence of faults in that previously considered fault-class* ?

In this paper, we present a case study illustrating these three issues in the context of a compositional and stepwise design of a repetitive agreement program that offers two tolerances: (a) it masks the effects of Byzantine failures, by which we mean that it continues to satisfy the specification of repetitive agreement in the presence of Byzantine faults, and (b) it is stabilizing in the presence of transient and Byzantine faults, by which we mean that upon starting from an arbitrary state –which may result from the occurrence of these faults– it eventually reaches a state from where it satisfies the specification of repetitive agreement. The resulting multitolerant program is, to the best of our knowledge, the first program for repetitive agreement that is both masking tolerant and stabilizing tolerant. (Previous designs are only masking tolerant, e.g. [4], or only nonmasking tolerant e.g. [5, 6], but none is multitolerant. In fact, we are not aware of any previous design that is stabilizing tolerant.)

The rest of the paper is organized as follows. In Section 2, we specify the problem of repetitive agreement. In Section 3, we identify a simple fault-intolerant program for solving the problem. In Section 4, we add a component to the program for masking the effect of Byzantine failure. In Section 5, we add another component for stabilizing from transient and Byzantine failures, while preserving the masking tolerance to Byzantine failures alone. We present an extension of our program in Section 6 and discuss its multitolerance-preserving refinement in Section 7. We comment on the general aspects of our method in Section 8, and make concluding remarks in Section 9.

2 Problem Statement: Repetitive Agreement

A system consists of a set of processes, including a "general" process, g. Each computation of the system consists of an infinite sequence of rounds; in each round, the general chooses a binary decision value $d.g$ and, depending upon this value, all other processes output a binary decision value of their own.

The system is subject to two fault-classes: The first one permanently and undetectably corrupts some processes to be Byzantine, in the following sense: each Byzantine process follows the program skeleton of its non-Byzantine version, i.e., it sends messages and performs output of the appropriate type whenever required by its non-Byzantine version, but the data sent in the messages and the output may be arbitrary. The second one transiently and undetectably corrupts the state of the processes in an arbitrary manner and possibly also permanently corrupts some processes to be Byzantine.

(Note that, if need be, the model of a Byzantine process can be readily weakened to handle the case when the Byzantine process does not send its messages or perform its output, by detecting their absence and generating arbitrary messages or output in response.)

The problem. In the absence of faults, repetitive agreement requires that each round in the system computation satisfies Validity and Agreement, defined below.

- *Validity:* If g is non-Byzantine, the decision value output by every non-Byzantine process is identical to $d.g$.
- *Agreement:* Even if g is Byzantine, the decision values output by all non-Byzantine processes are identical.

Masking tolerance. In the presence of the faults in the first fault-class, i.e., Byzantine faults, repetitive agreement requires that each round in the system computation satisfies Validity and Agreement.

Stabilizing tolerance. In the presence of the faults in the second fault-class, i.e., transient and Byzantine faults, repetitive agreement requires that eventually each round in the system computation satisfies Validity and Agreement. In other words, upon starting from an arbitrary state (which may be reached if transient and Byzantine failures occur), eventually a state must be reached in the system computation from where every future round satisfies Validity and Agreement. □

Before proceeding to compositionally design a masking as well as stabilizing tolerant repetitive agreement program, let us recall the wellknown fact that for repetitive agreement to be masking tolerant it is both necessary and sufficient for the system to have at least $3t+1$ processes, where t is the total number of Byzantine processes [4]. Therefore, for ease of exposition, we will initially restrict our attention, in Sections 3-5, to the special case where the total number of processes in the system (including g) is 4 and, hence, t is 1. In other words, the Byzantine failure fault-class may corrupt at most one of the four processes. Later, in Section 6, we will extend our multitolerant program for the case where t may exceed 1.

Programming notation. Each system process will be represented by a set of "variables" and a finite set of "actions". Each variable ranges over a predefined nonempty domain. Each action has a unique name and is of the form:

$$\langle\text{name}\rangle :: \langle\text{guard}\rangle \longrightarrow \langle\text{statement}\rangle$$

The guard of each action is a boolean expression over the variables of that process and possibly other processes. The execution of the statement of each action atomically and instantaneously updates the value of zero or more of variables of that process, possibly based on the values of the variables of that and other processes.

For convenience in specifying an action as a restriction of another action, we will use the notation

$$\langle\text{name}'\rangle :: \langle\text{guard}'\rangle \wedge \langle\text{name}\rangle$$

to define an action $\langle\text{name}'\rangle$ whose guard is obtained by restricting the guard of action $\langle\text{name}\rangle$ with $\langle\text{guard}'\rangle$, and whose statement is identical to the statement of action $\langle\text{name}\rangle$. Operationally speaking, $\langle\text{name}'\rangle$ is executed only if the guard of $\langle\text{name}\rangle$ and the guard $\langle\text{guard}'\rangle$ are both true.

Let S be a system. A "state" of S is defined by a value for each variable in the processes of S, chosen from the domain of the variable. A state predicate of S is a boolean expression over the variables in the processes of S. An action of S is "enabled" in a state iff its guard (state predicate) evaluates to true in that state.

Each computation of S is assumed to be a fair sequence of steps: In every step, an action in a process of S that is enabled in the current state is chosen and the statement of the action is executed atomically. Fairness of the sequence means that each action in a process in S that is continuously enabled along the states in the sequence is eventually chosen for execution.

3 Designing an Intolerant Program

The following simple program suffices in the absence of faults: In each round, the general sends its new $d.g$ value to all other processes. When a process receives this $d.g$ value, it outputs that value and sends an acknowledgment to the general. After the general receives acknowledgments from all the other processes, it starts the next round which repeats the same procedure.

We let each process j maintain a variable $d.j$, denoting the decision of j, that is set to $\perp$ when j has not yet copied the decision of the general. Also, we let j maintain a sequence number $sn.j$, $sn.j \in \{0..1\}$, to distinguish between successive rounds. (In Section 7, we consider the case where the sequence numbers are from the set $\{0..K-1\}$ where $K \geq 2$.)

The general process. The general executes only one action, RG1: when the sequence numbers of all processes become identical, the general starts a new round by choosing a new value for $d.g$ and incrementing its sequence number, $sn.g$. Thus, letting $\oplus$ denote addition modulo 2, the action of the general is:

$$RG1 :: \quad (\forall k :: sn.k = sn.g) \quad \longrightarrow \quad d.g, sn.g := new\_decision(), sn.g \oplus 1$$

The non-general processes. Each other process j executes two actions: The first action, $RO1$, is executed after the general has started a new round, in which case j copies the decision of the general. It then executes its second action, $RO2$, which outputs its decision, increments its sequence number to denote that it is ready to participate in the next round, and resets its decision to $\perp$ to denote that it has not yet copied the decision of the general in that round. Thus, the two actions of j are:

$$
\begin{array}{lll}
RO1 :: & d.j = \perp \ \wedge\ (sn.j \oplus 1\ =\ sn.g) & \longrightarrow\ d.j := d.g \\
RO2 :: & d.j \neq \perp & \longrightarrow\ \{\ \text{output } d.j\ \};\ d.j, sn.j := \perp, sn.j \oplus 1
\end{array}
$$

The correctness proof of R is straightforward. (The interested reader will find the proof in [7].)

4 Adding Masking Tolerance to Byzantine Faults

Program R is neither masking tolerant nor stabilizing tolerant to Byzantine failure. In particular, R may violate Agreement if the general becomes Byzantine and sends different values to the other processes. Note, however, that since these values are binary, at least two of them are identical. Therefore, for R to mask the Byzantine failure of any one process, it suffices to add a "masking" component to R that restricts action $RO2$ in such a way that each non-general process only outputs a decision that is the majority of the values received by the non-general processes.

For the masking component to compute the majority, it suffices that the non-general process obtain the values received by other non-general processes. Based on these values, each process can correct its decision value to that of the majority.

We associate with each process j an auxiliary boolean variable $b.j$ that is true iff j is Byzantine. For each process k (including j itself), we let j maintain a local copy of $d.k$ in $D.j.k$. Hence, the decision value of the majority can be computed over the set of $D.j.k$ values for all k. To determine whether a value $D.j.k$ is from the current round or from the previous round, j also maintains a local copy of the sequence number of k in $SN.j.k$, which is updated whenever $D.j.k$ is.

The general process. To capture the effect of Byzantine failure, one action, $MRG2$, is added to the original action $RG1$ (which we rename as $MRG1$): $MRG2$ lets g change its decision value arbitrarily and is executed only if g is Byzantine. Thus, the actions for g are:

$$
\begin{array}{llll}
MRG1 :: & RG1 & & \\
MRG2 :: & b.g & \longrightarrow & d.g := 0|1
\end{array}
$$

The non-general processes. We add the masking component "between" the actions $RO1$ and $RO2$ at j to get the five actions $MRO1-5$: $MRO1$ is identical to $RO1$. $MRO2$ is executed after j receives a decision value from g, to set $D.j.j$ to $d.j$, provided that all other processes had obtained a copy of $D.j.j$ in the previous round. $MRO3$ is executed after another process k has obtained a decision value for the new round, to set $D.j.k$ to $d.k$. $MRO4$ is executed if j needs to correct its decision value to the majority of the decision values of its neighbors in the current round. $MRO5$ is a restricted version of action $RO2$ that allows j to perform its output iff its decision value is that of majority. Together, the actions $MRO2-4$ and the restriction to action $RO2$ in $MRO5$ define the masking component (cf. the dashed box below).

To model Byzantine execution of j, we introduce action $MRO6$ that is executed only if $b.j$ is true: $MRO6$ lets j change $D.j.j$ and, thereby, affect the value read by process k when k executes $MRO3$. $MRO6$ also lets j obtain arbitrary values for $D.j.k$ and, thereby, affect the value of $d.j$ when j executes $MRO4$. Thus, the six actions of MRO are as follows:

$MRO1 :: RO1$

$MRO2 :: \; d.j \neq \perp \; \wedge SN.j.j = sn.j \wedge \; compl.j \quad \longrightarrow \; D.j.j,\, SN.j.j := d.j,\, SN.j.j \oplus 1$

$MRO3 :: \; SN.j.k \; \oplus 1 \; = \; SN.k.k \quad \longrightarrow \; D.j.k,\, SN.j.k := D.k.k,\, SN.k.k$

$MRO4 :: \; d.j \neq \perp \; \wedge \; majdefined.j \; \wedge \; d.j \neq maj.j \quad \longrightarrow \; d.j := maj.j$

$MRO5 :: \; d.j \neq \perp \; \wedge \; majdefined.j \; \wedge \; d.j = maj.j \quad \longrightarrow \; output\_decision(d.j) \,;\, d.j,\, sn.j := \perp, \; sn.j \oplus 1$

$MRO6 :: \; b.j \quad \longrightarrow \; D.j.j := 0 \mid 1;$
$\qquad (\| \, k : SN.j.k \oplus 1 = SN.k.k : D.j.k,\, SN.j.k := 0 \mid 1,\, SN.k.k)$

$$\begin{array}{lll} \text{where,} \quad compl.j & \equiv & (\forall k :: SN.j.j = SN.k.j) \\ \quad majdefined.j & \equiv & compl.j \; \wedge \; (\forall k :: SN.j.j = SN.j.k) \; \wedge \; (sn.j \neq SN.j.j) \\ \quad maj.j & = & (majority \; k :: D.j.k) \end{array}$$

Fault Actions. If the number of Byzantine processes is less than 1, the fault actions make some process Byzantine. Thus, letting l and m range over all processes, the fault actions are:

$$|\{l : b.l\}| < 1 \qquad \longrightarrow \qquad b.m := true$$

Proof of correctness. In accordance with the design issues discussed in the introduction, this proof consists of two parts: (1) The masking component offers masking tolerance to R in the presence of Byzantine faults, and (2) the masking component does not interfere with R in the absence of faults.

(1) For each round of the system computation, let $v.j$ denote the value obtained by j in that round when it executes $RO1$, and let $cordec$ be defined as follows.

$$\begin{array}{lll} cordec & = d.g & \text{if } \neg b.g \\ & = (majority \; j :: v.j) & \text{otherwise} \end{array}$$

Observe that in the start state of the round —where the sequence numbers of all processes are identical, i.e. $(\forall j, k :: sn.j = SN.j.k = sn.g)$, and no non-Byzantine process has read the decision of g, i.e. $(\forall j : \neg b.j : d.j = \perp)$— only action $RG1$ in g can be executed. Thereafter, the only action enabled at each non-Byzantine process j is $RO1$.

After j executes $RO1$, j can only execute its masking component. Moreover, j cannot execute $RO2$ until the masking component in j terminates in that round.

Thus, the masking component in j executes *between* the actions $RO1$ and $RO2$ in j.

The masking component in j first executes action $MRO2$ to increment $SN.j.j$. By the same token, the masking component in k increments $SN.k.k$. Subsequently, the masking component in j can execute $MRO3$, to update $SN.j.k$ and $D.j.k$. Likewise, each k can execute $MRO3$ to update $SN.k.j$ and $D.k.j$. Note that if k is non-Byzantine, $D.j.k$ is the same as $v.k$, which in turn is equal to $d.g$ if g is also non-Byzantine. It follows that eventually $majdefined.j \wedge maj.j = cordec$ holds, and the masking component in j can subsequently ensure that $d.j = maj.j$ before it terminates in that round.

After the masking component in j terminates, j can only execute action $RO2$. It follows that, in the presence of a Byzantine fault, each round of the system computation satisfies Validity and Agreement.

(2) Observe that, in the absence of a Byzantine fault, the masking component eventually satisfies $majdefined.j \wedge d.j = maj.j$ in each round and then terminates. Therefore, the masking component does not interfere with R in the absence of a Byzantine fault. □

5 Adding Stabilizing Tolerance to Transient & Byzantine Failures

Despite the addition of the masking component to the program R, the resulting program MR is not yet stabilizing tolerant to transient and Byzantine failures. For example, MR deadlocks if its state is transiently corrupted into one where some non-general process j incorrectly believes that it has completed its last round, i.e., $d.j = \perp \ \wedge \ SN.j.j \neq sn.j$. It therefore suffices to add a "stabilizing" component to MR that ensures stabilizing tolerance to transient and Byzantine failures while preserving the masking tolerance to Byzantine failure.

Towards designing the stabilizing component, we observe that in the absence of transient faults the following state predicates are invariantly true of MR: (i) whenever $d.j$ is set to $\perp$, by executing action $MRO5$, j increments $sn.j$, thus satisfying $SN.j.j = sn.j$; and (ii) whenever j sets $sn.j$ to be equal to $sn.g$, by executing action $MRO5$, $d.j$ is the same as $\perp$. In the presence of transient faults, however, these two state predicates may be violated. Therefore, to add stabilizing tolerance, we need to guarantee that these two state predicates are corrected.

To this end, we add two corresponding correction actions, namely $MRO7$ and $MRO8$, to the non-general processes. Action $MRO7$ is executed when $d.j$ is $\perp$ and $SN.j.j$ is different from $sn.j$, and it sets $SN.j.j$ to be equal to $sn.j$. Action $MRO8$ is executed when $sn.j$ is the same as $sn.g$ but $d.j$ is different from $\perp$, and it sets $d.j$ to be equal to $\perp$. With the addition of this stabilizing component to MR, we get a multitolerant program SMR.

$MRO7 ::$	$d.j = \perp \ \wedge \ SN.j.j \neq sn.j$	$\longrightarrow$	$SN.j.j := sn.j$
$MRO8 ::$	$d.j \neq \perp \ \wedge \ sn.j = sn.g$	$\longrightarrow$	$d.j := \perp$

Fault Actions. In addition to the Byzantine fault actions, we now consider the transient state corruption fault actions (let j and k range over non-general processes):

$$\begin{array}{lll} true & \longrightarrow & d.g, sn.g := 0|1, 0|1 \\ true & \longrightarrow & d.j, sn.j := 0|1, 0|1 \\ true & \longrightarrow & SN.j.k, D.j.k := 0|1, 0|1 \end{array}$$

Proof of Correctness. In accordance with the design issues discussed in the introduction, this proof consists of three parts: (1) The stabilizing component offers stabilizing tolerance to MR in the presence of transient and Byzantine faults, (2) the stabilizing component does not interfere with the execution of MR in the absence of faults, and (3) the stabilizing component does not interfere with the masking tolerance of MR in the presence of Byzantine faults only.

(1) Observe that execution of the component in isolation ensures that eventually the program reaches a state where the state predicate S holds, where

$$S \;=\; (d.j = \bot \;\Rightarrow\; SN.j.j = sn.j) \;\;\wedge\;\; (sn.j = sn.g \;\;\Rightarrow\;\; d.j = \bot)\;.$$

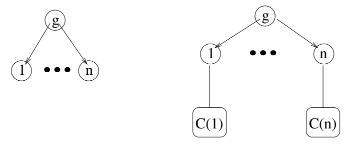
Since both disjuncts in S are preserved by the execution of all actions in MR, program MR does not interfere with the correction of S by stabilizing component. Starting from a state satisfying S, at most one round is executed incorrectly. For reasons of space, we omit the proof here and refer the interested reader to [7].

(2) Observe that, in the absence of faults, S continues to be preserved, and hence the stabilizing component is never executed. Therefore, the stabilizing component does not interfere with MR in the absence of faults.

(3) As in part (2), observe that, in the presence of Byzantine faults only, S continues to be preserved and, hence, the stabilizing component is never executed. Therefore, the stabilizing component does not interfere with MR in the presence of Byzantine faults. □

6 Extension to Tolerate Multiple Byzantine Faults

To motivate the generalization of SMR to handle t Byzantine failures given n non-general processes, where $n \geq 3t$, let us take a closer look at how program SMR is derived from R. To design SMR, we added to each process j a set of components $C(j)$ (see Figure 1).



(a) : Program R (b) : Program SMR

g : Actions $MRG1-2$
j : Actions $RO1-2$ and $MRO6$
$C(j)$: Actions $MRO2-4$, $MRO7-8$, and the restriction of $MRO5$

Figure 1: **Structure of R and SMR**

Note that action $MRO2$ is of the form of $RG1$ and that action $MRO3$ is of the form $RO1$ followed by $RO2$. ($D.j.j$ and $SN.j.j$ play the role of $d.g$ and $sn.g$ and $D.j.k$ and $SN.j.k$ play the role of the d values at the non-general processes.) In other words, $C(j)$ itself contains a repetitive agreement program !

With this insight, we are now ready to generalize program SMR to handle the multiple Byzantine faults, based on an idea that is essentially due to Lamport, Shostak, and Pease [4]. (Our generalization, of course, is distinguished by being multitolerant.) Let g denote the general process, X denote the set of non-general processes, t denote the maximum number of Byzantine processes. We define $SMR(g, X, t) = BYZ(g, X, t, \langle\rangle)$, where

$$
\begin{array}{lcl}
BYZ(g,X,t,s) & = & inp(g,X,t,s) \wedge MRG1(g,X,t,s) \quad [\!] \quad MRG2(g,X,t,s) \\
& & [\!] \\
& & (\ [\!]\ j : j \in X :\quad RO1(j,X,t,s) \quad [\!] \quad w(j,X,t,s) \wedge RO2(j,X,t,s) \quad [\!] \\
& & \qquad\qquad\qquad MRO6(j,X,t,s)\ ) \\
& & [\!] \\
& & (\ [\!]\ j : j \in X :\ \ C(j,X,t,s \circ g)\ \ )
\end{array}
$$

and

$$
\begin{array}{lcll}
inp(g,X,t,s) & = & d.(last(s), X \cup \{g\}, t{+}1, trlast(s)) \neq \bot \quad \wedge & \\
& & sn.(g,X,t,s) \ = \ sn.(last(s), X \cup \{g\}, t{+}1, trlast(s)) & \text{if } s \neq \langle\rangle \\
& = & new\_decision() & \text{otherwise} \\
w(j,X,t,s) & = & majdefined.(j,X,t,s) \quad \wedge \quad d.(j,X,t,s) = maj.(j,X,t,s) & \text{if } t > 0 \\
& = & true & \text{otherwise} \\
C(j,X,t,s) & = & MRO4(j,X,t,trlast(s)) \quad [\!] \quad MRO7(j,X,t,trlast(s)) \quad [\!] & \\
& & MRO8(j,X,t,trlast(s)) \quad [\!] \quad BYZ(j, X-\{j\}, t{-}1, s) & \text{if } t > 0 \\
& = & \text{the empty program} & \text{otherwise}
\end{array}
$$

Here s is a sequence ; $last(s)$ denotes the last element of s ; $trlast(s)$ denotes the sequence obtained by omitting the last element of s ; $s \circ j$ denotes the sequence obtained by appending j to s ; and action ac in program j is modified as follows:

- j is replaced with the quadruple (j, X, t, s)
- The quantification over k in $compl$ is over the set
  $\{\ (k, X-\{j\}, t{-}1, s \circ j)\ :\ k \in (X-\{j\})\ \} \ \cup\ \{\ (j, X, t, s)\ \}$
- The quantification over k in $majdefined$ and maj is over the set
  $\{\ (j, X{-}\{k\}, t{-}1, s \circ k)\ :\ k \in (X-\{j\})\ \} \ \cup\ \{\ (j, X, t, s)\ \}$
- If s is nonempty, the $output\_decision$ is assigned to the variable
  $D.(j, X, t, s).(j, X \cup last(s), t{+}1, trlast(s))$

Observe that if the definition of $SMR(g, X, t)$ is instantiated with $t = 0$, the resulting program is R. And, if the definition is instantiated with $t = 1$, the resulting program is SMR (with the previously noted exception that action $MRO3$ in j of SMR is implemented by $RO1$ and $RO2$ in the bottommost instantiation of BYZ, namely $BYZ(j, X-\{j\}, 0, \langle g\rangle)$).

Program $SMR(g, X, t)$ is multitolerant, i.e., it is masking tolerant to Byzantine faults and stabilizing tolerant to transient and Byzantine faults. We note that the structure of the proof of stabilization is the same as the proof for SMR: upon starting from any state, the program reaches a state where S holds; subsequently, g is guaranteed to start a new round infinitely often; and when g starts the $(t+1)-th$ round, the resulting computation satisfies Validity and Agreement. The proof of masking tolerance is similar to the one in [4].

7 Refining the Atomicity While Preserving Multitolerance

Our design thus far has assumed read-and-write atomicity, whereby each action of a process can atomically read the variables of the other processes and update the variables of that process. In this section, we show that our design can be refined into read-or-write atomicity, whereby each action of a process can either atomically read the variables of some other process or atomically write its variables but not both.

We choose a standard refinement [8]: In each process j a copy of every variable that j reads from another process is introduced. For each of these variables, an action is added to j that asynchronously reads that variable into the copy. Moreover, the actions of j are modified so that every occurrence of these variables is replaced by their corresponding copies.

We perform this refinement successively in each step of our design. Thus, we refine R first, the masking component second, and the stabilizing component last. Below, we prove the properties of the program resulting from each refinement step, in terms of the issues (1)-(3) discussed in the introduction.

Step 1: Correctness of the refined R. In the absence of faults, when g increments $sn.g$ by executing action $RG1$, the only actions of program R that can be executed are $RO1$ and then $RO2$ at each non-general process. Until each non-general process j executes $RO2$, g cannot execute any further action. Thus, by refining R, even if j first reads $d.g$ and $sn.g$, then updates its local copies of $d.g$ and $sn.g$, and later executes the refined action $RO1$, g cannot execute any other actions in the meanwhile. Hence, the computations of the refined R have the same effect as those of R.

Step 2: Correctness of the refined masking component. (1) In the presence of Byzantine faults, the refined actions of R do not interfere with the refined masking component. To see this, consider the refinement of action $MRO2$ of the masking component: To execute $MRO2$, j needs to read the variable $SN.k.j$ of process k. The refinement introduces a copy of $SN.k.j$ at j. For the refined action $MRO2$ to be enabled, j must first update these copies from other processes. Also, if $compl.j$ is true then it continues to be true unless j changes $SN.j.j$ by executing $MRO2$. Hence, $MRO2$ can be correctly refined. Likewise, actions $MRO4$ and $MRO5$ can be correctly refined. Regarding action $MRO3$, recall from Section 6 that $MRO3$ is equivalent to the simultaneous execution of $RO1$ and $RO2$ and, hence, it too can be correctly refined. Hence, the masking component executes only between the executions of $RO1$ and $RO2$ and thus the refined actions of R do not interfere with the refined masking component.

(2) In the absence of Byzantine faults, just as in Section 4, the refined masking component eventually satisfies $majdefined.j \ \wedge \ d.j = maj.j$ in each round and then terminates. Therefore, the refined masking component does not interfere with the refined R in the absence of Byzantine faults.

Step 3: Correctness of the refined stabilizing component. (1) Towards preserving stabilizing tolerance in the presence of transient and Byzantine faults while refining the stabilizing component, we claim that the set of possible sequence numbers has to be increased to $\{0..K-1\}$ where $K \geq 4$. (This claim follows from the fact that between g and j there are four sequence numbers, namely, $sn.g$, $sn.j$, and the copies of $sn.g$ and $sn.j$ at j and g respectively; for details, see [8].) Moreover, a deadlock has to be avoided in the states of refined version of R where $sn.j \neq sn.g$ and $sn.j \neq sn.g \ominus 1$. (These states do not exist in SMR since its sequence numbers are either 0 or 1.) Therefore, to preserve stabilization, we need to add actions to the stabilizing component that satisfy $sn.j \in \{sn.g, sn.g \ominus 1\}$. If these actions set $sn.j$ to $sn.g$, $d.j$ must also be set to $\perp$; otherwise, action $MRO2$ may interfere with this component by incrementing $sn.j$. Alternatively, these actions may set $sn.j$ to $sn.g \ominus 1$. Either alternative is acceptable. Since the refined R and the refined masking component preserve each constraint satisfied by the refined stabilizing component, the former does not interfere with the latter in the presence of transient and Byzantine faults.

(2) In the absence of Byzantine faults, just as in Section 5, the stabilizing component never executes. Therefore, the refined stabilizing component does not interfere with the other refined components in the absence of faults.

(3) In the presence of Byzantine faults only, again the stabilizing component never executes. Therefore, the refined stabilizing component does not interfere with the other refined components in the presence of Byzantine faults.

In summary, program SMR can be refined into read-or-write atomicity while preserving its multitolerance by asynchronously updating copies of the variables of "neighboring" processes. Note that the copies can be implemented by channels of unit length between the neighboring processes that lose any existing message in them when they receive a new message. It follows that the refined program is multitolerant for a message passing network where each channel has unit capacity. Moreover, using standard transformations, one can further refine the program into a message passing one with bounded capacity channels.

8 Generalizing From Our Design

In this section, we discuss the general aspects of our method in the context of the design for SMR.

We find that our stepwise method of adding a component to provide each desired tolerance property facilitated the solution of the problem at hand. It is worthwhile to point out that this method is general enough to design programs obtained from various existing fault-tolerance design methods such as replication, checkpointing and recovery, Schneider's state machine approach, exception handling, and Randell's recovery blocks.

Types of tolerance components. The stabilizing component we added to MR, to ensure that the state predicate S holds, is an instance of a *corrector*. Corrector components suffice for the design of stabilizing tolerance and, more generally, nonmasking tolerant programs. Wellknown examples of correctors include reset procedures, rollback-recovery, forward recovery, error correction codes, constraint (re)satisfaction, exception handlers, and alternate procedures in recovery blocks. Large correctors can be designed in a stepwise and hierarchical fashion by parallel and/or sequential composition of small correctors.

The masking component we added to R is itself composed of two sub-components, one a *detector* and the other a *corrector*. The detector consists of actions $MRO2-3$ and the restriction to $RO2$ in $MRO5$, while the corrector consists of actions $MRO2-4$. The task of the detector is to help preserve the safety properties (namely, Validity and Agreement) in the presence of Byzantine failure, by detecting the state predicate "the decision of j is that of the majority of the non-general processes", while the task of the corrector is to ensure that the same state predicate holds. Note that adding this detector but not the corresponding corrector would have yielded only fail-safe tolerance instead of masking tolerance. In other words, in the presence of Byzantine failure, Validity and Agreement would be satisfied if all processes output their decision, although some processes may never output their decision.

More generally, detector components suffice for the design of fail-safe tolerance and, together, detector and corrector components suffice for the design of masking tolerance. Wellknown examples of detectors include snapshot procedures, acceptance tests, error detection codes, consistency checkers, watchdog programs, snooper programs, and exception conditions. Analogous to the compositional design of large correctors, large detectors can be designed in a stepwise and hierarchical fashion, by parallel and/or sequential composition of small detectors. The interested reader is referred to a companion paper [1] for an in-depth study of detectors and correctors.

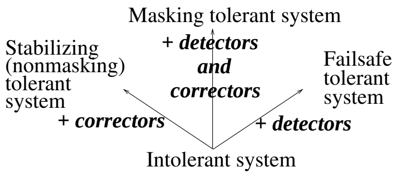


Figure 2: **Components that suffice for design of various tolerances**

Self-tolerances of components. Since a component that is added to tolerate a fault-class is itself subject to the fault-class, the question arises: what sort of tolerance should the component itself possess to that fault-class ?

We observe that in SMR, the masking component is itself masking tolerant to Byzantine faults and the stabilizing component is itself stabilizing tolerant to transient and Byzantine faults. In fact, in general, for adding stabilizing (nonmasking) tolerance, it suffices that the added component be stabilizing (nonmasking) tolerant. Likewise, for adding fail-safe tolerance, it suffices that the added component be fail-safe tolerant. And, for adding masking tolerance, it suffices that the added component be masking tolerant.

In practice, the detectors and correctors often possess the desired tolerance trivially. But if they do not, one way to design components to be self-tolerant is by the analogous addition of more detectors and correctors components to them. Alternative ways are exemplified by designs that yield self-checking, self-stabilizing, or inherently fault-tolerant programs.

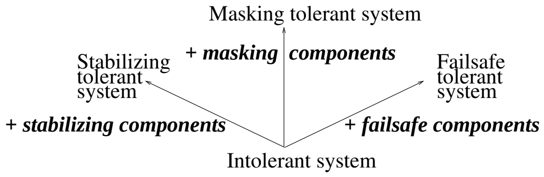


Figure 3: **Self-tolerances of components for various tolerances**

Stepwise design of tolerances. We observe that our decision to make the program masking tolerant first and stabilizing tolerant second is not crucial. *The same program could also be obtained by adding components in the reverse order, to deal with stabilization first and masking second.* In fact, in general, the same multitolerant program can be designed by adding the tolerance components in different orders.

For the special case of adding both detector and corrector components for masking tolerance, the design may be simplified by using a stepwise approach [5]: For instance, we may first augment the program with detectors and then augment the resulting fail-safe tolerant program with correctors. Alternatively, we may first augment the program with correctors and then augment the resulting nonmasking tolerant program with detectors.

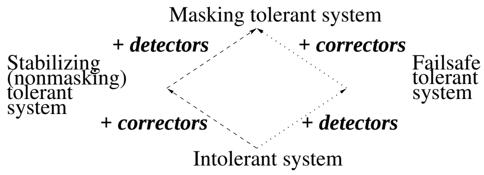


Figure 4: **Two approaches for stepwise design of masking tolerance**

On a related note, we observe that adding detectors to a stabilizing program suffices to enhance the tolerance of the correctors in the program from stabilizing to masking. Likewise, adding correctors to a fail-safe program suffices to enhance the tolerance of the detectors in the program from fail-safe to masking.

Stepwise refinement of tolerances. Our example illustrates the general principle that a component-based multitolerant program can be refined in the same manner as it is designed: in the first step, the original program is refined and its correctness in the absence of faults is shown. Then, the refined version of the first component is added and the tolerance properties in the presence of the first fault-class are shown. And so on for each fault-class. Thus, in obtaining the refined program, the design/proof of the original program can be reused.

Alternative tolerances to Byzantine failures. In the presence of Byzantine failures alone, SMR satisfies the specification of repetitive agreement in *each* round. The reader may wonder whether this strong guarantee is true of every repetitive agreement program that tolerates Byzantine failures. The answer is negative: Zhao and Bastani [6] have presented a program that is nonmasking tolerant to Byzantine failures, i.e., that could violate the specification in some finite number of rounds only. Moreover, we present here a program that is stabilizing tolerant —but not masking tolerant— to Byzantine failure. This program is composed from SMR and a nonmasking tolerant program outlined below. Again, for simplicity, we consider the special case where there are four processes and at most one is Byzantine.

In our nonmasking tolerant program, each non-general process j chooses a "parent" of j that is initially g. In each round, j receives the decision value of its parent and outputs that value as the decision of j. In parallel, j obtains the decision value of g and forwards it to other non-general processes. If the values that j receives from g and the other two processes are not all identical, j is allowed to change its parent, so that it will output a correct decision in the following rounds, as follows.

Let j, k, and l be the three non-general processes. We consider two cases: (1) g is Byzantine and (2) g is non-Byzantine. Case (1): If g sends the value B to l and $B \oplus 1$ to j and the remaining process k, j and k will suspect that l or g is Byzantine, and l will know that g is Byzantine and that j and k are non-Byzantine. Without loss of generality, let the id of j be greater than that of k. We let both j and l change their parent to k (to avoid forming a cycle in the parent relation, k retains g as its parent). In all future rounds, j and l output the value received from k and, hence, the decision output by j, k, and l is identical. Case (2): Since the values sent by both j and k are the same, both j and k are non-Byzantine. Again, assuming the id of k is greater than that of j, it is safe to let j change its parent to k. In all future rounds, the decision output by j and k is the same as that output by g.

It follows that the nonmasking tolerant program executes only a finite number of rounds incorrectly in the presence of at most one Byzantine failure. This program is made stabilizing by adding SMR to it, as follows: Each process j is in one of two modes: *nonmasking* or *stabilizing*. It executes the nonmasking tolerant program when it is in the nonmasking mode, and it executes SMR when it is in the stabilizing mode. Further, it is allowed to change from the nonmasking mode to the stabilizing mode, but not vice versa. Observe that the nonmasking tolerant program satisfies the state predicate "if the parent of j is k for some $k \neq g$, then k is non-Byzantine, the parent of k is g, and the parent of l is k provided l is non-Byzantine". Hence, if j suspects that this predicate is violated, i.e., in some round j detects that either g or k is Byzantine, or the parent of k is not g, or the parent of l is not k, then j changes to the stabilizing mode and starts executing SMR. Moreover, whenever j detects that some other process is in the stabilizing mode, it changes its mode to stabilizing. Thus, if the composite program is perturbed to a state that is not reached by the nonmasking tolerant program, eventually all processes execute actions of SMR. It follows that the composite program is stabilizing tolerant but not masking tolerant to Byzantine failures.

9 Concluding Remarks

In this paper, we presented a case study in the design of multitolerance. Starting with an intolerant program, we first added a component for masking tolerance and then added another component for stabilizing tolerance. Our design illustrated the issues of: (i) how to add a component for offering a tolerance in the presence of a fault-class, (ii) how to ensure that the added component does not interfere with the satisfaction of the program specification in the absence of faults, and (iii) how to ensure that the added component does not interfere with the tolerances offered to the previously considered fault-classes. While our proofs of interference-freedom were presented in informal terms, they are readily formalized, e.g., in the temporal logic framework of [1]. The formalization builds upon previous work on compositionality, e.g. [9, 10, 11].

We observed that a similar design is possible where we first add stabilizing tolerance and then masking tolerance. Also, not every repetitive agreement program that stabilizes in the presence of transient and Byzantine faults needs to be masking tolerant to Byzantine faults; in particular, it could be only nonmasking tolerant to Byzantine faults. Moreover, as discussed in Section 7, our initial design which assumed read-and-write atomicity is readily refined within the context of our design method into read-or-write atomicity or message-passing.

A reviewer of this paper correctly observed that our design method could benefit from using the principles of adaptive fault-tolerance [12]. Indeed, one way of simplifying the interference-freedom obligation between tolerance components is to choose only one of the tolerance components for execution at a time and to adapt this choice as the (fault) environment and internal state of the system changes. Note, however, that the mechanism for adapting the choice of tolerance component must itself be multitolerant.

References

[1] A. Arora and S. S. Kulkarni. Component-based design of multitolerance. Technical Report OSU-CISRC TR37, Ohio State University, 1996.

[2] Z. Liu and M. Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[3] K. P. Birman and R. van Renesse. *Reliable distributed computing using the Isis Toolkit.* IEEE Computer Society Press, 1994.

[4] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[5] A. Arora and S. S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 1997, to appear.

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

[7] S. S. Kulkarni and A. Arora. Compositional design of multitolerant repetitive Byzantine agreement (preliminary version). *Third Workshop on Self-Stabilizing Systems (WSS 97), University of California, Santa Barbara*, 1997.

[8] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[9] K. Apt, N. Francez, and W.-P. de Roever. A proof system for communicating sequential processes. *ACM transactions on Programming Languages and Systems*, pages 359–385, 1980.

[10] H. Schepers. *Fault Tolerance and Timing of Distributed Systems: Compositional specification and verification.* PhD thesis, Eindhoven University, 1994.

[11] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[12] J. Goldberg, I. Greenberg, and T. Lawrence. Adaptive fault-tolerance. *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–138, 1993.