

# Fault-tolerant Wait-free Shared Objects<sup>\*†</sup>

Prasad Jayanti<sup>‡</sup>      Tushar Deepak Chandra<sup>§</sup>      Sam Toueg<sup>¶</sup>

## Abstract

Wait-free implementations of shared objects tolerate the failure of processes, but not the failure of base objects from which they are implemented. We consider the problem of implementing shared objects that tolerate the failure of both processes and base objects.

We identify two classes of object failures: *responsive* and *non-responsive*. With responsive failures, a faulty object responds to every operation, but its responses may be incorrect. With non-responsive failures, a faulty object may also “hang” without responding. In each class, we define *crash*, *omission*, and *arbitrary* modes of failure.

We show that all responsive failure modes can be tolerated. More precisely, for all responsive failure modes  $\mathcal{F}$ , object types  $T$ , and  $t \geq 0$ , we show how to implement a shared object of type  $T$  which is  $t$ -tolerant for  $\mathcal{F}$ . Such an object remains correct and wait-free even if up to  $t$  base objects fail according to  $\mathcal{F}$ . In contrast to responsive failures, we show that even the most benign non-responsive failure mode cannot be tolerated. We also show that randomization can be used to circumvent this impossibility result.

*Graceful degradation* is a desirable property of fault-tolerant implementations: the implemented object never fails more severely than the base objects it is derived from, even if *all* the base objects fail. For several failure modes, we show whether this property can be achieved, and, if so, how.

## 1 Introduction

### 1.1 Problem addressed

We consider concurrent systems in which asynchronous processes communicate via typed linearizable shared objects. In such systems, complex (shared) objects, such as queues and stacks, are implemented in software from simple objects, such as registers and test&sets,

---

<sup>\*</sup>A preliminary version of this appeared in the proceedings of the 33rd IEEE Annual Symposium on Foundations of Computer Science, October, 1992.

<sup>†</sup>Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory. Second author supported also by an IBM graduate fellowship.

<sup>‡</sup>6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH 03755.

<sup>§</sup>IBM T.J. Watson Research Center, Hawthorne, NY 10532.

<sup>¶</sup>Department of Computer Science, Cornell University, Ithaca, NY 14853.

which are often supported in hardware. Traditional implementations (for example, [CHP71]) use lock-based techniques and are consequently not fault-tolerant: if any process crashes while holding the lock, the other processes are effectively prevented from accessing the implemented object. Wait-free implementations, which have been the focus of much recent research, were introduced to overcome this drawback [Lam77]. An implementation is *wait-free* if every access by a non-faulty process is guaranteed a response, regardless of whether the other processes are slow, fast, or have crashed.

Wait-free implementations of shared objects tolerate the failure of processes, but not the failure of base objects from which they are implemented. We consider the problem of implementing shared objects that tolerate the failure of both processes and base objects.

We divide object failures into two broad classes: *responsive* and *non-responsive*. With responsive failures, a faulty object responds to every operation, but its responses may be incorrect. With non-responsive failures, a faulty object may also “hang” without responding.

We divide the responsive class into three failure modes: *crash*, *omission*, and *arbitrary*. An object that fails by crash behaves correctly until it fails and, once it fails, it returns a distinguished response  $\perp$  to every operation. Clearly, crash is the most benign failure mode. The most severe responsive failure mode is the arbitrary mode. Objects experiencing arbitrary failures may “lie”, *i.e.*, they may return arbitrary responses. In terms of severity, omission falls between crash and arbitrary. When an object fails by omission, it returns normal responses to some operations and  $\perp$  to others, and satisfies the following property: the object would seem non-faulty if every operation that obtained the response  $\perp$  were treated like an incomplete operation that never obtained a response. Our study of omission failures is motivated by the fact that implementations tolerating such failures can be composed, but implementations tolerating the simpler crash failures cannot be.

Similarly, we divide the non-responsive class into *NR-crash*, *NR-omission*, and *NR-arbitrary* failure modes. An object that fails by NR-crash behaves correctly until it fails and, once it fails, it stops responding. An object that fails by NR-omission may fail to respond to the operations of an arbitrary subset of processes, but continue to respond to the operations of the remaining processes (forever). The behavior of an object that fails by NR-arbitrary is completely unrestricted: it may not respond to an operation and, even if it does, the response may be arbitrary.

An implementation  $\mathcal{I}$  is *t-tolerant for failure mode  $\mathcal{F}$*  if the implemented object remains wait-free and correct even if at most  $t$  base objects fail by  $\mathcal{F}$ . (We use the term *derived object* for the implemented object and the term *base objects* for the objects used in the implementation.) The *resource complexity* of  $\mathcal{I}$  is the number of base objects used in  $\mathcal{I}$ .  $\mathcal{I}$  is a *self-implementation* if all base objects are of the same type as the derived object.

Consider a  $t$ -tolerant implementation for failure mode  $\mathcal{F}$ . By definition, a derived object of this implementation is guaranteed to behave correctly even if up to  $t$  base objects fail by  $\mathcal{F}$ . But what happens if more than  $t$  base objects fail by  $\mathcal{F}$ ? In general, the derived object may experience a more severe failure than  $\mathcal{F}$ . In other words, implementations may “amplify” failures: derived objects may fail more severely than base objects. This

undesirable behavior is prevented by implementations that are “gracefully degrading”. An implementation is *gracefully degrading for failure mode  $\mathcal{F}$*  if it has the following property: if base objects only fail by  $\mathcal{F}$ , then the derived object does not fail more “severely” than  $\mathcal{F}$ . Thus, if  $\mathcal{F}$  is guaranteed to be the most severe failure mode that hardware objects may experience, the graceful degradation property of an implementation makes it possible to extend the same guarantee to software objects.

We study the problem of designing  $t$ -tolerant and/or gracefully degrading implementations for the various responsive and non-responsive failure modes. An independent work by Afek, Greenberg, Merritt, and Taubenfeld [AGMT92] has the same general goal, but differs in many respects. We present a comparison of the two works in Section 8.

## 1.2 Summary of results

The three main topics studied are: tolerating responsive failures, tolerating non-responsive failures, and achieving graceful degradation. The following are the main conclusions: (1) it is feasible to design deterministic implementations that tolerate even the most severe of the responsive failures, *viz.*, arbitrary failures, (2) Implementations cannot tolerate even the simplest of non-responsive failures, *viz.*, crash failures, without the use of randomization, and (3) Of the two benign failure modes, *viz.*, crash and omission, it is feasible to design gracefully degrading implementations for omission, but not for crash. Accordingly, we give three fault-tolerant universal constructions — a deterministic one for arbitrary failures, a randomized one for non-responsive arbitrary failures, and a deterministic one for omission failures that also guarantees graceful degradation.

In the following, we say *type  $T$  has an implementation from a set  $\mathcal{S}$  of types* if it is possible to wait-free implement an object of type  $T$  from objects whose types are in  $\mathcal{S}$ . (We use the type-writer font for the names of types.)

Herlihy and Plotkin showed that every type has an implementation from  $\{\texttt{consensus}, \texttt{register}\}$  [Her88, Her91b, Plo89].<sup>1</sup> Hence, if the types **consensus** and **register** have  $t$ -tolerant implementations, then every type has a  $t$ -tolerant implementation. We therefore focus on obtaining  $t$ -tolerant implementations of **consensus** and **register**.

### 1.2.1 Tolerating responsive failures

We give  $t$ -tolerant self-implementations of **consensus** for crash, omission, and arbitrary failures. For crash and omission failures, our self-implementation is optimal requiring only  $t + 1$  base consensus objects. For arbitrary failures, our self-implementation is efficient requiring  $O(t \log t)$  base consensus objects. We also give  $t$ -tolerant self-implementations of **register** for crash, omission, and arbitrary failures. Combining the above results

---

<sup>1</sup>The type **consensus** supports two operations, *propose 0* and *propose 1*, and has the following sequential specification: if *propose v* is the first operation, then every operation gets the response *v*. The **register** supports *read* and *write* operations with the standard specification that a read returns the most recently written value.

with the universality results in [Her91b, Plo89], we conclude that *every* type  $T$  has a  $t$ -tolerant implementation (from `{consensus, register}`) for *all* responsive failure modes. Moreover, if  $T$  implements both `consensus` and `register`, then  $T$  has a  $t$ -tolerant *self*-implementation. This implies that familiar types such as (2-process) `fetch&add`, `queue`, `stack`, `test&set`, and ( $N$ -process) `compare&swap`, `move`, `memory-to-memory swap` have  $t$ -tolerant self-implementations even for arbitrary failures.

### 1.2.2 Tolerating non-responsive failures

An object that fails non-responsively may not respond to operations. Thus, if a process invokes an operation on an object and waits for the response before proceeding further, then a non-responsive failure of the object can result in the process waiting for the response forever! To overcome this difficulty, we allow a process to have pending operations on more than one object. In other words, we allow a process to invoke an operation on some object  $O_1$  and, without waiting for a response from  $O_1$ , to proceed to invoke an operation on a different object  $O_2$ . Thus, it is conceivable that  $t$  non-responsive failures can be tolerated by invoking  $n$  operations in parallel and waiting for  $n-t$  responses. Unfortunately, this is not the case. We show that there is no 1-tolerant implementation of `consensus` even for NR-crash failures, the most benign of the non-responsive failure modes.<sup>2</sup> This immediately implies that any type  $T$  that implements `consensus`, such as `fetch&add`, `queue`, `stack`, `test&set`, `compare&swap`, `move`, `sticky-bit`, and `memory-to-memory swap`, has no 1-tolerant implementation for NR-crash.

We ask whether randomization can be used to circumvent these impossibility results. The answer is yes. Specifically, we show that `register` has a  $t$ -tolerant (deterministic) *self*-implementation even for NR-arbitrary failures. Furthermore, randomized implementations of `consensus` from `register` are well-known (for example, see [Asp90]). These two results, together with the universality results in [Her91b, Plo89], imply that every type has a *randomized*  $t$ -tolerant implementation from `register` even for NR-arbitrary failures.

### 1.2.3 Achieving graceful degradation

If an implementation is gracefully degrading for failure mode  $\mathcal{F}$ , the derived object never fails more severely than  $\mathcal{F}$  provided that base objects fail only by  $\mathcal{F}$  (this property holds even if *all* base objects fail). Graceful degradation is clearly desirable. In fact, it also provides a method for automatically boosting the fault-tolerance of an implementation: We show that, given a 1-tolerant gracefully degrading self-implementation of any type  $T$  for any failure mode  $\mathcal{F}$ , one can construct a  $t$ -tolerant gracefully degrading self-implementation of  $T$  for  $\mathcal{F}$ .

Requiring graceful degradation may increase the cost of an implementation. For instance, consider  $t$ -tolerant implementations of `consensus` for omission failures. We present

---

<sup>2</sup>The impossibility of implementing a fault-tolerant *consensus object* from any finite set of base *objects*, one of which may fail by NR-crash, is shown using the impossibility of solving the *consensus problem* among a finite number of *processes*, one of which may crash [FLP85, LAA87, DDS87].

two such implementations. One uses only  $t+1$  base objects, but is not gracefully degrading. The other is gracefully degrading, but requires  $2t+1$  base objects. In fact, we show that for all non-trivial deterministic types  $T$ , any  $t$ -tolerant gracefully degrading implementation of  $T$  for omission failures requires at least  $2t+1$  base objects (no matter what the types of the base objects are).

The main question, however, is whether graceful degradation can be achieved at all. We answer this question for the crash and omission failure modes. We show that there is a large class of types that have no gracefully degrading implementations for crash. This class includes many common types, such as `queue`, `stack`, `test&set`, and `compare&swap`. Intuitively, crash is so benign that it is impossible to ensure that the implemented object does not fail more severely than crash even when base objects fail only by crash. In contrast, we prove the following universality result for omission failures: Every type has a  $t$ -tolerant gracefully degrading implementation from `{consensus, register}` for omission.

### 1.2.4 Miscellaneous results

We also study the problem of *translating* severe failures into more benign failures [NT90]. In particular, given  $3t+1$  (base) consensus objects, at most  $t$  of which may experience arbitrary failures, we show how to implement a consensus object that can only fail by omission. We prove that this translation from arbitrary to omission is resource optimal.

Finally, we show that NR-arbitrary failures can be viewed as having two orthogonal components: NR-omission and arbitrary. Specifically, for any type  $T$ , given any  $t$ -tolerant self-implementations  $\mathcal{I}'$  and  $\mathcal{I}''$  of  $T$  for NR-omission failures and arbitrary failures, respectively, we show how to construct a  $t$ -tolerant self-implementation of  $T$  for NR-arbitrary failures. This decomposition simplifies the problem of tolerating NR-arbitrary failures.

## 1.3 Organization

In Section 2, we describe the model. In Section 3, we define the responsive and non-responsive classes of failures, and the failure modes within each class. We define the concepts of  $t$ -tolerant implementation and graceful degradation in Section 4. The three main topics — tolerating responsive failures, tolerating non-responsive failures, and the feasibility of graceful degradation for crash and omission failure modes — are studied in Sections 5, 6, and 7, respectively. In Section 8, we present a comparison with the results in [AGMT92]. In Appendix A, we show how to translate arbitrary failures to omission failures for the type `consensus`. In Appendix B, we define all the types that appear in this paper.

## 2 Model

Our model is similar to Herlihy’s [Her91b], but there are some differences due to the need to model implementations that are both wait-free and tolerant of non-responsive object failures. These differences will be pointed to as they arise.

## 2.1 I/O Automaton

A concurrent system consists of processes and objects. We model processes and objects as I/O automata [LT88].

An *I/O Automaton*  $A$  is a non-deterministic automaton with the following components:

1.  $States(A)$  is a finite/infinite set of states, including a distinguished set of starting states.
2.  $In(A)$  is a set of input events.
3.  $Out(A)$  is a set of output events.
4.  $Int(A)$  is a set of internal events.
5.  $Step(A)$  is a transition relation given by a set of tuples  $(s, e, s')$ , where  $s$  and  $s'$  are states, and  $e$  is an event. Such a triple is called a *step*, and it means that an automaton in state  $s$  can undergo a transition to state  $s'$  and that transition is associated with event  $e$ .

If  $(s, e, s')$  is a step, we say  $e$  is *enabled* in state  $s$ . I/O Automata (abbreviated hereafter as automata) must additionally satisfy the requirement that input, output, and internal events are disjoint, and every input event is enabled in every state. The latter captures the fact that an automaton has no control over when input events occur.

An *execution* of an automaton  $A$  is a finite sequence  $s_0, e_1, s_1, e_2, s_2, \dots, e_n, s_n$  or an infinite sequence  $s_0, e_1, s_1, e_2, s_2, \dots$  of alternating states and events such that  $s_0$  is a starting state and  $(s_i, e_{i+1}, s_{i+1})$  is a step of  $A$ . A *history* of an automaton is the subsequence of events in an execution.

A new automaton can be constructed by composing a set of compatible automata. A pair  $A, B$  of automata is *compatible* if (i) the internal events of either automaton are disjoint from the events of the other, and (ii) the output events of the two automata are disjoint; that is,  $Int(A) \cap (In(B) \cup Int(B) \cup Out(B)) = \emptyset$ , and  $Int(B) \cap (In(A) \cup Int(A) \cup Out(A)) = \emptyset$ , and  $Out(A) \cap Out(B) = \emptyset$ . A set of automata is compatible if every pair in the set is compatible. We compose a new automaton  $S$  from compatible (component) automata as follows. A state of  $S$  is a tuple of the components' states, and a starting state of  $S$  is a tuple of the components' starting states.  $Out(S)$ , the set of output events of  $S$ , is the union of the sets of output events of the component automata.  $Int(S)$ , the set of internal events of  $S$ , is the union of the sets of internal events of the component automata.  $In(S)$ , the set of input events of  $S$ , is  $IN - Out(S)$ , where  $IN$  is the union of the sets of input events of the component automata. A triple  $(s, e, s')$  is in  $Step(S)$  if and only if, for all the component automata  $A$ , one of the following holds: (1)  $e$  is an event of  $A$  and the projection of the step onto  $A$  is in  $Step(A)$ , or (2)  $e$  is not an event of  $A$  and the state of  $A$  is the same in  $s$  and  $s'$ .

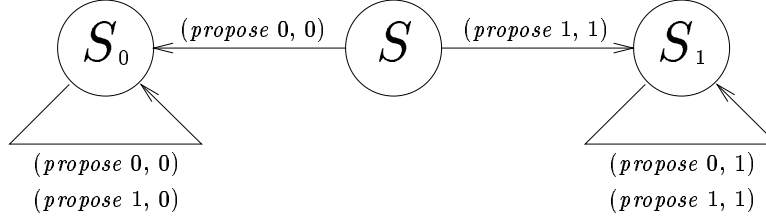


Figure 1: Sequential specification of **consensus**

---

Let  $E$  be an execution of an automaton composed from  $A_1, A_2, \dots, A_k$  and  $H$  be the corresponding history. The *history of a component  $A_i$  in  $E$* , denoted by  $H|A_i$ , is the subsequence of  $H$  consisting only of the events of  $A_i$ .

## 2.2 Object type

Every object has a type. The type specifies the expected behavior of the object. More precisely, a *type  $T$*  is a tuple  $(OP, RES, G, \tau)$  where  $OP$  and  $RES$  are sets of operations and responses respectively,  $G$  is a directed finite or infinite multi-graph in which each edge has a label of the form  $(op, res)$  where  $op \in OP$  and  $res \in RES$ , and  $\tau$  is a history transformation function. We refer to  $G$  as the *sequential specification* of  $T$  and the vertices of  $G$  as the *states* of  $T$ . Intuitively, if there is an edge, labeled  $(op, res)$ , from state  $s$  to state  $s'$ , it means that applying the operation  $op$  to an object in state  $s$  may change the state to  $s'$  and return the response  $res$ . We explain the history transformation function  $\tau$  later in Section 2.8.

A sequence  $\sigma = (op_1, res_1), (op_2, res_2), \dots, (op_l, res_l)$  is *legal from state  $s$  of  $T$*  if there is a path labeled  $\sigma$  in  $G$  from the state  $s$ .  $T$  is *deterministic* if, for all states  $s$  of  $T$  and for all operations  $op \in OP$ , there is at most one edge from  $s$  labeled  $(op, res)$  (for some  $res \in RES$ ).  $T$  is *non-deterministic* otherwise.  $T$  is *total* if, for all states  $s$  of  $T$  and for all operations  $op \in OP$ , there is at least one edge from  $s$  labeled  $(op, res)$  (for some  $res \in RES$ ). In this paper, we restrict our attention to total types.  $T$  is *finite* if it has only a finite number of states.  $T$  is *infinite* otherwise.

The types **consensus** and **consensus with safe-reset** are central to this paper. Their sequential specifications are presented in Figures 1 and 2. The sequential specifications of the remaining types mentioned in this paper are presented in Appendix B.

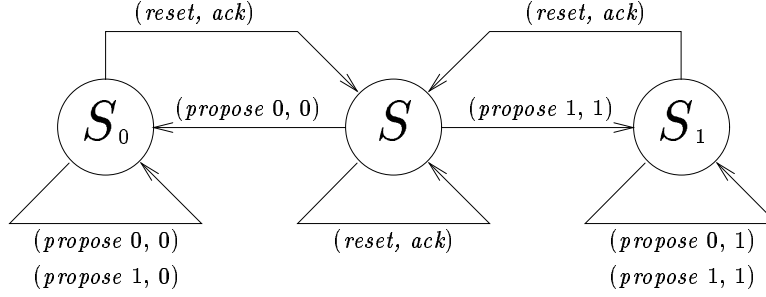


Figure 2: Sequential specification of consensus with `safe-reset`

---

### 2.3 Objects and Processes

As already mentioned, objects and processes are modeled as automata. Each object  $O$  has two attributes: a type  $T$  and a state  $s$  of  $T$  to which  $O$  is initialized.

We assume that a process can be made to crash (by an invisible adversary) at any point in an execution. We model this as follows. Every process  $P$  has a distinguished state  $FAIL(P)$ , an input event  $crash(P)$ , and an output event  $crashed(P)$ . From any state, the input event  $crash(P)$  moves  $P$  to state  $FAIL(P)$  and, once in state  $FAIL(P)$ , no event moves  $P$  out of that state. The output event  $crashed(P)$  is enabled only in  $FAIL(P)$ .

Unless mentioned otherwise, we assume that a process is deterministic. This implies that, for every state  $s$  of a process and event  $e$ , there is no more than one  $s'$  such that  $(s, e, s')$  is a step of the process.

### 2.4 Clock

A *clock* is an automaton with a single state *running*, a single internal event *tick*, and a single step  $(running, tick, running)$  in its transition relation. It has no input or output events. Thus, a clock does no more than generating ticks.

### 2.5 Concurrent system

A *concurrent system* consisting of processes  $P_1, P_2, \dots, P_n$  and objects  $O_1, \dots, O_m$  is defined as the automaton composed from the process automata  $P_i$ ,  $1 \leq i \leq n$ , the object automata  $O_j$ ,  $1 \leq j \leq m$ , and a clock automaton. We write  $(P_1, P_2, \dots, P_n; O_1, \dots, O_m)$  to denote such a concurrent system. The reader should notice that we have departed from the model in [Her91b] by including a clock as a component of a concurrent system, adding a *FAIL* state for every process, and making a fairness assumption on executions (see Section 2.6). These differences are motivated by the fact that our work introduces new concepts, such



as an implementation that is both wait-free and tolerant of non-responsive object failures. The fairness assumption guarantees that every process that attempts to take a step will eventually be able to do so. The clock ensures that, regardless of how processes and objects are specified, the system has infinite executions. Notice that the *ticks* generated by a clock are internal events of the clock. Thus, a process or an object cannot take advantage of the presence of a clock.

Let  $O_j$  be an object of type  $T = (OP, RES, G, \tau)$ . The input and output events of  $O_j$  include  $invoke(P_i, op, O_j)$  and  $respond(P_i, res, O_j)$ , respectively, where  $P_i$  is a process and  $op \in OP$ . We call these events *invocations* and *responses*, respectively. The input and output events of a process  $P_i$  include  $respond(P_i, res, O_j)$  and  $invoke(P_i, op, O_j)$ , respectively.

Let  $E$  be an execution of a concurrent system and  $H$  be the corresponding history. A response  $r$  *matches* an invocation  $i$  in  $H$  if  $i$  is the most recent invocation preceding  $r$  such that the process and object names of  $i$  and  $r$  agree. An *operation* in  $H$  is a pair of events, an invocation and its matching response.<sup>3</sup> An *incomplete operation* in  $H$  is an invocation with no matching response. History  $H$  is *complete* if it has no incomplete operations. We define a relation  $<_H$ , which reflects the partial “real time” order of operations in  $H$ , as follows. For any two operations  $oper$  and  $oper'$  in  $H$ ,  $oper <_H oper'$  if the response of  $oper$  precedes the invocation of  $oper'$ . We say that  $oper$  precedes  $oper'$  in  $H$ . Two operations unrelated by  $<_H$  (i.e., neither operation precedes the other) are said to be *concurrent* in  $H$ . History  $H$  is *sequential* if it has no concurrent operations.

We assume initially that a process is a single thread of control: after invoking an operation on an object, it waits to receive the response before it invokes another operation (on any object). We also assume that, for any process  $P_i$  and object  $O_j$ , the interaction between  $P_i$  and  $O_j$  is proper: first  $P_i$  invokes an operation on  $O_j$ , then  $O_j$  responds, and then  $P_i$  invokes on  $O_j$ , then  $O_j$  responds, and so on. We model these assumptions as follows. Let  $H$  be the history corresponding to an execution of a concurrent system. Recall that  $H|A$  denotes the history of component  $A$  in  $H$ , i.e., the subsequence of events in  $H$  which belong to the component  $A$ . Thus,  $(H|P_i)|O_j$  denotes the subsequence of events common to Process  $P_i$  and Object  $O_j$ . These events are invocations on  $O_j$  from  $P_i$  and responses to  $P_i$  from  $O_j$ . History  $H$  is *well-formed* if, for all processes  $P_i$  and objects  $O_j$ , the following conditions hold: (i) no prefix of  $H|P_i$  has more than one incomplete operation, and (ii)  $(H|P_i)|O_j$  begins with an invocation and has alternating invocations and responses. Except in Section 6 (where we study non-responsive failures), we restrict ourselves to well-formed histories of a concurrent system.

When a process is restricted to be a single thread of control, it will block if an object fails to respond to its invocation. Thus, it will be impossible to construct fault-tolerant implementations in the presence of non-responsive object failures. Hence, in Section 6, where such implementations are sought, we relax Condition (i) above and allow a process to have multiple incomplete operations. We however continue to insist on Condition (ii) which implies that a process can have no more than one incomplete operation on any one

---

<sup>3</sup> Thus, the term “operation” is overloaded. It will be however clear from the context whether a particular use of this term refers to an element of  $OP$  of a type  $T = (OP, RES, G, \tau)$  or to a pair of events in a history.

object.

## 2.6 Fairness assumption

An execution  $E$  of a concurrent system is *unfair* if  $E$  is infinite and the following holds: there is an internal or output event  $e$  and a suffix  $E'$  of  $E$  such that (i) for all states  $s$  in  $E'$ ,  $e$  is enabled in  $s$ , and (ii)  $e$  is not in  $E'$ . This definition does not consider input events since input events are, as mentioned before, enabled in every state of an execution. An execution  $E$  of a concurrent system is *fair* if it is not unfair. We restrict our attention to fair executions of a concurrent system.

The above fairness assumptions has two implications. First, every process that wishes to take a step will eventually be able to do so. Second, the presence of a clock, together with the fairness assumption, guarantees that every concurrent system, regardless of how its processes are specified, has infinite executions. As we will see, the latter property leads to simple definitions for a wait-free implementation and a wait-free implementation which is tolerant of non-responsive failures.

## 2.7 Linearizability

The sequential specification of a type specifies how an object behaves in the absence of concurrent operations. To characterize an object's behavior in the presence of concurrent operations, we additionally need the concept of *linearizability* [HW90]. Linearizability requires that each operation, spanning over an interval of time from the invocation of the operation to its response, must appear to take effect at some instant in this interval. We make this more precise below.

Let  $H$  be the history of some object in an execution of a concurrent system. Let  $T = (OP, RES, G, \tau)$  be a type and  $s$  be a state of  $T$ . A *linearization of  $H$  with respect to  $(T, s)$*  is a complete sequential history  $S$  with the following properties:

1.  $S$  is legal from state  $s$  of  $T$ .
2.  $S$  includes every complete operation in  $H$ .
3. Let  $\text{invoke}(P_i, op, \mathcal{O})$  be an incomplete operation in  $H$ . Then, either  $S$  does not include this incomplete operation or  $S$  includes a complete operation  $(\text{invoke}(P_i, op, \mathcal{O}), \text{respond}(P_i, res, \mathcal{O}))$  for some  $res \in RES$ .  
Intuitively, this captures the notion that some incomplete operations in  $H$  did not take effect, while the others did.
4.  $S$  includes no operations other than the ones mentioned in 1 or 2.
5. For all operations  $oper, oper'$  in  $S$ , if  $oper <_H oper'$  then  $oper <_S oper'$ .

Thus, the order of non-overlapping operations in  $H$  is preserved in  $S$ .

Notice that  $H$  may have no linearization or may have several different linearizations.  $H$  is *linearizable with respect to*  $(T, s)$  if  $H$  has a linearization with respect to  $(T, s)$ .

Let  $O$  be an object of type  $T$ , initialized to state  $s$  of  $T$ , and let  $H$  be the history of  $O$  in an execution  $E$  of a concurrent system. We say that  $O$  is *linearizable in*  $E$  if  $H$  is linearizable with respect to  $(T, s)$ .

## 2.8 Well-behavedness

It is tempting to say that an object is well-behaved in an execution if and only if it is linearizable in that execution. However some important objects that appeared in literature are not linearizable. Here are some examples.

- Consider the type **safe register**, defined by Lamport [Lam86]. It supports read and write operations and has the same sequential specification as **register**: every read returns the value written by the most recent write. However, in the presence of concurrent operations, a safe register extends fewer guarantees than a (linearizable or “atomic”) register. In particular, if a read operation on a safe register is concurrent with a write, then that read operation can return an arbitrary response. Thus, the history  $H$  of a safe register does not have to be linearizable. However,  $H$  satisfies the following weaker property [Lam86]: If  $H'$  is the result of removing all read operations in  $H$  that are concurrent with a write, then  $H'$  is linearizable.
- Consider the type **consensus with safe-reset** [Her91b]. Figure 2 presents its sequential specification. In using an object of this type, if a reset operation is concurrent with a propose or another reset operation, then the object is allowed to return arbitrary responses to all operations thereafter. Thus, the history  $H$  of an object of type **consensus with safe-reset** does not have to be linearizable. However,  $H$  satisfies the following weaker property [Her91b]: If  $H'$  is the maximal prefix of  $H$  in which no reset operation is concurrent with any other operation, then  $H'$  is linearizable.
- Consider the type **1-reader 1-writer register**. A history  $H$  of an object of this type does not have to be linearizable if either more than one process reads or more than one process writes. However,  $H$  satisfies the following weaker property: If  $H'$  is the maximal prefix of  $H$  in which no more than one process reads and no more than one process writes, then  $H'$  is linearizable.
- Consider the type **1-reader 1-writer safe register**. A history  $H$  of an object of this type satisfies the following property. Let  $H'$  be the maximal prefix of  $H$  in which no more than one process reads and no more than one process writes. Let  $H''$  be the result of removing all read operations in  $H'$  that are concurrent with a write. Then,  $H''$  is linearizable.

In all these examples, given a history  $H$  of an object of type  $T$ , we required that a transformation of  $H$ , not  $H$  itself, be linearizable with respect to  $T$ . This is the motivation for including a history transformation function  $\tau$  as a component in the 4-tuple

defining a type. We are now ready to define well-behavedness. Let  $O$  be an object of type  $T = (OP, RES, G, \tau)$  which is initialized to state  $s$  of  $T$ . Let  $H$  be the history of  $O$  in an execution  $E$  of a concurrent system. We say that  $O$  is *well-behaved in  $E$*  if  $\tau(H)$  is linearizable with respect to  $(T, s)$ .

For most types considered in this paper, such as **consensus**, **register**, and **queue**, the history transformation function is the identity function. Thus, for these types, well-behavedness is the same as linearizability. The following types are the exceptions in this paper: **1-reader 1-writer register**, **1-reader 1-writer safe register**, and **consensus with safe-reset**. The history transformation functions for these types should be obvious from the above discussion.

## 2.9 Wait-freedom and correctness

Recall that every process automaton has a *FAIL* state. A *process  $P$  crashes in an execution  $E$*  of a concurrent system if the state of  $P$  is  $FAIL(P)$  at any point in  $E$ .  $P$  is *correct in  $E$*  if it does not crash in  $E$ . An *object  $O$  is wait-free in  $E$*  if either  $E$  is finite or every invocation on  $O$  by a correct process has a matching response. An *object  $O$  is correct in  $E$*  if  $O$  is wait-free and well-behaved in  $E$ . *Object  $O$  fails in  $E$*  if  $O$  is not correct in  $E$ .

## 2.10 Implementation

Let  $T$  be a type and  $s$  be a state of  $T$ . Further, let  $\mathcal{L} = (T_1, T_2, \dots)$  be a list of types (the list may be infinite and the types in the list need not be distinct) and  $\Sigma = (s_1, s_2, \dots)$  be a list where  $s_i$  is a state of type  $T_i$ . An *implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$  for processes  $P_1, P_2, \dots, P_N$*  is a function  $\mathcal{I}(O_1, O_2, \dots)$  satisfying the following properties:

1. There exist process automata  $F_1, F_2, \dots, F_N$ , known as the *front-ends*, such that if  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots)$ , then  $\mathcal{O}$  is the automaton  $(F_1, F_2, \dots, F_N; O_1, O_2, \dots)$ .
2. Front-ends  $F_i$  and  $F_j$  ( $i \neq j$ ) have no common events.
3. Let  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots)$ . Each input event  $invoke(P_i, op, \mathcal{O})$  of  $\mathcal{O}$  is matched with an input event of  $F_i$ ; each output event  $respond(P_i, res, \mathcal{O})$  of  $\mathcal{O}$  is matched with an output event of  $F_i$ .
4. Each output event  $crashed(P_i)$  of Process  $P_i$  is matched with the input event  $crash(F_i)$  of the front-end  $F_i$ .
5. Let  $O_1, O_2, \dots$  be distinct objects of types  $T_1, T_2, \dots$ , initialized to states  $s_1, s_2, \dots$ , respectively. Then,  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots)$  is an object of type  $T$ , initialized to state  $s$ , with the following property: for every execution  $E$  of the concurrent system  $(P_1, P_2, \dots, P_N; \mathcal{O})$ , if  $O_1, O_2, \dots$  are well-behaved in  $E$ , then  $\mathcal{O}$  is well-behaved in  $E$ .

Informally, the front-end  $F_i$  is represented by a set of access procedures  $\text{Apply}(P_i, op, \mathcal{O})$  ( $op \in OP(T)$ ).  $\text{Apply}(P_i, op, \mathcal{O})$  specifies how process  $P_i$  should “simulate” the operation  $op$  on  $\mathcal{O}$  in terms of operations on  $O_1, O_2, \dots$ . We say that  $\mathcal{O}$  is a *derived object* of the implementation  $\mathcal{I}$ , and  $O_1, O_2, \dots$  are the *base objects* of  $\mathcal{O}$ . The *resource complexity* of  $\mathcal{I}$  is the number of base objects required by  $\mathcal{I}$  to implement a derived object.

Condition 1 above states that a derived object is constituted by base objects and access procedures (front-ends).

Condition 2 captures the notion that the execution of a step of the access procedure by one process  $P_i$  cannot affect the state of another process  $P_j$ .

Condition 3 captures the notion that (i) invoking an operation on  $\mathcal{O}$  by process  $P_i$  activates the front-end  $F_i$  or, equivalently, begins the execution of an access procedure, and (ii) the value returned by the front-end (access procedure)  $F_i$  is the response of  $\mathcal{O}$ .

Condition 4 captures our intuition that when a process  $P_i$  crashes, the front end  $F_i$  of that process must stop executing.

Condition 5 ensures that a derived object is well-behaved whenever all its base objects are well-behaved.

An implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$  is a *self-implementation* if every type in the list  $\mathcal{L}$  is  $T$ . Thus, in a self-implementation, base objects are of the same type as the derived object.

We say that  $\mathcal{I}$  is an *implementation of  $(T, s)$  from a set  $\mathcal{S}$  of types for  $N$  processes* if there is a list  $\mathcal{L} = (T_1, T_2, \dots)$  of types and a list  $\Sigma = (s_1, s_2, \dots)$  of states such that  $T_i \in \mathcal{S}$ ,  $s_i$  is a state of  $T_i$ , and  $\mathcal{I}$  is an implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$  for  $N$  processes. We say that a *type  $T$  has an implementation from a set  $\mathcal{S}$  of types for  $N$  processes* if, for all states  $s$  of  $T$ , there is an implementation of  $(T, s)$  from  $\mathcal{S}$  for  $N$  processes. Finally, we say that  $T$  *implements  $T'$*  if there is an implementation of  $T'$  from  $\{T\}$ .

## 2.11 Wait-free implementation

An *implementation for  $N$  processes is wait-free* if every derived object  $\mathcal{O}$  has the following property: if  $E$  is an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$  in which all base objects of  $\mathcal{O}$  are wait-free, then  $\mathcal{O}$  is wait-free in  $E$ .

Let us briefly examine how this definition captures our intuitive notion of what a wait-free implementation is. Consider an infinite execution  $E$  of the concurrent system  $(P_1, P_2, \dots, P_N; \mathcal{O})$ . (As already mentioned, the clock and the fairness assumption, together, guarantee that such an infinite execution exists.) Assume that all base objects are wait-free in  $E$ . Thus, every base object returns a response to every operation from every correct process. By the fairness assumption, every correct process succeeds in taking all the steps that it attempts in  $E$ . Hence, if the implementation is wait-free, we expect every correct process to succeed in completing every operation it attempts on the derived object  $\mathcal{O}$ . In other words, we expect that  $\mathcal{O}$  is wait-free in  $E$ . That is precisely what the definition states.

An *implementation for  $N$  processes is  $k$ -bounded wait-free* if it is wait-free and every derived object  $\mathcal{O}$  has the following property: For all executions of  $(P_1, P_2, \dots, P_N; \mathcal{O})$  and for all  $P_i$   $1 \leq i \leq N$ , between an invocation on  $\mathcal{O}$  by  $P_i$  and its matching response,  $P_i$  has no more than  $k$  invocations on all base objects of  $\mathcal{O}$  put together.

Intuitively, in a  $k$ -bounded wait-free implementation, a process completes its operation on a derived object in no more than  $k$  steps.

In this paper, we are primarily interested in wait-free implementations. From now on, we will therefore write “implementation” and “ $k$ -bounded implementation” as shorthand for “wait-free implementation” and “ $k$ -bounded wait-free implementation”, respectively.

### 3 Failure modes

An object is only an abstraction with a multitude of possible implementations. For instance, it may be built as a hardware module in a tightly coupled multi-processor system, or as a server machine in a message passing distributed system. Whatever the implementation, the reality is that hardware components sometimes fail and, when this happens, the object fails to provide the intended abstraction.

Object failures lead to undesirable system behavior. Therefore, it is important to implement derived objects that behave correctly even if some of the base objects of the implementation fail. The complexity of such a fault-tolerant implementation depends on the *failure mode*, *i.e.*, the manner in which a failed object departs from correct behavior. In this section, we define a spectrum of failure modes that fall into two broad classes: *responsive* and *non-responsive*.

As we will see, a failed object  $\mathcal{O}$  may sometimes return a distinguished response  $\perp$ . If a process  $P$  receives  $\perp$  from  $\mathcal{O}$ , it can immediately infer that  $\mathcal{O}$  is faulty. Thus, it is reasonable to assume that  $P$  does not invoke operations on  $\mathcal{O}$  thereafter. We restrict our attention to executions in which this assumption holds.

#### 3.1 Responsive failure modes

An object experiencing a responsive failure responds to every invocation, even though the response may be incorrect. Thus, responsive failure modes share the property that objects remain wait-free even if they fail. We describe below three increasingly severe responsive failure modes.

##### 3.1.1 Crash

crash is the most benign of all failure modes, responsive or non-responsive. Informally, an object that fails by crash behaves correctly until it fails and, once it fails, it returns a distinguished response  $\perp$  to every invocation. This failure mode is based on the premise that an object detects when it becomes faulty and responds with  $\perp$  thereafter.

Let  $\mathcal{O}$  be an object of type  $T = (OP, RES, G, \tau)$ , initialized to state  $s$  of  $T$ . *Object  $\mathcal{O}$  fails in an execution  $E$  by crash* if it is not well-behaved in  $E$ , but satisfies the following properties:

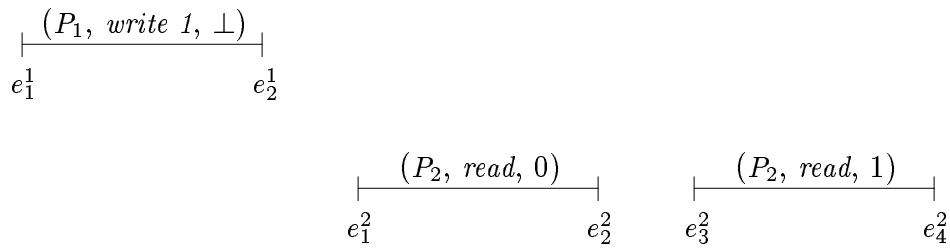
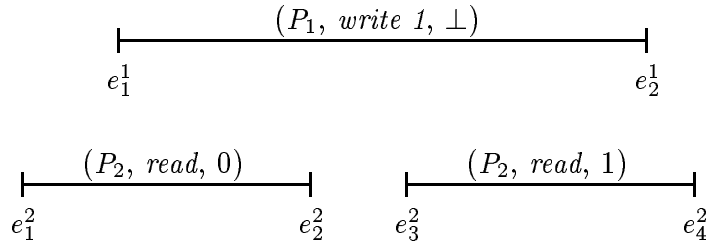
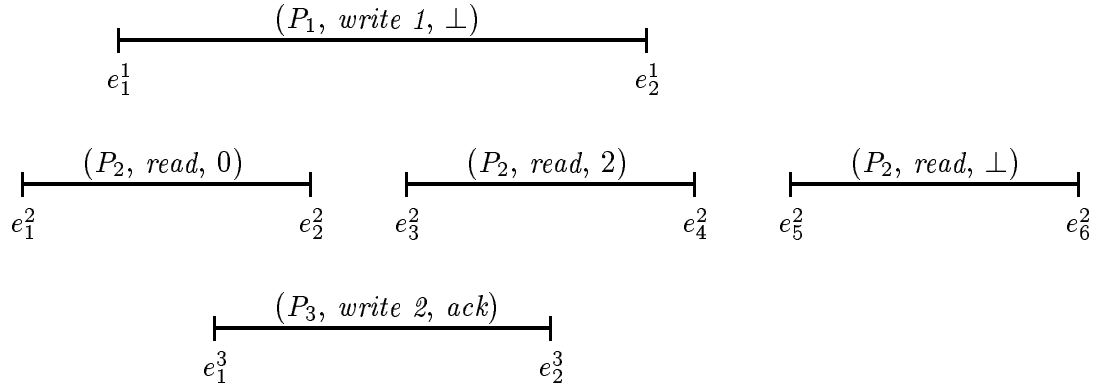
1.  $\mathcal{O}$  is wait-free in  $E$ .
2. Every response from  $\mathcal{O}$  in  $E$  either belongs to  $RES$  or is  $\perp$  (where  $\perp$  is a distinguished value not in  $RES$ ). An operation that returns  $\perp$  is an *aborted* operation.
3. Let  $\mathcal{H}$  be the history of  $\mathcal{O}$  in  $E$ , and let  $op$  and  $op'$  be two completed operations in  $\mathcal{H}$ . If  $op$  precedes  $op'$  and  $op$  is an aborted operation, then  $op'$  is also an aborted operation.
4. Let  $\mathcal{H}'$  be the history obtained by removing all aborted operations in  $\mathcal{H}$ . Then,  $\tau(\mathcal{H}')$  is linearizable with respect to  $(T, s)$ .

Property 3 is the “once  $\perp$ , everafter  $\perp$ ” property of crash. Property 4 captures the notion that  $\mathcal{O}$  behaves correctly until it fails and that aborted operations do not take effect. Let us consider some examples. Let  $\mathcal{R}$  be an object of type **register**, initialized to 0.

- Consider the history  $\mathcal{H}$  of  $\mathcal{R}$  in Figure 3. (In the figure, a line segment represents the duration of an operation, from invocation to response. A triple  $(P_i, op, res)$  over the line segment denotes that  $P_i$  is the invoking process,  $op$  is the operation invoked, and  $res$  is the response from  $\mathcal{R}$ .) The failure of  $\mathcal{R}$  is by crash, as verified below. Removing aborted operations in  $\mathcal{H}$  results in  $\mathcal{H}' = e_1^2, e_1^3, e_2^2, e_3^2, e_2^3, e_4^2$ . (Event  $e_i^j$  denotes the  $i^{th}$  event of process  $P_j$ .) Clearly,  $\mathcal{H}'$  is linearizable with respect to **(register, 0)**:  $e_1^2, e_2^2, e_1^3, e_2^3, e_3^2, e_4^2$  is a linearization. The history transformation function  $\tau$  for **register** is the identity function. Thus,  $\tau(\mathcal{H}') = \mathcal{H}'$ , and is linearizable with respect to **(register, 0)**. Thus, Property 4 holds in  $\mathcal{H}$ . Other properties also hold and are trivial to verify.
- Consider the history  $\mathcal{H}$  of  $\mathcal{R}$  in Figure 4. Now  $\mathcal{H}' = e_1^2, e_2^2, e_3^2, e_4^2$ . Clearly,  $\mathcal{H}'$  (and hence,  $\tau(\mathcal{H}')$ ) is not linearizable with respect to **(register, 0)**. Thus, the failure of  $\mathcal{R}$  is not by crash.

### 3.1.2 Omission

We begin with the motivation for the omission failure mode. Consider an implementation  $\mathcal{I}$ , and a derived object  $\mathcal{O}$  of  $\mathcal{I}$ . Even if the base objects of  $\mathcal{O}$  may only fail by crash,  $\mathcal{O}$  itself may experience a more severe failure than crash. To see this, suppose that a base object  $o$  of  $\mathcal{O}$  fails by crash. Consider a process  $P$  that invokes an operation  $op$  on  $\mathcal{O}$  and executes **Apply**( $P, op, \mathcal{O}$ ). If, during the execution of **Apply**( $P, op, \mathcal{O}$ ),  $P$  accesses  $o$ ,  $o$  returns  $\perp$  to  $P$ . This may cause **Apply**( $P, op, \mathcal{O}$ ) to terminate and also return  $\perp$ . Strictly after this occurs, suppose that another process  $Q$  invokes some operation  $op'$  on  $\mathcal{O}$ , and that **Apply**( $Q, op', \mathcal{O}$ ) is *not* required to access  $o$ . Then, while executing **Apply**( $Q, op', \mathcal{O}$ ),  $Q$  does





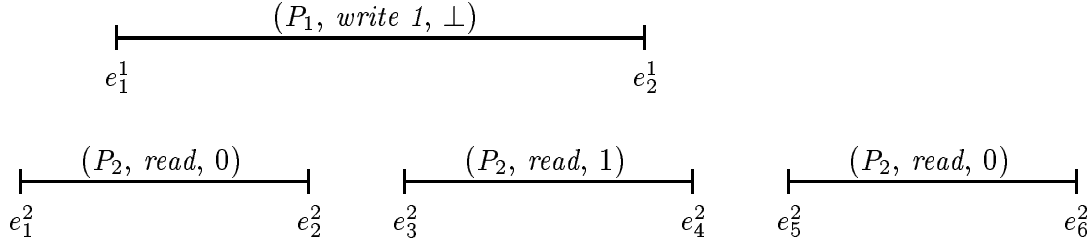


Figure 6: Safe register  $\mathcal{R}$ , initialized to 0, fails by omission

not notice the failure of  $o$ . So  $\text{Apply}(Q, op', \mathcal{O})$  terminates “normally” and returns a non- $\perp$  response. Thus,  $\mathcal{O}$ ’s behavior violates the “once  $\perp$ , everafter  $\perp$ ” property:  $\mathcal{O}$  returned  $\perp$  to  $P$ ’s operation and a non- $\perp$  response to a strictly later operation by  $Q$ . We conclude that  $\mathcal{O}$ ’s failure is more severe than crash. Does this mean that  $\mathcal{O}$ ’s failure is arbitrary? We now argue that this is not the case.

Recall that after  $P$  receives  $\perp$ , we assume that  $P$  refrains from accessing  $\mathcal{O}$  again. Thus, to  $Q$ , the above scenario is indistinguishable from one in which  $P$  had crashed in the middle of the procedure  $\text{Apply}(P, op, \mathcal{O})$ , while accessing  $o$ . Since the implementation  $\mathcal{I}$  (from which  $\mathcal{O}$  is derived) is wait-free,  $\mathcal{O}$  tolerates the apparent crash of process  $P$ . Thus,  $\mathcal{O}$ ’s response to  $Q$  must be correct. We conclude that the failure of  $\mathcal{O}$  is more severe than crash, but is not completely arbitrary. Our model of omission, formally defined below, captures this type of failure.

Let  $\mathcal{O}$  be an object of type  $T = (OP, RES, G, \tau)$ , initialized to state  $s$  of  $T$ . *Object  $\mathcal{O}$  fails in an execution  $E$  by omission* if it is not well-behaved in  $E$ , but satisfies the following properties:

1.  $\mathcal{O}$  is wait-free in  $E$ .
2. Every response from  $\mathcal{O}$  in  $E$  either belongs to  $RES$  or is  $\perp$ .
3. Let  $\mathcal{H}$  be the history of  $\mathcal{O}$  in  $E$ . Let  $\mathcal{H}'$  be the history obtained by removing the response events associated with the aborted operations in  $\mathcal{H}$ . Then,  $\tau(\mathcal{H}')$  is linearizable with respect to  $(T, s)$ .

Suppose that an operation by process  $P$  receives the response  $\perp$  from  $\mathcal{O}$ . Property 3 states that this aborted operation must appear like an incomplete operation to all processes other than  $P$ .

Notice the subtle difference in the way we obtain  $\mathcal{H}'$  from  $\mathcal{H}$  for crash and omission. For crash, both invocation and response events associated with aborted operations are removed to obtain  $\mathcal{H}'$ . For omission, only the response events associated with aborted operations are removed. Let us consider some examples.

- Let  $\mathcal{R}$  be an object of type **register**, initialized to 0. Consider the history  $\mathcal{H}$  of  $\mathcal{R}$  in Figure 4. The failure of  $\mathcal{R}$  is by omission, as verified below. Removing the

response events of aborted operations in  $\mathcal{H}$  results in  $\mathcal{H}' = e_1^2, e_1^1, e_2^2, e_3^2, e_4^2$ . ( $e_2^1$  is removed from  $\mathcal{H}$  to obtain  $\mathcal{H}'$ .) The write operation by  $P_1$  becomes an incomplete operation in  $\mathcal{H}'$ .  $\mathcal{H}'$  is linearizable with respect to  $(\text{register}, 0)$ :  $e_1^2, e_2^2, e_1^1, e, e_3^2, e_4^2$  is a linearization, where  $e$  is a response event returning *ack*. Thus, in the linearization of  $\mathcal{H}'$ , the first read by  $P_2$  takes effect first, then the write by  $P_1$  (which is incomplete in  $\mathcal{H}'$ ) takes effect, and then the second read by  $P_2$  takes effect. Since  $\tau$  is the identity for **register**, it follows that  $\tau(\mathcal{H}')$  is linearizable with respect to  $(\text{register}, 0)$ . Thus, Property 3 of omission holds in  $\mathcal{H}$ . Other properties also hold and are trivial to verify.

- Let  $\mathcal{R}$  be an object of type **register**, initialized to 0. Consider the history  $\mathcal{H}$  of  $\mathcal{R}$  in Figure 5. The failure of  $\mathcal{R}$  is by omission, as verified below. Removing the response event  $e_2^1$  of the aborted operation results in  $\mathcal{H}' = e_1^1, e_1^2, e_2^2, e_3^2, e_4^2$ .  $\mathcal{H}'$  (and hence,  $\tau(\mathcal{H}')$ ) is linearizable with respect to  $(\text{register}, 0)$ :  $e_1^2, e_2^2, e_1^1, e, e_3^2, e_4^2$  is a linearization, where  $e$  is a response event returning *ack*. Thus, in the linearization of  $\mathcal{H}'$ , the first read by  $P_2$  takes effect first, then the write by  $P_1$  (which is aborted in  $\mathcal{H}$  and incomplete in  $\mathcal{H}'$ ) takes effect, and then the second read by  $P_2$  takes effect.

This example shows that an aborted operation may take effect a long time after it completed.

- Let  $\mathcal{R}$  be an object of type **register**, initialized to 0. Consider the history  $\mathcal{H}$  of  $\mathcal{R}$  in Figure 6. Now,  $\mathcal{H}' = e_1^2, e_1^1, e_2^2, e_3^2, e_4^2, e_5^2, e_6^2$ . It is easy to verify that  $\mathcal{H}'$  (and hence,  $\tau(\mathcal{H}')$ ) is not linearizable with respect to  $(\text{register}, 0)$ . Thus, the failure of  $\mathcal{R}$  is not by omission.
- Same as the above example, but suppose that  $\mathcal{R}$  is of type **safe register**. Recall that the function  $\tau$  for **safe register** removes all read operations that overlap with a write. Thus,  $\tau(\mathcal{H}') = e_1^1$ , and is obviously linearizable with respect to  $(\text{safe register}, 0)$ . (The empty sequence is a linearization of  $\tau(\mathcal{H}')$ .) Thus, Property 3 of omission holds. Other properties also hold and are trivial to verify. Thus,  $\mathcal{R}$  fails by omission in  $\mathcal{H}$ .

### 3.1.3 Arbitrary

An object  $\mathcal{O}$  fails in an execution  $E$  by the *arbitrary failure mode* if it is not well-behaved in  $E$ , but is wait-free in  $E$ . Informally,  $\mathcal{O}$  responds to every invocation in  $E$ , but the responses may be arbitrary.

## 3.2 Non-responsive failure modes

With responsive failure modes, a faulty object remains wait-free. Non-responsive failure modes do not have this property.

### 3.2.1 NR-crash

NR-crash is the most benign of all non-responsive failure modes. Informally, an object that fails by NR-crash behaves correctly until it fails (Property 1 below) and, once it fails, it never responds to any invocation (Property 2 below).

An object  $\mathcal{O}$  fails in an execution  $E$  by NR-crash if it is not wait-free in  $E$ , but satisfies the following properties:

1.  $\mathcal{O}$  is well-behaved in  $E$ .
2. The total number of non- $\perp$  responses from  $\mathcal{O}$  in  $E$  is finite.

### 3.2.2 NR-omission

An object  $\mathcal{O}$  fails in an execution  $E$  by NR-omission if it is not wait-free in  $E$ , but is well-behaved in  $E$ .

NR-omission is more severe than NR-crash. In particular, an object that fails by NR-omission does not necessarily satisfy Property 2 of NR-crash. Thus, the object may not respond to invocations from some processes and always respond to invocations from others.

### 3.2.3 NR-arbitrary

An object  $\mathcal{O}$  fails in an execution  $E$  by NR-arbitrary if it fails in  $E$ .

Thus, the behavior of an object that experiences an NR-arbitrary failure is completely unrestricted. Such an object may not respond to an invocation; even if it does, the response may be arbitrary.

## 4 Fault-tolerance and graceful degradation — definitions and properties

In the following, let  $\mathcal{I}$  be an implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$  for processes  $P_1, P_2, \dots, P_N$ , where  $\mathcal{L} = (T_1, T_2, \dots)$  and  $\Sigma = (s_1, s_2, \dots)$ .

We say that  $\mathcal{I}$  is *t-tolerant for failure mode  $\mathcal{F}$*  if it satisfies the following:

Let  $O_1, O_2, \dots$  be distinct objects of types  $T_1, T_2, \dots$ , initialized to states  $s_1, s_2, \dots$ , respectively. Then,  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots)$  is an object of type  $T$ , initialized to state  $s$ , with the following property: for every execution  $E$  of the concurrent system  $(P_1, P_2, \dots, P_N; \mathcal{O})$ , if at most  $t$  objects among  $O_1, O_2, \dots$  fail, and they fail by  $\mathcal{F}$ , then  $\mathcal{O}$  is correct.

We say that  $\mathcal{I}$  is *gracefully degrading for failure mode  $\mathcal{F}$*  if it satisfies the following:

Let  $O_1, O_2, \dots$  be distinct objects of types  $T_1, T_2, \dots$ , initialized to states  $s_1, s_2, \dots$ , respectively. Then,  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots)$  is an object of type  $T$ , initialized to state  $s$ , with the following property: for every execution  $E$  of the concurrent system  $(P_1, P_2, \dots, P_N; \mathcal{O})$ , if all faulty objects among  $O_1, O_2, \dots$  fail by  $\mathcal{F}$ , then either  $\mathcal{O}$  is correct or  $\mathcal{O}$  fails by  $\mathcal{F}$ .

Let  $\mathcal{O}$  be a derived object of an implementation that is both  $t$ -tolerant and gracefully degrading for failure mode  $\mathcal{F}$ . The above definitions imply that: (i) if at most  $t$  base objects of  $\mathcal{O}$  fail, and they fail by  $\mathcal{F}$ , then  $\mathcal{O}$  does not fail, and (ii) if more than  $t$  base objects of  $\mathcal{O}$  fail, and they fail by  $\mathcal{F}$ , then  $\mathcal{O}$  may fail, but it does not experience a more severe failure than  $\mathcal{F}$ . Property (i) is guaranteed by  $t$ -tolerance and property (ii) by graceful degradation.

#### 4.1 Composing fault-tolerant implementations

Gracefully degrading implementations can be composed as stated by the following lemma. Given a list  $L$  of integers and an integer  $n$ , let  $\text{MinSum}(n, L)$  be the sum of the  $n$  smallest integers in  $L$ . If  $L_1$  and  $L_2$  are lists, let  $L_1 \cdot L_2$  denote the concatenation of  $L_1$  and  $L_2$ .

In the lemma below and in the rest of this paper, if we do not specify the number of processes for which an implementation is intended, it should be assumed that the implementation is for  $N$  processes, where  $N$  is arbitrary. Also, we say that a type  $T$  has a  $t$ -tolerant gracefully degrading implementation if, for all states  $s$  of  $T$ , there is a  $t$ -tolerant gracefully degrading implementation of  $(T, s)$ . The lemma is illustrated in Figure 7.

**Lemma 4.1 (Compositional Lemma)** *Suppose that  $T$  has a  $t$ -tolerant implementation from  $\mathcal{L}$  for failure mode  $\mathcal{F}$ , where  $\mathcal{L} = (T_1, T_2, \dots, T_n)$  is a list of types. Furthermore, suppose that each  $T_i$  has a  $t_i$ -tolerant gracefully degrading implementation from  $\mathcal{L}_i$  for failure mode  $\mathcal{F}$ . Then we have:*

1.  *$T$  has a  $t'$ -tolerant implementation from  $\mathcal{L}'$  for failure mode  $\mathcal{F}$ , where  $\mathcal{L}' = \mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \dots \cdot \mathcal{L}_n$  and  $t' = \text{MinSum}(t+1, \langle t_1+1, t_2+1, \dots, t_n+1 \rangle) - 1$ .*
2. *If the  $t$ -tolerant implementation of  $T$  from  $\mathcal{L}$  is gracefully degrading for  $\mathcal{F}$ , then  $T$  has a  $t'$ -tolerant gracefully degrading implementation from  $\mathcal{L}'$  for failure mode  $\mathcal{F}$ .*

*Proof Sketch* Let  $s$  be any state of  $T$ . By the statement of the lemma,  $(T, s)$  has a  $t$ -tolerant gracefully degrading implementation  $\mathcal{I}$  from  $(\mathcal{L}, \Sigma)$  for failure mode  $\mathcal{F}$ , for some  $\Sigma = (s_1, s_2, \dots, s_n)$  such that  $s_i$  is a state of  $T_i$ . For all  $i$ , let  $\mathcal{L}_i = (T_{i1}, T_{i2}, \dots, T_{ij_i})$ . By the statement of the lemma, each  $(T_i, s_i)$  has a  $t_i$ -tolerant gracefully degrading implementation  $\mathcal{I}_i$  from  $(\mathcal{L}_i, \Sigma_i)$  for failure mode  $\mathcal{F}$ , for some  $\Sigma_i = (s_{i1}, s_{i2}, \dots, s_{ij_i})$  such that  $s_{ik}$  is a state of  $T_{ik}$ .

Let  $o_{11}, \dots, o_{1j_1}, \dots, o_{n1}, \dots, o_{nj_n}$  be objects of types  $T_{11}, \dots, T_{1j_1}, \dots, T_{n1}, \dots, T_{nj_n}$ , initialized to states  $s_{11}, \dots, s_{1j_1}, \dots, s_{n1}, \dots, s_{nj_n}$ , respectively. Define an implementation  $\mathcal{I}'$  as follows:  $\mathcal{I}'(o_{11}, \dots, o_{1j_1}, \dots, o_{n1}, \dots, o_{nj_n}) = \mathcal{I}(O_1, \dots, O_n)$ , where  $O_i = \mathcal{I}_i(o_{i1}, o_{i2}, \dots, o_{ij_i})$ .

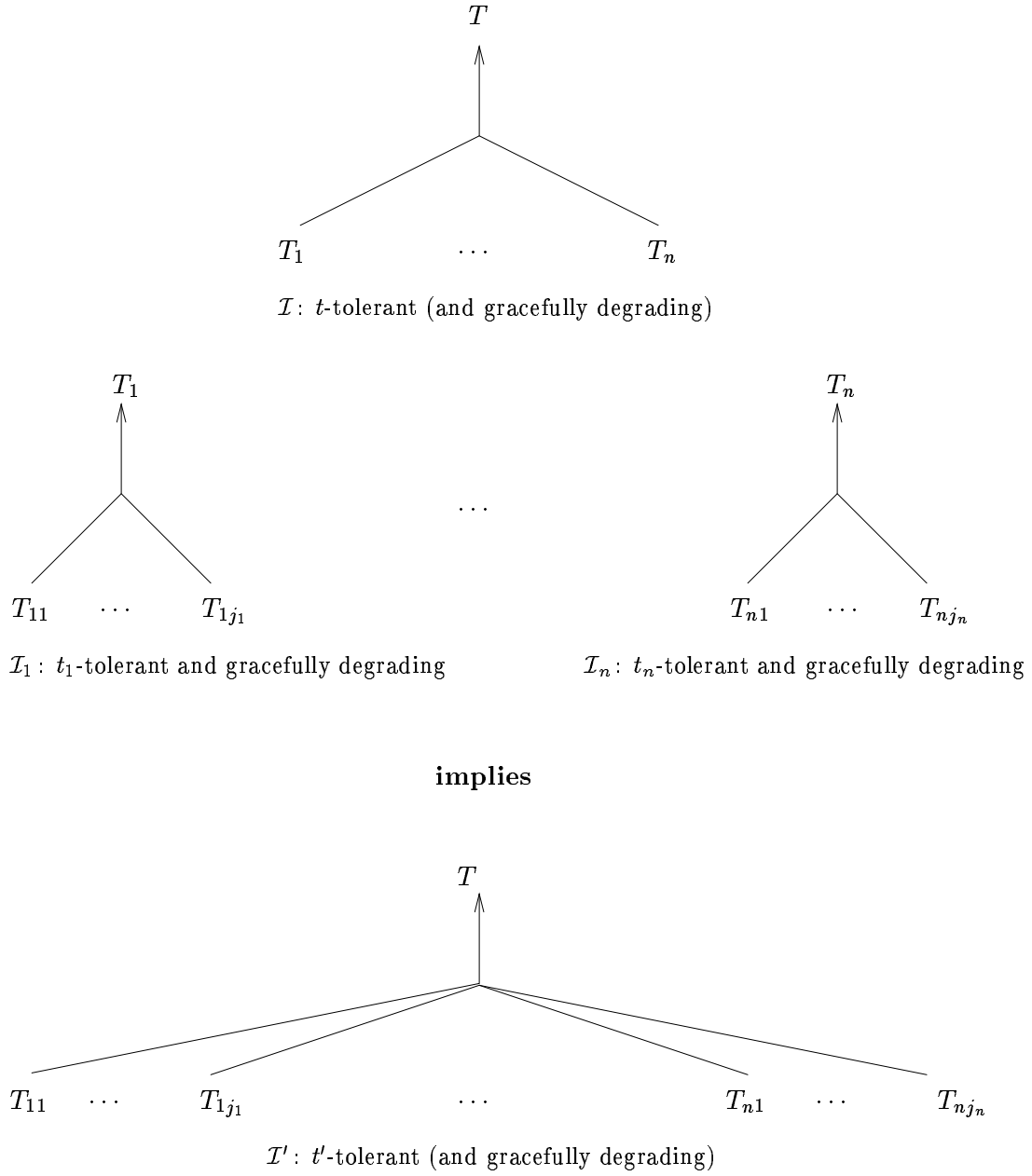


Figure 7: Illustration of the Compositional Lemma

Assume that each  $o_{kl}$ , if it fails, only fails by  $\mathcal{F}$ . Since  $\mathcal{I}_i$  is  $t_i$ -tolerant,  $O_i$  fails only if at least  $t_i + 1$  objects among  $o_{i1}, \dots, o_{ij_i}$  fail. Furthermore, since  $\mathcal{I}_i$  is gracefully degrading,  $O_i$  can only fail by  $\mathcal{F}$ , no matter how many base objects of  $O_i$  fail. From this and the fact that  $\mathcal{I}$  is  $t$ -tolerant for  $\mathcal{F}$ , it follows that  $\mathcal{I}(O_1, \dots, O_n)$  fails only if at least  $t + 1$  objects among  $O_1, \dots, O_n$  fail. Thus, for  $\mathcal{I}(O_1, \dots, O_n)$  to fail, at least  $\text{MinSum}(t+1, \langle t_1+1, t_2+1, \dots, t_n+1 \rangle) = t'+1$  objects among  $o_{11}, \dots, o_{1j_1}, \dots, o_{n1}, \dots, o_{nj_n}$  must fail. In other words,  $\mathcal{I}'$  is a  $t'$ -tolerant implementation of  $(T, s)$  from  $(\mathcal{L}', \Sigma')$ , where  $\Sigma' = \Sigma_1 \cdot \Sigma_2 \cdot \dots \cdot \Sigma_n$ . This completes the proof of the first part of the lemma.

Assume that the implementation  $\mathcal{I}$  is gracefully degrading for  $\mathcal{F}$ . Thus, if  $O_1, \dots, O_n$  (which are the base objects of  $\mathcal{O}$ ) only fail by  $\mathcal{F}$ , then  $\mathcal{O}$ , if it fails, only fails by  $\mathcal{F}$ . We have already argued that if objects  $o_{11}, \dots, o_{1j_1}, \dots, o_{n1}, \dots, o_{nj_n}$  only fail by  $\mathcal{F}$ , then each  $O_i$ , if it fails, only fails by  $\mathcal{F}$ . We conclude that if objects  $o_{11}, \dots, o_{nj_n}$  only fail by  $\mathcal{F}$ , then  $\mathcal{O}$ , if it fails, only fails by  $\mathcal{F}$ . Thus,  $\mathcal{I}'$  is gracefully degrading for  $\mathcal{F}$ . This completes the proof of the second part of the lemma.  $\square$

We now state a special case of the compositional lemma, obtained by setting  $t = 0$  and  $\forall 1 \leq i \leq n : t_i = t$ . This lemma is used frequently in later sections.

**Corollary 4.1** *Suppose that  $T$  has a (0-tolerant) implementation from  $(T_1, T_2, \dots, T_n)$ . Furthermore, suppose that each  $T_i$  has a  $t$ -tolerant gracefully degrading implementation from  $\mathcal{L}_i$  for failure mode  $\mathcal{F}$ , where  $\mathcal{L}_i$  is some list of types. Then we have:*

1.  *$T$  has a  $t$ -tolerant implementation from  $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \dots \cdot \mathcal{L}_n$  for failure mode  $\mathcal{F}$ .<sup>4</sup>*
2. *If the (0-tolerant) implementation of  $T$  from  $(T_1, T_2, \dots, T_n)$  is gracefully degrading for  $\mathcal{F}$ , then  $T$  has a  $t$ -tolerant gracefully degrading implementation from  $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \dots \cdot \mathcal{L}_n$  for failure mode  $\mathcal{F}$ .*

The compositional lemma can also be used to enhance the fault-tolerance of a self-implementation. This is the substance of the following corollary, obtained by setting  $T_i = T$ ,  $\mathcal{L}_i = \mathcal{L}$ , and  $t_i = t$  in Lemma 4.1. Below, we say that  $T$  has an implementation of resource complexity  $n$  if, for all states  $s$  of  $T$ ,  $(T, s)$  has an implementation of resource complexity  $n$ .

**Corollary 4.2** *If  $T$  has a  $t$ -tolerant gracefully degrading self-implementation  $\mathcal{I}$  of resource complexity  $n$  for failure mode  $\mathcal{F}$ , then  $T$  has a  $(t^2 + 2t)$ -tolerant gracefully degrading self-implementation  $\mathcal{I}'$  of resource complexity  $n^2$  for  $\mathcal{F}$ .*

Recursive application of the above corollary boosts the fault-tolerance of self-implementations.

**Corollary 4.3 (Booster Lemma)** *If  $T$  has a 1-tolerant gracefully degrading self-implementation of resource complexity  $k$  for failure mode  $\mathcal{F}$ , then  $T$  has a  $t$ -tolerant gracefully degrading self-implementation of resource complexity  $O(t^{\log_2 k})$  for  $\mathcal{F}$ .*

---

<sup>4</sup>This part holds even if the implementation of each  $T_i$  is  $t$ -tolerant, but not gracefully degrading.

## 4.2 Graceful degradation for arbitrary failures

We show that if  $T$  has a  $t$ -tolerant  $k$ -bounded implementation, then  $T$  has a  $t$ -tolerant gracefully degrading  $k$ -bounded implementation for arbitrary failures. Thus, if we know how to obtain a bounded implementation, graceful degradation for arbitrary failures comes automatically and at no extra cost.

Observe that if an implementation guarantees that the derived object is wait-free whenever the base objects are wait-free, the implementation is gracefully degrading for arbitrary failures. The lemma below is based on this observation.

**Lemma 4.2** *If  $T$  has a  $t$ -tolerant  $k$ -bounded implementation from  $\mathcal{L}$  for arbitrary failures, then  $T$  has a  $t$ -tolerant gracefully degrading  $k$ -bounded implementation from  $\mathcal{L}$  for arbitrary failures.*

*Proof Sketch* Let  $s$  be any state of  $T$ . By the statement of the lemma,  $(T, s)$  has a  $t$ -tolerant  $k$ -bounded implementation  $\mathcal{I}$  from  $(\mathcal{L}, \Sigma)$ , for some sequence  $\Sigma$  of states. Define the implementation  $\mathcal{I}'$  as follows. In  $\mathcal{I}'$ , a process applies an operation  $op$  on the derived object  $\mathcal{O}$  by first setting a local counter *count* to 0, and then proceeding as in the implementation  $\mathcal{I}$ . As the process executes the steps of  $\mathcal{I}$ , it increments *count* each time it applies an operation on a base object of  $\mathcal{O}$ . If *count* reaches  $k$  and the implementation  $\mathcal{I}$  has not yet returned a response, the process deduces that more than  $t$  base objects have failed (this deduction is sound since  $\mathcal{I}$  is a  $t$ -tolerant  $k$ -bounded implementation), and returns an arbitrary value as the response from  $\mathcal{O}$  to its operation  $op$ .

Since  $\mathcal{I}$  is a correct  $t$ -tolerant implementation, it follows that  $\mathcal{I}'$  is also a correct  $t$ -tolerant implementation. Clearly,  $\mathcal{I}'$  has the property that, if all base objects are wait-free, the derived object is also wait-free. Hence  $\mathcal{I}'$  is gracefully degrading for arbitrary failures. We conclude that  $\mathcal{I}'$  is a  $t$ -tolerant gracefully degrading  $k$ -bounded implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$  for arbitrary failures. Hence the lemma.  $\square$

## 5 Tolerating responsive failures

Herlihy [Her91b] and Plotkin [Plot89] showed that one can implement a (wait-free) object of any type using only consensus and register objects. Therefore, if **consensus** and **register** have  $t$ -tolerant implementations, then every type has a  $t$ -tolerant implementation. Hence we focus on fault-tolerant implementations of **consensus** and **register**.

### 5.1 Fault-tolerant implementation of consensus

In this section, we present a self-implementation of **consensus** that is  $t$ -tolerant for both crash and omission failures. This implementation requires  $t+1$  base consensus objects and is thus resource optimal. Following that, we present an efficient  $t$ -tolerant self-implementation of **consensus** for arbitrary failures.

Achieving consensus among processes, some of which may fail, is a widely studied problem in the literature ([PSL80, LSP82, FLM86, Coa87, ST87, BGP89, DRS90, CW92], to cite a few). The reader may notice some similarity between this problem and the one studied here, namely, obtaining  $t$ -tolerant implementations of **consensus**. We therefore begin by contrasting these two problems. The existing solutions to the consensus problem are for *synchronous message passing* systems. In such systems, processes communicate by passing messages to each other; furthermore, bounds on message delays and bounds on the relative speeds of processes are assumed to be known. In contrast, we study the consensus problem for *asynchronous shared-memory* systems. The asynchrony in the system rules out the common paradigm in which a correct process “waits” until every process has either crashed or taken a step. We require solutions to be wait-free: a process should be able to decide regardless of how fast or slow the other processes are. Also, in synchronous message passing systems, solutions to the consensus problem, where processes are subject to arbitrary failures, assume that fewer than a third of the processes fail (without this assumption, the problem cannot be solved). In contrast, our solutions tolerate the crash failure of any number of processes and, in addition, the arbitrary failure of up to  $t$  shared objects. As a result of these differences, the problem of  $t$ -tolerant implementation of **consensus** does not reduce to any previous problem considered in the literature.

The “State Machine” approach [Lam78, Sch90] of replicating objects, applying an operation to all objects, and returning the majority response is not useful in deriving  $t$ -tolerant implementations of **consensus**. For example, consider the following implementation which uses  $2t + 1$  base consensus objects  $(O_1, O_2, \dots, O_{2t+1})$  to tolerate the crash failure of any  $t$  of them. A process  $p$  proposes a value  $v_p$  to the derived consensus object  $\mathcal{O}$  by proposing  $v_p$  to each of  $O_1, O_2, \dots, O_{2t+1}$ . At the end of this,  $p$  will have obtained the response 0 from, say,  $n_0$  base objects, the response 1 from  $n_1$  base objects, and the response  $\perp$  from  $2t + 1 - n_0 - n_1$  base objects.  $p$  returns 0 (as the response of  $\mathcal{O}$ ) if  $n_0 > n_1$ . Otherwise, it returns 1. Unfortunately, this implementation is not  $t$ -tolerant for crash. The following is a counterexample.

Let  $t = 2$ . Suppose that processes  $p$  and  $q$  wish to propose 0 and 1, respectively, to the derived consensus object  $\mathcal{O}$ . Suppose that the steps of  $p$  and  $q$  interleave in the order specified below. Process  $p$  proposes 0 to  $O_1, O_2$ , and  $O_3$ , and all three return 0 to  $p$ . Objects  $O_1$  and  $O_2$  then fail by crash. Process  $q$  proposes 1 to all of  $O_1, O_2, \dots, O_5$ ; Objects  $O_1$  and  $O_2$  return  $\perp$  to  $q$ ,  $O_3$  returns 0, and  $O_4$  and  $O_5$  return 1. Process  $p$  resumes and proposes 0 to  $O_4$  and  $O_5$ , and both these objects return 1 to  $p$ . Thus,  $p$  obtained three 0’s and two 1’s, and  $q$  obtained two 1’s and one 0. By the above implementation,  $p$  returns 0 and  $q$  returns 1. This implies that the derived object  $\mathcal{O}$  did not satisfy the agreement property despite the fact that only two base objects failed by crash. Thus, the implementation is not 2-tolerant for crash.

In the following, we first state the properties of a consensus object and then present the implementations. We use the properties in proving our implementations correct.



### 5.1.1 Properties of consensus

**consensus** supports two operations, *propose 0* and *propose 1*, and has the sequential specification given in Figure 1. We will refer to the states  $S$ ,  $S_0$ , and  $S_1$  of **consensus** as the *uncommitted*, *0-committed*, and *1-committed* states, respectively. In this section, we state the properties that a consensus object satisfies in executions. To state these properties, we need the following definitions. Let  $\mathcal{O}$  be an object of type **consensus** and let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$ .

- Object  $\mathcal{O}$  satisfies *integrity* in  $E$  if and only if every response from  $\mathcal{O}$  in  $E$  is either 0 or 1.
- Object  $\mathcal{O}$  satisfies *weak integrity* in  $E$  if and only if every response from  $\mathcal{O}$  in  $E$  is either 0, 1, or  $\perp$ .
- Object  $\mathcal{O}$  satisfies *validity* in  $E$  if and only if the following holds in  $E$ . If there is a response of  $v$  from  $\mathcal{O}$  and  $v \in \{0, 1\}$ , then there is an invocation of *propose v* on  $\mathcal{O}$  preceding this response.
- Object  $\mathcal{O}$  satisfies *agreement* in  $E$  if and only if the following holds in  $E$ . If  $\mathcal{O}$  returns  $v_1, v_2$  to two invocations, and  $v_1, v_2 \in \{0, 1\}$ , then  $v_1 = v_2$ . (By this definition, if  $\mathcal{O}$  returns 0 to some processes and  $\perp$  to all others, it still satisfies agreement.)

The propositions below follow easily from the sequential specification of **consensus** and the definitions of linearizability and omission failures.

**Proposition 5.1** *Let  $\mathcal{O}$  be an object of type **consensus**, initialized to the uncommitted state. Let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$ . Object  $\mathcal{O}$  is correct in  $E$  if and only if it is wait-free in  $E$  and satisfies integrity, validity, and agreement in  $E$ .*

**Proposition 5.2** *Let  $\mathcal{O}$  be an object of type **consensus**, initialized to the uncommitted state. Let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$  in which  $\mathcal{O}$  fails. Object  $\mathcal{O}$  fails by omission in  $E$  if and only if it is wait-free in  $E$  and satisfies weak integrity, validity, and agreement in  $E$ .*

In the following sections, we present several fault-tolerant implementations of **consensus**. In describing these implementations, we write  $loc := \text{Propose}(P, v, \mathcal{O})$ <sup>5</sup> to denote that process  $P$  invokes *propose v* on  $\mathcal{O}$  and stores the response in its local variable  $loc$ .

Implementing a consensus object  $\mathcal{O}$  initialized to the 0-committed (respectively, 1-committed) state is trivial:  $\text{Propose}(P, v, \mathcal{O})$  simply returns 0 (respectively, 1). Thus, the only interesting case is to implement a consensus object initialized to the uncommitted state. Consequently, throughout this paper, we use the phrase “ $\mathcal{I}$  is an implementation of **consensus**” to mean “ $\mathcal{I}$  is an implementation of (**consensus**, uncommitted state)”.

---

<sup>5</sup> Throughout this paper, we write  $\text{Propose}$  (with upper case “P”) if the operation is on a derived object, and  $\text{propose}$  (with lower case “p”) if it is on a base object.

### 5.1.2 Tolerating crash and omission failures

We present a  $t$ -tolerant self-implementation of **consensus** for omission failures. The resource complexity is  $t + 1$  and is therefore optimal. Since omission failures are strictly more severe than crash, this self-implementation is also correct for crash.

Figure 8 presents a  $t$ -tolerant self-implementation of **consensus** for omission failures. (In all our algorithms, we use indentation to convey the scope of an **if** statement or a **for** statement.) This implementation uses  $t + 1$  base objects. A process  $p$  proposes to the derived object  $\mathcal{O}$  by accessing each of  $O_1, O_2, \dots, O_{t+1}$ , in that order. At any point in the algorithm,  $p$  holds an estimate of the eventual return value in  $estimate_p$ . When  $p$  proposes its current estimate to a base object  $O_k$ , if  $O_k$  returns a non- $\perp$  response  $w$  different from  $p$ 's current estimate,  $p$  changes its estimate to  $w$ . After accessing all  $t + 1$  base objects,  $p$  returns its estimate as the response of the derived object  $\mathcal{O}$ .

---

$O_1, O_2, \dots, O_{t+1}$  : consensus objects, initialized to the uncommitted state

```

Procedure Propose( $p, v_p, \mathcal{O}$ )      /*  $v_p \in \{0, 1\}$  */
     $estimate_p, w, k$  : integer local to  $p$ 
begin
     $estimate_p := v_p$ 
    for  $k := 1$  to  $t + 1$ 
         $w := \text{propose}(p, estimate_p, O_k)$ 
        if  $w \neq \perp$  then  $estimate_p := w$ 
    return( $estimate_p$ )
end

```

Figure 8:  $t$ -tolerant self-implementation of **consensus** for omission

---

**Theorem 5.1** *Figure 8 presents a  $t$ -tolerant self-implementation of **consensus** for omission failures.<sup>6</sup> The resource complexity of the implementation is  $t + 1$  and is optimal.*

*Proof* Let  $\mathcal{O}$  be a derived object of the implementation, and  $O_1, O_2, \dots, O_{t+1}$  be its base objects. Consider an execution  $E$  in which at most  $t$  base objects fail by omission, and the remaining objects are correct. We show that  $\mathcal{O}$  is correct in  $E$ .

1.  $\mathcal{O}$  satisfies validity: An easy induction on  $k$ , the variable in Figure 8, shows that if  $estimate_p$  equals some value  $u$  at any point in  $E$ , then there was a prior invocation (from some process  $q$ ) of **Propose**( $q, u, \mathcal{O}$ ). The induction will use Proposition 5.2, and the fact that  $p$  does not change  $estimate_p$  if a base object returns  $\perp$ .

---

<sup>6</sup> Recall our convention that, if we do not mention the number of processes for which an implementation is intended, then the implementation is for  $N$  processes, where  $N$  is arbitrary.

2.  $\mathcal{O}$  satisfies agreement: Since at most  $t$  base objects fail, there is an  $O_k$  ( $1 \leq k \leq t+1$ ) that is correct. So  $O_k$  returns the same response  $w \in \{0,1\}$  to every process that accesses it. This implies that for all  $p$  that access  $O_k$ ,  $estimate_p = w$  when  $p$  completes the  $k^{th}$  iteration of the loop. Since each base object in  $O_{k+1}, \dots, O_{t+1}$  is either correct or fails by omission in  $E$ , by Propositions 5.1 and 5.2, each of these base objects satisfies validity. From these facts, it is easy to conclude from the implementation that  $estimate_p$  never changes value from the  $(k+1)$ st iteration onwards. Thus  $\mathcal{O}$  returns the same response  $w$  to every  $p$ .
3.  $\mathcal{O}$  satisfies integrity: Obvious.

Since a base object that fails by omission remains wait-free, it is clear that  $\mathcal{O}$  is wait-free in  $E$ . By Proposition 5.1,  $\mathcal{O}$  is correct in  $E$ . It is obvious that the resource complexity of  $t+1$  of our self-implementation is optimal.  $\square$

We remark that the above implementation is *not* gracefully degrading. To see this, suppose that  $v_p = 0$  and  $v_q = 1$ , and all the  $t+1$  base objects fail by crash initially. It is easy to see that  $\mathcal{O}$  returns 0 to  $p$  and 1 to  $q$ . Thus,  $\mathcal{O}$  does not satisfy agreement and, by Proposition 5.2, the failure of  $\mathcal{O}$  is more severe than omission. Later, in Section 7, we will present a  $t$ -tolerant self-implementation of **consensus** that is also gracefully degrading (for omission). This implementation uses  $2t+1$  base objects. We will also prove that  $2t+1$  is a lower bound on the resource complexity of any  $t$ -tolerant gracefully degrading implementation of **consensus** for omission. Interestingly, as we will prove later in Section 7, **consensus** has no  $t$ -tolerant gracefully degrading implementation for crash.

### 5.1.3 Tolerating arbitrary failures

In this section, we present a  $t$ -tolerant self-implementation for arbitrary failures whose resource complexity is  $O(t \log t)$ . This self-implementation uses the divide-and-conquer strategy. The base step obtains a 1-tolerant self-implementation, and the recursive step obtains a  $t$ -tolerant self-implementation from a  $t/2$ -tolerant self-implementation.

Figure 9 presents the base step, the 1-tolerant self-implementation of **consensus** for arbitrary failures. This implementation uses six base objects  $O_1, \dots, O_6$ , divided into two groups. The first group consists of  $O_1, O_2$ , and  $O_3$ , and the second group consists of  $O_4, O_5$ , and  $O_6$ . To propose a value  $v$  to the derived consensus object  $\mathcal{O}$ , process  $p$  proceeds as follows: it proposes  $v$  to the first group; then, it proposes the response of the first group to the second group; it regards the response of the second group to be the response of  $\mathcal{O}$ . To propose a value  $v$  to a group,  $p$  simply proposes  $v$  to all three objects in the group and obtains their responses.  $p$  regards the majority response from these objects to be the response of the group.

Since a consensus object that experiences an arbitrary failure may return a non-binary response, we always “filter” the responses to get binary responses. We do this using the procedure **f-propose**( $p, v, \mathcal{O}$ ) which calls **propose**( $p, v, \mathcal{O}$ ) and returns the response if it is 0 or 1, and returns 0 otherwise.

---

$O_1, O_2, \dots, O_6$  : consensus objects, initialized to the uncommitted state

**Procedure** `Propose`( $p, v, \mathcal{O}$ )  
**begin**  
     $v := \text{Majority}(p, O_1, O_2, O_3, v)$   
     $v := \text{Majority}(p, O_4, O_5, O_6, v)$   
    `return`( $v$ )  
**end**

**Procedure** `Majority`( $p, O_1, O_2, O_3, v$ )  
     $count_p[0..1], w$ : integer local to  $p$   
**begin**  
     $count_p[0..1] := (0,0)$   
    **for**  $i := 1$  to 3  
         $w := \text{f-propose}(p, v, O_i)$   
         $count_p[w] := count_p[w] + 1$   
    **if**  $count_p[0] > count_p[1]$  **then**  
        `return`(0)  
    **else** `return`(1)  
**end**

Figure 9: 1-tolerant self-implementation of consensus for arbitrary failures

---

**Lemma 5.1** *Let  $i$  be either 1 or 4. If at most one object among  $O_i$ ,  $O_{i+1}$ , and  $O_{i+2}$  fails, then  $\text{Majority}(p, O_i, O_{i+1}, O_{i+2}, v)$  returns  $\bar{v}$  only if there is a concurrent or preceding execution of  $\text{Majority}(q, O_i, O_{i+1}, O_{i+2}, \bar{v})$ .*

*Proof* Clear from the algorithm. □

**Lemma 5.2** *Let  $i$  be either 1 or 4. If no object among  $O_i$ ,  $O_{i+1}$ , and  $O_{i+2}$  fails, then, for all  $p$  and  $q$ ,  $\text{Majority}(p, O_i, O_{i+1}, O_{i+2}, v_p)$  returns the same value as  $\text{Majority}(q, O_i, O_{i+1}, O_{i+2}, v_q)$ .*

*Proof* Clear from the algorithm. □

**Theorem 5.2** *Figure 9 presents a 1-tolerant gracefully degrading self-implementation of consensus for arbitrary failures.*

*Proof* Since the implementation is bounded, by Lemma 4.2, it is gracefully degrading for arbitrary failures. We now prove that the implementation is 1-tolerant.

Consider an execution  $E$  in which at most one of  $O_1, O_2, \dots, O_6$  fails by the arbitrary failure mode and the remaining are correct. Lemma 5.1 implies that  $\mathcal{O}$  satisfies validity in  $E$ . Clearly, either all of  $O_1, O_2$ , and  $O_3$  are correct in  $E$ , or all of  $O_4, O_5$ , and  $O_6$  are correct in  $E$ . In the latter case, Lemma 5.2 implies that  $\mathcal{O}$  satisfies agreement in  $E$ . In the former case, Lemmas 5.1 and 5.2 together imply that  $\mathcal{O}$  satisfies agreement in  $E$ . It is obvious that  $\mathcal{O}$  satisfies integrity and is wait-free in  $E$ . Thus, by Proposition 5.1,  $\mathcal{O}$  is correct in  $E$ . □

Given this 1-tolerant self-implementation, by Booster Lemma (Corollary 4.3) we obtain a  $t$ -tolerant self-implementation of consensus for arbitrary failures. However, the resulting resource complexity is  $O(t^{\log_2 6})$ .

A more efficient recursive algorithm is presented in Figure 10. This algorithm implements a  $t$ -tolerant consensus object  $\mathcal{O}$  from  $O_1$ , a  $\lceil \frac{t-1}{2} \rceil$ -tolerant consensus object,  $O_2$ , a  $\lfloor \frac{t-1}{2} \rfloor$ -tolerant consensus object, and  $10t + 3$  (0-tolerant) consensus objects —  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$ . Figure 11 illustrates the order in which the base objects of  $\mathcal{O}$  are accessed by a process proposing 0 on  $\mathcal{O}$  (the access pattern for a process proposing 1 on  $\mathcal{O}$  is symmetric). Before presenting a formal correctness proof, we provide some intuition for the implementation.

Consider an execution in which at most  $t$  base objects fail by the arbitrary failure mode. Since  $O_1$  is  $\lceil \frac{t-1}{2} \rceil$ -tolerant and  $O_2$  is  $\lfloor \frac{t-1}{2} \rfloor$ -tolerant, at least one of  $O_1$  and  $O_2$  is correct. The algorithm is based on this key observation.

The high level intuition behind the implementation of  $\text{Propose}(p, v_p, \mathcal{O})$  is as follows. Process  $p$  proposes  $v_p$  to  $O_1$  and then checks if there is evidence to believe that  $O_1$  has failed. If there is no such evidence,  $p$  adopts the value returned by  $O_1$  as the return value of  $\text{Propose}(p, v_p, \mathcal{O})$ . Otherwise,  $p$  proposes to  $O_2$  and adopts the value returned by  $O_2$  as the return value of  $\text{Propose}(p, v_p, \mathcal{O})$ .

Process  $p$  uses objects  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$  to determine whether  $O_1$  has failed.  $O_1$  can fail in one of three ways: (i) by returning a value outside  $\{0, 1\}$ , (ii) by returning a value  $v \in \{0, 1\}$  that was not proposed by any process, and (iii) by returning 0 to some processes and 1 to other processes. The first case is overcome by using **f-propose** as a “filter”. The second and third cases are detected with the help of  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$ .

The failure detection provided by  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$  is not perfect: if  $O_1$  fails, some processes may not detect the failure. (However, it is never the case that, if  $O_1$  is correct, some process believes that  $O_1$  is faulty.) Thus, a process  $p$  may detect that  $O_1$  failed, but a different process  $q$  may not. Then,  $q$  decides the value, say  $v$ , returned to it by  $O_1$ . Process  $p$ , on the other hand, proposes to  $O_2$  and decides the value returned by  $O_2$ . To avoid disagreement between the decisions of  $p$  and  $q$ , our implementation ensures that  $p$  proposes  $v$  (and not  $\bar{v}$ ) to  $O_2$ . Since  $O_2$  is correct (this follows from the fact that  $O_1$  is faulty),  $O_2$  returns  $v$  and, thus,  $p$  also decides  $v$ .

We state below two properties of our algorithm which are central to understanding its correctness.

- P1. If  $O_1$  is correct and  $O_1$  returns 0 to process  $p$ , then  $\text{count}_p[0] \geq 2t + 1$ . (The symmetric property, resulting from replacing 0 by 1, also holds.)

If  $O_1$  is correct and  $O_1$  returns 0, then some process  $q$  proposed 0 to  $O_1$  before any process got a response from  $O_1$ . It follows from our implementation that (i) process  $q$  had proposed 0 to each of  $A_0[1 \dots 3t + 1]$  before it proposed 0 to  $O_1$ , and (ii) no process proposed 1 to any of  $A_0[1 \dots 3t + 1]$  before  $q$  proposed 0 to  $O_1$ . Thus, when  $p$  accesses the objects  $A_0[1 \dots 3t + 1]$ , every correct object in  $A_0[1 \dots 3t + 1]$  returns 0. Since at least  $2t + 1$  of the objects in  $A_0[1 \dots 3t + 1]$  are correct, we have  $\text{count}_p[0] \geq 2t + 1$ .

- P2. If  $O_1$  is correct and  $O_1$  returns  $v$ , then, for all processes  $p$ ,  $\text{WitnessCount}_p[v] \geq 3t + 1$ .

If  $O_1$  is correct and  $O_1$  returns  $v$  to some process, then  $O_1$  returns  $v$  to every process. By the implementation, every process proposes  $v$  to every object in  $B[1 \dots 4t + 1]$ . Since at least  $3t + 1$  of the objects in  $B[1 \dots 4t + 1]$  are correct, we have  $\text{WitnessCount}_p[v] \geq 3t + 1$ .

Thus, if a process  $p$  receives  $v$  from  $O_1$ ,  $\text{count}_p[v] \geq 2t + 1$ , and  $\text{WitnessCount}_p[v] \geq 3t + 1$ , then  $O_1$  appears correct to  $p$  and, by line 13,  $p$  decides  $v$ . It is still possible that some process  $q$ , using the above properties, detected  $O_1$  to be faulty. However, since  $A_v[1 \dots 3t + 1]$  and  $B[1 \dots 4t + 1]$  are consensus objects and no more than  $t$  of them fail, we have  $\text{count}_q[v] \geq t + 1$  and  $\text{WitnessCount}_q[v] \geq 2t + 1$ . Thus, lines 12 through 18 of the implementation ensure that  $q$  proposes  $v$  to  $O_2$ . Since  $O_2$  is correct (this follows from the fact that  $O_1$  is faulty),  $O_2$  returns  $v$  and, thus,  $q$  also decides  $v$ .

We now provide a more rigorous proof of correctness for the implementation.

**Theorem 5.3** *Figure 10 presents a  $t$ -tolerant gracefully degrading self-implementation of consensus for arbitrary failures of resource complexity  $O(t \log t)$ .*

---

$A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ ,  $B[1 \dots 4t + 1]$  : (0-tolerant) consensus objects,  
 initialized to the uncommitted state  
 $O_1$  :  $\lceil \frac{t-1}{2} \rceil$ -tolerant consensus objects, initialized to the uncommitted state  
 $O_2$  :  $\lfloor \frac{t-1}{2} \rfloor$ -tolerant consensus objects, initialized to the uncommitted state

**Procedure Propose**( $p, v_p, \mathcal{O}$ )  
 $count_p[0..1]$ ,  $WitnessCount_p[0..1]$ ,  $belief_p$ ,  $ans1_p$ ,  $ans2_p$ ,  $v'_p$ ,  $i$ ,  $w$  : integer local to  $p$   
**begin**  
 1      $count_p[0..1]$ ,  $WitnessCount_p[0..1] := (0,0)$   
  
 2     **Phase 1:** **for**  $i := 1$  **to**  $3t + 1$   
 3          $w := \text{f-propose}(p, v_p, A_{v_p}[i])$   
 4         **if**  $w = v_p$  **then**  $count_p[v_p] := count_p[v_p] + 1$   
  
 5     **Phase 2:**  $ans1_p := \text{f-propose}(p, v_p, O_1)$   
  
 6     **Phase 3:** **for**  $i := 1$  **to**  $4t + 1$   
 7          $w := \text{f-propose}(p, ans1_p, B[i])$   
 8          $WitnessCount_p[w] := WitnessCount_p[w] + 1$   
  
 9     **Phase 4:** **for**  $i := 1$  **to**  $3t + 1$   
 10          $w := \text{f-propose}(p, v_p, A_{\overline{v_p}}[i])$   
 11         **if**  $w = \overline{v_p}$  **then**  $count_p[\overline{v_p}] := count_p[\overline{v_p}] + 1$   
  
 12     **Phase 5:** Choose  $belief_p$  such that  $WitnessCount_p[belief_p] > WitnessCount_p[\overline{belief_p}]$   
 13         **if**  $WitnessCount_p[belief_p] \geq 3t + 1$  and  $count_p[belief_p] \geq 2t + 1$  **then**  
 14             **return**( $belief_p$ )  
 15         **if**  $WitnessCount_p[belief_p] \geq 2t + 1$  and  $count_p[belief_p] \geq t + 1$  **then**  
 16              $v'_p := belief_p$   
 17         **else**  
 18              $v'_p := v_p$   
 19              $ans2_p := \text{propose}(p, v'_p, O_2)$   
 20             **return**( $ans2_p$ )  
**end**

Figure 10: Efficient  $t$ -tolerant self-implementation of consensus for arbitrary failures

---

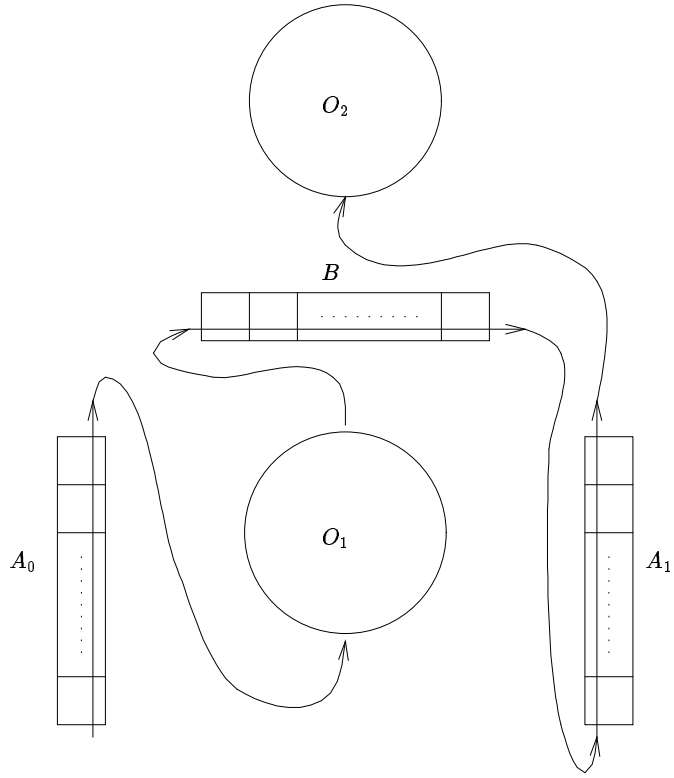


Figure 11: Execution trace of a process proposing 0 on  $\mathcal{O}$

---



*Proof* Since the implementation is bounded, by Lemma 4.2, it is gracefully degrading for arbitrary failures. We now prove that the implementation is  $t$ -tolerant.

Consider an execution  $E$  in which at most  $t$  base objects fail by the arbitrary failure mode, and the remaining are correct. We show below, through a series of lemmas, that  $\mathcal{O}$  is correct in  $E$ ; or equivalently (by Proposition 5.1), that  $\mathcal{O}$  satisfies validity, agreement, and integrity, and is wait-free in  $E$ . Proposition 5.1 is used very often in this proof. For brevity, we omit references to it.

**Lemma 5.3** *If  $O_1$  fails in  $E$ , then  $O_2$  is correct in  $E$ .*

*Proof* Suppose both  $O_1$  and  $O_2$  fail in  $E$ . Since  $O_1$  is derived from a  $\lceil \frac{t-1}{2} \rceil$ -tolerant implementation, at least  $\lceil \frac{t-1}{2} \rceil + 1$  base objects of  $O_1$  must fail in  $E$ . Similarly, at least  $\lfloor \frac{t-1}{2} \rfloor + 1$  base objects of  $O_2$  must fail in  $E$ . Thus a total of  $\lceil \frac{t-1}{2} \rceil + \lfloor \frac{t-1}{2} \rfloor + 2 > t$  base objects of  $\mathcal{O}$  fail in  $E$ , a contradiction to the definition of  $E$ .  $\square$

**Lemma 5.4** *If  $O_1$  is correct in  $E$ ,  $\mathcal{O}$  satisfies validity and agreement in  $E$ .*

*Proof* Suppose  $O_1$  is correct. Thus,  $O_1$  satisfies validity and agreement. By the agreement property of  $O_1$ ,  $ans1_p = ans1_q$  for all  $p, q$ . Let  $v = ans1_p$ . Thus, every process proposes the same value  $v$  to every  $B[i]$  in Phase 3. Since at most  $t$  objects in  $B[1 \dots 4t + 1]$  fail,  $belief_p = v$  and  $WitnessCount_p[belief_p] \geq 3t + 1$  (for every  $p$ ).

By the validity property of  $O_1$ , some process  $q$  will have invoked `propose`( $q, v, O_1$ ) before any process gets the response  $v$  from  $O_1$ . This implies that  $q$  will have finished Phase 1 before any process begins Phase 3. Since at least  $2t + 1$  objects in  $A_v[1 \dots 3t + 1]$  are correct, it follows that, for all  $p$ ,  $count_p[v] \geq 2t + 1$  by the end of Phase 4 of  $p$ . Thus, we have  $WitnessCount_p[belief_p] \geq 3t + 1$  and  $count_p[belief_p] \geq 2t + 1$  (for every  $p$ ). Hence, every  $p$  decides  $v$  (the proposal of  $q$ ) by line 14.  $\square$

**Lemma 5.5** *If  $O_1$  fails in  $E$ ,  $\mathcal{O}$  satisfies validity and agreement in  $E$ .*

*Proof* Suppose  $O_1$  fails. Then by Lemma 5.3,  $O_2$  is correct, and thus satisfies validity and agreement. We need to consider two cases.

**CASE 1** Suppose some process  $p$  returns by line 14. This implies that  $WitnessCount_p[belief_p] \geq 3t + 1$  and  $count_p[belief_p] \geq 2t + 1$ . Since at most  $t$  base objects fail, it follows that, for every  $q$ ,  $WitnessCount_q[belief_p] \geq 2t + 1$  and  $count_q[belief_p] \geq t + 1$ . By line 12, this implies that  $belief_q = belief_p$ . Let  $V = belief_p$ . Since  $WitnessCount_q[belief_q] \geq 2t + 1$  and  $count_q[belief_q] \geq t + 1$ , either  $q$  returns  $belief_q = V$  by line 14 and we have agreement between  $p$  and  $q$ , or  $q$  sets  $v'_q$  to  $belief_q$  by line 16, making  $v'_q$  equal to  $V$ . Thus every  $q$ , that does not return by line 14, proposes  $v'_q = V$  on  $O_2$ . By the validity property of  $O_2$ ,  $ans2_q = V$ , and  $q$  returns  $V$  by line 19. Again we have agreement between  $p$  and  $q$ .

To see that  $\mathcal{O}$  satisfies validity, note that  $count_p[belief_p] \geq 2t + 1$  implies that some process proposed  $belief_p = V$  on at least  $t + 1$  objects in  $A_{belief_p}[1 \dots 3t + 1]$ .

**CASE 2** Suppose no process returns by line 14. Then every  $q$  returns  $ans2_q$  by line 19. By the agreement property of  $O_2$ , for all  $p, q$ , we have  $ans2_p = ans2_q$ . Thus,  $\mathcal{O}$  satisfies agreement. In the following, let  $ans = ans2_p$ .

By the validity property of  $O_2$ , some process  $p$  must have proposed  $ans$  to  $O_2$ . That is  $v'_p = ans$ . In the algorithm,  $v'_p$  equals either  $v_p$  or  $belief_p$ . If  $v'_p = v_p$ , then clearly  $\mathcal{O}$  satisfies validity. If  $v'_p = belief_p \neq v_p$ , then  $p$  must have executed line 16. It follows that  $count_p[belief_p] \geq t + 1$ . Since at most  $t$  objects in  $A_{belief_p}[1 \dots 3t + 1]$  fail, some process  $q$  proposed  $v_q = belief_p$  on some object in  $A_{belief_p}[1 \dots 3t + 1]$ . Thus, process  $q$  proposed  $v_q$  on  $\mathcal{O}$ . Thus,  $\mathcal{O}$  satisfies validity.  $\square$

**Lemma 5.6** *The resource complexity of the implementation in Figure 10 is  $O(t \log t)$ .*

*Proof* Denoting the resource complexity of the  $t$ -tolerant self-implementation of **consensus** for arbitrary failures by  $f(t)$ , we have the following recurrence:  $f(t) = 2f(t/2) + 2(3t + 1) + (4t + 1)$  and  $f(1) = 6$ .  $\square$

It is obvious that  $\mathcal{O}$  satisfies integrity and is wait-free in  $E$ . By Lemmas 5.4 and 5.5,  $\mathcal{O}$  satisfies validity and agreement in  $E$ . Thus, by Proposition 5.1,  $\mathcal{O}$  is correct in  $E$ . This completes the proof of Theorem 5.3.  $\square$

As we will see later, to obtain fault-tolerant implementations of generic types, it is useful to have a fault-tolerant implementation of **consensus with safe-reset**, not just of **consensus**. Let us first recall the type **consensus with safe-reset**. The sequential specification of this type is in Figure 2 and its history transformation function is explained in Section 2.8. Intuitively, an object of this type is like a consensus object, but it also supports the reset operation. Applying reset causes the object to move to the uncommitted state. Thus, the object can be used for multiple rounds of consensus by resetting it between rounds. However, the reset operation is guaranteed to work only if it is executed in “isolation”: that is, if it is not concurrent with another reset operation or a propose operation. Otherwise the object may behave in an unrestricted manner.

Figures 10 and 12, with the following modifications, present a  $t$ -tolerant gracefully degrading self-implementation of **consensus with safe-reset**. In Figure 10, assume that objects  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$  are no longer just consensus objects, but are consensus-with-safe-reset objects, initialized to the uncommitted state. Also, assume that  $O_1$  and  $O_2$  are  $\lceil \frac{t-1}{2} \rceil$ -tolerant and  $\lfloor \frac{t-1}{2} \rfloor$ -tolerant consensus-with-safe-reset objects, initialized to the uncommitted state.

**Theorem 5.4** *Figures 10 and 12 present a  $t$ -tolerant gracefully degrading self-implementation of **consensus with safe-reset** for arbitrary failures.*

*Proof Sketch* Let  $E$  be an execution in which a reset operation on  $\mathcal{O}$  is not concurrent with any other operation on  $\mathcal{O}$ . It is obvious that at the end of an execution of **Reset**( $p, \mathcal{O}$ ), all correct objects among  $O_1$ ,  $O_2$ ,  $A_0[1 \dots 3t + 1]$ ,  $A_1[1 \dots 3t + 1]$ , and  $B[1 \dots 4t + 1]$  are in the uncommitted state. The implementation of **Propose**( $p, v_p, \mathcal{O}$ ), as well as its proof of correctness, is the same as before.  $\square$

---

```

Procedure Reset( $p, \mathcal{O}$ )
   $i$  : integer local to  $p$ 
begin
  reset( $p, O_1$ )
  reset( $p, O_2$ )
  for  $i := 1$  to  $3t + 1$ 
    reset( $p, A_0[i]$ )
    reset( $p, A_1[i]$ )
  for  $i := 1$  to  $4t + 1$ 
    reset( $p, B[i]$ )
  return( $ack$ )
end

```

Figure 12: Reset procedure of the  $t$ -tolerant self-implementation of consensus with `safe-reset` for arbitrary failures

---

## 5.2 Fault-tolerant implementation of register

The type  $n$ -valued `register` supports the operations `read` and `write`  $v$  ( $0 \leq v < n$ ), and has a simple sequential specification: `read` returns the last value written. We write `unbounded register` for  $\infty$ -valued `register`, and `boolean register` for 2-valued `register`. If a result holds for  $n$ -valued `register`, for all finite  $n$  and for  $n = \infty$ , in stating that result we simply write `register` without qualifying it as  $n$ -valued. The main result of this section is that `register` has a  $t$ -tolerant gracefully degrading self-implementation for arbitrary failures.

First, we present a  $t$ -tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register` in Figure 13.<sup>7</sup> The implementation uses  $2t + 1$  base registers. To read the derived register, the reader process  $P_r$  reads all  $2t + 1$  base registers and collects their responses in  $S$ . It then returns  $mode(S)$ , a value that occurs at least as many times in  $S$  as any other value. To write a value  $v$  into the derived register, the writer process  $P_w$  simply writes  $v$  to all  $2t + 1$  base registers.

**Lemma 5.7** *Figure 13 presents a  $t$ -tolerant gracefully degrading self-implementation of 1-reader 1-writer safe register for arbitrary failures.*

*Proof Sketch* Since the implementation is bounded, by Lemma 4.2, it is gracefully degrading

---

<sup>7</sup>Recall that this type has the same sequential specification as `register`, but has a different history transformation function, as explained in Section 2.8. Intuitively, if a read operation on an object of this type overlaps with a write, then that read operation is allowed to return any value [Lam86]. Furthermore, the object's behavior is unrestricted if either more than one process invokes read operations or more than one process invokes write operations.

---

$R_1, R_2, \dots, R_{2t+1}$ : 1-reader 1-writer safe registers, initialized to  
the initial value of the derived register

<u>Apply(<math>P_r, read, \mathcal{R}</math>)</u> $val, i$ : integers, local to $P_r$ $S$ : multi-set of integers, local to $P_r$ <b>begin</b> $S := \emptyset$ <b>for</b> $i := 1$ to $2t + 1$ $val := \text{apply}(P_r, read, R_i)$ $S := S \cup \{val\}$ return $mode(S)$ <b>end</b>	<u>Apply(<math>P_w, write\ v, \mathcal{R}</math>)</u> $i$ : integer, local to $P_w$ <b>begin</b> <b>for</b> $i := 1$ to $2t + 1$ $\text{apply}(P_w, write\ v, R_i)$ return $ack$ <b>end</b>
--	---

Figure 13:  $t$ -tolerant self-implementation of 1-reader 1-writer safe register for arbitrary failures

---

for arbitrary failures. We now prove that the implementation is  $t$ -tolerant.

Let  $\mathcal{R}$  be a derived register of the implementation, and  $R_1, \dots, R_{2t+1}$  be its base registers. Let  $E$  be an execution in which at most one process, call it  $P_r$ , reads  $\mathcal{R}$ , and at most one process, call it  $P_w$ , writes  $\mathcal{R}$ . Also, assume that at most  $t$  base registers fail in  $E$  and they fail by the arbitrary failure mode. Consider a read operation  $r$  on  $\mathcal{R}$  by  $P_r$  that is not concurrent with any write operation on  $\mathcal{R}$  by  $P_w$ . Let  $\text{Apply}(P_w, write\ v, \mathcal{R})$  be the latest write operation that precedes  $r$ . It is clear from the implementation that all correct base registers return  $v$  during the operation  $r$ . Since there are at least  $t + 1$  correct base registers, it follows that  $P_r$  receives  $v$  from at least  $t + 1$  base registers, and returns  $v$ . Hence the correctness of the implementation.  $\square$

There are many results presenting bounded implementations of one type of register from another [Pet83, Lam86, VA86, Blo87, BP87, NW87, PB87, SAG87, Sch88, Vid88, Vid89, HV91]. Some of them (for example, [Lam86, SAG87, Sch88]) can be combined to implement a multi-reader, multi-writer, atomic register using 1-reader, 1-writer, safe registers. In our terminology, this means that **register** has a bounded implementation from **1-reader 1-writer safe register**. This implies, by Lemma 4.2, that **register** has a 0-tolerant gracefully degrading implementation from **1-reader 1-writer safe register** for arbitrary failures. Using this result and Lemma 5.7, and applying Corollary 4.1, we conclude that **register** has a  $t$ -tolerant gracefully degrading implementation from **1-reader 1-writer safe register** for arbitrary failures. This trivially implies the following theo-

rem.

**Theorem 5.5** *register has a  $t$ -tolerant gracefully degrading self-implementation for arbitrary failures.*

### 5.3 Fault-tolerant implementations of generic types

In this section, we describe how to obtain fault-tolerant gracefully degrading implementations of generic types for arbitrary failures. Since arbitrary failures are more severe than the benign crash and omission failures, these implementations tolerate such benign failures as well. They are however not gracefully degrading for crash or omission. We study the feasibility of gracefully degrading implementations for benign failure modes in Section 7.

The theorems of this section depend on the universality results due to Herlihy and Plotkin [Her91b, Plo89]. These results are stated below.

**Theorem 5.6 (Herlihy)** *For all types  $T$ , there is a  $k$  such that  $T$  has a (0-tolerant)  $k$ -bounded implementation from {consensus with safe-reset, unbounded register}.*

Herlihy's universal construction requires unbounded registers even to implement finite types. Plotkin's construction, on the other hand, requires only boolean registers in such a situation [Plo89]. (Jayanti and Toueg achieve the same result as Plotkin, but with a more intuitive construction [JT92].)

**Theorem 5.7 (Plotkin)** *For all finite types  $T$ , there is a  $k$  such that  $T$  has a (0-tolerant)  $k$ -bounded implementation from {consensus with safe-reset, boolean register}.*

From Plotkin's theorem and Lemma 4.2, it follows that every finite type has a (0-tolerant) gracefully-degrading implementation from {consensus with safe-reset, boolean register} for arbitrary failures. Using this, together with Theorems 5.4, 5.5, and Lemma 4.1, we obtain:

**Corollary 5.1** *Let  $T$  be any finite type.*

- *$T$  has a  $t$ -tolerant gracefully degrading implementation from {consensus with safe-reset, boolean register} for arbitrary failures.*
- *If each of consensus with safe-reset and boolean register has a 0-tolerant gracefully degrading implementation from  $T$  for arbitrary failures, then  $T$  has a  $t$ -tolerant gracefully degrading self-implementation for arbitrary failures.*

From Theorem 5.6 and Lemma 4.2, it follows that every type has a (0-tolerant) gracefully-degrading implementation from {consensus with safe-reset, unbounded register} for arbitrary failures. Using this, together with Theorems 5.4, 5.5, and Lemma 4.1, we obtain:

**Corollary 5.2** *Let  $T$  be any type.*

- *$T$  has a  $t$ -tolerant gracefully degrading implementation from `{consensus with safe-reset, unbounded register}` for arbitrary failures.*
- *If each of `consensus with safe-reset` and `unbounded register` has a 0-tolerant gracefully degrading implementation from  $T$  for arbitrary failures, then  $T$  has a  $t$ -tolerant gracefully degrading self-implementation for arbitrary failures.*

We now apply the above corollaries to show that several common types have  $t$ -tolerant self-implementations for arbitrary failures. However, to do this, we have to first show that common types implement both `consensus with safe-reset` and `register`.

It is known that `fetch&add`, `queue`, `stack`, and `test&set` implement `consensus with safe-reset` for two processes, and that `compare&swap`, `move`, and `memory-to-memory swap` (henceforth `m-m swap`) implement `consensus with safe-reset` for any number of processes [Her91b, KM93].<sup>8</sup> These are all bounded implementations and, by Lemma 4.2, are gracefully degrading for arbitrary failures.

We claim that `compare&swap`, `move`, `m-m swap`, and `test&set` implement `1-reader 1-writer boolean safe register`, and that `fetch&add`, `queue`, and `stack` implement `1-reader 1-writer unbounded safe register`. We will show a bounded implementation of `1-reader 1-writer boolean register` from `test&set`, and this trivially implies that `test&set` implements `1-reader 1-writer boolean safe register`. The other implementations claimed above are also bounded and are easy to obtain. We have therefore omitted their descriptions. As already mentioned, it is known that `register` has a bounded implementation from `1-reader 1-writer safe register`. From these results, we conclude that `boolean register` has a bounded implementation from each of `compare&swap`, `move`, `m-m swap`, and `test&set`, and that `unbounded register` has a bounded implementation from each of `fetch&add`, `queue`, and `stack`. By Lemma 4.2, these implementations are gracefully degrading for arbitrary failures.

In Figure 14, we implement a `1-reader 1-writer boolean register`  $\mathcal{R}$  from a `test&set` object  $TS$ . To complement the value in  $\mathcal{R}$ , the writer flips the state of  $TS$ . It does this by applying the `test&set` operation on  $TS$ . If this operation returns 0, the writer knows that it has flipped the state of  $TS$ . Otherwise, the writer applies the `reset` operation to flip the state of  $TS$ . To read  $\mathcal{R}$ , the reader obtains the current state of  $TS$  by applying the `test&set` operation on it. If the state of  $TS$  is 0, the reader deduces that, since its last read, the writer complemented the value of  $\mathcal{R}$  an odd number of times. Therefore, the reader returns the complement of the last value it returned. We omit the proof of correctness.

From the above, we have

---

<sup>8</sup> Our definition of the types `move` and `m-m swap` are weaker than the corresponding ones given by Herlihy [Her91b]. In our definition (see Appendix B), an object of either type consists of only a pair of cells whose contents can be moved or swapped. In [Her91b], a `move/swap` operation can `move/swap` the contents of any cell into any other cell in an infinite array of cells. Kleinberg and Mullainathan showed that, even with the weaker definitions, `move` and `m-m swap` can implement `consensus with safe-reset` for any number of processes [KM93].

---

$TS$  : test&set object, initialized to state 1  
 $LastValueReturned$  : boolean, local to the reader process  $P_r$ , initialized to the initial value of the implemented register  $\mathcal{R}$   
 $LastValueWritten$  : boolean, local to the writer process  $P_w$ , initialized to the initial value of the implemented register  $\mathcal{R}$   
 $state_r$  : boolean, local to the reader process  $P_r$ , uninitialized  
 $state_w$  : boolean, local to the writer process  $P_w$ , uninitialized

<p> <u>Apply(<math>P_r, read, \mathcal{R}</math>)</u>  <math>state_r := \text{test\&amp;set}(P_r, TS)</math>  <b>if</b> (<math>state_r = 0</math>) <b>then</b>            <math>LastValueReturned := \neg LastValueReturned</math>        return <math>LastValueReturned</math> </p>	<p> <u>Apply(<math>P_w, write\ v, \mathcal{R}</math>)</u>  <b>if</b> (<math>v \neq LastValueWritten</math>) <b>then</b>            <math>LastValueWritten := v</math>            <math>state_w := \text{test\&amp;set}(P_w, TS)</math>            <b>if</b> (<math>state_w = 1</math>) <b>then</b>                reset(<math>P_w, TS</math>)        return <math>ack</math> </p>
--	---

Figure 14: 1-reader 1-writer boolean register from test&set

---

**Corollary 5.3** *compare&swap, move, and m-m swap have  $t$ -tolerant self-implementations for arbitrary failures.*

**Corollary 5.4** *queue, stack, test&set, and fetch&add have  $t$ -tolerant self-implementations for arbitrary failures. These implementations are for two processes.*

## 6 Tolerating non-responsive failures

So far we have considered objects that remain responsive (*i.e.*, wait-free) even if they fail. Thus, after invoking an operation, a process could afford to wait for a response before proceeding to invoke the next operation. Consequently, there has been no need so far for a process to have more than one incomplete operation at any time. With non-responsive failures, the situation is different. Since a failed object may not respond, waiting for a response could block the process forever. To overcome this difficulty, we allow a process to access base objects “in parallel”. In other words, a process can have multiple incomplete operations at any time. However, we still restrict a process to have no more than one incomplete operation on any particular object.

The ability to access base objects in parallel allows us to build a  $t$ -tolerant implementation of **register**, even for NR-arbitrary failures. In contrast, we show that **consensus** does not have an implementation that can tolerate the failure of a single base object, even if we assume that the faulty object can only fail by NR-crash and even if we do not restrict the number or the type of base objects that can be used in the implementation. Consequently, **test&set**, **compare&swap**, **queue**, **stack**, and several other common types, which can implement **consensus**, have no fault-tolerant implementations for any non-responsive failure mode. However, we show that randomization can be used to circumvent this impossibility result. *Every* type has a  $t$ -tolerant *randomized* implementation from **register**, even for NR-arbitrary failures. These results are the subject of this section.

### 6.1 Impossibility of fault-tolerant implementation of consensus

In this section, we first prove that **consensus** has no 1-tolerant implementation for NR-crash. We then define an extremely weak non-responsive failure mode, called *unfairness to a known process*, and prove that **consensus** has no 1-tolerant implementation even for this failure mode.

In each case, to prove that a certain implementation  $\mathcal{I}$  does not exist, we show that if  $\mathcal{I}$  exists, it would violate well-known the impossibility result due to Loui and Abu-Amara [LAA87] and Dolev, Dwork, and Stockmeyer [DDS87]. This result is about the *consensus problem for  $n$  processes*, defined informally as follows. Each process  $P_i$  is initially given an input  $v_i \in \{0, 1\}$ . Each correct process  $P_i$  must eventually decide a value  $d_i$  such that (i)  $d_i \in \{v_1, v_2, \dots, v_n\}$ , and (ii) for all processes  $P_i$  and  $P_j$  that decide,  $d_i = d_j$ .

**Theorem 6.1** (Loui and Abu-Amara, Dolev, Dwork, and Stockmeyer)



*The consensus problem for  $n$  processes has no solution if processes may communicate only via registers and at most one process may crash.*

**Theorem 6.2** *There is no 1-tolerant implementation of consensus, even for two processes, for NR-crash.*

*Proof* Suppose, for contradiction, there is a finite list  $\mathcal{L} = (T_1, T_2, \dots, T_l)$  of types and a list  $\Sigma = (s_1, s_2, \dots, s_l)$  of states such that there is a 1-tolerant implementation  $\mathcal{I}$  of **consensus** from  $(\mathcal{L}, \Sigma)$ , for two processes, for NR-crash. We will use this implementation to obtain a protocol for the consensus problem for  $l + 2$  processes. This protocol will require only registers for communication between processes and solves the consensus problem even if at most one process may crash.

Consider the concurrent system  $S$  consisting of  $l + 2$  processes, named  $\{p_1, p_2\} \cup \{q_j \mid 1 \leq j \leq l\}$ , and  $4l + 1$  registers, named  $\{invocation(i, j), response(j, i) \mid 1 \leq i \leq 2, 1 \leq j \leq l\} \cup \{decision\}$ . We claim that the consensus problem for processes in  $S$  is solvable, even if at most one process may crash and processes communicate exclusively via the registers in  $S$ . The following is the protocol. Let  $v_i \in \{0, 1\}$  be the initial input of  $p_i$ . The basic idea consists of two steps:

1. Let  $O_1, O_2, \dots, O_l$  be objects of type  $T_1, T_2, \dots, T_l$ , initialized to states  $s_1, s_2, \dots, s_l$ , respectively. Let  $\mathcal{O} = \mathcal{I}(O_1, \dots, O_l)$ . Thus,  $\mathcal{O}$  is a consensus object that can be shared by two processes. Moreover, by definition of  $\mathcal{I}$ ,  $\mathcal{O}$  remains correct even if one of its base objects fails by NR-crash.
2. In system  $S$ , process  $q_j$  ( $1 \leq j \leq l$ ) simulates the base object  $O_j$ , and process  $p_i$  ( $i = 1, 2$ ) simulates the execution of **Propose**( $p_i, v_i, \mathcal{O}$ ) on the derived object  $\mathcal{O}$ .

The details of the protocol are given below. Here, *decision* is used as a multi-writer multi-reader register. All other registers are used as 1-reader 1-writer registers:  $p_i$  writes *invocation*( $i, j$ ) and  $q_j$  reads it;  $q_j$  writes *response*( $j, i$ ) and  $p_i$  reads it.

Initialize all  $4l + 1$  registers to  $\perp$ . Process  $p_i$  simulates **Propose**( $p_i, v_i, \mathcal{O}$ ) as follows. If **Propose**( $p_i, v_i, \mathcal{O}$ ) requires  $p_i$  to invoke some operation  $op$  on  $O_j$ ,  $p_i$  appends  $op$  to the contents of *invocation*( $i, j$ ). (Since  $p_i$  is the only process that writes *invocation*( $i, j$ ), appending  $op$  to the previous contents can be performed in one step.) If **Propose**( $p_i, v_i, \mathcal{O}$ ) requires  $p_i$  to check if a response to some outstanding invocation on  $O_j$  has arrived,  $p_i$  checks if a response has been appended by  $q_j$  (which simulates  $O_j$ ) to *response*( $j, i$ ). If **Propose**( $p_i, v_i, \mathcal{O}$ ) returns a value  $v$ ,  $p_i$  first writes  $v$  in *decision* register, and then decides  $v$ . In addition to (and concurrently with) the above,  $p_i$  periodically checks if the register *decision* contains a non- $\perp$  value. If so, it decides that value.

Process  $q_j$  simulates the base object  $O_j$  as follows. Periodically  $q_j$  checks the registers *invocation*( $1, j$ ) and *invocation*( $2, j$ ), in a round-robin fashion. If  $q_j$  notices that some operation  $op$  has been appended to *invocation*( $i, j$ ),  $q_j$  simulates the application of  $op$  to  $O_j$  (using the sequential specification of the type  $T_j$ ) and appends the corresponding response

to  $response(j, i)$ . In addition to (and concurrently with) the above,  $q_j$  periodically checks if the register *decision* contains a non- $\perp$  value. If so, it decides that value.

The above simulation protocol solves the consensus problem among the  $l + 2$  processes in the concurrent system  $S$ , even if one of them crashes. To see this, consider any execution  $E$  of the concurrent system  $S$  in which at most one process crashes. Let  $E'$  be the corresponding “simulated” execution of the derived object  $\mathcal{O}$ . Note that the crash of one process in  $S$  corresponds to the NR-crash of at most one (simulated) base object of the (simulated) derived object  $\mathcal{O}$  in  $E'$ . Since  $\mathcal{I}$ , the **consensus** implementation from which  $\mathcal{O}$  is derived, is 1-tolerant for NR-crash,  $\mathcal{O}$  is correct in  $E'$  (despite the NR-crash of one of its base objects). Thus, by Proposition 5.1,  $\mathcal{O}$  satisfies integrity, validity, and agreement, and is wait-free in  $E'$ . Since  $\mathcal{O}$  is wait-free (in  $E'$ ), if  $p_i$  does not crash,  $\text{Propose}(p_i, v_i, \mathcal{O})$  eventually returns some value  $v$  (in  $E'$ ). Since  $\mathcal{O}$  satisfies integrity,  $v \in \{0, 1\}$ . Since  $\mathcal{O}$  satisfies validity,  $v$  is either  $v_1$  or  $v_2$ . Since  $\mathcal{O}$  satisfies agreement,  $\text{Propose}(p_1, v_1, \mathcal{O})$  and  $\text{Propose}(p_2, v_2, \mathcal{O})$  never return different values. Thus, from the protocol,  $p_1$  and  $p_2$  do not write different values in register *decision*. Since at most one process crashes, at least one of  $p_1$  and  $p_2$  will eventually write a binary value  $v$  in register *decision*. Since all correct processes periodically check the *decision* register, they eventually decide  $v$ .

We showed that we can use  $\mathcal{I}$  to solve the consensus problem in system  $S$ . This contradicts Theorem 6.1. Thus,  $\mathcal{I}$  cannot exist.  $\square$

We can strengthen the above result as follows. Suppose that *at most one* base object may fail and that it can only do so by being “unfair” (*i.e.*, by not responding) to *at most one* process. Furthermore, suppose that the identity of this process is a priori “common knowledge” among all the processes. Even with this extremely weak failure mode, called *unfairness to a known process*, we can prove the following:

**Theorem 6.3** *There is no 1-tolerant implementation of consensus, even for two processes, for unfairness to a known process.*

*Proof Sketch* Suppose, for contradiction, there is a finite list  $\mathcal{L} = (T_1, T_2, \dots, T_l)$  of types and a list  $\Sigma = (s_1, s_2, \dots, s_l)$  of states such that there is a 1-tolerant implementation  $\mathcal{I}$  of **consensus** from  $(\mathcal{L}, \Sigma)$ , for two processes, for unfairness to, say, process  $p_1$ . Consider the concurrent system  $S$ , as defined in the proof of Theorem 6.2. Suppose processes in  $S$  run the same simulation protocol as in that proof. There are two cases:

1. No process  $q_k$  crashes. In this case, it is easy to see that processes in  $S$  solve the consensus problem (exactly as before).
2. Some process  $q_k$  crashes. In this case, processes in  $S$  may fail to solve the consensus problem for the following reason. The crash of  $q_k$  corresponds to the NR-crash of the simulated base object  $O_k$ . This object is now potentially unfair to *both*  $p_1$  and  $p_2$ . But  $\mathcal{I}$  tolerates unfairness to only  $p_1$ . So the derived consensus object  $\mathcal{O}$  of  $\mathcal{I}$  is not necessarily correct.

To circumvent the problem that arises in Case 2, we modify the simulation protocol as follows: If  $\text{Propose}(p_2, v_2, \mathcal{O})$  requires  $p_2$  to invoke some operation  $op$  on some  $O_j$ ,  $p_2$  appends  $op$  to the contents of  $\text{invocation}(2, j)$ , as before, but now it also waits until a corresponding response is appended to  $\text{response}(j, 2)$  by process  $q_j$ . The rest of the simulation protocol remains exactly as before. We now reconsider the above two cases with the modified simulation protocol.

1. No process  $q_k$  crashes. As before, it is easy to see that processes in  $S$  solve the consensus problem.
2. Some process  $q_k$  crashes. If  $p_2$  attempts to access  $O_k$  after the crash of  $q_k$ , it will simply wait for the response forever.<sup>9</sup> Therefore, at worst, it appears to process  $p_1$  that  $O_k$  is unfair to  $p_1$  and that  $p_2$  is extremely slow. Since  $\mathcal{I}$  tolerates the unfairness of one base object to  $p_1$ ,  $\mathcal{O}$  remains correct. Since  $p_1$  does not crash (we assumed that only one process in  $S$  crashes, and this is  $q_k$ ),  $\text{Propose}(p_1, v_1, \mathcal{O})$  returns a value that  $p_1$  writes into  $\text{decision}$ . The rest of the proof is as in Theorem 6.2.

Again, we have a contradiction to Theorem 6.1. □

From the above two theorems we have:

**Corollary 6.1** *If a type  $T$  implements consensus for two processes, then  $T$  has no 1-tolerant implementation, for two processes, for NR-crash or for unfairness to a known process.*

As mentioned in Section 5.3, consensus has an implementation, for two processes, from each of the following types: `compare&swap`, `fetch&add`, `move`, `queue`, `stack`, `sticky-bit`, `m-m swap`, and `test&set`. Thus, we have:

**Corollary 6.2** *None of the following types has a 1-tolerant implementation, for two processes, for NR-crash or for unfairness to a known process: `compare&swap`, `fetch&add`, `move`, `queue`, `stack`, `sticky-bit`, `m-m swap`, and `test&set`.*

## 6.2 Fault-tolerant implementation of register

In contrast to the above impossibility results, we show in this section that `register` has a  $t$ -tolerant self-implementation even for NR-arbitrary failures.

First, we present a  $t$ -tolerant self-implementation of `1-reader 1-writer safe register` in Figure 15. The implementation uses  $5t + 1$  base registers. To read the derived register, the reader process  $P_r$  invokes `read` on each base register ( $P_r$  delays this read if its previous read on the base register is still incomplete). When  $P_r$  gets responses from  $4t + 1$  base registers, which are collected in the multi-set  $\text{Responses}$ , it returns  $\text{mode}(\text{Responses})$ . (Recall

---

<sup>9</sup>Of course, it also continues to read the `decision` register periodically and decides if a non- $\perp$  value is found there.

---

$R_1, R_2, \dots, R_{5t+1}$ : 1-reader 1-writer safe registers, initialized to the initial value of the derived register	
$Pending_r$ : set, local to the reader process $P_r$ , initialized to $\emptyset$ $Pending_w$ : set, local to the writer process $P_w$ , initialized to $\emptyset$	
<b>Apply</b> ( $P_r, read, \mathcal{R}$ )	<b>Apply</b> ( $P_w, write\ v, \mathcal{R}$ )
$Invoked_r$ : set, local to $P_r$ $Responses_r$ : multi-set, local to $P_r$ $val, i$ : integers, local to $P_r$	$Invoked_w$ : set, local to $P_w$ $Responses_w$ : multi-set, local to $P_w$ $val, i$ : integers, local to $P_w$
<b>begin</b>	<b>begin</b>
$Invoked_r := \emptyset$ $Responses_r := \emptyset$ $i := 0$	$Invoked_w := \emptyset$ $Responses_w := \emptyset$ $i := 0$
<b>Loop</b>	<b>Loop</b>
$i := (i \bmod 5t + 1) + 1$ <b>if</b> $R_i \in Pending_r$ <b>then</b> Check if $R_i$ responded <b>if</b> ( <i>yes</i> ) <b>then</b> $Pending_r := Pending_r - \{R_i\}$ Let $val$ be the response <b>if</b> $R_i \in Invoked_r$ <b>then</b> $Responses_r := Responses_r \cup \{val\}$ <b>if</b> $(R_i \notin Pending_r) \wedge (R_i \notin Invoked_r)$ <b>then</b> Invoke read on $R_i$ $Invoked_r := Invoked_r \cup \{R_i\}$ $Pending_r := Pending_r \cup \{R_i\}$	$i := (i \bmod 5t + 1) + 1$ <b>if</b> $R_i \in Pending_w$ <b>then</b> Check if $R_i$ responded <b>if</b> ( <i>yes</i> ) <b>then</b> $Pending_w := Pending_w - \{R_i\}$ Let $val$ be the response <b>if</b> $R_i \in Invoked_r$ <b>then</b> $Responses_w := Responses_w \cup \{val\}$ <b>if</b> $(R_i \notin Pending_w) \wedge (R_i \notin Invoked_w)$ <b>then</b> Invoke write $v$ on $R_i$ $Invoked_w := Invoked_w \cup \{R_i\}$ $Pending_w := Pending_w \cup \{R_i\}$
<b>Until</b> $ Responses_r  = 4t + 1$ return $mode(Responses_r)$	<b>Until</b> $ Responses_w  = 4t + 1$ return $ack$
<b>end</b>	<b>end</b>

---

Figure 15:  $t$ -tolerant self-implementation of 1-reader 1-writer safe register for NR-arbitrary failures

---

that  $mode(S)$  is a value that occurs at least as many times in  $S$  as any other value.) To write a value  $v$  into the derived register, the writer process  $P_w$  invokes *write*  $v$  on each base register (again, the writer delays invoking this write if its previous write on the base register is still incomplete). The writing of the derived register completes when the writer receives the response *ack* from  $4t + 1$  base registers.

In the implementation, the reader and the writer maintain three sets each in their local memory. *Pending* is the set of base registers on which the process has incomplete operations. *Invoked* is the set of base registers on which the process has already invoked operations in the current execution of the operation on the derived object. *Responses* is the set of responses, from base registers, to the invocations made during the current execution of the operation on the derived object.

**Lemma 6.1** *Figure 15 presents a  $t$ -tolerant self-implementation of 1-reader 1-writer safe register for NR-arbitrary failures.*

*Proof Sketch* Let  $\mathcal{R}$  be a derived register of the implementation, and  $R_1, \dots, R_{5t+1}$  be its base registers. Let  $E$  be an execution in which at most one process  $P_r$  reads  $\mathcal{R}$ , and at most one process  $P_w$  writes  $\mathcal{R}$ . Also, assume that at most  $t$  base registers fail in  $E$  and that they fail by the NR-arbitrary mode. Consider a completed read operation  $r$  on  $\mathcal{R}$  by  $P_r$  that is not concurrent with any write operation on  $\mathcal{R}$  by  $P_w$ . Let  $\text{Apply}(P_w, \text{write } v, \mathcal{R})$  be the latest write operation that precedes  $r$ . We will refer to this operation as  $w$ . From the implementation, it is clear that, of the base registers on which *write*  $v$  was invoked during  $w$ ,  $4t + 1$  base registers responded. Let  $S_w$  denote the set of these  $4t + 1$  base registers. Similarly, it is clear that, of the base registers on which *read* was invoked during  $r$ ,  $4t + 1$  base registers responded. Let  $S_r$  denote the set of these  $4t + 1$  base registers. Let  $S = S_r \cap S_w$ . Clearly,  $|S| \geq 3t + 1$ . Since we assumed that at most  $t$  base registers fail in  $E$ , there are at least  $2t + 1$  correct base registers in  $S$ . From the implementation, it is clear that each correct base register in  $S$  responds with  $v$  to the invocation of *read* by  $P_r$  during  $r$ . Thus, at the end of  $r$ ,  $v$  occurs at least  $2t + 1$  times in the multi-set  $\text{Responses}_r$ . This implies that  $r$  returns  $v$ . Hence the correctness of the implementation.  $\square$

As mentioned in Section 5.2, it is known that **register** has an implementation from **1-reader 1-writer safe register**. Using this result and Lemma 6.1, and applying Corollary 4.1,<sup>10</sup> we conclude that **register** has a  $t$ -tolerant implementation from **1-reader 1-writer safe register** for NR-arbitrary failures. This implies the following theorem.

**Theorem 6.4** ***register** has a  $t$ -tolerant self-implementation for NR-arbitrary failures.*

### 6.3 Randomized fault-tolerant implementations of generic types

So far we assumed that processes are deterministic. Suppose instead that processes have access to “fair coins”. A process can toss a coin and, based on the outcome of the toss, choose its step. Furthermore, let us informally define a *randomized implementation* as an implementation in which every correct process completes its operation on the derived object in a finite expected number of operations on the base objects. Interestingly, every type has a randomized implementation from **register** [Her91a], but most types have no (deterministic) implementations from **register** [Her91b]. In the following, we present a generalization of the former result.

**consensus with safe-reset** has a randomized implementation from **register** [Asp90]. Together with Theorem 6.4, this implies that **consensus with safe-reset** has a  $t$ -tolerant randomized implementation from **register** for NR-arbitrary failures. Combining this with Theorem 6.4, and Theorems 5.6 and 5.7 of Herlihy and Plotkin, we have

---

<sup>10</sup> Observe that every implementation is automatically gracefully degrading for NR-arbitrary failures. Thus, we are able to apply Corollary 4.1.

**Theorem 6.5** *Every finite type has a  $t$ -tolerant randomized implementation from `boolean register` for NR-arbitrary failures. Every infinite type has a  $t$ -tolerant randomized implementation from `unbounded register` for NR-arbitrary failures.*

Thus, if a finite (respectively, infinite) type  $T$  implements `boolean register` (respectively, `unbounded register`), then  $T$  has a  $t$ -tolerant randomized self-implementation for NR-arbitrary failures. As mentioned in Section 5.3, each of `test&set`, `compare&swap`, `move`, and `m-m swap` implements `boolean register`, and each of `fetch&add`, `queue`, and `stack` implements `unbounded register`. Thus, each of the above types has a  $t$ -tolerant randomized self-implementation even for NR-arbitrary failures.

## 6.4 Decomposability of NR-arbitrary failures

The final result of this section concerns the nature of NR-arbitrary failures. It states that the problem of tolerating NR-arbitrary failures can be reduced to two strictly simpler problems: tolerating arbitrary failures and tolerating NR-omission failures.

**Lemma 6.2** (Decomposability of NR-arbitrary failures) *A type  $T$  has a  $t$ -tolerant self-implementation for NR-arbitrary failures if and only if  $T$  has  $t$ -tolerant self-implementations for arbitrary failures and for NR-omission failures.*

*Proof Sketch* The “only if” direction is obvious. We now sketch the proof for the “if” direction. Let  $s$  be any state of  $T$ . Let  $\mathcal{I}_a$  be a  $t$ -tolerant self-implementation of  $(T, s)$  for arbitrary failures and  $\mathcal{I}_o$  be a  $t$ -tolerant self-implementation of  $(T, s)$  for NR-omission failures. Let  $m$  and  $n$  be the resource complexity of the implementations  $\mathcal{I}_a$  and  $\mathcal{I}_o$ , respectively. Define an implementation  $\mathcal{I}$ , of resource complexity  $m \cdot n$ , as follows:  $\mathcal{I}(o_1, o_2, \dots, o_{nm}) = \mathcal{I}_o(\mathcal{I}_a(o_1, \dots, o_m), \dots, \mathcal{I}_a(o_{(n-1)m+1}, \dots, o_{nm}))$ . We will verify below that  $\mathcal{I}$  is a  $t$ -tolerant self-implementation of  $(T, s)$  for NR-arbitrary failures.

Let  $\mathcal{O}$  be a derived object of  $\mathcal{I}$  and  $o_1, o_2, \dots, o_{nm}$  be the base objects of  $\mathcal{O}$ . Thus,  $\mathcal{O} = \mathcal{I}_o(O_1, O_2, \dots, O_n)$  where  $O_k = \mathcal{I}_a(o_{(k-1)m+1}, o_{(k-1)m+2}, \dots, o_{km})$  ( $1 \leq k \leq n$ ). Assume that at most  $t$  objects among  $o_1, o_2, \dots, o_{nm}$  fail, and they fail in an arbitrary manner. This trivially implies that, for each  $O_k$ , at most  $t$  base objects of  $O_k$  fail. Since  $O_k$  is not derived from an implementation that tolerates NR-arbitrary failures,  $O_k$  may not respond to an invocation; however, if it does respond, since it is derived from an implementation that is  $t$ -tolerant for arbitrary failures, its response is correct. We conclude that, if  $O_k$  fails, it fails by NR-omission. We also conclude that at most  $t$  objects among  $O_1, O_2, \dots, O_n$  fail (this follows from the fact that at most  $t$  objects among  $o_1, o_2, \dots, o_{nm}$  fail). From these conclusions and the fact that  $\mathcal{I}_o$  is  $t$ -tolerant for NR-omission, it follows that  $\mathcal{O}$  is correct. Hence the lemma.  $\square$

## 7 Graceful degradation for benign failure modes

Graceful degradation is a desirable property of implementations: it ensures that an implemented object never fails more severely than any of its components. Furthermore, if fault-tolerant implementations are gracefully degrading, then they can be composed (Lemma 4.1) and their degree of fault-tolerance can be automatically boosted (Lemma 4.3). In this section, we investigate the cost and the feasibility of achieving graceful degradation for the benign crash and omission failure modes. As one might expect, graceful degradation comes at a cost: for omission, `consensus` has a  $t$ -tolerant self-implementation of resource complexity  $t+1$ , but it has no  $t$ -tolerant gracefully degrading implementation of resource complexity less than  $2t+1$ . With respect to feasibility, our results are as follows. We identify a class of “order sensitive” types that includes many common types such as `queue`, `stack`, `test&set`, and `compare&swap`. We prove that no type in this class has a fault-tolerant gracefully degrading implementation for crash. Thus, when an object of such a type is implemented in software from a set of hardware objects, the software object can fail more severely than crash even if the underlying hardware objects only fail by crash. In contrast, we show that graceful degradation for omission is achievable in a strong sense: For omission, every type has a  $t$ -tolerant gracefully degrading implementation from every universal set of types. (A set  $S$  of types is *universal* if every type has an implementation from  $S$ .)

### 7.1 Cost of achieving graceful degradation

We have seen that, for omission, `consensus` has a  $t$ -tolerant self-implementation of resource complexity  $t+1$  (see Figure 8). But this implementation is *not* gracefully degrading for omission. In this section, we describe a self-implementation of `consensus` that is both  $t$ -tolerant *and* gracefully degrading for omission. The resource complexity of this implementation is  $2t+1$ . We will then prove that, for any “non-trivial” type (such as `consensus`),  $2t+1$  is a lower bound on the resource complexity of any  $t$ -tolerant gracefully degrading implementation. From these results, we conclude that graceful degradation comes at a cost.

First, let us recall why the implementation in Figure 8 is *not* gracefully degrading. Suppose that  $v_p = 0$  and  $v_q = 1$ , and all the  $t+1$  base objects  $O_1, O_2, \dots, O_{t+1}$  fail by crash initially. It is easy to see that  $\mathcal{O}$  returns 0 to  $p$  and 1 to  $q$ . Thus,  $\mathcal{O}$  does not satisfy agreement and, by Proposition 5.2, the failure of  $\mathcal{O}$  is more severe than omission.

In Figure 16, we present a  $t$ -tolerant gracefully degrading self-implementation of `consensus` for omission.<sup>11</sup> The implementation uses  $2t+1$  base consensus objects. A process  $p$  proposes to the derived object  $\mathcal{O}$  by accessing each of  $O_1, O_2, \dots, O_{2t+1}$ , in that order. At any point in the algorithm,  $p$  holds an estimate of the eventual return value in  $estimate_p$ . When  $p$  proposes its current estimate to a base object  $O_k$ , if  $O_k$  returns a non- $\perp$  response different from  $p$ ’s current estimate,  $p$  deduces that all of  $O_1, O_2, \dots, O_{k-1}$  have failed. Accordingly,  $p$  sets each location in its local vector  $V_p[1 \dots (k-1)]$  to  $\perp$  and changes its estimate to the response it received from  $O_k$ . This deduction by  $p$  is the most

---

<sup>11</sup>As will be shown later in Theorem 7.4, there is *no*  $t$ -tolerant gracefully degrading implementation of `consensus` for crash (for  $t > 0$ ).

---

$O_1, O_2, \dots, O_{2t+1}$  : consensus objects, initialized to the uncommitted state

```

Procedure Propose( $p, v_p, \mathcal{O}$ )      /*  $v_p \in \{0, 1\}$  */
     $V_p[1..2t+1]$ ,  $estimate_p, k$ : integer local to  $p$ 
begin
1       $estimate_p := v_p$ 
2      for  $k := 1$  to  $2t+1$ 
3         $V_p[k] := \text{propose}(p, estimate_p, O_k)$ 
4        if ( $V_p[k] \neq \perp$ )  $\wedge$  ( $V_p[k] \neq estimate_p$ ) then
5           $estimate_p := V_p[k]$ 
6           $V_p[1 \dots (k-1)] := (\perp, \perp, \dots, \perp)$ 
7        if  $V_p$  has more than  $t$   $\perp$ 's then
8          return( $\perp$ )
        else
9          return( $estimate_p$ )
    end

```

Figure 16:  $t$ -tolerant *gracefully degrading* self-implementation of **consensus** for omission

---

important step in the algorithm and is intuitively justified as follows. Suppose that some  $O_l$  ( $1 \leq l \leq k-1$ ) were correct. By the integrity and agreement property of  $O_l$ , every process would receive the same non- $\perp$  response, call it  $est$ , from  $O_l$ . Thus, every process will have the same estimate  $est$ , at the end of accessing  $O_l$ . Furthermore, since even objects that fail by omission satisfy validity and agreement, if a base object in  $O_{l+1} \dots O_{2t+1}$  returns a non- $\perp$  response, the response must be  $est$ . Thus, we conclude that, if  $O_k$  returns a response in  $\{0, 1\}$  which is different from  $p$ 's current estimate, objects  $O_1, O_2, \dots, O_{k-1}$  are faulty. At the end of accessing all  $2t+1$  base objects, if  $p$  believes that no more than  $t$  base objects failed, it returns its current estimate. Otherwise it returns  $\perp$ .

**Lemma 7.1** *For every  $k$ ,  $1 \leq k \leq 2t+1$ , at the end of the  $k^{th}$  iteration of the **for** loop of **Propose**( $p, v_p, \mathcal{O}$ ) in Figure 16,  $estimate_p \in \{0, 1\}$ , and  $V_p[1..k]$  contains only  $\perp$ 's and  $estimate_p$ 's.*

*Proof* By an easy induction on  $k$ . □

**Theorem 7.1** *Figure 16 presents a  $t$ -tolerant gracefully degrading self-implementation of **consensus** for omission. The resource complexity of the implementation is  $2t+1$ .*

*Proof* Let  $\mathcal{O}$  be a derived object of the implementation, and  $O_1, O_2, \dots, O_{2t+1}$  be its base objects. Consider an execution  $E$  in which all base objects that fail, fail by omission. (Note that we do not restrict the number of base objects that may fail in  $E$ .)



1.  $\mathcal{O}$  is wait-free: Obvious since base objects that fail by omission remain wait-free.
2.  $\mathcal{O}$  satisfies validity: An easy induction on  $k$ , the loop variable in Figure 8, shows that, if  $estimate_p$  equals some value  $u$  at any point in  $E$ , then there is an invocation (from some process  $q$ ) of  $\text{Propose}(q, u, \mathcal{O})$  earlier in  $E$ . The induction will use Proposition 5.2, and the fact that  $p$  does not change  $estimate_p$  if a base object returns  $\perp$ .
3.  $\mathcal{O}$  satisfies agreement: Suppose, for a contradiction, there exist two processes  $p$  and  $q$  such that  $\text{Propose}(p, v_p, \mathcal{O})$  returns 0 and  $\text{Propose}(q, v_q, \mathcal{O})$  returns 1. From Lemma 7.1 and lines 7, 8, and 9 of the algorithm, it follows that  $V_p$  has at least  $t+1$  0's at the end of the execution of  $\text{Propose}(p, v_p, \mathcal{O})$  and  $V_q$  has at least  $t+1$  1's at the end of the execution of  $\text{Propose}(q, v_q, \mathcal{O})$ . This is possible only if there is a  $k$  ( $1 \leq k \leq 2t+1$ ) such that  $\text{propose}(p, estimate_p, O_k)$  returned 0 and  $\text{propose}(q, estimate_q, O_k)$  returned 1. Thus  $O_k$  does not satisfy agreement. By Proposition 5.2, the failure of  $O_k$  in  $E$  is not by omission, a contradiction.
4.  $\mathcal{O}$  satisfies weak integrity: Obvious.
5.  $\mathcal{O}$  satisfies integrity if at most  $t$  base objects fail: Let  $O_{k_1}, O_{k_2}, \dots, O_{k_l}$  ( $k_1 < k_2 < \dots < k_l$ ) be all the correct base objects. Since at most  $t$  fail, we have  $l \geq t+1$ . By Proposition 5.1,  $O_{k_1}$  satisfies integrity and agreement. Thus, there is a  $v \in \{0, 1\}$  such that, for all  $p$ ,  $\text{propose}(p, estimate_p, O_{k_1})$  returns  $v$ . Thus, for all  $p$ ,  $estimate_p = v$  at the end of  $k_1$  iterations of the for-loop in  $\text{Propose}(p, v_p, \mathcal{O})$ . Using this and Proposition 5.2, it is easy to verify that, at the end of the execution of  $\text{Propose}(p, v_p, \mathcal{O})$ ,  $V_p[k_i] = v$  and  $estimate_p = v$  for all  $p$  and for all  $i$ ,  $1 \leq i \leq l$ . This implies, by lines 7, 8 of the algorithm, that  $\text{Propose}(p, v_p, \mathcal{O})$  returns  $v$ .

From 1, 2, 3, and 4 above and Proposition 5.2, we conclude that either  $\mathcal{O}$  is correct in  $E$  or  $\mathcal{O}$  fails by omission in  $E$ . From 1, 2, 3, and 5 above and Proposition 5.1, we conclude that if at most  $t$  base objects of  $\mathcal{O}$  fail in  $E$ ,  $\mathcal{O}$  is correct in  $E$ . Thus, Figure 16 is a  $t$ -tolerant gracefully degrading self-implementation of **consensus** for omission.  $\square$

We now prove a general lower bound on the resource complexity of gracefully degrading implementations of any non-trivial type for omission. Informally, a type is trivial if each operation has a fixed response. More precisely,  $T = (OP, RES, G, \tau)$  is *trivial* if, for all states  $s$  of  $T$ , there is a function  $f : OP \rightarrow RES$  such that for all finite sequences  $op_1, op_2, \dots, op_k$  of operations,  $(op_1, f(op_1)), (op_2, f(op_2)), \dots, (op_k, f(op_k))$  is legal from state  $s$  of  $T$ . A type is *non-trivial* if it is not trivial. The following proposition is immediate.

**Proposition 7.1** *Let  $T = (OP, RES, G, \tau)$  be a deterministic non-trivial type. Then, there exists a state  $s$  of  $T$  and operations  $op_1, op_2 \in OP$  with the following property. Let  $f : OP \rightarrow RES$  be the function such that, for all  $op \in OP$ ,  $(op, f(op))$  is legal from state  $s$ . Then,  $(op_1, f(op_1)), (op_2, f(op_2))$  is not legal from  $s$ .*

For an illustration of the proposition, consider **consensus**, which is clearly a deterministic and non-trivial type. Let  $s$  be the *uncommitted state*, and  $op_1$  and  $op_2$  be *propose*

0 and *propose 1*, respectively. Then, the function  $f$  is as follows:  $f(\text{propose } 0) = 0$  and  $f(\text{propose } 1) = 1$ . Now, as the proposition claims, the sequence  $(\text{propose } 0, f(\text{propose } 0))$ ,  $(\text{propose } 1, f(\text{propose } 1))$  is not legal from the uncommitted state.

**Theorem 7.2** *Let  $T = (\text{OP}, \text{RES}, G, \tau)$  be any deterministic non-trivial type such that, for all sequential histories  $H$ ,  $\tau(H) = H$ . The resource complexity of any  $t$ -tolerant gracefully degrading implementation of  $T$ , for two processes, for omission is at least  $2t + 1$ .*

*Proof* Let  $s, op_1, op_2, f$  be as in Proposition 7.1. Assume that the theorem is false. Then,  $(T, s)$  has a  $t$ -tolerant gracefully degrading implementation  $\mathcal{I}$  from  $(\mathcal{L}, \Sigma)$ , for two processes, for omission, where  $\mathcal{L} = (T_1, T_2, \dots, T_{2t})$  is some list of types and  $\Sigma = (s_1, s_2, \dots, s_{2t})$  is a list of states. Let  $O_1, O_2, \dots, O_{2t}$  be objects of type  $T_1, T_2, \dots, T_{2t}$ , initialized to  $s_1, s_2, \dots, s_{2t}$ , respectively. Let  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots, O_{2t})$  be the derived object of type  $T$ , initialized to state  $s$ . We will describe a scenario  $S$  in which two processes  $P$  and  $Q$  apply operations on the derived object  $\mathcal{O}$ . At the start of Scenario  $S$ , assume that all base objects of  $\mathcal{O}$  fail, as described below.

Objects  $O_i$  ( $1 \leq i \leq t$ ) fail as follows: Whenever  $P$  invokes an operation on  $O_i$ ,  $O_i$  returns a correct response to  $P$  and undergoes an appropriate change of state; but whenever  $Q$  invokes an operation on  $O_i$ ,  $O_i$  returns  $\perp$  and does not undergo any change of state. Objects  $O_j$  ( $t + 1 \leq j \leq 2t$ ) fail in a symmetric manner, as follows: Whenever  $P$  invokes an operation on  $O_j$ ,  $O_j$  returns  $\perp$  and does not undergo any change of state; but whenever  $Q$  invokes an operation on  $O_j$ ,  $O_j$  returns a correct response to  $Q$  and undergoes an appropriate change of state.

#### Scenario S

1. Process  $Q$  applies the operation  $op_1$  on  $\mathcal{O}$ . Let  $v_1$  be the response of  $\mathcal{O}$ .
2. Process  $P$  applies the operation  $op_2$  on  $\mathcal{O}$ .

(When we describe a scenario as above, we mean that all steps in Item 1 strictly precede every step in Item 2.) Note that:

1. The failure of each base object is by omission.
2. The scenario  $S$  is indistinguishable to  $Q$  from a scenario  $S'$  in which  $O_1, O_2, \dots, O_t$  fail exactly as in  $S$ , but  $O_{t+1}, O_{t+2}, \dots, O_{2t}$  are correct. Since  $\mathcal{O}$  is derived from a  $t$ -tolerant implementation, the response of  $\mathcal{O}$  to  $Q$  in  $S'$  must be correct. By definition of  $f$ , it follows that this response is  $f(op_1)$ . Since  $S$  and  $S'$  are indistinguishable to  $Q$ ,  $Q$  returns  $f(op_1)$  as the response of  $\mathcal{O}$  also in  $S$ .
3. When  $P$  applies  $op_2$  on  $\mathcal{O}$  (in Scenario  $S$ ), the manner in which base objects have failed makes it impossible for  $P$  to know whether  $Q$  previously executed any operations on  $\mathcal{O}$ . Thus, Scenario  $S$  is indistinguishable to  $P$  from a scenario  $S''$  in which (i)  $P$  is the first process to invoke an operation on  $\mathcal{O}$ , and (ii) objects  $O_{t+1}, O_{t+2}, \dots, O_{2t}$  fail

exactly as in Scenario  $S$ , but objects  $O_1, O_2, \dots, O_t$  are correct. Since  $\mathcal{O}$  is derived from a  $t$ -tolerant implementation, the response of  $\mathcal{O}$  to  $P$  in  $S''$  must be correct. By definition of  $f$ , it follows that this response is  $f(op_2)$ . Since  $S$  is indistinguishable to  $P$  from  $S''$ ,  $P$  returns  $f(op_2)$  as the response of  $\mathcal{O}$  also in  $S$ .

By Proposition 7.1,  $(op_1, f(op_1)), (op_2, f(op_2))$  is not legal from state  $s$ . So, the history  $H$  of object  $\mathcal{O}$  in Scenario  $S$  is not linearizable with respect to  $(T, s)$ . Since  $H$  is a sequential history, by the premise of the theorem,  $\tau(H) = H$ . Thus,  $\tau(H)$  is not linearizable with respect to  $(T, s)$ . In other words,  $\mathcal{O}$  does not satisfy Property 3 of omission. We conclude that the failure of  $\mathcal{O}$  is not by omission, even though the base objects of  $\mathcal{O}$  have failed only by omission. This implies that  $\mathcal{I}$ , the implementation from which  $\mathcal{O}$  is derived, is not gracefully degrading for omission.  $\square$

**Corollary 7.1** *Let  $\mathcal{I}$  be any  $t$ -tolerant gracefully degrading implementation of consensus, for two processes, for omission. The resource complexity of  $\mathcal{I}$  is at least  $2t + 1$ .*

## 7.2 Feasibility of achieving graceful degradation

In this section, we study the feasibility of achieving gracefully degrading implementations for the crash and omission failure modes. We identify a large class of types and prove that no type in this class has a fault-tolerant gracefully degrading implementation for crash. In contrast, we show that graceful degradation for omission is achievable in a strong sense: every type has a  $t$ -tolerant gracefully degrading implementation from every universal set of types for omission.

### 7.2.1 Graceful degradation for crash

Consider a system that supports a given set  $S$  of “hardware” objects. Assume that these objects may fail but, if they do, they are guaranteed to only fail by crash. Suppose that we wish to implement an object  $\mathcal{O}$  of type  $T$  using objects in  $S$ . We do not require  $\mathcal{O}$  to be fault-tolerant. However, if  $\mathcal{O}$  fails because one or more objects in  $S$  fail by crash, we would like  $\mathcal{O}$  to fail only by crash. This last requirement is desirable for two reasons:

- The benign failure semantics of crash are desirable.
- Such an object  $\mathcal{O}$  appears like any other hardware object of the system. In other words, with this “software implementation” of  $\mathcal{O}$ , the system would be no different, in functionality *and* failure semantics, from one that directly supports the objects in  $S \cup \{\mathcal{O}\}$  in hardware.

In our terminology, we are seeking a gracefully degrading implementation of  $T$  for crash from the types (of the objects) in  $S$ . Unfortunately, as we show shortly, many types do not have such implementations, even from very powerful types. This negative result implies

that, in many cases, the simple and desirable failure semantics of crash cannot be achieved. Our negative result applies to the class of order-sensitive types, defined below.

A type  $T = (OP, RES, G, \tau)$  is *order-sensitive* if it is deterministic,  $\tau$  is the identity, and there is a state  $s$  with the following property. There exist operations  $op, op'$  (not necessarily distinct) in  $OP$  and values  $u, v, u', v'$  in  $RES$  such that each of  $(op, u)$ ,  $(op', u')$  and  $(op', v')$ ,  $(op, v)$  is legal from state  $s$  of  $T$ , and  $u \neq v$  and  $u' \neq v'$ . Intuitively, when an object  $\mathcal{O}$  of type  $T$  is in the state  $s$ , and two processes  $p$  and  $q$  invoke operations  $op$  and  $op'$ , respectively, concurrently on  $\mathcal{O}$ , they can both determine, based on the return values, the order in which their operations are linearized. It is easy to see that every order-sensitive type implements **consensus** for two processes.

**queue** is an example of an order-sensitive type. To see this, let  $s$  be the state in which there are two elements 5 and 10 in the queue (5 at the front), and let both  $op$  and  $op'$  be *deq*. Now we have  $u = 5$ ,  $u' = 10$ ,  $v' = 5$ , and  $v = 10$ . Thus  $u \neq v$  and  $u' \neq v'$ , as required. **compare&swap**, **consensus**, **stack**, and **test&set** are some other examples of order-sensitive types.

A type is *non-order-sensitive* if it is deterministic and is not order-sensitive. Examples of non-order-sensitive types include **register**, **sticky-bit**, **move**, and **m-m swap**. Thus, while every order-sensitive type implements **consensus** for two processes, not every type that implements **consensus** for two processes is order-sensitive. In other words, the set of order-sensitive types is a proper subset of the set of types that implement **consensus** for two processes. (Hereafter we will refer to the latter set as *CONS2*.)

We now present two theorems for crash. To prevent their long proofs from interrupting the flow, we state both theorems and discuss their implications before presenting the proofs.

**Theorem 7.3** *Let  $T$  be any order-sensitive type and  $\mathcal{S}$  be any set of non-order-sensitive types.  $T$  has no gracefully degrading implementation from  $\mathcal{S}$  for crash.*

This negative result is significant in two ways. First, it holds even though we are not requiring the implementation to be fault-tolerant. Second, the set of non-order-sensitive types includes some universal types, such as **sticky-bit**, **move**, and **m-m swap**. The above result holds despite including such powerful types in  $\mathcal{S}$ .

Requiring a derived object to inherit the crash failure semantics of its base objects is even more difficult if we add the requirement that the derived object be 1-tolerant: Even if we do not restrict the types of primitives available in the underlying system, such implementations do not exist for many objects of interest. This is the substance of the next theorem.

**Theorem 7.4** *There is no 1-tolerant gracefully degrading implementation of any order-sensitive type for crash.*

The above two theorems raise serious concerns about the “practicality” of the crash mode: Even if “hardware” objects are designed to fail only by crash, “software” objects

usually don't. The omission mode does not have this severe limitation. In fact, we show in the next subsection that, for any  $t \geq 0$ , *every* type has a  $t$ -tolerant *gracefully degrading* implementation from every universal set of types for omission. In other words, implementations preserving the omission failure semantics of the underlying system always exist. This is a formal justification for adopting the omission failure mode.

We remark that there are no obvious ways to strengthen Theorem 7.4. For instance, consider the statement “There is no 1-tolerant gracefully degrading implementation of any type in *CONS2* for crash”.<sup>12</sup> This statement is false. In fact, even the weaker version “There is no 1-tolerant gracefully degrading implementation of any type in *CONS2* from any set of non-order-sensitive types for crash” does not hold: We can show that **sticky-bit** has a  $t$ -tolerant gracefully degrading implementation from  $\{\mathbf{sticky-bit}, \mathbf{register}\}$  for crash. Since **sticky-bit** belongs to *CONS2*, and both **sticky-bit** and **register** are non-order-sensitive, such an implementation is a counter-example to the above statement. The details of this implementation are long and tedious, and are therefore omitted.

We now end Section 7.2.1 with the proofs of Theorems 7.3 and 7.4.

### Proof of Theorem 7.3

Suppose that the theorem is false. Then, there is an order-sensitive type  $T$  which has a gracefully degrading implementation from some set of non-order-sensitive types for crash. For type  $T$ , let  $op, op', s, u, v, u', v'$  be as in the definition of an order-sensitive type. It follows that there is a list  $\mathcal{L} = (T_1, T_2, \dots, T_n)$  of non-order-sensitive types and a list  $\Sigma = (s_1, s_2, \dots, s_n)$  of states ( $s_i$  is a state of  $T_i$ ) such that  $(T, s)$  has a gracefully degrading implementation  $\mathcal{I}$  from  $(\mathcal{L}, \Sigma)$  for crash. We arrive at a contradiction after a series of lemmas involving bivalency arguments [FLP85] and indistinguishable scenarios.

Let  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots, O_n)$ , where  $O_1, O_2, \dots, O_n$  are objects of type  $T_1, T_2, \dots, T_n$ , initialized to states  $s_1, s_2, \dots, s_n$ , respectively. Thus,  $\mathcal{O}$  is a (derived) object of type  $T$ , initialized to state  $s$ . Consider the concurrent system consisting of processes  $p, q$  and the object  $\mathcal{O}$ . In the following, we will refer to a state of the concurrent system as a *configuration*. Let  $C_0$  denote a configuration in which  $\mathcal{O}$  is in state  $s$  and processes  $p, q$  are about to execute  $\mathbf{Apply}(p, op, \mathcal{O})$  and  $\mathbf{Apply}(q, op', \mathcal{O})$ , respectively.

**Lemma 7.2** *Suppose all base objects are correct. For any interleaving of the steps in the complete executions of  $\mathbf{Apply}(p, op, \mathcal{O})$  and  $\mathbf{Apply}(q, op', \mathcal{O})$ , either  $\mathbf{Apply}(p, op, \mathcal{O})$  returns  $u$  and  $\mathbf{Apply}(q, op', \mathcal{O})$  returns  $u'$ , or  $\mathbf{Apply}(p, op, \mathcal{O})$  returns  $v$  and  $\mathbf{Apply}(q, op', \mathcal{O})$  returns  $v'$ .*

*Proof* In the linearization of the history of object  $\mathcal{O}$ , either  $\mathbf{Apply}(p, op, \mathcal{O})$  immediately precedes  $\mathbf{Apply}(q, op', \mathcal{O})$ , or  $\mathbf{Apply}(q, op', \mathcal{O})$  immediately precedes  $\mathbf{Apply}(p, op, \mathcal{O})$ . This, together with the definitions of  $u, u', v, v'$ , and the fact that  $T$  is a deterministic type, implies the lemma.  $\square$

<sup>12</sup>This statement is stronger than Theorem 7.4 since, as remarked earlier, the set of order-sensitive types is a proper subset of *CONS2*.

Let  $C$  denote a configuration reached from  $C_0$  after some interleaving of (partial) executions of  $\text{Apply}(p, op, \mathcal{O})$  and  $\text{Apply}(q, op', \mathcal{O})$ . We say  $C$  is  $X$ -valent if, in the absence of base object failures,  $\text{Apply}(p, op, \mathcal{O})$  returns  $X$ , no matter how the steps of  $\text{Apply}(p, op, \mathcal{O})$  and  $\text{Apply}(q, op', \mathcal{O})$  interleave when execution resumes from  $C$ . By Lemma 7.2, if  $C$  is  $X$ -valent, either  $X = u$  or  $X = v$ .  $C$  is *monovalent* if  $C$  is either  $u$ -valent or  $v$ -valent.  $C$  is *bivalent* if it is neither  $u$ -valent nor  $v$ -valent.

**Lemma 7.3**  $C_0$  is bivalent.

*Proof* Starting from  $C_0$ , if  $p$  completes all the steps of  $\text{Apply}(p, op, \mathcal{O})$  before  $q$  starts  $\text{Apply}(q, op', \mathcal{O})$ , then  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$ . Thus  $C_0$  is not  $v$ -valent.

Similarly, starting from  $C_0$ , if  $q$  completes all the steps of  $\text{Apply}(q, op', \mathcal{O})$  before  $p$  starts  $\text{Apply}(p, op, \mathcal{O})$ , then  $\text{Apply}(q, op', \mathcal{O})$  returns  $v$ . Thus, by Lemma 7.2, when  $\text{Apply}(p, op, \mathcal{O})$  completes, it returns  $v$ . Thus  $C_0$  is not  $u$ -valent.

Since  $C_0$  is neither  $u$ -valent nor  $v$ -valent, it is bivalent.  $\square$

We say  $C'$  is a *reachable configuration* from  $C$  if, starting from the configuration  $C$ , there is some interleaving of the steps of  $p$  and  $q$  such that  $C'$  is the configuration at the end of that interleaving. Given a configuration  $C$ , let  $C(p)$  denote the configuration that results when  $p$  takes a single step of  $\text{Apply}(p, op, \mathcal{O})$  from  $C$ .  $C(q)$  is similarly defined.

**Lemma 7.4** There is a bivalent configuration  $C_{crit}$  reachable from  $C_0$  such that  $C_{crit}(p)$  and  $C_{crit}(q)$  are both monovalent.

*Proof* Interleave the steps of  $\text{Apply}(p, op, \mathcal{O})$  and  $\text{Apply}(q, op', \mathcal{O})$  as shown in Figure 17. Since  $\mathcal{O}$  is wait-free, the **repeat** ... **until** loop in the figure must terminate after a finite number of iterations. Let  $C_{crit}$  be the value of  $C$  just when the loop terminates. It is easy to verify that  $C_{crit}$  satisfies the properties required by the lemma.  $\square$

---

```

 $C := C_0$ 
repeat
  if  $C(p)$  is bivalent then
     $C := C(p)$ 
  if  $C(q)$  is bivalent then
     $C := C(q)$ 
until  $(C(p) \text{ is monovalent}) \wedge (C(q) \text{ is monovalent})$ 

```

Figure 17: Reaching a *critical* bivalent configuration

---

Since  $C_{crit}$  is bivalent,  $C_{crit}(p)$  and  $C_{crit}(q)$  cannot both be  $X$ -valent for the same  $X$ . Thus, either  $C_{crit}(p)$  is  $u$ -valent and  $C_{crit}(q)$  is  $v$ -valent, or  $C_{crit}(p)$  is  $v$ -valent and  $C_{crit}(q)$  is  $u$ -valent. Without loss of generality, we will assume the former.

**Lemma 7.5** *The enabled steps of  $p$  and  $q$  in  $C_{crit}$  access the same base object.*

*Proof* Suppose not. Then  $(C_{crit}(p))(q)$  and  $(C_{crit}(q))(p)$  are identical configurations, and yet, the former is  $u$ -valent and the latter  $v$ -valent. This is impossible since  $u \neq v$ .  $\square$

Assume that  $O_k$  is the base object mentioned in the above lemma, and  $\text{Apply}(p, oper, O_k)$ ,  $\text{Apply}(q, oper', O_k)$  are the enabled steps of  $p$  and  $q$  respectively in  $C_{crit}$ . Since  $O_k$  is an object of a non-order-sensitive type, either  $\text{Apply}(q, oper', O_k)$  returns the same value whether applied in  $C_{crit}$  or  $C_{crit}(p)$ , or  $\text{Apply}(p, oper, O_k)$  returns the same value whether applied in  $C_{crit}$  or  $C_{crit}(q)$ . In the following, we will deal with the former case. The latter case can be handled similarly and is omitted.

**Lemma 7.6** *Consider*

**Scenario S1** (Starts from the configuration  $C_{crit}$ )

1. *Process  $q$  takes the step  $\text{Apply}(q, oper', O_k)$ .*
2. *Process  $p$  completes the execution of  $\text{Apply}(p, op, \mathcal{O})$ .*
3. *All base objects  $O_1, O_2, \dots, O_n$  fail by crash.*
4. *Process  $q$  resumes and completes the execution of  $\text{Apply}(q, op', \mathcal{O})$ .*

*Then  $\text{Apply}(p, op, \mathcal{O})$  returns  $v$  and  $\text{Apply}(q, op', \mathcal{O})$  returns  $v'$ .*

*Proof* Since  $q$  takes the step from  $C_{crit}$ , and  $C_{crit}(q)$  is  $v$ -valent, and no base object failures occur before  $p$  completes the execution of  $\text{Apply}(p, op, \mathcal{O})$  in Item 2,  $\text{Apply}(p, op, \mathcal{O})$  returns  $v$  in Item 2 of the scenario.

Suppose  $\text{Apply}(q, op', \mathcal{O})$  returns  $\perp$ . Since  $\mathcal{I}$  is gracefully degrading,  $\mathcal{O}$  must either be correct or fail by crash. Given that  $\text{Apply}(p, op, \mathcal{O})$  returns a non- $\perp$  response, this requires that  $\text{Apply}(p, op, \mathcal{O})$  precedes  $\text{Apply}(q, op', \mathcal{O})$  in the linearization order. Doing so, however, implies that  $(op, v)$  is legal from state  $s$  of  $T$ . This is false since  $(op, u)$  is the only sequence legal from state  $s$  of  $T$ , and  $v \neq u$ . Thus  $\text{Apply}(q, op', \mathcal{O})$  cannot return  $\perp$ .

Suppose  $\text{Apply}(q, op', \mathcal{O})$  returns  $w$ , where  $\perp \neq w \neq v'$ . Since in the linearization, either  $\text{Apply}(p, op, \mathcal{O})$  precedes  $\text{Apply}(q, op', \mathcal{O})$ , or  $\text{Apply}(q, op', \mathcal{O})$  precedes  $\text{Apply}(p, op, \mathcal{O})$ , it follows that either  $(op, v), (op', w)$  or  $(op', w), (op, v)$  is legal from state  $s$  of  $T$ . This is false since  $(op, u), (op', u')$  and  $(op', v'), (op, v)$  are the only sequences legal from state  $s$  of  $T$ , and  $u \neq v, w \neq v' \neq v$ .

We conclude that  $\text{Apply}(q, op', \mathcal{O})$  must return  $v'$ .  $\square$

**Lemma 7.7** *Consider*

**Scenario S2** (Starts from the configuration  $C_{crit}$ )

1. *Process  $p$  takes the step  $\text{Apply}(p, oper, O_k)$ .*

2. Process  $q$  takes the step  $\text{Apply}(q, \text{oper}', O_k)$ .
3. Process  $p$  resumes and completes the execution of  $\text{Apply}(p, \text{op}, \mathcal{O})$ .
4. All base objects  $O_1, O_2, \dots, O_n$  fail by crash.
5. Process  $q$  resumes and completes the execution of  $\text{Apply}(q, \text{op}', \mathcal{O})$ .

Then  $\text{Apply}(p, \text{op}, \mathcal{O})$  returns  $u$  and  $\text{Apply}(q, \text{op}', \mathcal{O})$  returns  $v'$ .

*Proof* Since  $p$  takes the step from  $C_{crit}$ ,  $C_{crit}(p)$  is  $u$ -valent, and no base object failures occur before  $p$  completes the execution of  $\text{Apply}(p, \text{op}, \mathcal{O})$  in Item 3,  $\text{Apply}(p, \text{op}, \mathcal{O})$  returns  $u$  in Item 3 of the scenario. Since  $S2 \approx_q S1$ ,  $\text{Apply}(q, \text{op}', \mathcal{O})$  returns  $v'$  as in  $S1$ .  $\square$

Neither  $(\text{op}, u), (\text{op}', v')$  nor  $(\text{op}', v'), (\text{op}, u)$  is legal from state  $s$  of  $T$ . Hence, the execution in Lemma 7.7 is not linearizable. Thus, the failure of  $\mathcal{O}$  in  $S2$  is not by crash. We conclude that  $\mathcal{I}$  is not a gracefully degrading implementation for crash, a contradiction. This concludes the proof of Theorem 7.3.  $\square$

#### Proof of Theorem 7.4

Suppose that the theorem is false. Then, there is an order-sensitive type  $T$  which has a 1-tolerant gracefully degrading implementation for crash. For type  $T$ , let  $\text{op}, \text{op}', s, u, v, u', v'$  be as in the definition of an order-sensitive type. It follows that there is a list  $\mathcal{L} = (T_1, T_2, \dots, T_n)$  of types and a list  $\Delta = (s_1, s_2, \dots, s_n)$  of states ( $s_i$  is a state of  $T_i$ ) such that  $(T, s)$  has a 1-tolerant gracefully degrading implementation  $\mathcal{I}$  from  $(\mathcal{L}, \Delta)$  for crash. We arrive at a contradiction after a series of lemmas involving indistinguishable scenarios.

Let  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots, O_n)$ , where  $O_1, O_2, \dots, O_n$  are objects of type  $T_1, T_2, \dots, T_n$ , initialized to states  $s_1, s_2, \dots, s_n$ , respectively. Thus,  $\mathcal{O}$  is a (derived) object of type  $T$ , initialized to state  $s$ . Consider the concurrent system consisting of processes  $p, q$  and the object  $\mathcal{O}$ . Suppose that  $\mathcal{O}$  is in state  $s$ , and  $p, q$  are about to execute  $\text{Apply}(p, \text{op}, \mathcal{O})$  and  $\text{Apply}(q, \text{op}', \mathcal{O})$ , respectively.

**Lemma 7.8** *Suppose all base objects are correct. For any interleaving of the steps in the complete executions of  $\text{Apply}(p, \text{op}, \mathcal{O})$  and  $\text{Apply}(q, \text{op}', \mathcal{O})$ , either  $\text{Apply}(p, \text{op}, \mathcal{O})$  returns  $u$  and  $\text{Apply}(q, \text{op}', \mathcal{O})$  returns  $u'$ , or  $\text{Apply}(p, \text{op}, \mathcal{O})$  returns  $v$  and  $\text{Apply}(q, \text{op}', \mathcal{O})$  returns  $v'$ .*

*Proof* Same as Lemma 7.2.  $\square$

**Lemma 7.9** *There exists a (possibly empty) sequence  $\Sigma$  of steps of  $p$  and a step  $\sigma$  of  $p$  such that the following Scenarios  $S1$  and  $S2$  are possible.*

**Scenario  $S1$**  (scenario starts with  $\mathcal{O}$  in state  $s$ )

1. Process  $p$  initiates and partially executes  $\text{Apply}(p, \text{op}, \mathcal{O})$  by completing the steps in  $\Sigma$ .



2. Process  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ , returning  $v'$ .
3.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ , returning  $v$ .

**Scenario S2** (scenario starts with  $\mathcal{O}$  in state  $s$ )

1.  $p$  initiates and (partially) executes  $\text{Apply}(p, op, \mathcal{O})$  by completing the steps in  $\Sigma \cdot \sigma$ .
2.  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ , returning  $u'$ .
3.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ , returning  $u$ .

*Proof* Clearly, if process  $p$  executes no steps of  $\text{Apply}(p, op, \mathcal{O})$  before process  $q$  initiates and completes  $\text{Apply}(q, op', \mathcal{O})$ , then  $\text{Apply}(q, op', \mathcal{O})$  must return  $v'$ . Further, if  $p$  initiates and completes all the steps of  $\text{Apply}(p, op, \mathcal{O})$  (let  $\Gamma$  be this sequence of steps) before  $q$  initiates and completes  $\text{Apply}(q, op', \mathcal{O})$ , then  $\text{Apply}(q, op', \mathcal{O})$  must return  $u'$ . Together with Lemma 7.8 by which  $\text{Apply}(q, op', \mathcal{O})$  must return either  $u'$  or  $v'$ , the above implies that there exists a sequence  $\Sigma$  of steps and a step  $\sigma$  such that  $\Sigma \cdot \sigma$  is a prefix of  $\Gamma$  for which the lemma holds.  $\square$

Hereafter we will assume  $O_k$  is the base object accessed by  $p$  in step  $\sigma$ .

**Lemma 7.10** *Consider*

**Scenario S3** (scenario starts with  $\mathcal{O}$  in state  $s$ )

1.  $p$  initiates and (partially) executes  $\text{Apply}(p, op, \mathcal{O})$  by completing the steps in  $\Sigma \cdot \sigma$ .
2.  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ , returning  $u'$  (as in S2).
3.  $O_1, O_2, \dots, O_n$  fail by crash.
4.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ .

Then  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$ .

*Proof* Suppose  $\text{Apply}(p, op, \mathcal{O})$  returns  $\perp$ . Since  $\mathcal{I}$  is gracefully degrading,  $\mathcal{O}$  must either be correct or fail by crash. This requires, given that  $\text{Apply}(q, op', \mathcal{O})$  returns a non- $\perp$  response, that  $\text{Apply}(q, op', \mathcal{O})$  precedes  $\text{Apply}(p, op, \mathcal{O})$  in the linearization order. Doing so, however, implies that  $(op', u')$  is legal from state  $s$  of  $T$ . This is false since  $u' \neq v'$ ,  $T$  is deterministic, and  $(op', v')$  is legal from state  $s$  of  $T$ . Thus  $\text{Apply}(p, op, \mathcal{O})$  cannot return  $\perp$ .

Suppose  $\text{Apply}(p, op, \mathcal{O})$  returns  $w$  where  $\perp \neq w \neq u$ . Since in the linearization, either  $\text{Apply}(p, op, \mathcal{O})$  precedes  $\text{Apply}(q, op', \mathcal{O})$  or  $\text{Apply}(q, op', \mathcal{O})$  precedes  $\text{Apply}(p, op, \mathcal{O})$ , it follows that either  $(op, w), (op', u')$  or  $(op', u'), (op, w)$  is legal from state  $s$  of  $T$ . This is false since  $(op, u), (op', u')$  and  $(op', v'), (op, v)$  are the only sequences legal from state  $s$  of  $T$ , and  $w \neq u$ ,  $u' \neq v'$ .

We conclude that  $\text{Apply}(p, op, \mathcal{O})$  must return  $u$ .  $\square$

**Lemma 7.11** *Consider*

Scenario S4 (scenario starts with  $\mathcal{O}$  in state  $s$ )

1.  $p$  initiates and (partially) executes  $\text{Apply}(p, op, \mathcal{O})$  by completing the steps in  $\Sigma \cdot \sigma$ .
2.  $O_k$  fails by crash.
3.  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ .
4.  $O_1, \dots, O_{k-1}$  and  $O_{k+1}, \dots, O_n$  also fail by crash.
5.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ .

Then  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$  and  $\text{Apply}(q, op', \mathcal{O})$  returns  $u'$ .

*Proof* Clearly,  $S4 \approx_p S3$ . Therefore, as in S3,  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$  in S4. Since  $\mathcal{I}$  is 1-tolerant, and since only  $O_k$  has failed by the completion of  $\text{Apply}(q, op', \mathcal{O})$ ,  $\text{Apply}(q, op', \mathcal{O})$  must return a non- $\perp$  response. From the definitions of  $u, u', v$ , and  $v'$ , it is easy to verify that the only non- $\perp$  response that satisfies linearizability is  $u'$ .  $\square$

**Lemma 7.12** *Consider*

Scenario S5 (scenario starts with  $\mathcal{O}$  in state  $s$ )

1.  $p$  initiates and partially executes  $\text{Apply}(p, op, \mathcal{O})$  by completing the steps in  $\Sigma$ .
2.  $O_k$  fails by crash.
3.  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ .
4.  $O_1, \dots, O_{k-1}$  and  $O_{k+1}, \dots, O_n$  also fail by crash.
5.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ .

Then  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$ .

*Proof* Clearly  $S5 \approx_q S4$ . Therefore  $\text{Apply}(q, op', \mathcal{O})$  returns  $u'$  as in S4. By arguments similar to those in Lemma 7.10, it can be shown that  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$ .  $\square$

**Lemma 7.13** *Consider*

Scenario S6 (scenario starts with  $\mathcal{O}$  in state  $s$ )

1.  $p$  initiates and partially executes  $\text{Apply}(p, op, \mathcal{O})$  by completing the steps in  $\Sigma$ .
2.  $q$  initiates and completes (all the steps of)  $\text{Apply}(q, op', \mathcal{O})$ .
3. All base objects  $O_1, O_2, \dots, O_n$  fail by crash.

4.  $p$  completes the remaining steps of  $\text{Apply}(p, op, \mathcal{O})$ .

Then  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$ , and  $\text{Apply}(q, op', \mathcal{O})$  returns  $v'$ .

*Proof* Since  $S6 \approx_p S5$ ,  $\text{Apply}(p, op, \mathcal{O})$  returns  $u$  as in  $S5$ . Since  $S6 \approx_q S1$ ,  $\text{Apply}(q, op', \mathcal{O})$  returns  $v'$  as in  $S1$ .  $\square$

Neither  $(op, u), (op', v')$  nor  $(op', v'), (op, u)$  is legal from state  $s$  of  $T$ . Hence the execution in Lemma 7.13 is not linearizable. Thus the failure of  $\mathcal{O}$  in  $S6$  is not by crash. We conclude that  $\mathcal{I}$  is not a gracefully degrading implementation for crash, a contradiction which concludes the proof of Theorem 7.4.  $\square$

### 7.2.2 Graceful degradation for omission

In this subsection, we study the feasibility of achieving gracefully degrading implementations for omission. In this subsection, if we make a statement and omit to mention the failure mode in consideration, the failure mode is understood to be omission.

A set  $\mathcal{S}$  of types is *universal* if every type has an implementation from  $\mathcal{S}$ . An example of such a set is  $\{\text{consensus with safe-reset}, \text{register}\}$  [Her91b]. The main result of this section is the *graceful degradation theorem for omission*, stated as follows: Every type has a  $t$ -tolerant gracefully degrading implementation from every universal set of types for omission. We prove this result through three key lemmas. Below, we list these lemmas and explain how they are used in proving the main result.

- *Lemma 7.14* Every 0-tolerant implementation can be transformed into a 0-tolerant implementation which is gracefully degrading for omission.
- *Lemma 7.18* **register** has a  $t$ -tolerant gracefully degrading self-implementation for omission.
- *Lemma 7.19* **consensus with safe-reset** has a  $t$ -tolerant gracefully degrading implementation from  $\{\text{consensus with safe-reset}, \text{register}\}$  for omission.

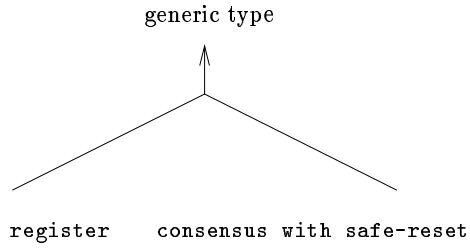
We now explain the steps involved in obtaining the graceful degradation theorem for omission. Figures 18 and 19 depict these steps.

Step 1. Every type has a 0-tolerant implementation from  $\{\text{register}, \text{consensus with safe-reset}\}$ .

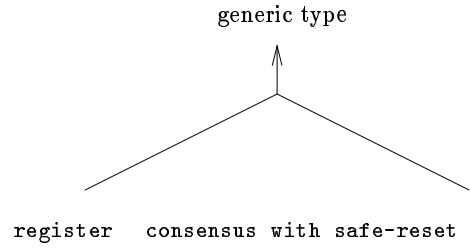
This follows from Herlihy's universality result [Her91b].

Step 2. Every type has a 0-tolerant gracefully degrading implementation from  $\{\text{register}, \text{consensus with safe-reset}\}$ .

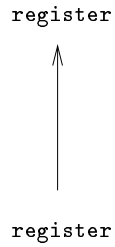
This follows from Step 1 and Lemma 7.14.



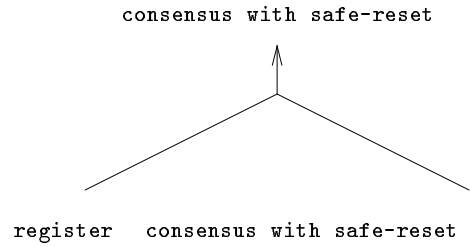
Step 1: 0-tolerant



Step 2: 0-tolerant gracefully degrading



Step 3:  $t$ -tolerant gracefully degrading



Step 4:  $t$ -tolerant gracefully degrading

**imply**

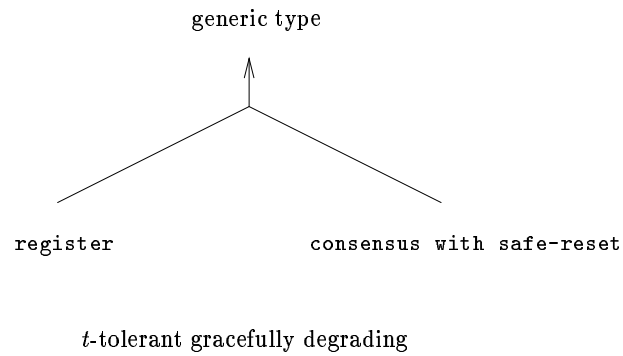


Figure 18: First steps in the derivation of the graceful degradation theorem for omission

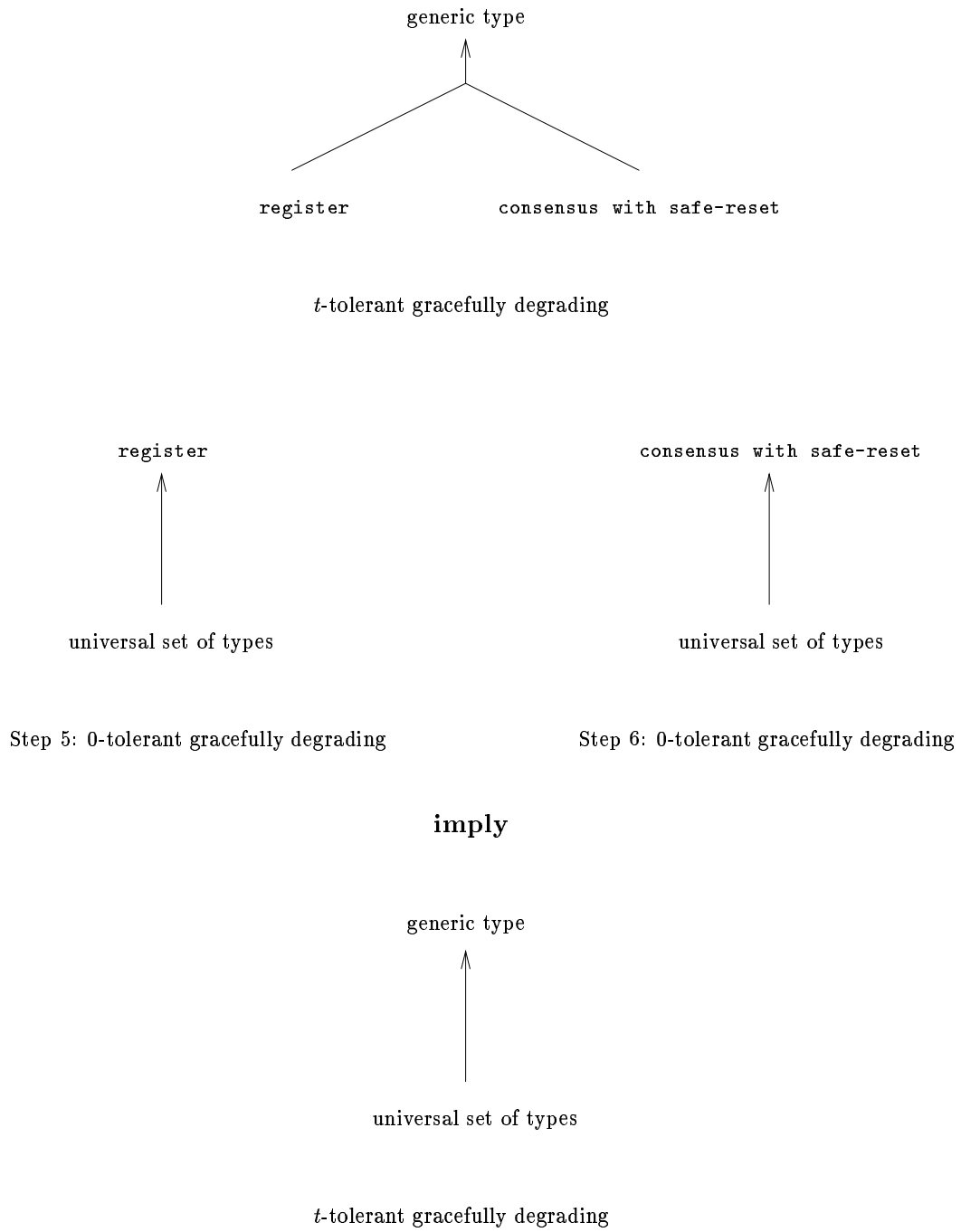


Figure 19: Later steps in the derivation of the graceful degradation theorem for omission

Step 3. **register** has a  $t$ -tolerant gracefully degrading self-implementation. This is Lemma 7.18.

Step 4. **consensus with safe-reset** has a  $t$ -tolerant gracefully degrading implementation from **{register, consensus with safe-reset}**. This is Lemma 7.19.

From Steps 2, 3, and 4, and Corollary 4.1, we conclude that every type has a  $t$ -tolerant gracefully degrading implementation from **{register, consensus with safe-reset}** for omission. From this conclusion, Steps 5 and 6 below, and the compositional lemma (Lemma 4.1), we have the main theorem: Every type has a  $t$ -tolerant gracefully degrading implementation from every universal list of types for omission.

Step 5. **register** has a 0-tolerant gracefully degrading implementation from any universal set of types.

By definition of a universal set of types, **register** has a 0-tolerant implementation from such a set. This, together with Lemma 7.14, implies Step 5.

Step 6. **consensus with safe-reset** has a 0-tolerant gracefully degrading implementation from any universal set of types.

The reasoning is the same as for Step 5.

We now prove the three lemmas mentioned above.

## A transformation to realize graceful degradation

We present a transformation  $\mathcal{G}$  such that if  $\mathcal{I}$  is any 0-tolerant implementation, then  $\mathcal{G}(\mathcal{I})$  is a 0-tolerant implementation which is gracefully degrading for omission. For all implementations  $\mathcal{I}$ ,  $\mathcal{G}(\mathcal{I})$  is obtained as follows. Let  $\mathcal{O}$  be a derived object of  $\mathcal{G}(\mathcal{I})$ . A process  $P$  applies an operation  $op$  on  $\mathcal{O}$  as in the implementation  $\mathcal{I}$ . However, as  $P$  executes the procedure to apply  $op$  on  $\mathcal{O}$ , if some base object of  $\mathcal{O}$  returns  $\perp$  to  $P$ ,  $P$  immediately terminates its operation on  $\mathcal{O}$  and returns  $\perp$  as the response of  $\mathcal{O}$  to  $op$ .

**Lemma 7.14** *Let  $T$  be a type,  $s$  be a state of  $T$ , and  $\mathcal{I}$  be a 0-tolerant implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$ , for processes  $P_1, \dots, P_N$ . Then,  $\mathcal{G}(\mathcal{I})$  is a 0-tolerant gracefully degrading implementation of  $(T, s)$  from  $(\mathcal{L}, \Sigma)$ , for processes  $P_1, \dots, P_N$ , for omission.*

*Proof Sketch* In the absence of base object failures, it is obvious that a derived object of  $\mathcal{G}(\mathcal{I})$  behaves identically as a derived object of  $\mathcal{I}$ . Since  $\mathcal{I}$  is a 0-tolerant implementation of  $(T, s)$ , it follows that  $\mathcal{G}(\mathcal{I})$  is also a 0-tolerant implementation of  $(T, s)$ . We now show that  $\mathcal{G}(\mathcal{I})$  is gracefully degrading for omission. In the following, let  $T = (OP, RES, G, \tau)$ .

Let  $\mathcal{O}$  be a derived object of  $\mathcal{G}(\mathcal{I})$ . Let  $E$  be an execution of  $(P_1, \dots, P_N; \mathcal{O})$  in which (i) one or more base objects of  $\mathcal{O}$  fail, (ii) each base object that fails, fails by omission, and (iii) if a process gets the response  $\perp$  from  $\mathcal{O}$ , that process does not subsequently invoke an

operation on  $\mathcal{O}$ . We claim that if  $\mathcal{O}$  fails in  $E$ , it fails by omission. This claim implies that  $\mathcal{G}(\mathcal{I})$  is gracefully degrading for omission. To prove the claim, we must show that all three properties stated in the definition of omission hold for  $\mathcal{O}$  in the execution  $E$ . Property 2, that every response of  $\mathcal{O}$  is from  $RES \cup \{\perp\}$ , is obvious. We verify Properties 1 and 3 below.

Let  $H(E)$  denote the history in execution  $E$ . Let  $H_{proc} = H(E)|\{P_1, \dots, P_N\}$ , the subsequence of  $H(E)$  consisting of the events of processes. Thus,  $H_{proc}$  contains the internal events of processes, invocations of processes on  $\mathcal{O}$  and on the base objects of  $\mathcal{O}$ , and the responses from  $\mathcal{O}$  and from the base objects of  $\mathcal{O}$ .<sup>13</sup> Construct a sequence  $H'_{proc}$  from  $H_{proc}$  as follows: for all response events  $e$  which correspond to a base object  $O$  returning  $\perp$  to a process  $P$ , replace  $e$  with  $Crash(P)$  and remove all events of  $P$  following  $e$ . Intuitively, by transforming  $H_{proc}$  to  $H'_{proc}$ , we “shift the blame” from the base object  $O$ , by stopping  $O$  from returning  $\perp$  to  $P$ , to the process  $P$ , by crashing  $P$  after  $P$ ’s invocation on  $O$ . We claim that there exists an execution  $E'$  of  $(P_1, \dots, P_N; \mathcal{O})$  such that  $H'_{proc} = H(E')|\{P_1, \dots, P_N\}$ . (We leave the proof of this claim to the reader.)

We make two claims below which, together, imply that each base object of  $\mathcal{O}$  is correct in the execution  $E'$ . The justification of each claim follows its statement. We write  $H(E, O)$  to denote the subsequence of events in  $E$ , consisting of only invocations on  $O$  and responses from  $O$ .

- Each base object  $O$  is well-behaved in  $E'$ .

We assumed earlier that either  $O$  is correct in  $E$  or  $O$  fails by omission in  $E$ . Suppose that  $O$  is correct in  $E$ . Then, from the definition of  $E'$ ,  $H(E, O) = H(E', O)$ . Thus,  $O$  is correct also in  $E'$ . In particular,  $O$  is well-behaved in  $E'$ .

Suppose that  $O$  fails by omission in  $E$ . Let  $H'(E, O)$  be the history obtained by removing response events associated with the aborted operations in  $H(E, O)$ . By Property 3 of omission,  $\tau(H'(E, O))$  is linearizable with respect to  $(T', s')$ , where  $T'$  is the type of  $O$  and  $s'$  is the state of  $T'$  to which  $O$  was initialized. From the definition of  $E'$ , observe that  $H(E', O) = H'(E, O)$ . It follows that  $\tau(H(E', O))$  is also linearizable with respect to  $(T', s')$ . That is,  $O$  is well-behaved in  $E'$ .

- Each base object  $O$  is wait-free in  $E'$ .

We assumed earlier that either  $O$  is correct in  $E$  or  $O$  fails by omission in  $E$ . Suppose that  $O$  is correct in  $E$ . Then, from the definition of  $E'$ ,  $H(E, O) = H(E', O)$ . Thus,  $O$  is correct also in  $E'$ . In particular,  $O$  is wait-free in  $E'$ .

Suppose that  $O$  fails by omission in  $E$ . By Property 1 of omission,  $O$  is wait-free in  $E$ . From the definition of  $E'$ , observe that if  $O$  responds to an invocation by a process  $P$  in  $E$ , but does not respond to the corresponding invocation by process  $P$  in  $E'$ , then  $P$  is crashed in  $E'$ . From the above, we conclude that  $O$  is wait-free in  $E'$ .

---

<sup>13</sup> Recall that  $\mathcal{O} = (F_1, \dots, F_N; O_1, \dots, O_M)$  where  $F_1, \dots, F_N$  are the front-ends and  $O_1, \dots, O_M$  are the base objects of  $\mathcal{O}$ . Thus, strictly speaking, if  $H_{proc} = H_E|\{P_1, \dots, P_N\}$ ,  $H_{proc}$  does not contain invocations on  $O_i$ ’s or responses from  $O_i$ ’s. However, in this proof sketch, we will refer to the events of  $F_i$  as the events of  $P_i$ . Thus,  $H_{proc}$  contains the events of  $P_i$ ’s and also the events of  $F_i$ ’s.

Thus, all base objects are correct in  $E'$ . It follows that  $\mathcal{O}$  is correct in  $E'$ . In particular,  $\mathcal{O}$  is wait-free and well-behaved in  $E'$ .

We now argue that  $\mathcal{O}$  is wait-free in  $E$ . Assume, for a contradiction, that it is not. Then,  $E$  is infinite and there is a process  $P$  such that  $P$  is correct in  $E$  and  $P$  has an incomplete operation on  $\mathcal{O}$  in  $E$ . We claim that, in  $E$ ,  $P$  did not receive the response  $\perp$  from any base object of  $\mathcal{O}$ . Because, if it did,  $P$  would return  $\perp$  as the response of  $\mathcal{O}$  and would not subsequently invoke an operation on  $\mathcal{O}$ ; thus,  $P$  would have no incomplete operation on  $\mathcal{O}$  in  $E$ , a contradiction. Thus, in  $E$ ,  $P$  is correct,  $P$  never receives  $\perp$  from any base object of  $\mathcal{O}$ , and  $P$  has an incomplete operation on  $\mathcal{O}$ . From this and the definition of  $E'$ ,  $P$  is correct in  $E'$  and  $P$  has an incomplete operation on  $\mathcal{O}$  in  $E'$ . Furthermore, since  $E$  is infinite, so is  $E'$ . The above two facts imply that  $\mathcal{O}$  is not wait-free in  $E'$ . This contradicts the conclusion reached in the previous paragraph. Thus,  $\mathcal{O}$  is wait-free in  $E$  and, consequently, Property 1 of omission holds for  $\mathcal{O}$  in  $E$ .

Let  $H'(E, \mathcal{O})$  be the history obtained by removing response events associated with the aborted operations in  $H(E, \mathcal{O})$ . From the definition of  $E'$ , observe that  $H(E', \mathcal{O}) = H'(E, \mathcal{O})$ . We already concluded that  $\mathcal{O}$  is well-behaved in  $E'$ ; that is,  $\tau(H(E', \mathcal{O}))$  is linearizable with respect to  $(T, s)$ . It follows that  $\tau(H'(E, \mathcal{O}))$  is also linearizable with respect to  $(T, s)$ . The latter implies that Property 3 of omission holds for  $\mathcal{O}$  in  $E$ . This completes the proof of the lemma.  $\square$

## Graceful degradation for register

We show that **register** has a  $t$ -tolerant gracefully degrading self-implementation for omission. The following are the steps involved.

- S1.** We present a 1-tolerant gracefully degrading self-implementation of **1-reader 1-writer safe register**.
- S2.** As mentioned before, it is known that there is a 0-tolerant implementation of **register** from **1-reader 1-writer safe register**. It follows from Lemma 7.14 that there is a 0-tolerant gracefully degrading implementation of **register** from **1-reader 1-writer safe register**.
- S3.** Combining the results in Steps **S1** and **S2** with Corollary 4.1, we obtain a 1-tolerant gracefully degrading self-implementation of **register**. By Booster Lemma, this can be turned into a  $t$ -tolerant gracefully degrading self-implementation of **register**.

Figure 20 presents a 1-tolerant gracefully degrading self-implementation of **1-reader 1-writer safe register**. The implementation uses four base registers. The reader process  $P_r$  maintains a local variable  $FAILED_r$  to remember the faulty base registers it has so far encountered. The writer process  $P_w$  similarly maintains  $FAILED_w$ . To read the derived register,  $P_r$  reads each base register that has so far not appeared faulty to it. It adds base registers that return  $\perp$  to the set  $FAILED_r$  and collects the responses from other base registers in the multi-set  $ValuesRead$ . If, at the end,  $P_r$  has detected two or more base registers



---

$R_1, R_2, R_3, R_4$ : 1-reader 1-writer safe register, initialized to  
the same value as the initial value of the derived register  
 $FAILED_w$ : set, local to the writer process  $P_w$ , initialized to  $\emptyset$   
 $FAILED_r$ : set, local to the reader process  $P_r$ , initialized to  $\emptyset$   
 $ValuesRead$ : multi-set, local to  $P_r$

Apply( $P_r, read, \mathcal{R}$ )

```

ValuesRead :=  $\emptyset$ 
for  $i := 1$  to 4
  if  $R_i \notin FAILED_r$  then
    resp := read( $P_r, R_i$ )
    if resp =  $\perp$  then
      FAILED_r := FAILED_r  $\cup$  { $R_i$ }
    else ValuesRead := ValuesRead  $\cup$  {resp}
if |FAILED_r|  $\geq 2$  then
  return  $\perp$ 
else return mode(ValuesRead)

```

Apply( $P_w, write\ v, \mathcal{R}$ )

```

for  $i := 1$  to 4
  if  $R_i \notin FAILED_w$  then
    resp := write( $P_w, v, R_i$ )
    if resp =  $\perp$  then
      FAILED_w := FAILED_w  $\cup$  { $R_i$ }
if |FAILED_w|  $\geq 2$  then
  return  $\perp$ 
else return ack

```

Figure 20: 1-tolerant gracefully degrading self-implementation of 1-reader 1-writer safe register for omission

---

to be faulty, it returns  $\perp$ . Otherwise it returns  $\text{mode}(\text{ValuesRead})$ , a value that occurs at least as many times in  $\text{ValuesRead}$  as any other value. To write a value  $v$  in the derived register, the writer process  $P_w$  writes  $v$  in each base register that has so far not appeared faulty to it. Like  $P_r$ ,  $P_w$  also adds base registers that return  $\perp$  to the set  $\text{FAILED}_w$ . If, at the end,  $P_w$  has detected two or more base registers to be faulty, it returns  $\perp$ . Otherwise it returns  $\text{ack}$ .

We now prove that the implementation is correct. Consider the concurrent system  $\mathcal{S} = (P_r, P_w; \mathcal{R})$ , where  $\mathcal{R}$  is a derived object of the implementation. Let  $R_1, R_2, R_3$ , and  $R_4$  be the base objects of  $\mathcal{R}$ . We present two lemmas below. The first proves that it is a gracefully degrading implementation of **1-reader 1-writer safe register**, and the second proves that it is 1-tolerant.

**Lemma 7.15** *Let  $E$  be any execution of  $\mathcal{S}$  which satisfies the following.*

- A1.**  *$P_r$  invokes only **Read** operations on  $\mathcal{R}$  and  $P_w$  invokes only **Write** operations on  $\mathcal{R}$ .*
- A2.** *If a process ( $P_r$  or  $P_w$ ) gets the response  $\perp$  from  $\mathcal{R}$ , it does not subsequently invoke an operation on  $\mathcal{R}$ .*
- A3.** *If a base object of  $\mathcal{R}$  fails, it fails by omission.*

*Then, if  $\mathcal{R}$  fails in  $E$ , it fails by omission.*

*Proof* To prove the lemma, it suffices to show that  $\mathcal{R}$  satisfies Properties 1, 2, and 3 of omission in  $E$ . By **A3**, each base object of  $\mathcal{R}$  either fails by omission or is correct in  $E$ . It follows that each base object is wait-free in  $E$ . From this and the implementation, it is easy to see that  $\mathcal{R}$  is wait-free in  $E$ . Thus,  $\mathcal{R}$  satisfies Property 1 of omission in  $E$ . Property 2 of omission, that every response from  $\mathcal{R}$  is either  $\perp$  or from  $\text{RES}$ , is obvious. Below, we verify that  $\mathcal{R}$  satisfies Property 3 of omission in  $E$ .

Let  $H$  be the history of  $\mathcal{R}$  in  $E$ . Let  $H'$  be obtained by removing response events in  $H$  that return  $\perp$ . (As a result, a read operation  $r$  and a write operation  $w$ , which are not concurrent in  $H$ , may become concurrent in  $H'$ . This will happen if  $w$  returned  $\perp$  and  $w$  preceded  $r$  in  $H$ .) To verify that  $\mathcal{R}$  satisfies Property 3 of omission in  $E$ , it suffices to show that, in the history  $H'$ , every complete read operation, which is not concurrent with a write operation, returns the most recent value written.

Let  $r$  be any complete read operation in  $H'$  that is not concurrent with a write operation in  $H'$ . Let  $V$  be the response returned by  $r$ . Let  $\text{Apply}(P_w, \text{write } V', \mathcal{R})$ , denoted by  $w$ , be the latest write operation in  $H'$  that precedes  $r$ . By construction of  $H'$  and the fact that  $r$  and  $w$  are complete operations in  $H'$ , we have (i)  $V \neq \perp$  and (ii)  $w$  returned  $\text{ack}$  (as opposed to  $\perp$ ). Let  $\mathbf{F}_r$  be the value of  $\text{FAILED}_r$  at the end of the read operation  $r$  in  $E$ . Since  $r$  returned  $V \neq \perp$ , it follows from the implementation that  $|\mathbf{F}_r| \leq 1$ . Let  $\mathbf{F}_w$  be the value of  $\text{FAILED}_w$  at the end of  $w$ . Since  $w$  returned  $\text{ack}$ , it follows from the implementation that  $|\mathbf{F}_w| \leq 1$ . Let  $S = \{R_1, R_2, R_3, R_4\} - (\mathbf{F}_r \cup \mathbf{F}_w)$ . The above implies that either  $|S| > 2$  or  $\mathbf{F}_r = 1$  and  $|S| = 2$ . Also, when the reader  $P_r$  reads a register  $R \in S$  during the execution

of  $r$ , it is obvious that  $R$  returns  $V'$ . Therefore, at the end of  $r$ , either  $V'$  occurs at least three times in *ValuesRead*, or  $V'$  occurs exactly twice in *ValuesRead* and  $\mathbf{F}_r = 1$ . In either case, at the end of  $r$ ,  $\text{mode}(\text{ValuesRead}) = V'$ . Hence  $r$  returns  $V'$ . We conclude that  $V = V'$ . In other words, every complete read operation in  $H'$ , which is not concurrent with a write operation in  $H'$ , returns the most recent value written. This verifies that  $\mathcal{R}$  satisfies Property 3 of omission in  $E$ . Hence the lemma.  $\square$

**Lemma 7.16** *Let  $E$  be any execution of  $\mathcal{S}$  which satisfies conditions **A1**, **A2**, and **A3** listed in the previous lemma. Additionally, assume that at most one base object of  $\mathcal{R}$  fails in  $E$ . Then,  $\mathcal{R}$  is correct in  $E$ .*

*Proof* We have to show that  $\mathcal{R}$  is well-behaved and wait-free in  $E$ . Consider any complete read operation  $r$  in  $E$  that is not concurrent with a write operation. Let  $\text{Apply}(P_w, \text{write } V, \mathcal{R})$  be the latest write operation in  $E$  that precedes  $r$ . Since at most one base object fails, it is obvious that  $P_r$  reads  $V$  from at least three base registers during the execution of  $r$ . Hence the value returned by the read operation  $r$  is  $V$ . This implies that  $\mathcal{R}$  is well-behaved in  $E$ .

Each base register  $R_i$  either fails by omission or is correct in  $E$ . In either case,  $R_i$  is wait-free in  $E$ . From this and the implementation, it is obvious that  $\mathcal{R}$  is wait-free in  $E$ .  $\square$

**Lemma 7.17** *Figure 20 presents a 1-tolerant gracefully degrading self-implementation of 1-reader 1-writer safe register for omission.*

*Proof* Immediate from Lemmas 7.15 and 7.16.  $\square$

By the reasoning presented in Steps **S1**, **S2**, and **S3** earlier, we have:

**Lemma 7.18** *register has a  $t$ -tolerant gracefully degrading self-implementation for omission.*

### Graceful degradation for consensus with safe-reset

We present a  $t$ -tolerant gracefully degrading implementation of **consensus with safe-reset** from  $\{\text{consensus with safe-reset}, \text{register}\}$  for omission. This implementation is similar to, but more complex than, the  $t$ -tolerant gracefully degrading self-implementation of **consensus** presented earlier in Figure 16. The added complexity is due to the fact that a reset operation has to be supported.

To understand the difficulty in supporting the reset operation, we first extend the implementation in Figure 16 in the obvious manner and show why it does not work. First, assume that the base objects  $O_1, O_2, \dots, O_{2t+1}$  are not just consensus objects, but are consensus-with-safe-reset objects. Second, implement  $\text{Reset}(P, \mathcal{O})$ , a reset of the derived object  $\mathcal{O}$  by Process  $P$ , by resetting each base object of  $\mathcal{O}$ . If no more than  $t$  base objects return  $\perp$ ,  $P$  returns *ack*; otherwise,  $P$  returns  $\perp$ . Assume that the implementation of the propose operation on  $\mathcal{O}$  remains as in Figure 16. Unfortunately, the above implementation is

not correct. To see this, suppose that the steps of processes  $P$  and  $Q$  interleave in the order described below. Process  $P$  wishes to reset  $\mathcal{O}$  and begins the execution of  $\text{Reset}(P, \mathcal{O})$ . As  $P$  resets each base object of  $\mathcal{O}$ , assume that each of  $O_1, O_2, \dots, O_{2t}$  is correct and returns *ack* to  $P$ , but  $O_{2t+1}$  fails by omission and returns  $\perp$  to  $P$ .  $P$  completes  $\text{Reset}(P, \mathcal{O})$ , returning *ack*. Process  $Q$  wishes to propose 0 to  $\mathcal{O}$  and begins the execution of  $\text{Propose}(P, 0, \mathcal{O})$ . As  $Q$  proposes 0 to each base object, each of  $O_1, O_2, \dots, O_{2t}$ , being correct, returns 0 to  $P$ . Therefore, at the end of  $2t$  iterations of the for-loop in Figure 16,  $estimate_q = 0$ . Thus, in the last iteration of the for-loop,  $Q$  proposes 0 to  $O_{2t+1}$ . Since  $O_{2t+1}$  has failed by omission, it behaves as if the aborted reset operation of  $P$  on  $O_{2t+1}$  were an incomplete operation (see Property 3 of omission failure). Thus, from  $O_{2t+1}$ 's point of view, the propose operation by  $Q$  on  $O_{2t+1}$  is concurrent with the “incomplete” reset operation by  $P$  on  $O_{2t+1}$ . Recall that a consensus-with-safe-reset object may return arbitrary responses to operations if any operation is concurrent with a reset. Thus, the response from  $O_{2t+1}$  to the propose operation by  $Q$  is arbitrary. Assume that this response is 1. From Figure 16, it is clear that  $estimate_q$  changes to 1 and  $Q$  terminates  $\text{Propose}(P, 0, \mathcal{O})$ , returning 1. This violates the validity property of  $\mathcal{O}$ :  $\mathcal{O}$  returned 1 to  $Q$  even though no process proposed 1 to  $\mathcal{O}$ . We conclude that the implementation is not even 1-tolerant.

Before presenting the correct implementation, we state two propositions that characterize the type **consensus with safe-reset**. These propositions will be useful when we prove the correctness of our implementation. For ease of stating the propositions, we need some definitions.

In the following, let  $\mathcal{O}$  be an object of type **consensus with safe-reset**, initialized to the uncommitted state. Let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$ . As just mentioned, if a reset overlaps with any other operation, including another reset operation,  $\mathcal{O}$  can behave in an unrestricted manner, though still responsive. This leads us to define  $\phi(E)$  to be the maximal prefix of  $E$  in which a reset operation is not concurrent with any other operation.

- Object  $\mathcal{O}$  satisfies *integrity* in  $E$  if and only if every response from  $\mathcal{O}$  to a propose operation in  $\phi(E)$  is either 0 or 1, and every response from  $\mathcal{O}$  to a reset operation in  $\phi(E)$  is *ack*.
- Object  $\mathcal{O}$  satisfies *weak integrity* in  $E$  if and only if every response from  $\mathcal{O}$  to a propose operation in  $\phi(E)$  is either 0, 1, or  $\perp$ , and every response from  $\mathcal{O}$  to a reset operation in  $\phi(E)$  is either *ack* or  $\perp$ .

An *epoch of  $\mathcal{O}$  in  $E$*  is any of the following: (i) a subsequence of  $\phi(E)$  beginning with the event immediately following the response of a reset operation to the event immediately preceding the invocation of the next reset operation, or (ii) the prefix of  $\phi(E)$  up to the event immediately preceding the first invocation of reset, or (iii) the suffix of  $\phi(E)$  ranging from the the event immediately following the response of the last reset in  $\phi(E)$ . Notice that there may be several epochs of  $\mathcal{O}$  in  $E$ . An epoch is *clean* if every operation (reset or propose) that precedes the epoch returns a non- $\perp$  response. Thus, all operations which complete before the start of a clean epoch return non- $\perp$  responses. Notice that if  $\mathcal{O}$  satisfies integrity in  $E$ , then every epoch of  $\mathcal{O}$  in  $E$  is clean.

- Object  $\mathcal{O}$  satisfies *epoch-validity* in  $E$  if and only if the following holds. If  $\mathcal{O}$  returns a response  $v$  to a propose operation in some clean epoch and  $v \in \{0, 1\}$ , then there is an invocation of *propose*  $v$  on  $\mathcal{O}$ , in the same epoch, preceding this response.
- Object  $\mathcal{O}$  satisfies *epoch-agreement* in  $E$  if and only if the following holds. If  $\mathcal{O}$  returns  $v_1, v_2$  to two propose operations in some clean epoch and  $v_1, v_2 \in \{0, 1\}$ , then  $v_1 = v_2$ . (By this definition, if  $\mathcal{O}$  returns 0 to some processes and  $\perp$  to all others, it still satisfies epoch-agreement.)

Notice how these definitions generalize the ones in Section 5.1.1. The propositions below follow easily from the specification of **consensus with safe-reset**, and the definitions of linearizability and omission failures. These propositions are similar to Propositions 5.1 and 5.2.

**Proposition 7.2** *Let  $\mathcal{O}$  be an object of type **consensus with safe-reset** and let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$ . Object  $\mathcal{O}$  is correct in  $E$  if and only if  $\mathcal{O}$  is wait-free in  $E$  and satisfies integrity, epoch-validity, and epoch-agreement in  $E$ .*

**Proposition 7.3** *Let  $\mathcal{O}$  be an object of type **consensus with safe-reset** and let  $E$  be an execution of  $(P_1, P_2, \dots, P_N; \mathcal{O})$  in which  $\mathcal{O}$  fails. Object  $\mathcal{O}$  fails by omission in  $E$  if and only if it is wait-free in  $E$  and satisfies weak-integrity, epoch-validity, and epoch-agreement in  $E$ .*

Figure 21 presents a  $t$ -tolerant gracefully degrading implementation of **consensus with safe-reset** from **{consensus with safe-reset, register}** for omission. The implementation uses  $2t + 1$  consensus-with-safe-reset objects  $(O_1, O_2, \dots, O_{2t+1})$  and  $2t + 1$   $t$ -tolerant gracefully degrading boolean registers  $(\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_{2t+1})$ . (By Lemma 7.18,  $\mathcal{R}_i$ 's can be implemented from registers.) The register  $\mathcal{R}_i$  is set to 1 if any process detects  $O_i$  to be faulty, *i.e.*, if any process obtains the response  $\perp$  from  $O_i$ . The following is an important running feature of our implementation: If, during the execution of an operation on the derived object  $\mathcal{O}$ , a process  $P$  gets a response of  $\perp$  from any  $\mathcal{R}_i$ ,  $P$  returns  $\perp$  as the response of  $\mathcal{O}$ . This is justified on the basis that  $\mathcal{R}_i$  is  $t$ -tolerant, and thus, more than  $t$  base objects of  $\mathcal{R}_i$  must have failed for  $\mathcal{R}_i$  to fail. Since  $\mathcal{O}$  needs to be only  $t$ -tolerant,  $\mathcal{O}$  may fail and return  $\perp$  if more than  $t$  base objects of  $\mathcal{O}$  fail, or equivalently, if any  $\mathcal{R}_i$  fails. We now describe the procedures **Reset** $(P_i, \mathcal{O})$  and **Propose** $(P_i, v_i, \mathcal{O})$ .

To reset  $\mathcal{O}$ , a process  $P_i$  first reads all  $\mathcal{R}_k$ 's and collects the identities of the faulty objects among  $\{O_1, O_2, \dots, O_{2t+1}\}$ .  $P_i$  then resets each non-faulty object in  $\{O_1, O_2, \dots, O_{2t+1}\}$ . If, during this resetting, an object  $O_k$  responds with  $\perp$  to  $P_i$ ,  $P_i$  writes 1 in  $\mathcal{R}_k$  to record the fact that  $O_k$  is faulty. At the end of this,  $P_i$  returns with the response *ack*.

To propose  $v_i$  to  $\mathcal{O}$ , a process  $P_i$  first reads all  $\mathcal{R}_k$ 's and collects the identities of the faulty objects among  $\{O_1, O_2, \dots, O_{2t+1}\}$ . With a few minor differences, the rest of the implementation parallels the one in Figure 16. At any point in the algorithm,  $P_i$  holds an estimate of the eventual return value in  $estimate_i$ . To start with,  $estimate_i$  is set to  $v_i$ .

---

$\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_{2t+1}$ :  $t$ -tolerant gracefully degrading boolean registers, initialized to 0  
 $O_1, O_2, \dots, O_{2t+1}$ : (0-tolerant) consensus-with-safe-reset objects

---

<p><b>Procedure</b> <math>\text{Propose}(P_i, v_i, \mathcal{O})</math>  <math>V_i[1 \dots 2t+1]</math>, <math>\text{estimate}_i</math>, <math>\text{resp}</math>, <math>k</math>,  <math>\text{set-of-failed}</math>: local to <math>P_i</math>  <b>begin</b>  <math>\text{estimate}_i := v_i</math>  <math>\text{set-of-failed} := \emptyset</math>  <b>for</b> <math>k := 1</math> <b>to</b> <math>2t+1</math>  <math>\text{resp} := \text{Read}(P_i, \mathcal{R}_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{return } \perp</math>  <b>else if</b> <math>\text{resp} = 1</math> <b>then</b>  <math>\text{set-of-failed} := \text{set-of-failed} \cup \{O_k\}</math>  <b>for</b> <math>k := 1</math> <b>to</b> <math>2t+1</math>  <b>if</b> <math>O_k \in \text{set-of-failed}</math> <b>then</b>  <math>V_i[k] := \perp</math>  <b>else</b>  <math>\text{resp} := \text{propose}(P_i, \text{estimate}_i, O_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{resp} := \text{Write}(P_i, 1, \mathcal{R}_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{return } \perp</math>  <b>else if</b> <math>\text{resp} \neq \text{estimate}_i</math> <b>then</b>  <math>\text{estimate}_i := \text{resp}</math>  <math>V_i[1 \dots (k-1)] := (\perp, \perp, \dots, \perp)</math>  <b>if</b> <math>V_i</math> has more than <math>t</math> <math>\perp</math>'s <b>then</b>  <math>\text{return } \perp</math>  <b>else return</b> <math>\text{estimate}_i</math>  <b>end</b></p>	<p><b>Procedure</b> <math>\text{Reset}(P_i, \mathcal{O})</math>  <math>\text{set-of-failed}</math>, <math>\text{resp}</math>, <math>k</math>: local to <math>P_i</math>  <b>begin</b>  <math>\text{set-of-failed} := \emptyset</math>  <b>for</b> <math>k := 1</math> <b>to</b> <math>2t+1</math>  <math>\text{resp} := \text{Read}(P_i, \mathcal{R}_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{return } \perp</math>  <b>else if</b> <math>\text{resp} = 1</math> <b>then</b>  <math>\text{set-of-failed} := \text{set-of-failed} \cup \{O_k\}</math>  <b>for</b> <math>k := 1</math> <b>to</b> <math>2t+1</math>  <b>if</b> <math>O_k \notin \text{set-of-failed}</math> <b>then</b>  <math>\text{resp} := \text{reset}(P_i, O_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{resp} := \text{Write}(P_i, 1, \mathcal{R}_k)</math>  <b>if</b> <math>\text{resp} = \perp</math> <b>then</b>  <math>\text{return } \perp</math>  <math>\text{return } \text{ack}</math>  <b>end</b></p>
--	--

Figure 21:  $t$ -tolerant gracefully degrading implementation of consensus with safe-reset for omission

---

$P_i$  then goes through  $O_1, O_2, \dots, O_{2t+1}$ , in that order, and performs the following steps on each of them. If  $O_k$  is known to be faulty,  $P_i$  does not access  $O_k$ ; it simply pretends that  $O_k$  returned  $\perp$ . Otherwise,  $P_i$  proposes its current estimate to  $O_k$ . If  $O_k$  returns a non- $\perp$  response,  $P_i$  proceeds exactly as in Figure 16. If  $O_k$  returns  $\perp$ ,  $P_i$  writes 1 in  $\mathcal{R}_k$  to record the fact that  $O_k$  is faulty. After going through all of  $O_1, O_2, \dots, O_{2t+1}$ ,  $P_i$  applies the same rules as in Figure 16 to compute the return value.

**Lemma 7.19** *Figure 21 presents a  $t$ -tolerant gracefully degrading implementation of consensus with safe-reset from {consensus with safe-reset, register} for omission.*

*Proof* Let  $\mathcal{R}_i$  ( $1 \leq i \leq 2t+1$ ) be a derived object of the  $t$ -tolerant gracefully degrading implementation of **register** (such an implementation exists by Lemma 7.18). Let

$R_{i,1}, R_{i,2}, \dots, R_{i,m}$  be the base registers of  $\mathcal{R}_i$ . Let  $\mathcal{O}$  be derived from the implementation in Figure 21 using  $O_1, O_2, \dots, O_{2t+1}$  and  $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_{2t+1}$ . Thus,  $O_1, O_2, \dots, O_{2t+1}$  and  $R_{i,j}$  ( $1 \leq i \leq 2t+1, 1 \leq j \leq m$ ) are the base objects of  $\mathcal{O}$ . Consider an execution  $E$  of  $(P_1, P_2, \dots, P_N; \mathcal{O})$  in which all base objects that fail, fail by omission. Let  $\mathcal{E}$  be a clean epoch of  $\mathcal{O}$  in  $E$ . Let  $FAILED(\mathcal{E})$  be the set of all  $O_j$  ( $1 \leq j \leq 2t+1$ ) such that some process had written 1 in  $\mathcal{R}_j$  before epoch  $\mathcal{E}$  started. Thus,  $FAILED(\mathcal{E})$  is the subset of  $\{O_1, O_2, \dots, O_{2t+1}\}$  that failed before the start of  $\mathcal{E}$ . We make the following observations.

- O1.** For each base object  $O \in \{O_1, O_2, \dots, O_{2t+1}\} - FAILED(\mathcal{E})$ ,  $\mathcal{E}$  is a clean epoch of  $O$ .
- O2.** In epoch  $\mathcal{E}$ , no process invokes an operation on a base object in  $FAILED(\mathcal{E})$ .
- O3.** In the execution of **Propose** $(P_i, v_i, \mathcal{O})$ , at the end of the  $k^{th}$  iteration of the **for-loop** ( $1 \leq k \leq 2t+1$ ),  $estimate_i \in \{0, 1\}$ , and  $V_i[1..k]$  contains only  $\perp$ 's and  $estimate_i$ 's.

We now use these observations to show that  $\mathcal{O}$  satisfies the required properties in  $E$ .

1.  $\mathcal{O}$  is wait-free: Recall that base objects that fail by omission remain wait-free. From this and the implementation, it is obvious that  $\mathcal{O}$  is wait-free.
2.  $\mathcal{O}$  satisfies epoch-validity: Suppose that an execution of **Propose** $(P_i, v_i, \mathcal{O})$  in epoch  $\mathcal{E}$  returns  $v \in \{0, 1\}$ . (Let  $e_{ret}$  denote the event of completion of this execution.) It follows that, during this execution, some base object  $O_j$  returns  $v$  to  $P_i$  when  $P_i$  performs **propose** $(P_i, estimate_i, O_j)$ . Let  $e_f$  denote the first response event in  $\mathcal{E}$  in which a base object among  $\{O_1, O_2, \dots, O_{2t+1}\}$  returns the response  $v$ . Let  $O_f$  be the base object associated with the event  $e_f$ . By **O2**,  $O_f \in \{O_1, O_2, \dots, O_{2t+1}\} - FAILED(\mathcal{E})$ . By **O1**,  $\mathcal{E}$  is a clean epoch of  $O_f$ . Since  $O_f$  either is correct or fails by omission, by Propositions 7.2 and 7.3,  $O_f$  satisfies epoch-validity. That is, there is an invocation of **propose** $(P_i, v, O_f)$  in  $\mathcal{E}$  before the response event  $e_f$ . From the implementation and the definition of  $e_f$ , this invocation of **propose** $(P_i, v, O_f)$  is possible only during the execution of **Propose** $(P_i, v, \mathcal{O})$ . Thus, the invocation of **Propose** $(P_i, v, \mathcal{O})$  precedes the invocation of **propose** $(P_i, v, O_f)$ , which, in turn, precedes  $e_f$ . Furthermore,  $e_f$  precedes  $e_{ret}$ . This implies that the invocation of **Propose** $(P_i, v, \mathcal{O})$  precedes  $e_{ret}$ . We conclude that  $\mathcal{O}$  satisfies epoch-validity in  $E$ .
3.  $\mathcal{O}$  satisfies epoch-agreement: Suppose that, in  $\mathcal{E}$ , there is an execution of **Propose** $(P_i, v_i, \mathcal{O})$  and one of **Propose** $(P_j, v_j, \mathcal{O})$ , which return 0 and 1, respectively. We will refer to these executions as *exec1* and *exec2*. From **O3** and the implementation, it follows that  $V_i$  has at least  $t+1$  0's at the end of *exec1*. Similarly,  $V_j$  has at least  $t+1$  1's at the end of *exec2*. This implies that there is a  $k$  ( $1 \leq k \leq 2t+1$ ) such that  $O_k$  returns 0 when  $P_i$  performs **propose** $(P_i, estimate_i, O_k)$  in *exec1* and returns 1 when  $P_j$  performs **propose** $(P_j, estimate_j, O_k)$  in *exec2*. By **O2**,  $O_k \in \{O_1, O_2, \dots, O_{2t+1}\} - FAILED(\mathcal{E})$ . It follows from **O1** that  $\mathcal{E}$  is a clean epoch for  $O_k$ . Since  $O_k$  either is correct or fails by omission, by Propositions 7.2 and 7.3,  $O_k$  satisfies epoch-agreement. This contradicts the earlier conclusion that  $O_k$  returns 0 to  $P_i$  and 1 to  $P_j$ . We conclude that  $\mathcal{O}$  satisfies epoch-agreement in  $E$ .

4.  $\mathcal{O}$  satisfies weak integrity: Obvious.
5.  $\mathcal{O}$  satisfies integrity if at most  $t$  base objects fail: Suppose that no more than  $t$  base objects of  $\mathcal{O}$  fail. For all  $j$ ,  $1 \leq j \leq 2t + 1$ , since  $\mathcal{R}_j$  is  $t$ -tolerant,  $\mathcal{R}_j$  will be correct. It follows from the implementation that every reset operation on  $\mathcal{O}$  in  $E$  returns *ack*. We now make some observations to show that every propose operation on  $\mathcal{O}$  in  $\phi(E)$  returns either 0 or 1. In the following, let  $\mathcal{E}$  be any (not necessarily clean) epoch of  $\mathcal{O}$  in  $E$ .
  - (a) Let  $O_{k_1}, O_{k_2}, \dots, O_{k_l}$  ( $k_1 < k_2 < \dots < k_l$ ) be all the base objects among  $\{O_1, O_2, \dots, O_{2t+1}\}$  which are correct in  $E$ . Since at most  $t$  fail, we have  $l \geq t+1$ .
  - (b) From the fact that  $O_{k_1}$  is correct in  $E$ , it is easy to verify that  $\mathcal{E}$  is a clean epoch for  $O_{k_1}$ . Since  $O_{k_1}$  is correct and  $\mathcal{E}$  is a clean epoch for  $O_{k_1}$ , by Proposition 7.2,  $O_{k_1}$  satisfies integrity and epoch-agreement in epoch  $\mathcal{E}$ . Thus, there is a  $v \in \{0, 1\}$  such that every propose operation on  $O_{k_1}$  in epoch  $\mathcal{E}$  returns  $v$ . This implies that, for every execution of  $\text{Propose}(P_i, v_i, \mathcal{O})$  in  $\mathcal{E}$ ,  $estimate_i = v$  at the end of  $k_1$  iterations of the for-loop.
  - (c) For all  $1 \leq j \leq l$ ,  $O_{k_j}$  is correct in  $E$ . From this, it is easy to verify that  $\mathcal{E}$  is a clean epoch for  $O_{k_j}$ . Since  $O_{k_j}$  is correct and  $\mathcal{E}$  is a clean epoch for  $O_{k_j}$ , by Proposition 7.2,  $O_{k_j}$  satisfies integrity, epoch-validity, and epoch-agreement in epoch  $\mathcal{E}$ . In particular, if every process that proposes to  $O_{k_j}$  in epoch  $\mathcal{E}$  proposes the value  $v$ , then  $O_{k_j}$  returns only  $v$  in  $\mathcal{E}$ .
  - (d) Let  $O_j \in \{O_1, O_2, \dots, O_{2t+1}\} - \{O_{k_1}, O_{k_2}, \dots, O_{k_l}\}$ . By definition,  $O_j$  fails by omission in  $E$ , returning  $\perp$  to some process. Let  $P$  be the first process to receive  $\perp$  from  $O_j$  and let *oper* denote the execution of  $P$ 's operation on the derived object  $\mathcal{O}$  during which  $P$  received  $\perp$  from  $O_j$ . Consider the following two cases. In the first case, assume that  $O_j$  returned  $\perp$  to  $P$  before epoch  $\mathcal{E}$  started. Since  $\mathcal{E}$  is a clean epoch, it follows that *oper* completed before  $\mathcal{E}$  started. This implies that  $P$  wrote 1 in  $\mathcal{R}_j$  before the start of epoch  $\mathcal{E}$ . It follows from the implementation that no process invokes an operation on  $O_j$  in epoch  $\mathcal{E}$ . In the second case, assume that  $O_j$  never returned  $\perp$  to any process before the start of epoch  $\mathcal{E}$ . Then, it is easy to see that  $\mathcal{E}$  is a clean epoch for  $O_j$ . Thus, by Proposition 7.3, if every process that proposes to  $O_j$  in epoch  $\mathcal{E}$  proposes the value  $v$ ,  $O_j$  returns either  $v$  or  $\perp$  in  $\mathcal{E}$ .

Consider any execution of  $\text{Propose}(P_i, v_i, \mathcal{O})$  in epoch  $\mathcal{E}$ . We claim that  $estimate_i = v$  at the end of  $k_1$  iterations of the for-loop and the value of  $estimate_i$  does not change in the subsequent iterations. The claim follows directly from the above observations and the fact that a process does not change its estimate if a base object  $O_j$  returns  $\perp$ . This claim, together with the fact that  $O_{k_1}, O_{k_2}, \dots, O_{k_l}$  are correct, implies that, at the end of the execution, (i)  $estimate_i = v$  and (ii) for all  $1 \leq j \leq l$ ,  $V_i[k_j] = v$ . From the implementation, it follows that  $\text{Propose}(P_i, v_i, \mathcal{O})$  returns  $v$ . We conclude that  $\mathcal{O}$  satisfies integrity.



From 1, 2, 3, and 4 above, and Proposition 7.3, we conclude that either  $\mathcal{O}$  is correct in  $E$  or  $\mathcal{O}$  fails by omission in  $E$ . Thus, the implementation is gracefully degrading for omission. From 1, 2, 3, and 5 above, and Proposition 7.2, we conclude that if at most  $t$  base objects of  $\mathcal{O}$  fail in  $E$ , and they fail by omission, then  $\mathcal{O}$  is correct in  $E$ . Thus, the implementation is  $t$ -tolerant for omission. This completes the proof of the lemma.  $\square$

## Graceful degradation theorem for omission

From the previous three lemmas, and the argument presented at the beginning of Section 7.2.2, we have

**Theorem 7.5** *Every type has a  $t$ -tolerant gracefully degrading implementation from every universal set of types for omission.*

## 8 Related work

In an independent work, Afek *et al.* consider the problem of coping with shared memory subject to *memory failures* [AGMT92]. Informally, each failure is modeled as a *faulty write*. The following failure modes are considered:

- A. There is a bound  $m$  on the total number of faulty writes.
- B. There is a bound  $f$  on the total number of data objects that may be affected by memory failures, and a bound  $k$  on the number of faulty writes on each faulty object. A different failure model is obtained for  $k = \infty$ .

In our terminology, these failure modes are responsive. The second one, with  $k = \infty$ , corresponds to our arbitrary failure mode.

[AGMT92] focuses on fault-tolerant implementations of the following types of objects: safe, atomic, binary, and  $V$ -valued **register** from various types of registers;  $N$ -process **test&set** from  $N$ -process **test&set** and bounded **register**; and **N-consensus** from **read-modify-write** (RMW). [AGMT92] also gives a universal fault-tolerant implementation from unbounded RMW, based on Herlihy’s universal implementation. The main differences between [AGMT92] and this paper are as follows:

1. [AGMT92] does not consider any non-responsive failure mode.
2. Amongst the responsive failure modes, benign ones, such as crash and omission, are also not considered in [AGMT92].
3. This paper does not consider failure modes that bound the number of times faulty objects can fail (in [AGMT92], each “faulty write” is counted as a failure).

4. The two approaches to modeling failures appear to be fundamentally different. There is no direct way to model benign failures, such as crash and omission failures, with “faulty writes”. On the other hand, our approach—defining how each faulty object deviates from its type—is not suited to handle Model A above.
5. This paper introduces the concept of *graceful degradation*, and presents several related results, in particular, for crash and omission failure modes. For arbitrary failures, graceful degradation reduces to the “*strong wait-freedom*” concept introduced in [AGMT92].
6. In the Open Problems section of [AGMT92] it is stated:

“It would be particularly interesting to implement memory-fault tolerant data objects directly from similar, faulty objects, such as test-and-set from test-and-set, without using atomic registers, or read-modify-write from read-modify-write, without using an unbounded universal construction.”

It is interesting to note that both of these types do have fault-tolerant self-implementations. For bounded RMW, this is a direct consequence of Corollary 5.1. For  $N$ -process `test&set`, one can combine the fault-tolerant implementation of `test&set` from `{test&set, bounded register}` [AGMT92], with the implementation of bounded `register` from `test&set` presented in Figure 14.

7. The existence of a fault-tolerant *self*-implementation of `consensus`, shown in this paper, does not follow from the results in [AGMT92].
8. The fault-tolerant implementation of  $N$ -process `test&set` from `{test&set, bounded register}`, shown in [AGMT92], does not follow from our results (when  $N > 2$ ).

## Acknowledgements

We thank Vassos Hadzilacos for many interesting discussions. The comments of Vassos Hadzilacos, Jon Kleinberg, Sendhil Mullainathan, Gil Neiger, King Tan, and the anonymous referees on earlier drafts helped us improve the presentation.

## A Translation from arbitrary failure mode to omission failure mode

In this section, we define the notion of a translation between failure modes. We also present a  $t$ -tolerant translation from arbitrary failure mode to omission failure mode for the type `consensus`. We prove that its resource complexity of  $3t + 1$  is optimal. This translation can be used along with the  $t$ -tolerant self-implementation of `consensus` for omission (presented in Section 5.1.2) to obtain a  $t$ -tolerant self-implementation of `consensus` for

arbitrary failures. However, the resource complexity of the resulting implementation will be  $(3t + 1) \cdot (t + 1)$ , which is more than the  $O(t \log t)$  complexity achieved by the direct implementation presented earlier in Figure 10.

A  $t$ -tolerant translation of  $(T, s)$  from failure mode  $\mathcal{F}$  to failure mode  $\mathcal{F}'$  is a self-implementation  $\mathcal{I}$  with the following property:

Let  $\mathcal{O}$  be a derived object of  $\mathcal{I}$  and  $E$  be an execution. If at most  $t$  base objects of  $\mathcal{O}$  fail in  $E$ , and they fail by  $\mathcal{F}$ , then either  $\mathcal{O}$  is correct in  $E$  or  $\mathcal{O}$  fails by  $\mathcal{F}'$  in  $E$ .

A type  $T$  has a  $t$ -tolerant translation from failure mode  $\mathcal{F}$  to failure mode  $\mathcal{F}'$  if, for all states  $s$  of  $T$ ,  $(T, s)$  has a  $t$ -tolerant translation from  $\mathcal{F}$  to  $\mathcal{F}'$ .

The motivation for translation is as follows. Suppose that the hardware of a system supports objects of type  $T$ . Suppose further that these objects, if they fail, fail by (a severe mode)  $\mathcal{F}$ . If the translation defined above is available, then it is easy to make it seem as if the system supported objects of type  $T$  which, if they fail, will fail by (the less severe mode)  $\mathcal{F}'$ .

$A[1 \dots 2t + 1], B[1 \dots t]$  : consensus objects, initialized to the uncommitted state

```

Procedure Propose( $p, v_p, \mathcal{O}$ )
     $count_p[0..1], w, i, belief_p$  : integer local to  $p$ 
begin
1      Phase 1:  $count_p[0..1] := (0, 0)$ 
2          for  $i := 1$  to  $2t + 1$ 
3               $w := \mathbf{f-propose}(p, v_p, A[i])$ 
4               $count_p[w] := count_p[w] + 1$ 
5      Phase 2: Choose  $belief_p$  such that
               $count_p[belief_p] > count_p[\overline{belief_p}]$ .
6          for  $i := 1$  to  $t$ 
7              if  $belief_p \neq \mathbf{f-propose}(p, belief_p, B[i])$  then
8                  return( $\perp$ )
9      return( $belief_p$ )
end

```

Figure 22:  $t$ -tolerant translation from arbitrary to omission for consensus

In Figure 22, we present a  $t$ -tolerant translation of consensus from arbitrary failure mode to omission failure mode. Below, we prove its correctness through a series of lemmas. Let  $\mathcal{O}$  be a consensus object, initialized to the uncommitted state, derived from this translation. The base objects of  $\mathcal{O}$  are  $A[1 \dots 2t + 1], B[1 \dots t]$ .

**Lemma A.1**  $\mathcal{O}$  satisfies integrity in any execution in which all base objects of  $\mathcal{O}$  are correct.

*Proof* Clear from the algorithm.  $\square$

**Lemma A.2**  $\mathcal{O}$  is wait-free in any execution in which all base objects of  $\mathcal{O}$  are wait-free.

*Proof* Clear from the algorithm.  $\square$

In the following lemmas, let  $E$  be an execution in which at most  $t$  base objects experience arbitrary failures, and the remaining are correct.

**Lemma A.3**  $\mathcal{O}$  satisfies weak integrity in  $E$ .

*Proof* Clear from the algorithm.  $\square$

**Lemma A.4**  $\mathcal{O}$  satisfies validity in  $E$ .

*Proof* Suppose  $\mathcal{O}$  returns  $v \in \{0, 1\}$  to the invocation  $\text{Propose}(p, v_p, \mathcal{O})$  (from process  $p$ ). Then  $v = \text{belief}_p$  (by line 9), and  $\text{count}_p[v] = \text{count}_p[\text{belief}_p] \geq t + 1$  (by line 5). So there is at least one correct base object  $A[i]$  such that  $\text{propose}(p, v_p, A[i])$  returned  $v$ . By Proposition 5.1,  $A[i]$  satisfies validity. It follows that some process  $q$  invoked  $\text{propose}(q, v_q, A[i])$  where  $v_q = v$ . This implies that  $q$  invoked  $\text{Propose}(q, v, \mathcal{O})$ .  $\square$

**Lemma A.5**  $\mathcal{O}$  satisfies agreement in  $E$ .

*Proof* Suppose  $\mathcal{O}$  fails to satisfy agreement by returning 0 to some process  $p$  and 1 to a different process  $q$ . Since  $\mathcal{O}$  returns 0 to  $p$ , it follows that  $\text{belief}_p = 0$  at the end of  $E$ . Similarly,  $\text{belief}_q = 1$ . Thus,  $\text{belief}_p \neq \text{belief}_q$ . It is easy to verify that if all of  $A[1 \dots 2t + 1]$  are correct, then  $\text{belief}_p = \text{belief}_q$ . It follows that at least one of  $A[1 \dots 2t + 1]$  fails.

Further, since  $\mathcal{O}$  returns 0 to  $p$ , it follows that, for all  $1 \leq i \leq t$ ,  $\text{propose}(p, \text{belief}_p, B[i])$  returns 0 to  $p$ . Similarly, for all  $1 \leq i \leq t$ ,  $\text{propose}(q, \text{belief}_q, B[i])$  returns 1 to  $q$ . Thus all  $t$  base objects  $B[1 \dots t]$  fail by not satisfying agreement. Counting the failed  $A[i]$ 's and  $B[i]$ 's, we have more than  $t$  failed base objects, a contradiction.  $\square$

From the above lemmas, and Propositions 5.1 and 5.2, we conclude that: (i)  $\mathcal{O}$  is correct in every execution in which all base objects of  $\mathcal{O}$  are correct; and (ii)  $\mathcal{O}$  is either correct or it fails by omission in every execution in which at most  $t$  base objects of  $\mathcal{O}$  fail by the arbitrary failure mode, and the remaining base objects are correct. Thus,

**Theorem A.1** Figure 22 presents a  $t$ -tolerant translation from arbitrary failures to omission failures for consensus. The resource complexity of the translation is  $3t + 1$ .

**Theorem A.2** The resource complexity of any  $t$ -tolerant translation  $\mathcal{I}$  from arbitrary to omission for consensus is at least  $3t + 1$ .

*Proof* For a contradiction, assume the resource complexity of  $\mathcal{I}$  is  $n \leq 3t$ . We prove the theorem through a series of lemmas, involving “indistinguishable” scenarios. Let  $\mathcal{O} = \mathcal{I}(O_1, O_2, \dots, O_n)$ . In the following, we say that a process  $p$  *accesses a base object*  $O_i$  if, during the execution of  $\text{Propose}(p, v_p, \mathcal{O})$ ,  $p$  executes  $\text{propose}(p, *, O_i)$ .

**Lemma A.6** *Suppose  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  to completion (and no other process interleaves with  $p$ ). If all base objects are correct, then  $p$  accesses at least  $t + 1$  base objects.*

*Proof* Suppose the lemma is false, and  $p$  accesses only  $O_{i_1}, O_{i_2}, \dots, O_{i_m}$  ( $m \leq t$ ) before completing  $\text{Propose}(p, 0, \mathcal{O})$ . Since all base objects are correct,  $\mathcal{O}$  satisfies validity and integrity. Hence  $\text{Propose}(p, 0, \mathcal{O})$  returns 0. Now consider the following two scenarios. In these and other scenarios, unless mentioned otherwise, assume that objects are correct.

**Scenario S1**

1.  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  to completion accessing only  $O_{i_1}, O_{i_2}, \dots, O_{i_m}$ .  $\text{Propose}(p, 0, \mathcal{O})$  returns 0.
2.  $q$  executes  $\text{Propose}(q, 1, \mathcal{O})$  to completion.

**Scenario S2**

1. Each of  $O_{i_1}, O_{i_2}, \dots, O_{i_m}$  fails and spontaneously gets into the same state as it is in at the end of Item 1 in Scenario S1.
2.  $q$  executes  $\text{Propose}(q, 1, \mathcal{O})$  to completion; objects  $O_{i_1}, O_{i_2}, \dots, O_{i_m}$  behave exactly as they do in Item 2 of Scenario S1.

Since no base objects fail in S1,  $\mathcal{O}$  must be correct in S1. By Proposition 5.1,  $\mathcal{O}$  satisfies integrity and agreement. Thus  $\text{Propose}(q, 1, \mathcal{O})$  returns 0 in S1. Clearly  $S1 \approx_q S2$ . So  $\text{Propose}(q, 1, \mathcal{O})$  returns 0 in S2 also, violating validity. By Propositions 5.1 and 5.2,  $\mathcal{O}$  is neither correct nor does it fail by omission. Since at most  $t$  base objects fail in S2, and they fail by the arbitrary failure mode, the translation  $\mathcal{I}$  is incorrect, a contradiction.  $\square$

**Lemma A.7** *Consider*

**Scenario S3**

1.  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  up to the point where it has accessed exactly  $t$  base objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$ .
2.  $q$  executes  $\text{Propose}(q, 1, \mathcal{O})$  to completion.

*Then  $\text{Propose}(q, 1, \mathcal{O})$  returns 1.*

*Proof* Let  $S = \{\text{base objects accessed by } q\} - \{O_{i_1}, O_{i_2}, \dots, O_{i_t}\}$ . Let  $O_{j_1}, O_{j_2}, \dots, O_{j_k}$  be all the base objects in  $S$  arranged in the order in which they are first invoked by  $q$ . Note that  $k \leq n - t \leq 2t$ .

Let  $S2'$  represent the scenario obtained by textually substituting  $t$  for  $m$  in Scenario  $S2$ . Since at most  $t$  base objects fail in  $S2'$ , and they fail by the arbitrary failure mode,  $\mathcal{O}$  must either be correct or fail by omission. Hence, by Propositions 5.1 and 5.2,  $\mathcal{O}$  satisfies validity and weak integrity in  $S2'$ . So  $\text{Propose}(q, 1, \mathcal{O})$  returns 1 or  $\perp$  in  $S2'$ . Since  $S2' \approx_q S3$ , we conclude  $\text{Propose}(q, 1, \mathcal{O})$  returns 1 or  $\perp$  in  $S3$ . Since no base object fails in  $S3$ ,  $\mathcal{O}$  must be correct. By Proposition 5.1,  $\mathcal{O}$  satisfies integrity in  $S3$ . So  $\text{Propose}(q, 1, \mathcal{O})$  returns either 0 or 1 in  $S3$ . Together with the above conclusion, this implies the lemma.  $\square$

**Lemma A.8** *Consider*

Scenario S4

1.  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  up to the point where it has accessed exactly  $t$  base objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$ .
2. Let  $O_{j_1}, O_{j_2}, \dots, O_{j_k}$  and  $S$  be as defined above (note  $k \leq 2t$ ).  $q$  executes  $\text{Propose}(q, 1, \mathcal{O})$  up to the point where, of the objects in  $S$ , it has accessed exactly  $\{O_{j_1}, O_{j_2}, \dots, O_{j_{k-t}}\}$ .
3.  $p$  completes the execution of  $\text{Propose}(p, 0, \mathcal{O})$ .

Then  $\text{Propose}(p, 0, \mathcal{O})$  returns 0.

*Proof* Consider

Scenario S5

1.  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  up to the point where it has accessed exactly  $t$  base objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$ .
2. Each of  $O_{j_1}, O_{j_2}, \dots, O_{j_{k-t}}$  fails and spontaneously gets into the same state as it is in at the end of Item 2 in Scenario S4.
3.  $p$  completes the execution of  $\text{Propose}(p, 0, \mathcal{O})$ .

We claim that  $S4 \approx_p S5$ . The only subtlety in verifying this claim is to understand why objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$  cannot help  $p$  distinguish S4 from S5. This is explained below. Objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$  are consensus objects and are correct in both scenarios. Further,  $p$  is the first process to access these objects in both scenarios. Thus, the response from each of these objects is identical in both scenarios.

Since  $k \leq 2t$ , the number of base objects that fail in  $S5 = k - t \leq t$ . Since they fail by the arbitrary failure mode in S5, either  $\mathcal{O}$  is correct in S5, or  $\mathcal{O}$  fails by omission in S5. Thus, by Propositions 5.1 and 5.2,  $\mathcal{O}$  satisfies validity and weak integrity in S5. So  $\text{Propose}(p, 0, \mathcal{O})$  returns either 0 or  $\perp$  in S5. Since  $S4 \approx_p S5$ ,  $\text{Propose}(p, 0, \mathcal{O})$  returns either 0 or  $\perp$  in S4 also. However since no base object fails in S4,  $\mathcal{O}$  is correct in S4, and by Proposition 5.1, it satisfies integrity in S4. Thus  $\text{Propose}(p, 0, \mathcal{O})$  returns 0 in S4.  $\square$

**Lemma A.9** *Consider*

Scenario S6

1.  $p$  executes  $\text{Propose}(p, 0, \mathcal{O})$  up to the point where it has accessed exactly  $t$  base objects  $O_{i_1}, O_{i_2}, \dots, O_{i_t}$ .
2.  $q$  executes  $\text{Propose}(q, 1, \mathcal{O})$  to completion, returning 1, by Lemma A.7.
3. Let  $O_{j_1}, O_{j_2}, \dots, O_{j_k}$  be as defined above (note  $k \leq 2t$ ). Each of  $\{O_{j_{k-t+1}}, O_{j_{k-t+2}}, \dots, O_{j_k}\}$  fails and behaves as though it was never accessed by  $q$ .
4.  $p$  completes the execution of  $\text{Propose}(p, 0, \mathcal{O})$ .

Then  $\text{Propose}(p, 0, \mathcal{O})$  returns 0.

*Proof* Note that  $\text{S4} \approx_p \text{S6}$ . By Lemma A.8,  $\text{Propose}(p, 0, \mathcal{O})$  returns 0 in S4. So  $\text{Propose}(p, 0, \mathcal{O})$  returns 0 in S6.  $\square$

From the above lemma, it is clear that  $\mathcal{O}$  does not satisfy agreement in S6. Hence, by Propositions 5.1 and 5.2,  $\mathcal{O}$  fails in S6, but not by omission. Since at most  $t$  base objects fail in S6, and they fail by the arbitrary failure mode, the translation  $\mathcal{I}$  is incorrect, a contradiction. This completes the proof of Theorem A.2.  $\square$

## B Type definitions

In this section, we specify the types mentioned in the paper. Recall that a type is defined as a 4-tuple  $(OP, RES, G, \tau)$ . For all types specified here,  $\tau$  is the identity function. We describe the graph  $G$  with a set of procedures.

---

$OP = \{\text{write}(v) \mid 0 \leq v < n\} \cup \{\text{read}()\}$   
 $RES = \{v \mid 0 \leq v < n\} \cup \{ack\}$   
 State:  
 $X \in \{0, 1, \dots, n-1\}$

`read()`  
     `return(X)`

`write(v)`  
      $X := v$   
     `return(ack)`

Figure 23:  $n$ -valued register

---



---

$OP = \{\text{compare\&swap}(v_1, v_2) \mid v_1, v_2 \in \{0, 1, 2\}\}$   
 $RES = \{0, 1, 2\}$   
 State:  
 $X \in \{0, 1, 2\}$

`compare\&swap( $v_1, v_2$ )`  
     if  $X = v_1$  then  
          $X := v_2$   
     `return(X)`

Figure 24: compare\&swap

---



---

$OP = \{\text{test\&set}(), \text{reset}()\}$   
 $RES = \{0, 1, ack\}$   
State:  
 $X \in \{0, 1\}$

**test&set()**  
   $y := X$   
   $X := 1$   
  return( $y$ )

**reset()**  
   $X := 0$   
  return( $ack$ )

Figure 25: **test&set**

---

---

$OP = \{\text{fetch\&add}(v) | v \text{ is an integer}\}$   
 $RES = \text{Set of integers}$   
State:  
 $X$ , an integer

**fetch&add( $v$ )**  
   $X := X + v$   
  return( $X$ )

Figure 26: **fetch&add**

---

---

$OP = \{\text{enq}(v) \mid v \text{ is integer}\} \cup \{\text{deq}()\}$

$RES = \{v \mid v \text{ is integer}\} \cup \{\text{nil}, \text{ack}\}$

State:

$X$ , a sequence of integers

**enq**( $v$ )

$X := X \cdot v$

return( $\text{ack}$ )

**deq**()

if  $X$  is empty then

return( $\text{nil}$ )

else if  $X = v \cdot X'$  then

$X := X'$

return( $v$ )

Figure 27: **queue**

---

---

$OP = \{\text{push}(v) \mid v \text{ is integer}\} \cup \{\text{pop}()\}$

$RES = \{v \mid v \text{ is integer}\} \cup \{\text{nil}, \text{ack}\}$

State:

$X$ , a sequence of integers

**push**( $v$ )

$X := X \cdot v$

return( $\text{ack}$ )

**pop**()

if  $X$  is empty then

return( $\text{nil}$ )

else if  $X = X' \cdot v$  then

$X := X'$

return( $v$ )

Figure 28: **stack**

---

---

$OP = \{\text{read}(i), \text{write}(v, i), \text{move}(i) \mid v, i \in \{0, 1\}\}$

$RES = \{0, 1, \text{ack}\}$

State:

$X_0, X_1 \in \{0, 1\}$

**read**(i)

return( $X_i$ )

**write**( $v, i$ )

$X_i := v$

return( $\text{ack}$ )

**move**( $i$ )

$X_{\bar{i}} := X_i$

return( $\text{ack}$ )

Figure 29: move

---

---

$OP = \{\text{read}(i), \text{write}(v, i), \text{swap}() \mid v, i \in \{0, 1\}\}$

$RES = \{0, 1, \text{ack}\}$

State:

$X_0, X_1 \in \{0, 1\}$

**read**(i)

return( $X_i$ )

**write**( $v, i$ )

$X_i := v$

return( $\text{ack}$ )

**swap**()

$temp := X_0$

$X_0 := X_1$

$X_1 := temp$

return( $\text{ack}$ )

Figure 30: memory-to-memory swap

---

---

$OP = \{\text{write}(v) | v \in \{0, 1\}\} \cup \{\text{read}()\}$   
 $RES = \{0, 1, \text{ack}\}$

State:

$X \in \{0, 1, \perp\}$ , initially  $\perp$

`read()`  
     `return(X)`

`write(v)`  
     if  $X = \perp$  then  
          $X := v$   
     `return(ack)`

Figure 31: sticky-bit

---

## References

- [AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 47–58, August 1992. To appear in *JACM*.
- [Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990.
- [BGP89] P. Berman, J. Garay, and K.J. Perry. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 410–415, 1989.
- [Blo87] B. Bloom. Constructing two writer atomic registers. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.
- [BP87] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.
- [CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.
- [Coa87] B.A. Coan. *Achieving consensus in fault-tolerant distributed computer systems: protocols, lower bounds, and simulations*. PhD thesis, Massachusetts Institute of Technology, June 1987.

- [CW92] B.A. Coan and J.L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, 1992.
- [DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
- [DRS90] D. Dolev, R. Reischuk, and H.R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.
- [FLM86] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.
- [FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.
- [Her88] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, 1988.
- [Her91a] M.P. Herlihy. Randomized wait-free concurrent objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 11–21, August 1991.
- [Her91b] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.
- [HV91] S. Haldar and K. Vidyasankar. A simple construction of 1-writer multi-reader multi-valued atomic variable from regular variables. Technical Report Technical Report No: 9108, Memorial University of Newfoundland, Department of Computer Science, Memorial University of Newfoundland, St. John’s, NF, Canada, A1C 5S7, 1991.
- [HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
- [JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. (Appeared in *Lecture Notes in Computer Science*, Springer-Verlag, No: 647).
- [KM93] J. Kleinberg and S. Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, August 1993.
- [LAA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

- [Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.
- [Lam78] L. Lamport. The implementation of reliable distributed multi-process systems. *Computer Networks*, 2:95–114, 1978.
- [Lam86] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.
- [LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [LT88] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT, MIT Laboratory for Computer Science, 1988.
- [NT90] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.
- [NW87] R. Newman-Wolf. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 232–248, 1987.
- [PB87] G. Peterson and J. Burns. Concurrent reading while writing ii: the multi-writer case. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.
- [Pet83] G. L. Peterson. Concurrent reading while writing. *ACM TOPLAS*, 5(1):56–65, 1983.
- [Plo89] S. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.
- [PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
- [SAG87] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register, revisited. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 206–221, 1987.
- [Sch88] R. Schaffer. On the correctness of atomic multi-writer registers. Technical report, TR No: MIT/LCS/TM-364, MIT Laboratory for Computer Science, 1988.
- [Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
- [ST87] T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

- [VA86] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986.
- [Vid88] K. Vidyasankar. Converting lamport's regular register to atomic register. *IPL*, 28:287–290, 1988.
- [Vid89] K. Vidyasankar. An elegant 1-writer multireader multivalued atomic register. *IPL*, 30:221–223, 1989.