Fault-Tolerant Wait-Free Shared Objects*

Prasad Jayanti
Tushar D. Chandra**
Sam Toueg

TR 92-1281
April 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG 2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp.
**Also supported by an IBM graduate fellowship.

Fault-tolerant Wait-free Shared Objects*

Prasad Jayanti Tushar D. Chandra† Sam Toueg

Department of Computer Science, Cornell University
Ithaca, New York 14853, USA

{prasad,chandra,sam}@cs.cornell.edu

Abstract

A *concurrent system* consists of processes communicating via shared objects, such as shared variables, queues, etc. The concept of *wait-freedom* was introduced to cope with *process* failures: each process that accesses a wait-free object is guaranteed to get a response even if all the other processes crash. But what if these wait-free objects themselves fail? For example, if a wait-free object "crashes", all the processes that access that object are prevented from making progress. In this paper, we introduce the concept of *fault-tolerant wait-free objects*, and study the problem of implementing them. We give a *universal* method to construct fault-tolerant wait-free objects, for *all* types of "responsive" failures (including one in which faulty objects may "lie"). In sharp contrast, we prove that many common and interesting object types (such as queues, sets, and test&set) have no fault-tolerant wait-free implementations even under the most benign of the "non-responsive" types of failure. We also introduce several concepts and techniques that are central to the design of fault-tolerant concurrent systems: the concepts of self-implementation and graceful degradation, and techniques to automatically increase the fault-tolerance of implementations. We prove matching lower bounds on the resource complexity of most of our algorithms.

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp.

†Also supported by an IBM graduate fellowship.

1 Introduction

1.1 Background and motivation

A *concurrent system* consists of processes communicating via shared objects. Examples of shared object types include data structures such as read/write `register`, `queue`, `set`, and `tree`, and synchronization primitives such as `test&set`, `fetch&add`, and `compare&swap`. Even though different processes may concurrently access a shared object, the object must behave as if all these accesses occur in some sequential order. More precisely, the behavior of a shared object must be *linearizable* ([HW90]). One way to ensure linearizability is to implement shared objects using critical sections [CHP71]. This approach, however, is not fault-tolerant: The crash of a process while in the critical section of a shared object can permanently prevent the rest of the processes from accessing that object. This lack of fault-tolerance led to the concept of *wait-free implementations* of shared objects. Informally, a shared object is wait-free if every operation invocation on that object is returned a response even if some or all other processes in the system crash.

Thus, a concurrent system in which all shared objects are wait-free is resilient to *process* crashes. However, such a system is not resilient to *shared object* failures.[1] For example, the "crash" of a single shared object stops all the processes that need to access that object. Motivated by this observation, we study the problem of implementing wait-free shared objects that are also *fault-tolerant*. With such objects, the system is guaranteed to make progress despite process crashes *and* the failures of some underlying objects. To the best of our knowledge, the issue of *fault-tolerant wait-free shared objects* has not been addressed before. (To simplify notation, hereafter "object" denotes a "shared object".)

1.2 Object failures

We classify object failures into two broad categories: *Responsive* and *non-responsive*. We require that objects subject to responsive failures continue to respond (in finite time) to operation invocations. The responses may be incorrect. In contrast, objects subject to non-responsive failures are exempt from responding to operation invocations. Such objects may "hang" on the invoking process.

We divide responsive failures into three sub-classes: *R-crash*, *R-omission*, and *R-arbitrary*. An object subject to R-crash failure behaves correctly until it fails, and once it fails, it returns a distinguished response $\perp$ to every invocation. As with R-crash, an object subject to R-omission failures may return the correct response or a $\perp$. However, even if it responds $\perp$ to a process p, a subsequent operation invocation by a different process q may get a correct response. This behavior models an object $\mathcal{O}$ made of several components, some of which failed. The operation by p "ran into" a failed component of $\mathcal{O}$, while the one by q only encountered correct components of $\mathcal{O}$. Finally, objects subject to R-arbitrary failures may "lie", *i.e.*, return arbitrary responses to operation invocations.

[1] Even "software" objects have underlying hardware components. The software and/or the hardware could be faulty.

Similarly, we divide non-responsive failures into *crash*, *omission*, and *arbitrary*. An object subject to crash failure behaves correctly until it fails, and once it fails, it never responds to operation invocations. An object subject to omission failures may fail to respond to the invocations of an arbitrary subset of processes, but continue to respond to the invocations of the remaining processes (forever). The behavior of an object subject to an arbitrary failure is completely unrestricted: it may not respond to an invocation, and even if it does, the response may be arbitrary.

1.3 Fault-tolerant objects

Let T be an object type and let $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ be a list of object types. A function $\mathcal{I} : T_1 \times T_2 \times \ldots \times T_n \rightarrow T$ is an *implementation of T from $\mathcal{L}$* if $\mathcal{O} = \mathcal{I}(o_1, o_2, \ldots, o_n)$ is a wait-free object of type T whenever o_i $(1 \leq i \leq n)$ is a wait-free object of type T_i. We call $\mathcal{O}$ a *derived object (of $\mathcal{I}$)* and o_i's the *base objects* of $\mathcal{O}$. $\mathcal{I}$ is *t-tolerant for a failure model $\mathcal{M}$* if $\mathcal{O}$ behaves correctly even if a maximum of t base objects of $\mathcal{O}$ fail according to $\mathcal{M}$.

The implementation $\mathcal{I}$ is a *self-implementation* if $T_1 = T_2 = \ldots = T_n = T$. In other words, in a self-implementation the base objects are required to be of the same type as the derived object. For example, consider the object type 2-process **queue** (i.e., a queue that can be accessed by at most two processes). In Section 6.3, we show that (for every t) there is a t-tolerant self-implementation of 2-process **queue** for R-arbitrary failures. Intuitively, this means that using a set of wait-free 2-process queues, t of which are subject to R-arbitrary failures, we can implement a *failure-free* wait-free 2-process queue. Thus in a self-implementation fault-tolerance is achieved through replication.

1.4 Results

To study whether a general object type has a t-tolerant implementation, we focus on two particular object types: **consensus**[2] and **register**. Herlihy [Her91] and Plotkin [Plo89] showed that one can implement a wait-free object of any type using only consensus and register objects. Thus, if **consensus** and **register** have t-tolerant implementations, then every object type has a t-tolerant implementation.

We first study the problem of tolerating responsive failures. We give t-tolerant *self*-implementations of **consensus** for R-crash, R-omission, and R-arbitrary failures. For R-crash and R-omission failures, our self-implementation is optimal requiring only $t+1$ base consensus objects if t of them may fail. For R-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 R-crash, R-omission, and R-arbitrary failures. Combining the above results with [Her91, Plo89], we conclude that *every* object type T has a t-tolerant implementation (from **consensus** and **register**) for *all* responsive models of failures. Moreover, if T implements **consensus** and **register**, then T has a t-tolerant *self*-implementation. This

[2] A consensus object supports two operations, *propose 0* and *propose 1*, and satisfies the following two properties. An operation gets a response v only if there is some prior invocation of *propose* v. Further, the response is the same for all invocations of both operations.

implies that familiar object types such as (2-process) `queue`, `stack`, `test&set`, `fetch&add`, and (N-process) `compare&swap` have t-tolerant self-implementations even for R-arbitrary failures!

What about tolerating non-responsive failures? Unfortunately, the results are mostly negative. We show that there is no 1-tolerant implementation of `consensus` even for crash failures, the most benign of the non-responsive models of failures.[3] This immediately implies that any object type T that implements `consensus` (such as `queue`, `stack`, `test&set`, `swap`, `compare&swap`, etc.) has no 1-tolerant implementations for crash failures. In contrast, we show that `register` has a t-tolerant *self*-implementation even for arbitrary failures. In addition to these universality and impossibility results, this paper contains the following results.

Let $\mathcal{I}$ be a t-tolerant implementation for failure model $\mathcal{M}$. By definition, every derived object of $\mathcal{I}$ is guaranteed to behave correctly even if up to t base objects fail according to $\mathcal{M}$. But what happens if more than t base objects fail? In general, the derived object may experience a more severe failure than $\mathcal{M}$! We say a t-tolerant implementation for a failure model $\mathcal{M}$ is *gracefully degrading* if the failure of more than t base objects (according to $\mathcal{M}$) cannot cause the derived object to experience a more severe failure than $\mathcal{M}$. From a 1-tolerant gracefully degrading self-implementation of any object type T for a failure model $\mathcal{M}$, we show how to recursively construct a t-tolerant self-implementation of T for $\mathcal{M}$. This provides a method for automatically increasing the fault-tolerance of an object.

In general, graceful degradation increases the cost of an implementation. For instance, consider t-tolerant implementations of `consensus` for R-omission failures. As already mentioned, there is such an implementation using only $t + 1$ base objects. However, this implementation is *not* gracefully degrading. In fact, we show that, in this case, graceful degradation requires at least $2t + 1$ base objects, and we give a matching algorithm.

We prove that there is a large class of object types that have no gracefully degrading implementations for R-crash. Intuitively, this means that whatever the implementation, the failure of the implemented object will be more severe than R-crash, even if all its base objects can only fail by R-crash.

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

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

[3] The impossibility of implementing a fault-tolerant consensus *object* from any finite list of base *objects*, one of which may crash, is shown using the impossibility of solving the consensus *problem* among a finite number of *processes*, one of which may crash [LAA87].

2 Preliminaries

A *concurrent system* consists of processes communicating via (shared) objects. A process interacts with an object by invoking an operation, and receiving a corresponding response from the object. Processes may exhibit arbitrary variations in their execution speeds. Further processes may crash. That is, a process may stop at any point in its execution and never take any steps thereafter.

An object is specified by a *type*. An object type T is defined by $N(T)$, $OP(T)$ and $A(T)$, where $N(T)$ is the maximum number of processes that may access an object $\mathcal{O}$ of type T, $OP(T)$ is the set of operations supported by $\mathcal{O}$, and $A(T)$ specifies how $\mathcal{O}$ behaves when these operations are applied *sequentially*. For concreteness, we assume $A(T)$ is a finite/infinite state non-deterministic automaton where some states are designated as initial states. There is a transition from state s to state t labeled (op, v) iff invoking the operation op when the object is in state s may leave the object in state t, returning the response v. We say a sequential execution $\mathcal{S} = (op_1, v_1), (op_2, v_2), \ldots, (op_k, v_k)$ *from state* s is *consistent with* T iff, viewing $A(T)$ as a directed graph with states as nodes and transitions as directed edges, there is a directed path labeled $\mathcal{S}$ from state s. Further $\mathcal{S}$ is *consistent with* T if there is some initial state s of T such that $\mathcal{S}$ from s is *consistent with* T.

Each process may have at most one pending invocation on any given object. That is, a process p cannot invoke an operation on an object $\mathcal{O}$ unless the previous operation of p on object $\mathcal{O}$ has already received a response. However, operations from different processes may overlap on an object. The sequential specification is therefore not sufficient to understand the behavior of an object. We use *linearizability* defined by Herlihy and Wing [HW90] as the criterion for the correctness of an object. Informally, linearizability requires every operation execution to appear to take effect instantaneously at some point in time between its invocation and response. We make this more formal below.

Let $\mathcal{O}$ be an object shared by the processes p_i, $i = 1, N$. Let E_t be an execution of the concurrent system $(p_1, p_2, \ldots, p_N, \mathcal{O})$ up to time t. Define $\mathcal{H}(E_t)$, the *history of the execution* E_t, as follows: $(p_i, op, v, t_s, t_e) \in \mathcal{H}(E_t)$ iff process p_i invokes operation op in E_t at time t_s, and that operation completes at time t_e returning the response v. Further, $(p_i, op, *, t_s, \infty) \in \mathcal{H}(E_t)$ iff process p_i invokes operation op in E_t at time t_s, and that operation does not complete by time t. We say $\mathcal{H}(E_t)$ is *linearizable with respect to type* T if and only if there exist a sequence $\mathcal{S}$ of (operation, response) pairs and a one-to-one correspondence f from $\mathcal{H}(E_t)$ to $\mathcal{S}$ satisfying the following:

- $\mathcal{S}$ is consistent with respect to T.
- $|\mathcal{S}| = |\mathcal{H}(E_t)|$, *i.e.*, there are exactly as many elements in the sequence $\mathcal{S}$ as there are in the set $\mathcal{H}(E_t)$.
- If $\eta = (p_i, op, v, t_s, t_e) \in \mathcal{H}(E_t)$ and $f(\eta) = \mathcal{S}_j$, then $\mathcal{S}_j = (op, v)$. (Here $\mathcal{S}_j$ denotes the j^{th} element of the sequence $\mathcal{S}$.)
- If $\eta = (p_i, op, *, t_s, \infty) \in \mathcal{H}(E_t)$ and $f(\eta) = \mathcal{S}_j$, then $\mathcal{S}_j = (op, v)$ for some $v \in \mathcal{Z}$.

- If $\eta' = (p_i, op', v', t'_s, t'_e), \eta'' = (p_j, op'', v'', t''_s, t''_e) \in \mathcal{H}(E_t)$, and $t'_e < t''_s$, then $f(\eta') = \mathcal{S}_k$, and $f(\eta'') = \mathcal{S}_l$ for some $k < l$.

An object $\mathcal{O}$ *is of type* T if for every t, and every execution E_t of the concurrent system $(p_1, p_2, \ldots, p_N, \mathcal{O})$ up to time t, $\mathcal{H}(E_t)$ is linearizable with respect to T. We say that that T is an N*-process type*, if $N = N(T)$. Any object of an N-process type is an N*-process object.*

Objects are either *primitive* or *derived.* A primitive object is completely "external" to the invoking process. In other words, after a process invokes an operation on a primitive object, it may simply wait for the object to return the response. In contrast, a derived object $\mathcal{O}$ is "implemented" in software from base objects (each one of which is either derived or primitive). Such an implementation provides a procedure $\texttt{Apply}(p_i, op, \mathcal{O})$ (for each $op \in OP(T)$ and $1 \leq i \leq N(T)$) that process p_i must execute in order to invoke an operation op on $\mathcal{O}$ and receive the corresponding response from $\mathcal{O}$. Each *step* in $\texttt{Apply}(p_i, op, \mathcal{O})$ is either an invocation on a base object of $\mathcal{O}$, or checking if a base object has returned a response to a previous invocation[4], or some local computation.

We now define wait-freedom for primitive and derived objects. A *primitive object is wait-free* if every operation invocation by every process gets a response in finite time. A *derived object* $\mathcal{O}$ *is wait-free* if $\texttt{Apply}(p_i, op, \mathcal{O})$ (for each $op \in OP(T)$ and $1 \leq i \leq N(T)$) returns a response in a finite number of steps, regardless of the execution speeds of the remaining processes. Unless mentioned otherwise, all the objects considered in this paper are wait-free.

3 Models of failure

An object is only an abstraction with a multitude of possible implementations. For instance, it may be implemented 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 implementation fails to provide the intended abstraction.

Object failures may lead to unsatisfactory system behavior. For instance, the "crash" of an object prevents the progress of all processes that access the object. Similarly, if the object returns "incorrect" responses, the system behavior also becomes incorrect. It is therefore important to implement derived objects that behave correctly even if some of the base objects of the implementation fail. The cost and the complexity of such a fault-tolerant implementation depends on the *failure model, i.e.,* the manner in which a failed base object departs from its expected behavior. In this paper, we define a spectrum of failure models that fall into two broad classes: *Responsive* and *non-responsive.*

[4]Note that p_i does not "block" for the response from the object; It only "polls" for the response, then proceeds to the next step.

3.1 Responsive models of failure

An object subject to responsive failures responds to every operation invocation. The response is possibly incorrect, but the object never fails to respond. We describe below three increasingly severe models of responsive failures.

3.1.1 R-crash

R-crash is the most benign model of object failure. This model is based on the premise that an object detects when it becomes faulty. Informally, an object subject to R-crash behaves correctly until it fails, and once it fails, it returns a distinguished response $\perp$ to every operation invocation. More precisely, an object $\mathcal{O}$ of type T subject to R-crash failure satisfies the following three properties. Let E_t be any execution of the concurrent system $(p_1, p_2, \ldots, p_N, \mathcal{O})$ up to time t, and $\mathcal{H}(E_t)$ be the corresponding history, as defined before.

1. $\mathcal{O}$ is wait-free.
2. If $(p, op, \perp, t_s, t_e) \in \mathcal{H}(E_t)$, $(p', op', v', t'_s, t'_e) \in \mathcal{H}(E_t)$, and $t_e < t'_s$, then $v' = \perp$.
3. Let $\mathcal{H}'(E_t) = \mathcal{H}(E_t) - \{(p, op, \perp, t_s, t_e) \in \mathcal{H}(E_t)\}$. Then $\mathcal{H}'(E_t)$ is linearizable with respect to T.

3.1.2 R-omission

Suppose $\mathcal{O}$ is a wait-free object implemented from some "hardware components". We informally argue that $\mathcal{O}$ may exhibit a more severe failure than R-crash, even if one of its "hardware components", say f, fails by R-crash. If a process p executes an operation op on $\mathcal{O}$ that accesses f, f returns $\perp$ to p, causing p to return $\perp$ for op. Suppose a different process q later executes some operation op' on $\mathcal{O}$ and op' does not require q to access f. Process q does not "notice" the failure of f, and thus completes op' returning a non-$\perp$ response. This violates the "once $\perp$, everafter $\perp$" property of R-crash.

Suppose that after p gets $\perp$ it does not access $\mathcal{O}$ again. To q, this scenario is indistinguishable from one in which p had crashed just before accessing f. Since the implementation of $\mathcal{O}$ from its components is wait-free, it is designed to tolerate p's apparent crash, and the non-$\perp$ response to q must be correct.

In view of these considerations[5], we formalize the R-omission model of failure as follows. An object $\mathcal{O}$ of type T subject to R-omission failures satisfies the following properties.

1. $\mathcal{O}$ is wait-free.
2. Let E_t be any execution of $(p_1, p_2, \ldots, p_N, \mathcal{O})$ up to time t with the following property: If a process p_i gets a response $\perp$ from $\mathcal{O}$ for some invocation in E_t, then p_i does not

[5] A formal justification for the R-omission model is given in Section 8.

invoke any operation on $\mathcal{O}$ subsequently in E_t. Defining $\mathcal{H}(E_t)$ as before, obtain $\mathcal{H}'(E_t)$ by replacing every tuple of the form $(p, op, \perp, t_s, t_e)$ by $(p, op, *, t_s, \infty)$. Then $\mathcal{H}'(E_t)$ is linearizable with respect to T.

3.1.3 R-arbitrary

An object subject to R-arbitrary failures is free to return arbitrary responses to operation invocations. The only property we require from such an object is that it be wait-free.

3.2 Non-responsive models of failure

Each responsive model of failure has its non-responsive counter-part. The difference lies in the fact that an object subject to a non-responsive failure model may also fail to respond to operation invocations.

3.2.1 Crash

Crash is the most benign of all non-responsive models of failure. Informally, an object subject to a crash failure behaves correctly until it fails, and once it fails, it never responds to any operation invocations. More precisely, an object $\mathcal{O}$ of type T subject to a crash failure satisfies the following properties.

1. If in a (temporally) infinite execution of the concurrent system $(p_1, p_2, \ldots, p_N, \mathcal{O})$, $\mathcal{O}$ never responds to an invocation of some process p_i, then the total number of responses from $\mathcal{O}$ in that (temporally) infinite execution is finite.
2. If E_t is any execution of the concurrent system $(p_1, p_2, \ldots, p_N, \mathcal{O})$ up to time t, and $\mathcal{H}(E_t)$ is the corresponding history, then $\mathcal{H}(E_t)$ is linearizable with respect to T.

3.2.2 Omission

Omission failures are more severe than crash. An object subject to omission failures satisfies only property 2 of the crash model.

3.2.3 Arbitrary

An object subject to arbitrary failures is not required to satisfy any properties at all. Thus the behavior of such an object is completely unrestricted. In particular, the object may choose not to respond to an invocation. Even if it does, the response can be arbitrary.

4 Definition of fault-tolerant implementations

Let T be an object type and let $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ be a list of object types (T_i's are not necessarily distinct). A function $\mathcal{I} : T_1 \times T_2 \times \ldots \times T_n \rightarrow T$ is an *implementation of T from $\mathcal{L}$* if $\mathcal{O} = \mathcal{I}(o_1, o_2, \ldots, o_n)$ is a wait-free object of type T whenever o_i $(1 \leq i \leq n)$ is a wait-free object of type T_i. We call $\mathcal{O}$ a *derived object (of $\mathcal{I}$)* and o_i's the *base objects* of $\mathcal{O}$. The *resource complexity* of $\mathcal{I}$ is n, the number of base objects that make up a derived object of the implementation. $\mathcal{I}$ is *t-tolerant for a failure model $\mathcal{M}$* if $\mathcal{O}$ behaves correctly[6] even if a maximum of t base objects of $\mathcal{O}$ fail according to $\mathcal{M}$. Note that, in general, if *more than t* base objects fail according to $\mathcal{M}$, $\mathcal{O}$ may experience a more severe failure than $\mathcal{M}$. We say that $\mathcal{I}$ is *gracefully degrading* if: when base objects only fail according to $\mathcal{M}$, $\mathcal{O}$ is only subject to failures of type $\mathcal{M}$.[7]

The implementation $\mathcal{I}$ is a *self-implementation* if $T_1 = T_2 = \ldots = T_n = T$. In other words, in a self-implementation the base objects are required to be of the same type as the derived object.

5 Some basic results

Gracefully degrading self-implementations have the desirable property that they can be composed recursively to realize any extent of fault-tolerance. This is formalized in the following lemma.

Lemma 5.1 (Booster Lemma) *If a type T has a t-tolerant gracefully degrading self-implementation $\mathcal{I}$ of resource complexity n for a failure model $\mathcal{M}$, then T has a $(t^2 + 2t)$-tolerant gracefully degrading self-implementation of resource complexity n^2 for $\mathcal{M}$.*

Proof (sketch) Let $\mathcal{I} = \lambda(o_1, o_2, \ldots, o_n)\, F(o_1, o_2, \ldots, o_n)$. Define $\mathcal{I}' = \lambda(o_1, o_2, \ldots, o_{n^2})\, F(F(o_1, \ldots, o_n), F(o_{n+1}, \ldots, o_{2n}), \ldots, F(o_{(n-1)n+1}, \ldots, o_{n^2}))$. It is easy to verify that $\mathcal{I}'$ is a gracefully degrading $(t^2 + 2t)$-tolerant self-implementation of T for $\mathcal{M}$. □

Recursive application of the booster lemma gives the following corollary.

Corollary 5.1 *If a type T has a 1-tolerant gracefully degrading self-implementation of resource complexity k for a failure model $\mathcal{M}$, then T has a t-tolerant gracefully degrading self-implementation of resource complexity $O(t^{\log k})$ for $\mathcal{M}$.*

In Section 6.1.4, we illustrate how this corollary can be applied to construct a t-tolerant self-implementation of **consensus** for R-arbitrary failures.

Our next result states that arbitrary failures have a responsive (R-arbitrary) and a non-responsive (omission) component. Thus the problem of tolerating arbitrary failures can be

[6] That is, $\mathcal{O}$ remains wait-free and linearizable with respect to T.

[7] Even if all the base objects of $\mathcal{O}$ fail!

reduced to two strictly simpler problems: tolerating R-arbitrary failures and tolerating omission failures.

Lemma 5.2 (Decomposability of arbitrary failures) *A type T has a t-tolerant self-implementation for arbitrary failures if and only if T has a t-tolerant self-implementation $\mathcal{I}'$ for R-arbitrary failures and $\mathcal{I}''$ for omission failures.*

Proof (sketch) The "only if" direction is obvious. To prove the "if" direction, suppose there are $\mathcal{I}' = \lambda(o_1, o_2, \ldots, o_m)\, F_A(o_1, o_2, \ldots, o_m)$ and $\mathcal{I}'' = \lambda(o_1, o_2, \ldots, o_n)\, F_O(o_1, o_2, \ldots, o_n)$. Define $\mathcal{I} = \lambda(o_1, o_2, \ldots, o_{nm})\, F_O(F_A(o_1, \ldots, o_m), \ldots, F_A(o_{(n-1)m+1}, \ldots, o_{nm}))$. It can be verified that $\mathcal{I}$ is a t-tolerant self-implementation of T for arbitrary failures. □

6 Tolerating responsive failures

To study whether an arbitrary object type has a t-tolerant implementation, we focus on two particular object types: **consensus** and **register**. Herlihy [Her91] and Plotkin [Plo89] showed that one can implement a wait-free object of any type using only consensus and register objects. Thus, if **consensus** and **register** have t-tolerant implementations, then every object type has a t-tolerant implementation.

6.1 Fault-tolerant implementation of consensus

In the following, we first define the object type **N-consensus**. We then present a t-tolerant self-implementation of **N-consensus** that works for both R-crash and R-omission failures. This implementation requires $t + 1$ base N-consensus objects, and is resource optimal. Following that, we show how to translate R-arbitrary failures of N-consensus objects to R-omission failures. Our translation is also proved to be resource optimal. Although the above two results can be chained together to obtain a t-tolerant self-implementation of **N-consensus** for R-arbitrary failures, the resultant self-implementation is not resource efficient: it requires $O(t^2)$ base consensus objects. We therefore present an alternative efficient self-implementation of resource complexity $O(t \log t)$.

6.1.1 N-consensus object type

The *consensus problem* for a system of N processes is defined as follows. Each process p_i is given a binary input v_i initially. The consensus problem requires each correct process to eventually reach the same (irrevocable) decision value d such that $d \in \{v_1, v_2, \ldots, v_N\}$. The object type **N-consensus** is defined so that an object of this type makes the consensus problem solvable in a system of N processes.

N-consensus is an N-process type that supports two operations, *propose 0* and *propose 1*, and has the following sequential specification. If the first operation invoked is *propose v*, then every invocation (including the first) is returned the response v. Together with

linearizability, this sequential specification implies that an object $\mathcal{O}$ is of type `N-consensus` iff it satisfies the following three properties:

- Validity: $\mathcal{O}$ returns a response $v \in \{0,1\}$ to an invocation (from process p) only if there is a prior invocation of *propose* v on $\mathcal{O}$ (by some process, possibly p itself).
- Agreement: If $\mathcal{O}$ returns v_1, v_2 to two invocations, and $v_1, v_2 \in \{0,1\}$, then $v_1 = v_2$.
- Integrity: The response returned to an invocation by $\mathcal{O}$ is either 0 or 1.

Let *loc* := Propose(p, v, $\mathcal{O}$) denote that process p invokes *propose* v on $\mathcal{O}$ and stores the response returned in its local variable *loc*.

6.1.2 Tolerating R-crash and R-omission failures

We present a t-tolerant self-implementation of `N-consensus` for R-omission failures. Since R-omission failures are strictly more severe than R-crash, the same implementation also works for R-crash failures.

A consensus object satisfies *weak integrity* if every response returned by the object is in $\{0, 1, \perp\}$.

Proposition 6.1 *Any N-consensus object that fails by R-omission satisfies validity, agreement, and weak integrity. Conversely, if a failed N-consensus object satisfies validity, agreement, and weak integrity, then the failure is R-omission.*

Proof Follows from the definitions. □

$O_1, O_2, \ldots, O_{t+1}$: `N-consensus` objects

```
Procedure Propose(p, v_p, O)        /* v_p ∈ {0,1} */
    estimate_p, w, k : integer local to p
begin
    estimate_p := v_p
    for k := 1 to t + 1 do
        w := propose(p, estimate_p, O_k)
        if w ≠ ⊥ then estimate_p := w
    return(estimate_p)
end
```

Figure 1: t-tolerant self-implementation of `N-consensus` for R-omission

---

Theorem 6.1 *Figure 1 gives a t-tolerant self-implementation of* `N-consensus` *for R-omission failures. The resource complexity of the implementation is* $t+1$ *and is optimal.*

Proof (Sketch)

Assume that at most t base objects fail by R-omission. We show below that the derived object $\mathcal{O}$ is a correct N-consensus object.

1. $\mathcal{O}$ satisfies validity: Using Proposition 6.1, and the fact that p does not change $estimate_p$ if a base object returns $\perp$, it is easy to verify by an induction on k that if $estimate_p$ equals some value u at any point, then there is a prior invocation (from some process q) of `Propose`$(q, u, \mathcal{O})$.

2. $\mathcal{O}$ satisfies agreement: Since at most t base objects fail, there is an O_k $(1 \leq k \leq t+1)$ that does not fail. 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, and due to Proposition 6.1, it never changes thereafter. Thus $\mathcal{O}$ returns the same response w to every p.

It is obvious that $\mathcal{O}$ always returns 0 or 1, and that $\mathcal{O}$ is wait-free.

Any t-tolerant self-implementation for R-omission failures must handle the case where t base objects fail (by R-crash) initially. It is therefore obvious that the resource complexity of $t+1$ of our self-implementation is optimal. □

The above (self) implementation is *not* gracefully degrading. For instance, suppose that $v_p = 0$ and $v_q = 1$, and the $t+1$ base objects fail by R-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 6.1, the failure of $\mathcal{O}$ is more severe than R-omission. In fact, we will now show that $2t+1$ is both a lower and upper bound on the resource complexity of a t-tolerant *gracefully degrading* self-implementation of `N-consensus` for R-omission[8]. The self-implementation that requires $2t+1$ base objects is given in Figure 2.

Claim 6.1 *Let v be the value of $estimate_p$ and V be the value of V_p at the end of k iterations $(1 \leq k \leq 2t+1)$ of the* for-loop *of* `Propose`$(p, v_p, \mathcal{O})$ *in Figure 2. Then $v \in \{0,1\}$, and $V_p[1..k]$ contains only $\perp$'s and v's.*

Proof By an easy induction on k. □

Theorem 6.2 *Figure 2 gives a t-tolerant gracefully degrading self-implementation of* `N-consensus` *for R-omission.*

Proof Assume all failures of base objects are by R-omission. We first show that, even if more than t base objects fail, $\mathcal{O}$ satisfies validity, agreement, and weak integrity:

[8] As will be shown later in Theorem 8.2, there is *no* gracefully degrading implementation of `N-consensus` for R-crash.

$O_1, O_2, \ldots, O_{2t+1}$: N-consensus objects

```
    Procedure Propose(p, v_p, O)        /* v_p ∈ {0,1} */
        V_p[1..2t+1], estimate_p, w, k: integer local to p
    begin
1       estimate_p := v_p
2       for k := 1 to 2t+1 do
3           w := propose(p, estimate_p, O_k)
4           V_p[k] := w
5           if (w ≠ ⊥)∧(w ≠ estimate_p) then
6               estimate_p := w
7               V_p[1...(k-1)] := (⊥, ⊥, ..., ⊥)
8       if V_p has more than t ⊥'s then
9           return(⊥)
10      else return(estimate_p)
    end
```

Figure 2: t-tolerant *gracefully degrading* self-implementation of N-consensus for R-omission

1. $\mathcal{O}$ satisfies validity: Using Proposition 6.1, and the fact that a process p does not change $estimate_p$ if a base object returns $\perp$, it is easy to verify by an induction on k that if $estimate_p$ equals some value u at any point, then there is a prior invocation (from some process q) of $\texttt{Propose}(q, u, \mathcal{O})$.

2. $\mathcal{O}$ satisfies agreement: Suppose, for a contradiction, there exist two processes p and q such that $\texttt{Propose}(p, v_p, \mathcal{O})$ returns 0 and $\texttt{Propose}(q, v_q, \mathcal{O})$ returns 1. From Claim 6.1, and lines 8, 9 of the algorithm, it follows that V_p has at least $t+1$ 0's at the end of the execution of $\texttt{Propose}(p, v_p, \mathcal{O})$ and V_q has at least $t+1$ 1's at the end of the execution of $\texttt{Propose}(q, v_q, \mathcal{O})$. This is possible only if there is a k $(1 \leq k \leq 2t+1)$ such that $\texttt{Propose}(p, estimate_p, O_k)$ returned 0 and $\texttt{Propose}(q, estimate_q, O_k)$ returned 1. Thus O_k does not satisfy agreement. By Proposition 6.1, the failure of O_k is not R-omission, a contradiction.

3. $\mathcal{O}$ satisfies weak integrity: Trivial to verify.

4. $\mathcal{O}$ satisfies integrity if at most t base objects fail: Let $O_{k_1}, O_{k_2}, \ldots, O_{k_l}$ $(k_1 < k_2 < \ldots < k_l)$ be all the correct base objects. Since at most t fail, we have $l \geq t+1$. By the integrity and agreement properties of O_{k_1}, there is a $v \in \{0,1\}$ such that for all p, $\texttt{Propose}(p, estimate_p, O_{k_1})$ returns v. Thus for all p $estimate_p = v$ at the end of k_1 iterations of the for-loop in $\texttt{Propose}(p, v_p, \mathcal{O})$. Using this and Proposition 6.1, it is easy to verify that at the end of the execution of $\texttt{Propose}(p, v_p, \mathcal{O})$, $V_p[k_i] = v$

and $estimate_p = v$ for all p and for all $1 \leq i \leq l$. This implies, by lines 8, 9 of the algorithm, that $\texttt{Propose}(p, v_p, \mathcal{O})$ returns v.

From 1, 2, and 4 above, we conclude that the self-implementation is t-tolerant for R-omission. From 1, 2, and 3 above, together with Proposition 6.1, we conclude that the self-implementation is gracefully degrading for R-omission. □

Theorem 6.3 *The resource complexity of any t-tolerant gracefully degrading implementation of* `N-consensus` *($N \geq 2$) for R-omission is at least $2t + 1$.*

Proof For a contradiction, assume that there is a t-tolerant gracefully degrading implementation $\mathcal{I}$ from $\mathcal{L} = \{T_1, T_2, \ldots, T_n\}$, of `N-consensus` for R-omission, where $n \leq 2t$. Let $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$. Consider the following interleaving of processes p and q.

Scenario

1. Process p invokes $\texttt{Propose}(p, 0, \mathcal{O})$ and executes the steps of $\texttt{Propose}(p, 0, \mathcal{O})$ until either it accesses exactly t base objects or it completes the execution of $\texttt{Propose}(p, 0, \mathcal{O})$, whichever is earlier. Let S_p denote the set of base objects accessed by p. Every base object $O \in S_p$ behaves correctly to p's invocations. Note that $|S_p| \leq t$.

2. Process q invokes and completes the execution of $\texttt{Propose}(q, 1, \mathcal{O})$. Let S_q denote the set of base objects accessed by q, and $T_q = S_q - S_p$. The base objects behave as follows: Every base object $O \in S_p$ accessed by q returns $\perp$ to q and undergoes no change in its state; every base object $O \in T_q$ behaves correctly to q's invocations. So q sees at most $|S_p| \leq t$ failures of base objects.

3. Process p resumes execution (thus $|S_p| = t$), and completes any remaining steps of $\texttt{Propose}(p, 0, \mathcal{O})$. The base objects behave as follows: Every $O \in T_q$ accessed by p returns $\perp$ to p; every $O \in S_p - T_q$ accessed by p behaves correctly to q's invocations. Note that $T_q = S_q - S_p \subseteq \{O_1, O_2, \ldots, O_n\} - S_p$, and thus $|T_q| \leq n - t \leq t$. So p sees at most $|T_q| \leq t$ failures of base objects.

In a scenario such as the above, we assume that all steps in item k strictly precede every step in item $k + 1$.

We make the following conclusions from the above scenario.

1. From the characterization of how the failed base objects behave, it is clear that all failures are by R-omission. Since $\mathcal{I}$ is gracefully degrading, the failure of $\mathcal{O}$ is no more severe than R-omission. Thus, by Proposition 6.1, $\mathcal{O}$ satisfies validity, agreement, and weak integrity.

2. In the scenario described, neither process "knows" that the other process is also running. Thus, by validity and weak integrity, $\texttt{Propose}(p, 0, \mathcal{O})$ must return either 0 or $\perp$, and $\texttt{Propose}(q, 1, \mathcal{O})$ must return either 1 or $\perp$.

3. In the scenario described, neither process sees more than t base object failures. Since $\mathcal{I}$ is t-tolerant, it follows that neither $\texttt{Propose}(p, 0, \mathcal{O})$ nor $\texttt{Propose}(q, 1, \mathcal{O})$ may return $\perp$. Together with Conclusion 2, this implies that $\texttt{Propose}(p, 0, \mathcal{O})$ returns 0 and $\texttt{Propose}(q, 1, \mathcal{O})$ returns 1. Thus object $\mathcal{O}$ violates agreement (required by Conclusion 1). We conclude that $\mathcal{I}$ is not a gracefully degrading t-tolerant implementation.

□

6.1.3 Translation from R-arbitrary to R-omission

A *t-tolerant translation from a failure model $\mathcal{M}$ to a* (less severe) *failure model $\mathcal{M}'$ for object type T* is a self-implementation $\mathcal{I} : T \times T \times \ldots \times T \rightarrow T$ such that $\mathcal{O} = \mathcal{I}(o_1, o_2, \ldots, o_n)$ fails according to $\mathcal{M}'$ if a maximum of t base objects of $\mathcal{O}$ fail according to $\mathcal{M}$ (and the remaining base objects are correct). Note that if no base objects fail, by definition of an implementation, $\mathcal{O}$ does not fail either.

In this section, we present a t-tolerant translation from R-arbitrary to R-omission for `N-consensus`. It is easy to see that this translation can be used along with the t-tolerant self-implementation for R-omission to obtain a t-tolerant self-implementation of `N-consensus` for R-arbitrary failures. This is the principal motivation for studying such a translation. We will also show that the resource complexity, $3t + 1$, of our translation is optimal.

Since a consensus object that suffers an R-arbitrary failure may return a non-binary response, we find it convenient to define $\texttt{f-propose}(p,v,O)$ as in Figure 3.

```
Procedure f-propose(p, v, O)
begin
    loc := propose(p, v, O)
    if loc ∈ {0, 1} then
        return(loc)
    else return(0)
end
```

Figure 3: Filtering an arbitrary response to a binary response

---

Let $\mathcal{O}$ be the derived object of the translation in Figure 4. The base objects of $\mathcal{O}$ are $A[1 \ldots 2t + 1]$, $B[1 \ldots t]$. In the following claims, assume that at most t base objects suffer R-arbitrary failures, and the remaining are correct.

Claim 6.2 *$\mathcal{O}$ satisfies weak integrity. Further, if no base object fails, $\mathcal{O}$ satisfies integrity.*

Claim 6.3 *$\mathcal{O}$ satisfies validity.*

$A[1 \ldots 2t+1]$, $B[1 \ldots t]$: wait-free N-consensus objects

```
    Procedure Propose(p, v_p, 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 do
3                    w := 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[¬belief_p].
6                for i := 1 to t do
7                    if belief_p ≠ f-propose(p, belief_p, B[i])
8                        return(⊥)
9                        exit Propose
10               return(belief_p)
    end
```

Figure 4: t-tolerant translation from R-arbitrary to R-omission for N-consensus

Proof Suppose $\mathcal{O}$ returns $v \in \{0,1\}$ to the invocation $\texttt{Propose}(p, v_p, \mathcal{O})$ (from process p). Then $v = belief_p$ (by line 10), and $count_p[v] = count_p[belief_p] \geq t+1$ (by line 5). So there is at least one correct base object $A[i]$ such that $\texttt{propose}(p, v_p, A[i])$ returned v. By validity of $A[i]$, it follows that some process q invoked $\texttt{propose}(q, v_q, A[i])$ where $v_q = v$. This implies that q invoked $\texttt{Propose}(q, v, \mathcal{O})$. □

Claim 6.4 *$\mathcal{O}$ satisfies agreement.*

Proof Suppose $\mathcal{O}$ fails to satisfy agreement by returning $v_1 \in \{0,1\}$ to some process p, and $v_2 \in \{0,1\}$ to a different process q where $v_1 \neq v_2$. $\mathcal{O}$ returns v_1 to p implies $v_1 = belief_p$. Similarly $v_2 = belief_q$. Since $v_1 \neq v_2$, we have $belief_p \neq belief_q$. It is easy to verify that if all of $A[1 \ldots 2t+1]$ are correct, then $belief_p = belief_q$. It follows that at least one of $A[1 \ldots 2t+1]$ fails.

Further $\mathcal{O}$ returns v_1 to p implies for all $1 \leq i \leq t$ $\texttt{propose}(p, belief_p, B[i])$ returns $belief_p = v_1$ to p. Similarly, for all $1 \leq i \leq t$ $\texttt{propose}(q, belief_q, B[i])$ returns $belief_q = v_2$ to q. Thus all t base objects $B[1 \ldots t]$ fail by not satisfying agreement. Thus counting the failed $A[i]$'s and $B[i]$'s, we have more than t failed base objects, a contradiction. □

Together with Proposition 6.1, the above claims trivially imply the following theorem.

Theorem 6.4 *Figure 4 presents a t-tolerant translation from R-arbitrary failures to R-*

omission failures for N-consensus. *The resource complexity of the translation is* $3t + 1$.

Theorem 6.5 *The resource complexity of any translation* $\mathcal{I}$ *from R-arbitrary to R-omission for* N-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 claims, involving "indistinguishable" scenarios. Let $\mathcal{O} = \mathcal{I}(o_1, o_2, \ldots, o_n)$. In the following we say a process p *touches a base object* o_i if during the execution of Propose$(p, v_p, \mathcal{O})$, p executes propose$(p, *, o_i)$.

Claim 6.5 *Suppose* p *executes* Propose$(p, 0, \mathcal{O})$ *to completion. If all base objects are correct, then* p *touches at least* $t + 1$ *base objects.*

Proof Suppose the claim is false, and p touches only $o_{i_1}, o_{i_2}, \ldots, o_{i_m}$ $(m \leq t)$ before exiting Propose$(p, 0, \mathcal{O})$. Since all base objects are correct, $\mathcal{O}$ satisfies validity and integrity. Hence Propose$(p, 0, \mathcal{O})$ returns 0. Now consider the following two scenarios.

Scenario S1

1. p executes Propose$(p, 0, \mathcal{O})$ to completion touching only $o_{i_1}, o_{i_2}, \ldots, o_{i_m}$ $(m \leq t)$. Propose$(p, 0, \mathcal{O})$ returns 0.
2. q executes Propose$(q, 1, \mathcal{O})$ to completion.

Scenario S2

1. $o_{i_1}, o_{i_2}, \ldots, o_{i_m}$ fail and behave as though they are touched by p exactly as in scenario S1. This is possible since $m \leq t$.
2. q executes Propose$(q, 1, \mathcal{O})$ to completion.

Since no base objects fail in S1, $\mathcal{O}$ behaves correctly. In particular, $\mathcal{O}$ satisfies integrity and agreement. Thus Propose$(q, 1, \mathcal{O})$ returns 0 in S1. Clearly S1 $\approx_q$ S2 (We write S1 $\approx_q$ S2 to denote that Scenarios S1 and S2 are indistinguishable to process q). So Propose$(q, 1, \mathcal{O})$ returns 0 in S2 also, violating validity. By Proposition 6.1, this failure of $\mathcal{O}$ in S2 is not R-omission. Since fewer than $t + 1$ base objects fail in S2, the translation $\mathcal{I}$ is incorrect, a contradiction. □

Claim 6.6 *Consider*

Scenario S3

1. p *executes* Propose$(p, 0, \mathcal{O})$ *up to the point where it has exactly touched* t *base objects* $o_{i_1}, o_{i_2}, \ldots, o_{i_t}$.

2. *q executes* **Propose**$(q, 1, \mathcal{O})$ *to completion.*

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

Proof Let $S = \{\text{base objects touched by } q\} - \{o_{i_1}, o_{i_2}, \ldots, o_{i_t}\}$. Let $o_{j_1}, o_{j_2}, \ldots, o_{j_k}$ be all the base objects in S arranged so that the first invocation of q on o_{j_l} is before the first invocation of q on $o_{j_{l+1}}$. Note that $k \leq n - t \leq 2t$.

Let S2′ represent scenario S2 when $m = t$. Since fewer than $t + 1$ base objects fail in S2′, the failure of $\mathcal{O}$ cannot be more severe than R-omission. Hence, by Proposition 6.1, $\mathcal{O}$ satisfies validity and weak integrity in S2′. So **Propose**$(q, 1, \mathcal{O})$ returns 1 or $\perp$ in S2′. Since S2′ $\approx_q$ S3, we conclude **Propose**$(q, 1, \mathcal{O})$ returns 1 or $\perp$ in S3. Further since no base object fails in S3, $\mathcal{O}$ satisfies integrity in S3. So **Propose**$(q, 1, \mathcal{O})$ returns either 0 or 1 in S3. Together the above two conclusions imply the claim. □

Claim 6.7 *Consider*

Scenario S4

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

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

Proof Consider

Scenario S5

1. p executes **Propose**$(p, 0, \mathcal{O})$ up to the point where it has exactly touched t base objects $o_{i_1}, o_{i_2}, \ldots, o_{i_t}$.
2. The base objects $o_{j_1}, o_{j_2}, \ldots, o_{j_{k-t}}$ fail and behave as though they are touched by q exactly as in S4.
3. p completes the execution of **Propose**$(p, 0, \mathcal{O})$.

Since $k \leq 2t$, the number of failed base objects in S5 $= k - t \leq t$, and therefore (by Proposition 6.1) $\mathcal{O}$ satisfies validity and weak integrity. So **Propose**$(p, 0, \mathcal{O})$ returns either 0 or $\perp$ in S5. Since clearly S4 $\approx_p$ S5, **Propose**$(p, 0, \mathcal{O})$ returns either 0 or $\perp$ in S4 also. However since no base object fails in S4, $\mathcal{O}$ must satisfy integrity in S4. Thus **Propose**$(p, 0, \mathcal{O})$ returns 0 in S4. □

Claim 6.8 *Consider*

Scenario S6

1. *p executes* `Propose`$(p, 0, \mathcal{O})$ *up to the point where it has exactly touched t base objects* $o_{i_1}, o_{i_2}, \ldots, o_{i_t}$.
2. *q executes* `Propose`$(q, 1, \mathcal{O})$ *to completion, returning 1, by Claim 6.6.*
3. *Let* $o_{j_1}, o_{j_2}, \ldots, o_{j_k}$ *be as defined above (note* $k \leq 2t$*).* $\{o_{j_{k-t+1}}, o_{j_{k-t+2}}, \ldots, o_{j_k}\}$ *fail and behave as though they are never touched by q.*
4. *p completes the execution of* `Propose`$(p, 0, \mathcal{O})$.

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

Proof Since S5 $\approx_p$ S6, `Propose`$(p, 0, \mathcal{O})$ returns 0 in S6. □

From the above claim, it is clear that $\mathcal{O}$ does not satisfy agreement in S6. Hence, by Proposition 6.1, the failure of $\mathcal{O}$ in S6 is more severe than R-omission. Since fewer than $t+1$ base objects fail in S6, the translation $\mathcal{I}$ is incorrect, a contradiction. This completes the proof of Theorem 6.5. □

6.1.4 Tolerating R-arbitrary failures

Since `N-consensus` has a t-tolerant self-implementation for R-omission failures, and has a t-tolerant translation from R-arbitrary to R-omission failures, it follows that `N-consensus` has a t-tolerant self-implementation for R-arbitrary failures also. However the resulting self-implementation is expensive, requiring $(3t+1)(t+1)$ base objects. Our main goal in this section is to present a t-tolerant self-implementation for R-arbitrary failures whose resource complexity is only $O(t \log t)$. This implementation employs the divide-and-conquer strategy. In the following, we first present the base step: obtaining a 1-tolerant self-implementation (Figure 5). This requires 6 base consensus objects, while the above mentioned approach through translation requires 8 base consensus objects. Then we show the recursive step of obtaining a t-tolerant self-implementation from a $t/2$-tolerant self-implementation (Figure 6).

Claim 6.9 *If at most one of* O_i, O_{i+1}, *and* O_{i+2} *(*$i = 1$ *or 4) fails, then an execution e of* `Access`$(p, O_i, O_{i+1}, O_{i+2}, v)$ *(See Figure 5) returns* $\overline{v}$ *only if there is some other execution* e' *of* `Access`$(q, O_i, O_{i+1},$ and $O_{i+2}, \overline{v})$ *(for some q) that either precedes or is concurrent with e.*

Claim 6.10 *If none of* O_i, O_{i+1}, *and* O_{i+2} *(*$i = 1$ *or 4) fails, then, for all p and q,* `Access`$(p,$ $O_i, O_{i+1}, O_{i+2}, v_p)$ *returns the same value as* `Access`$(q, O_i, O_{i+1}, O_{i+2}, v_q)$.

Theorem 6.6 *Figure 5 gives a* 1*-tolerant gracefully degrading self-implementation of* `N-consensus` *for R-arbitrary failures.*

$\mathcal{O}_i$: **N-consensus** objects $(1 \leq i \leq 6)$

```
Procedure Access(p, O1, O2, O3, v)
    count_p[0..1], w: integer local to p
begin
    count_p[0..1] := (0,0)
    for i := 1 to 3 do
        w := 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

Procedure Propose(p, v, O)
begin
    v := Access(p, O1, O2, O3, v)
    v := Access(p, O4, O5, O6, v)
    return(v)
end
```

Figure 5: 1-tolerant self-implementation of **N-consensus** for R-arbitrary failures

Proof Suppose that at most one of O_i $(1 \leq i \leq 6)$ fails. Then either none of O_1, O_2, and O_3 fails or none of O_4, O_5, and O_6 fails. Validity of $\mathcal{O}$ follows from Claim 6.9. If none of O_4, O_5, and O_6 fails, agreement of $\mathcal{O}$ follows from Claim 6.10. If none of O_1, O_2, and O_3 fails, agreement of $\mathcal{O}$ follows from Claims 6.9 and 6.10. It is obvious that $\mathcal{O}$ always returns 0 or 1, is wait-free, and gracefully-degrading. □

Given the 1-tolerant gracefully degrading self-implementation in Figure 5, by applying the Booster lemma (Lemma 5.1) we can obtain a t-tolerant self-implementation of **N-consensus** for R-arbitrary failures. However, the resulting resource complexity is $O(t^{\log_2 6})$, which is even higher than the complexity of the implementation through translation mentioned above. We therefore present below an alternative efficient recursive strategy. See Figure 6.

Theorem 6.7 *Figure 6 gives a t-tolerant (gracefully degrading) self-implementation of* **N-consensus** *for R-arbitrary failures of resource complexity $O(t \log t)$.*

Proof We prove the theorem through a series of claims. In all of them we assume that at most t base objects fail.

$A_0[1 \ldots 3t+1]$, $A_1[1 \ldots 3t+1]$, $B[1 \ldots 4t+1]$: (0-tolerant) N-consensus objects
O_1 : $\lceil \frac{t-1}{2} \rceil$-tolerant N-consensus object
O_2 : $\lfloor \frac{t-1}{2} \rfloor$-tolerant N-consensus object

```
    Procedure Propose(p, v_p, 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] := (0,0,0,0)

2       Phase 1: for i := 1 to 3t + 1 do
3                    w := 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 := f-propose(p, v_p, O_1)

6       Phase 3: for i := 1 to 4t + 1 do
7                    w := f-propose(p, ans1_p, B[i])
8                    WitnessCount_p[w] := WitnessCount_p[w]+1

9       Phase 4: for i := 1 to 3t + 1 do
10                   w := f-propose(p, v_p, A_{v̄_p}[i])
11                   if w = v̄_p then count_p[v̄_p] := count_p[v̄_p]+1

12      Phase 5: Choose belief_p such that WitnessCount_p[belief_p] > WitnessCount_p[belief̄_p].
13               if WitnessCount_p[belief_p] ≥ 3t + 1 and count_p[belief_p] ≥ 2t + 1 then
14                   return(belief_p); exit Propose
15               if WitnessCount_p[belief_p] ≥ 2t + 1 and count_p[belief_p] ≥ t + 1 then
16                   v'_p := belief_p
17               else v'_p := v_p
18               ans2_p := propose(p, v'_p, O_2)
19               return(ans2_p)
    end
```

Figure 6: Efficient t-tolerant self-implementation of N-consensus for R-arbitrary failures

Claim 6.11 *If O_1 fails, then O_2 does not fail.*

Proof Since O_1 and O_2 are derived objects of $\lceil \frac{t-1}{2} \rceil$-tolerant, and $\lfloor \frac{t-1}{2} \rfloor$-tolerant self-implementations of `N-consensus` respectively, O_1 and O_2 tolerate up to $\lceil \frac{t-1}{2} \rceil$ and $\lfloor \frac{t-1}{2} \rfloor$ failed base objects respectively. Since at most t base objects fail, both O_1 and O_2 cannot fail. □

Claim 6.12 *If O_1 does not fail, then $\mathcal{O}$ satisfies validity and agreement.*

Proof Suppose O_1 does not fail. Since a correct O_1 satisfies agreement, we have $ans1_p = ans1_q = v$ for all p, q. Thus every process proposes the same value v to every $B[i]$ in Phase 3. Since at most t objects in $B[1 \ldots 4t+1]$ lie (fail), $belief_p = v$ and $WitnessCount_p[belief_p] \geq 3t+1$ (for every p).

Also by the validity of O_1, some process q will have invoked $\texttt{propose}(q, v, O_1)$ 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 most t objects in $A_v[1 \ldots 3t+1]$ may lie, 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. □

Claim 6.13 *If O_1 fails, $\mathcal{O}$ satisfies validity and agreement.*

Proof Suppose O_1 fails. Then by Claim 6.11, O_2 does not fail. 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 may fail, it follows that $WitnessCount_q[belief_p] \geq 2t+1$ and $count_q[belief_p] \geq t+1$ (for every q). This implies, by line 12, $belief_q = belief_p$, and let $val = belief_p$. Since $WitnessCount_q[belief_q] \geq 2t+1$ and $count_q[belief_q] \geq t+1$ (for every q), either q returns $belief_q = val$ by line 14 and we have agreement between p and q, or q sets v'_q to $belief_q = val$ by line 16. Thus every q that does not return by line 14 proposes $v'_q = val$ on O_2. Since O_2 does not fail, by validity of O_2, $ans2_q = v'_q = val$, and q returns $ans2_q = val$ 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 $belief_p = val$ on at least $t+1$ objects in $A_{belief_p}[1 \ldots 3t+1]$.

CASE 2 Suppose no process returns by line 14. Then every q returns $ans2_q$ by line 19. Since O_2 does not fail, we have (for all p, q) $ans2_p = ans2_q = val$. Thus $\mathcal{O}$ satisfies agreement.

By the validity of O_2, some process p must have proposed val to O_2. That is $v'_p = val$. 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 $count_p[belief_p] \geq t+1$. This implies, considering that at most t objects in $A_{belief_p}[1 \ldots 3t+1]$ fail, that some process

q proposed $v_q = belief_p$ on some object in $A_{belief_p}[1 \ldots 3t+1]$. Thus $val = v'_p = belief_p = v_q$ and v_q is the initial proposal of q. Thus $\mathcal{O}$ satisfies validity. □

Claim 6.14 *The resource complexity of the implementation in Figure 4 is $O(t \log t)$.*

Proof Denoting the resource complexity of the t-tolerant (gracefully degrading) self-implementation of `N-consensus` for R-arbitrary failures by $f(t)$, we have the following recurrence: $f(t) = 2f(t/2) + 2(3t+1) + (4t+1)$ and $f(1) = 6$. Hence the result. □

To complete the proof of Theorem 6.7, note that agreement and validity follow from Claims 6.12 and 6.13. It is obvious that the implementation is wait-free, gracefully degrading, and that $\mathcal{O}$ satisfies integrity. □

6.2 Fault-tolerant implementation of register

The `register` type supports two operations, *read* and *write v*. The sequential specification is simple: *read* returns the most recent value written. Lamport defined a weaker (non-linearizable) object known as *safe register* [Lam86]. In the following, we first show how to build a fault-tolerant safe register from safe registers, some of which may suffer R-arbitrary failures. We then resort to the register construction results in the literature to show that `register` has a self-implementation for R-arbitrary failures.

Lemma 6.1 *Using $2t+1$ 1-reader, 1-writer safe registers, at most t of which may suffer R-arbitrary failures, we can implement a failure-free 1-reader, 1-writer, safe register.*

Proof (sketch) To read the safe register, the reader reads all base registers, and returns the majority response. If there is no majority, it returns an arbitrary value. To write a value v into the register, the writer writes v to all base registers. It is easy to verify that the above strategy implements a safe register that behaves correctly even if a maximum of t base registers suffer R-arbitrary failures. □

It is possible to implement a multi-reader, multi-writer, atomic register using 1-reader, 1-writer, safe registers [Blo87, BP87, CW90, HV91, Lam86, NW87, Pet83, PB87, Sch88, SAG87, Vid88, Vid89, VA86]. Thus we have the following theorem.

Theorem 6.8 `register` *has a t-tolerant self-implementation for R-arbitrary failures.*

6.3 Universality results

We now describe how to implement fault-tolerant wait-free shared objects of a generic type. An object type T is *finite* if $A(T)$ has only a finite number of states. Also let `N-consensus with reset` be an N-process object type informally defined as follows: An object O of this type behaves exactly like an object of type `N-consensus` with the difference that O supports an extra operation *reset*. Applying "reset" to O will initialize O and make it available for

a fresh round of consensus. The operation "reset" is required to work only in the absence of concurrent operations[9].

Herlihy showed that every finite object type[10] has an implementation from (`N-consensus with reset, unbounded register`) ([Her91]). The use of unbounded registers was replaced by boolean registers by Plotkin ([Plo89]). Using Plotkin's result, together with Theorems 6.7 and 6.8, we obtain the following corollary.

Corollary 6.1

- *Every finite object type has a t-tolerant implementation from* (`N-consensus with reset, boolean register`) *for R-arbitrary failures.*
- *If a finite object type implements* `N-consensus with reset` *and* `boolean register` *then T has a t-tolerant* self-*implementation for R-arbitrary failures.*

Herlihy's construction can be easily modified to yield a universal implementation from (`N-consensus with reset, unbounded register`) even for *infinite* object types. Thus Corollary 6.1 holds even if T is an infinite object type, provided that `boolean register` is replaced by `unbounded register` in the statement of the corollary.

Herlihy showed that `queue, stack, test&set, fetch&add` *etc.* implement `2-consensus`, and `compare&swap` implements `N-consensus` [Her91]. It is easy to show that `test&set` and `compare&swap` implement `boolean register`, and `queue, stack`, and `fetch&add` implement `unbounded register`. Thus,

Corollary 6.2 *The following object types have t-tolerant self-implementations for R-arbitrary failures: (2-process)* `queue, stack, test&set, fetch&add`, *and (N-process)* `compare&swap`.

7 Tolerating non-responsive failures

Unlike responsive failures, non-responsive failures are almost always impossible to cope with. We first show the impossibility of implementing a consensus *object* from any finite list of base *objects*, one of which may crash. We do so by a reduction from the consensus *problem* among a finite number of *processes*, one of which may crash. The latter problem is known to be unsolvable [FLP85, LAA87].

Theorem 7.1 *There is no 1-tolerant implementation of* `2-consensus` *for crash failures.*

[9] Therefore `N-consensus with reset` cannot be defined modularly through sequential specification and linearizability.

[10] An object type T is finite if $A(T)$, the automaton giving the sequential specification of T, has only a finite number of states.

Proof Suppose the theorem is false and there is a finite list $\mathcal{L} = \{T_1, T_2, \ldots, T_l\}$ of object types such that there is a 1-tolerant implementation $\mathcal{I}$ of 2-**consensus** from $\mathcal{L}$ for crash failures.

Now consider the following concurrent system S in which *all* objects are registers. Processes in S are $\{p_1, p_2\} \cup \{q_j | 1 \leq j \leq l\}$, and the registers are $\{decision\} \cup \{invocation(i,j), response(j,i) \mid 1 \leq i \leq 2, 1 \leq j \leq l\}$. We claim that the consensus problem is solvable in S even if at most one process in S may crash. The following is the protocol. Let $v_i \in \{0,1\}$ be the input of p_i. The idea is that process q_j $(1 \leq j \leq l)$ simulates an object o_j of type T_j, and process p_i $(i = 1,2)$ simulates the execution of **propose**(v_i) on the derived object $\mathcal{I}(o_1, \ldots, o_l)$. The details are as follows.

Initialize all registers to $\perp$. The process p_i simulates the execution of the procedure **propose**(v_i) of the implementation $\mathcal{I}$ as explained below. If **propose**(v_i) requires p_i to invoke some operation op on o_j, p_i appends op to the contents of $invocation(i,j)$. If **propose**(v_i) 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) to $response(j,i)$. If **propose**(v_i) requires p_i to decide some value v, p_i first writes v in *decision* register, then decides it, and halts execution. Also p_i periodically checks if the register *decision* contains a $v \in \{0,1\}$. If so, it decides v and halts execution.

Process q_j simulates the base object o_j as follows. q_j checks the registers $invocation(1,j)$ and $invocation(2,j)$ in a round-robin fashion. When it notices that some operation op has been appended to $invocation(i,j)$, it applies op to the local copy of o_j that it maintains and appends the corresponding response to $response(j,i)$. Also q_j periodically checks if the register *decision* contains a $v \in \{0,1\}$. If so, it decides v and halts execution.

It is easy to verify that the above protocol solves the consensus problem among the $l+2$ processes in S even if at most one of them crashes. To see this, consider the following cases:

1. No process crashes: Since every q_j, the process simulating object o_j, is correct and **propose**(v_i) executed by p_i $(i = 1,2)$ is a wait-free procedure, it follows that one of p_1 and p_2 or both eventually write a value $v \in \{0,1\}$ into *decision*. Thus every correct process eventually decides v.

2. p_1 crashes: By our assumption that at most one process crashes, process p_2 and q_j $(1 \leq j \leq l)$, the process simulating object o_j, are all correct. Together with the fact that **propose**(v_2) is a wait-free procedure, this implies that p_2 eventually writes a decision value v into *decision* and decides v. Every other correct process eventually observes v in *decision* and decides v.

3. p_2 crashes: By a symmetric argument.

4. q_k crashes (for some $1 \leq k \leq l$): This corresponds to the crash of the simulated base object o_k. Since $\mathcal{I}$ is 1-tolerant, the execution of **propose**(v_i) by process p_i $(i = 1,2)$ eventually terminates. Thus one of p_1 and p_2 or both write a value v into *decision*. Thus every correct process eventually decides v.

In all the above cases, since $\mathcal{I}$ is an implementation of 2-consensus the following holds: if both p_1 and p_2 write into the *decision* register, then they both write the same value, and this value is either v_1 or v_2.

We showed that we can use $\mathcal{I}$ to solve the consensus problem in system S, and this contradicts the impossibility result of Louis and Abu-Amara [LAA87]. □

We can strengthen the above result as follows. Suppose that *at most one* base object may fail, and it can only do so by being "unfair" (i.e., by not responding) to *at most one* process. Furthermore, suppose, the identity of this process is a priori "common knowledge" among all the processes. Even with this extremely weak model of object failure, called *1-unfairness to a known process*, we can prove the following:

Theorem 7.2 *There is no 1-tolerant implementation of* 2-consensus *for 1-unfairness to a known process.*

Proof (Sketch) Assume the theorem is false, namely, there is a 1-tolerant implementation of 2-consensus for 1-unfairness to process p_1. Now proceed as in the proof of Theorem 7.1. Cases 1, 2, and 3 still hold. Consider Case 4, where q_k crashes (for some $1 \leq k \leq l$). This corresponds to the 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. We circumvent this difficulty by modifying p_2's protocol as follows. If propose(v_2) requires p_2 to invoke some operation *op* on some o_j, p_2 appends *op* to the contents of *invocation*$(2, j)$, as before, but now it also waits until a corresponding response is appended to *response*$(j, 2)$ (by process q_j).[11] Thus, if p_2 attempts to access o_k after the crash of q_k, it will simply wait for the response forever. Therefore, at worst, the crash of q_k looks like o_k is unfair to p_1, and p_2 is extremely slow. Since $\mathcal{I}$ tolerates the unfairness of one base object to p_1, $\mathcal{I}(o_1, \ldots, o_l)$ continues to behave as a wait-free consensus object. Hence the procedure propose(v_1) executed by p_1 eventually terminates returning the decision value. As before, this value is written into *decision*, and eventually every correct process decides. Again, we have a contradiction to the impossibility result in [LAA87]. □

Let $\mathcal{C}$ be the class of all object types that can implement 2-consensus. From the above two theorems we have

Corollary 7.1 *For all $T \in \mathcal{C}$, there is no 1-tolerant implementation of T for crash or 1-unfairness to a known process.*

From [Her91] and this corollary, we conclude that Queue, Stack, Test&Set, Fetch&Add, Compare&Swap, and several other common types do not have a 1-tolerant implementation for crash or 1-unfairness to a known process. In contrast to the above impossibility results we show

Theorem 7.3 register *has a t-tolerant self-implementation for arbitrary failures.*

[11] It is easy to see that with this modification Cases 1, 2, and 3 still hold.

This follows from

Lemma 7.1 *Using $5t + 1$ 1-reader, 1-writer safe registers, at most t of which may suffer arbitrary failures, we can implement a failure-free 1-reader, 1-writer, safe register.*

Proof (Sketch) Informally, the reader invokes 'read' on all registers (on which it has no pending invocation) and waits until $4t + 1$ respond. It then returns the majority value. If there is no majority, it returns an arbitrary value. The writer writes to all registers (on which it has no pending write). It waits until $4t + 1$ of them return a "operation completed" response. It is easy to verify that the above strategy implements a safe register that works correctly even if a maximum of t base registers suffer arbitrary failures. □

8 Other basic results

Consider a system that supports a given set H of primitive hardware objects. Assume that these objects may fail, but if they do, they are guaranteed to only fail by R-crash. Suppose we wish to build an object $\mathcal{O}$ using only objects in H, and $\mathcal{O}$ is only required to function correctly in the absence of failures. However, when objects in H fail by R-crash, we would like $\mathcal{O}$ to fail only by R-crash. This last requirement is desirable for two reasons:

- The simple "once $\perp$, everafter $\perp$" property of R-crash is the most benign type of failure.
- Such an object $\mathcal{O}$ appears like any other primitive hardware object of the system: With $\mathcal{O}$, the system would be no different, in functionality *and* failure semantics, from one that supports $H \cup \{\mathcal{O}\}$ as its primitive hardware objects.

In our terminology, a (0-tolerant) gracefully degrading implementation is exactly what we are looking for. The existence of such an implementation depends on the type of $\mathcal{O}$ and the types of the objects in H. Unfortunately, as we show below, most objects do not have such implementations even when H includes very powerful objects.

An object type T is *order-sensitive* if it is a deterministic N-process type ($N \geq 2$) and the following holds: There exist state S in $A(T)$, operations op, op' (not necessarily distinct) in $OP(T)$, and values u, v, u', v' such that each of (op, u),(op', u') and (op', v'),(op, v) is a sequential execution from state S consistent with T, and $u \neq v$ and $u' \neq v'$. `Queue` is an example of an order-sensitive object type. To see this, instantiate S to the state in which there are two elements 5 and 10 in the queue (5 in the front), and both op and op' to *deq*. Now we have $u = 5$, $u' = 10$, $v' = 5$, and $v = 10$. Thus $u \neq v$ and $u' \neq v'$, as required. `Stack`, `Test&Set`, `Compare&Swap` are some other examples of order-sensitive object types. An object type is *non order-sensitive* if it is deterministic and not order-sensitive. Examples of non order-sensitive types include `register`, `sticky bit`, `move`, and `swap`.

Theorem 8.1 *There is no (0-tolerant) gracefully degrading implementation of any order-sensitive object type for R-crash from any list of non order-sensitive object types.*

Proof Omitted. □

Preserving the failures semantics of the underlying system is a highly desirable property of an implementation. For R-crash, the above theorem shows that this property is not achievable in many cases: implementations necessarily amplify the severity of the R-crash failures of the underlying system. For example, consider a system that supports registers and sticky bits in "hardware". In such a system *any* object can be implemented [Plo89], including (for example) queues. Assume the given registers and sticky bits only fail by R-crash. Can we implement a queue that also fails by R-crash? The above theorem shows that this cannot be done!

Requiring a derived object to inherit the R-crash semantics of its base objects is even more difficult if 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 most objects of interest! This is shown by the theorem below.

Theorem 8.2 *There is no 1-tolerant gracefully degrading implementation of any order-sensitive object type for R-crash.*

Proof For a contradiction, assume $\mathcal{L} = \{T_1, T_2, \ldots, T_n\}$ is a list of types such that there is a 1-tolerant gracefully degrading implementation $\mathcal{I}$ of T from $\mathcal{L}$ for R-crash. We prove the theorem through a series of claims, involving "indistinguishable" scenarios. Let $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots O_n)$, and op, op', S, u, v, u', v' be as given in the definition of order-sensitive types.

Claim 8.1 *Suppose $\mathcal{O}$ is in state S, and processes p and q execute* `Apply`$(p, op, \mathcal{O})$ *and* `Apply`$(q, op', \mathcal{O})$ *respectively. For any interleaving of* `Apply`$(p, op, \mathcal{O})$ *and* `Apply`$(q, op', \mathcal{O})$, *either* `Apply`$(p, op, \mathcal{O})$ *returns u and* `Apply`$(q, op', \mathcal{O})$ *returns u' or* `Apply`$(p, op, \mathcal{O})$ *returns v and* `Apply`$(q, op', \mathcal{O})$ *returns v'.*

Proof In the linearization of the execution history, either `Apply`$(p, op, \mathcal{O})$ precedes `Apply`$(q, op', \mathcal{O})$ or `Apply`$(q, op', \mathcal{O})$ precedes `Apply`$(p, op, \mathcal{O})$. This, together with the definitions of u, u', v, v', and the fact that T is a deterministic type, trivially imply the claim. □

Claim 8.2 *There exists a sequence α of steps (of p) and a step s (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* `Apply`$(p, op, \mathcal{O})$ *by completing the steps in α.*
2. *Process q initiates and completes (all the steps of)* `Apply`$(q, op', \mathcal{O})$, *returning v'.*
3. *p completes the remaining steps of* `Apply`$(p, op, \mathcal{O})$, *returning v.*

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

1. *p initiates and (partially) executes* $\texttt{Apply}(p, op, \mathcal{O})$ *by completing the steps in* $\alpha.s$.

2. *q initiates and completes (all the steps of)* $\texttt{Apply}(q, op', \mathcal{O})$, *returning* u'.

3. *p completes the remaining steps of* $\texttt{Apply}(p, op, \mathcal{O})$, *returning* u.

Proof Clearly if process p executes no steps of $\texttt{Apply}(p, op, \mathcal{O})$ before process q initiates and completes $\texttt{Apply}(q, op', \mathcal{O})$, then $\texttt{Apply}(q, op', \mathcal{O})$ must return v'. Further if p initiates and completes all the steps of $\texttt{Apply}(p, op, \mathcal{O})$ (let β be this sequence of steps) before q initiates and completes $\texttt{Apply}(q, op', \mathcal{O})$, then $\texttt{Apply}(q, op', \mathcal{O})$ must return u'. Together with Claim 8.1 by which $\texttt{Apply}(q, op', \mathcal{O})$ must return either u' or v', the above implies that there exists a sequence α of steps and a step s such that $\alpha.s$ is a prefix of β for which the claim holds. □

Hereafter we will assume O_k is the base object accessed by p in step s.

Claim 8.3 *Consider*

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

1. *p initiates and (partially) executes* $\texttt{Apply}(p, op, \mathcal{O})$ *by completing the steps in* $\alpha.s$.

2. *q initiates and completes (all the steps of)* $\texttt{Apply}(q, op', \mathcal{O})$, *returning* u' *(as in* S2*)*.

3. $O_1, O_2, \ldots, O_n$ *fail by R-crash.*

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

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

Proof Suppose $\texttt{Apply}(p, op, \mathcal{O})$ returns $\perp$. Since $\mathcal{I}$ is gracefully degrading, the failure of $\mathcal{O}$ must appear like R-crash. This requires, given that $\texttt{Apply}(q, op', \mathcal{O})$ returns a non-$\perp$ response, that $\texttt{Apply}(q, op', \mathcal{O})$ precede $\texttt{Apply}(p, op, \mathcal{O})$ in the linearization order. Doing so, however, implies that (op', u') is a sequential execution from S consistent with T. This cannot be true since $u' \neq v'$, T is deterministic, and (op', v') is a sequential execution from S consistent with T. Thus $\texttt{Apply}(p, op, \mathcal{O})$ cannot return $\perp$.

Suppose $\texttt{Apply}(p, op, \mathcal{O})$ returns w where $\perp \neq w \neq u$. Since in the linearization, either $\texttt{Apply}(p, op, \mathcal{O})$ precedes $\texttt{Apply}(q, op', \mathcal{O})$ or $\texttt{Apply}(q, op', \mathcal{O})$ precedes $\texttt{Apply}(p, op, \mathcal{O})$, it follows that either (op, w),(op', u') or (op', u'),(op, w) is a sequential execution from S consistent with T. This cannot be true since T is deterministic and (op, u),(op', u') and (op', v'),(op, v) are sequential executions from S consistent with T and $w \neq u$, $u' \neq v'$.

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

Claim 8.4 *Consider*

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

1. *p initiates and (partially) executes* $\texttt{Apply}(p, op, \mathcal{O})$ *by completing the steps in* $\alpha.s$.

2. O_k *fails by R-crash.*

3. *q initiates and completes (all the steps of)* $\texttt{Apply}(q, op', \mathcal{O})$.

4. $O_1, \ldots, O_{k-1}$ *and* $O_{k+1}, \ldots, O_n$ *also fail by R-crash.*

5. *p completes the remaining steps of* $\texttt{Apply}(p, op, \mathcal{O})$.

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

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

Claim 8.5 *Consider*

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

1. *p initiates and partially executes* $\texttt{Apply}(p, op, \mathcal{O})$ *by completing the steps in* α.

2. O_k *fails by R-crash.*

3. *q initiates and completes (all the steps of)* $\texttt{Apply}(q, op', \mathcal{O})$.

4. $O_1, \ldots, O_{k-1}$ *and* $O_{k+1}, \ldots, O_n$ *also fail by R-crash.*

5. *p completes the remaining steps of* $\texttt{Apply}(p, op, \mathcal{O})$.

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

Proof Clearly $\texttt{S5} \approx_q \texttt{S4}$. Therefore $\texttt{Apply}(q, op', \mathcal{O})$ returns u' as in S4. By similar arguments as in Claim 8.3, it can be shown that $\texttt{Apply}(p, op, \mathcal{O})$ returns u. □

Claim 8.6 *Consider*

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

1. *p initiates and partially executes* $\texttt{Apply}(p, op, \mathcal{O})$ *by completing the steps in* α.

2. *q initiates and completes (all the steps of)* $\texttt{Apply}(q, op', \mathcal{O})$.

3. *All base objects* $O_1, O_2, \ldots, O_n$ *fail by R-crash.*

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

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

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

Neither (op, u),(op', v') nor (op', v'),(op, u) is a sequential execution from S consistent with T. Hence the execution in Claim 8.6 is not linearizable. Thus the failure of $\mathcal{O}$ in S6 is more severe than R-crash. We conclude that $\mathcal{I}$ is not a gracefully degrading implementation for R-crash, a contradiction which concludes the proof of Theorem 8.2. □

The above discussion raises some questions on the "practicality" of the R-crash model: Even if "hardware" objects fail by R-crash, "software" objects don't. The R-omission model defined in this paper does not have this serious limitation. In fact, for any $t \geq 0$ *every* object type has a t-tolerant *gracefully degrading* implementation from (**universal type, register**) for R-omission. In other words, implementations preserving the R-omission semantics of the underlying system always exist. This is a formal justification for adopting the R-omission model of failure.

References

[Blo87] Bard Bloom. Constructing two writer atomic registers. In *The 6th Annual Symposium on Principles of Distributed Compu ting*, pages 249–259, 1987.

[BP87] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *The 6th Annual Symposium on Principles of Distributed Compu ting*, 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.

[CW90] Soma Chaudhuri and Jennifer Welch. Bounds on the costs of register implementations. Technical report, University of North Carolina at Chapel Hill, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, 1990.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[Her91] 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 ato mic variable from regular variables. Technical Report Technical Report No: 9108, Memorial University of Newfoundland, Department of Computer Science, Memorial University of Newfou ndland, 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.

[LAA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[Lam86] Leslie Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[NT90] Gil Neiger and Sam 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 *The 6th Annual Symposium on Principles of Distributed Compu ting*, pages 232–248, 1987.

[PB87] G. Peterson and J. Burns. Concurrent reading while writing ii: the multi-writer case. In *The 28th Annual Symposium on Foundations of Computer Scienc e*, 1987.

[Pet83] Gary L. Peterson. Concurrent reading while writing. *ACM TOPLAS*, 5(1):56–65, 1983.

[Plo89] Serge Plotkin. Sticky bits and universality of consensus. In *The 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.

[SAG87] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register, revisited. In *The 6th Annual Symposium on Principles of Distributed Compu ting*, 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 Scienc e, 1988.

[VA86] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *The 27th Annual Symposium on Foundations of Computer Scienc e*, 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.