Fault-Tolerant Wait-Free Shared Objects**

Prasad Jayanti
Tushar Deepak Chandra*
Sam Toueg

TR 92-1298
(Revision of TR 92-1281, April 1992)
August 1992

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

**A preliminary version of this will appear in the proceedings of the 33rd Annual Symposium on Foundations of Computer Science, October 1992.
**Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG 2-593 and grants form the IBM Endicott Programming Laboratory.
*Also supported by an IBM graduate fellowship.

Fault-tolerant Wait-free Shared Objects*†

Prasad Jayanti   Tushar Deepak Chandra‡   Sam Toueg

{prasad, chandra, sam}@cs.cornell.edu
Department of Computer Science
Cornell University
Ithaca, New York 14853

August 21, 1992

Abstract

A concurrent system consists of *processes* and *shared objects*. Previous research focused on the problem of tolerating process failures. We study the complementary problem of tolerating object failures.

We divide object failures into two broad classes: *responsive* and *non-responsive*. With responsive failures, a faulty object responds to every invocation, but responses may be incorrect. With non-responsive failures, a faulty object may also "hang" without responding. For each class, we consider *crash*, *omission*, and *arbitrary* types of failures.

For each type of failure, we are seeking a universal implementation for *fault-tolerant* wait-free shared objects. We present (deterministic) implementations for all types of responsive failures, including arbitrary failures. In contrast, we show that even the most benign type of non-responsive failures requires the use of randomization.

Of special interest is the problem of implementing fault-tolerant objects using only objects of the same type. We present such fault-tolerant *self*-implementations for many common object types.

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 models, we show whether this property can be achieved, and, if so, how.

In addition to the above possibility/impossibility results, we also consider the resource complexity of fault-tolerant implementations. In many cases, we present lower bounds and give matching algorithms.

*A preliminary version of this will appear in the proceedings of the 33rd Annual Symposium on Foundations of Computer Science, October, 1992.

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

‡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`, and `set`, 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 by every process is guaranteed a response in finite time irrespective of the speed of the other processes, 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 the failures of the *shared objects* themselves.[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 simplify notation, hereafter "object" denotes a "shared object".)

The problem addressed in this paper is novel. A preliminary version appeared in [JCT92a], and a summary of the results in [JCT92b]. An independent work by Afek, Greenberg, Merritt, and Taubenfeld [AGMT92] has the same general goal, but differs in many respects. We present a brief comparison of the two works in Section 8.

1.2 Object failures

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

We divide responsive failures into three models: *R-crash*, *R-omission*, and *R-arbitrary.* An object that fails by R-crash behaves correctly until it fails, and once it fails, it returns a distinguished response $\perp$ to every operation. As with R-crash, an object that fails by R-omission may return a correct response or a $\perp$. However, even if it responds $\perp$ to a process p, a subsequent operation 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

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

operation by p "ran into" a failed component of $\mathcal{O}$ (and returned $\perp$), while the later one by q only encountered correct components of $\mathcal{O}$ (and returned a correct response). Finally, objects experiencing R-arbitrary failures may "lie", *i.e.*, return arbitrary responses.

Similarly, we divide non-responsive failures into *crash*, *omission*, and *arbitrary*. An object that fails by crash behaves correctly until it fails, and once it fails, it stops responding. An object that fails by omission 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 that experiences an arbitrary failure is completely unrestricted: it may not respond, and even if it does, the response may be arbitrary.

1.3 Fault-tolerant objects

Let T be an object type and $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ be a list of object types (T_i's are not necessarily distinct). A *wait-free implementation* of T from $\mathcal{L}$ is a function $\mathcal{I}$ such that given any distinct objects $O_1, O_2, \ldots, O_n$ of type $T_1, T_2, \ldots, T_n$, respectively, $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$ is an object of type T that behaves correctly if all O_i's behave correctly. Roughly speaking, an object behaves correctly if it is wait-free and its behavior is consistent with its type. We say $\mathcal{O}$ is a *derived object* of the implementation $\mathcal{I}$, and $O_1, O_2, \ldots, O_n$ are the *base objects* of $\mathcal{O}$. The *resource complexity* of $\mathcal{I}$ is n, the number of base objects required by $\mathcal{I}$ to implement a derived object. Such a wait-free implementation $\mathcal{I}$ is *t-tolerant for failure model $\mathcal{M}$* if $\mathcal{O}$ behaves correctly even if at most t base objects of $\mathcal{O}$ fail by $\mathcal{M}$. In this Introduction, we write "implementation" as a shorthand for "wait-free 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 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 5.3, we show that 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, at most t of which may experience R-arbitrary failures, one 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 (for which a sequential implementation exists) using only consensus and register objects. Thus, if `consensus` and `register` have t-tolerant implementations, then every object type has a t-tolerant implementation.

[2] A consensus object supports two operations *propose 0* and *propose 1*, and has the following sequential specification: If the first operation on the object is *propose v* ($v \in \{0, 1\}$), then every operation is returned the response v.

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

What about tolerating non-responsive failures? We first 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 **fetch&add**, **queue**, **stack**, **test&set**, **compare&swap**, **move**, **sticky-bit**, **swap**, has no 1-tolerant implementation for crash failures. In contrast, we show that **register** has a t-tolerant *self*-implementation even for arbitrary failures. Since randomized implementations of **consensus** from **register** are well known (for example, see [Asp90]), the above result implies that every object type has a *randomized* t-tolerant implementation from **register** even for arbitrary failures. In addition to these universality and impossibility results, this paper contains the following results.

Consider a t-tolerant implementation for failure model $\mathcal{M}$. By definition, a derived object of this implementation is guaranteed to behave correctly even if up to t base objects fail by $\mathcal{M}$. But what happens if more than t base objects fail? In general, the derived object may experience a more severe failure than $\mathcal{M}$. 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 model $\mathcal{M}$ if it has the following property: if base objects only fail by $\mathcal{M}$, then derived objects also fail by $\mathcal{M}$.

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 gracefully degrading self-implementation of T for $\mathcal{M}$. Thus, graceful degradation provides a method for automatically increasing the fault-tolerance of an implementation.

Requiring graceful degradation may increase the cost of an implementation. For instance, consider t-tolerant implementations of **consensus** for R-omission failures. We present 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 graceful degradation for R-omission failures requires at least $2t + 1$ base

[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 [FLP85, LAA87].

objects (this lower bound holds for every deterministic non-trivial type).

In some cases, graceful degradation cannot be even achieved. In particular, we show 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. In other words, with R-crash, implementations necessarily amplify failures. In contrast, we prove the following strong possibility result for R-omission: Every object type has a t-tolerant gracefully degrading implementation from `consensus` and `register` for R-omission.

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 may experience R-arbitrary failures, we can implement a consensus object that can only fail by R-omission. We prove 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.

The paper is organized as follows. We give an informal system model and define several types of object failures in Sections 2 and 3. We define the concepts of t-tolerant wait-free implementation and graceful degradation in Section 4. We provide a formal presentation of the material of Sections 2, 3, and 4 in Appendices A, B, and C, respectively. In Section 5, we show how to implement objects that tolerate responsive failures. We present t-tolerant implementations of `consensus` in Section 5.1, of `register` in Section 5.2, and of arbitrary types in Section 5.3. The results on the cost of graceful degradation, and on the translation between failure models are also presented in Section 5.1. In Section 6, we study the feasibility of fault-tolerant implementations for non-responsive object failures. We first prove that many common object types including `consensus` have no 1-tolerant implementations for crash. In contrast, we show that `register` has a t-tolerant self-implementation even for arbitrary failures. We finally show that *every* object type has a t-tolerant *randomized* implementation from `register` even for arbitrary failures. In Section 7, we study graceful degradation for the R-crash and R-omission failure models. We present impossibility results for R-crash and a universality result for R-omission. In Section 8, we present a brief comparison with the results in [AGMT92]. In Appendix D, we define the object types that appear in this paper.

2 Informal model

A *concurrent system* consists of processes and shared objects. Associated with each object is a *type*. The type characterizes the expected behavior of the object. More precisely, an *object type* T is a tuple (N, OP, RES, G), where N is an integer greater than one, OP and

RES are sets of operations and responses respectively, and G is a directed finite or infinite graph in which each edge has a label of the form (op, res) where $op \in OP$ and $res \in RES$. Intuitively, if $\mathcal{O}$ is an object of type T, then $\mathcal{O}$ supports the operations in OP and may be shared by N processes (we say T is an *N-process type*). G specifies the expected behavior of $\mathcal{O}$ in the absence of concurrent operations on $\mathcal{O}$.

The vertices of G are the *states of* T. One state of T is the *initial* state. A state s of T is *reachable* if there is a path in G from the initial state to s. We assume that every state of T is reachable. A sequence $S = (op_1, res_1), (op_2, res_2), \ldots, (op_l, res_l)$ is *consistent from a state s of T* if there is a path labeled S in G from the state s. S is *consistent with respect to T* if it is consistent from the initial state of T. T is *deterministic* if for every state s of T and every operation $op \in OP$, there is at most one edge from s labeled (op, res). T is *non-deterministic* otherwise. T is *finite* if G is finite; T is *infinite* otherwise.

An object $\mathcal{O}$ of type T supports the set of procedures $\texttt{Apply}(P, op, \mathcal{O})$, for each process P and operation op in $OP(T)$. A process P *invokes* operation op on object $\mathcal{O}$ by calling $\texttt{Apply}(P, op, \mathcal{O})$, and *executes* the operation by executing this procedure. The operation *completes* when the procedure terminates. The *response* for an operation is the value returned by the procedure.

The sequential specification of an object $\mathcal{O}$, given by its type, is not sufficient to predict $\mathcal{O}$'s behavior in the presence of concurrent operations. To characterize such behavior, we use the concept of *linearizability* [HW90, Lam86]. Roughly speaking, linearizability requires every operation execution to appear to take effect instantaneously at some point in time between its invocation and response. We make it more precise below.

An *execution* of a concurrent system is an interleaving of the steps of the processes and the invocations and responses of the objects. Consider an execution E of a concurrent system consisting of an object $\mathcal{O}$ that is shared by processes $P_1, P_2, \ldots, P_N$. The *history* $\mathcal{H}$ of $\mathcal{O}$ in E is a set defined as follows: $(P_i, op, v, t_s, t_e) \in \mathcal{H}$ iff in execution E, process P_i invokes op at time t_s, and this operation completes at time t_e returning the response v. Further, $(P_i, op, *, t_s, \infty) \in \mathcal{H}$ iff process P_i invokes op at time t_s, and this operation does not complete. A history is *complete* if it has no incomplete operations. Given two operations (P_i, op, v, t_s, t_e) and $(P_j, op', v', t'_s, t'_e)$ in a history, we say (P_i, op, v, t_s, t_e) *precedes* $(P_j, op', v', t'_s, t'_e)$ if $t_e < t'_s$. A *complete history $\mathcal{H}$ is linearizable with respect to a type T* if there is a sequencing $\mathcal{S}$ of the tuples (operations) in $\mathcal{H}$ such that $\mathcal{S}$ respects the 'precedes' relation, and is consistent with respect to T. A *history $\mathcal{H}$ is linearizable with respect to a type T* if a linearizable complete history $\mathcal{H}'$ can be obtained from $\mathcal{H}$ as follows: each incomplete operation $(P_i, op, *, t_s, \infty)$ in $\mathcal{H}$ is either removed or replaced by a complete operation (P_i, op, v, t_s, t_e), for some response v and time t_e. This definition captures the notion that some incomplete operations in $\mathcal{H}$ had a "visible" effect, while the others did not.

Processes are *asynchronous*: *i.e.*, there are no bounds on the relative speeds of the processes. Furthermore, a process may *crash*: *i.e.*, a process may stop at an arbitrary point in an execution and never take any steps thereafter. The concept of wait-freedom was introduced to cope with such processes (for example, see [Her91]). An *object $\mathcal{O}$ is wait-free* *in an execution* E if either (i) E is finite, or (ii) every operation on $\mathcal{O}$ invoked by a process that does not crash in E gets a response from $\mathcal{O}$.

An object $\mathcal{O}$ is *correct in execution* E iff (i) $\mathcal{O}$ is wait-free in E, and (ii) the history of $\mathcal{O}$ in E is linearizable with respect to the type of $\mathcal{O}$. We say that $\mathcal{O}$ *fails in* E iff $\mathcal{O}$ is not correct in E. Even a faulty object may satisfy certain properties which depend on the type of failure it suffered. We postpone the definition of the failure models to next section.

Let T be an object type and $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ be a list of object types (T_i's are not necessarily distinct). A *wait-free implementation of* T *from* $\mathcal{L}$ is a function $\mathcal{I}$ such that given any distinct objects $O_1, O_2, \ldots, O_n$ of type $T_1, T_2, \ldots, T_n$, respectively, $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$ is an object of type T with the following property: In every execution, if $O_1, O_2, \ldots, O_n$ are correct, then $\mathcal{O}$ is correct. We say $\mathcal{O}$ is a *derived object* of the implementation $\mathcal{I}$, and $O_1, O_2, \ldots, O_n$ are the *base objects* of $\mathcal{O}$. All implementations studied in this paper are wait-free. Hereafter we write "implementation" as shorthand for "wait-free implementation".

We define the terms *self-implementation* of T and *resource complexity* as in Section 1.3. Our interest lies not just in implementations, but in implementations that tolerate the failures of base objects. Thus, we also need to define a *fault-tolerant* implementation. We present such a definition in Section 4, after defining failure models in Section 3.

3 Failure models

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 implementation 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 model*, *i.e.*, the manner in which a failed base object departs from correct behavior. In this paper, we define a spectrum of failure models that fall into two broad classes: *responsive* and *non-responsive.*

As we will see, in most models of failure, an object $\mathcal{O}$ of type T may fail by returning a response that is not allowed by its type; that is, a response not in *RES*(T). When a process P gets such a response from $\mathcal{O}$, it knows 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 models of failure

An object experiencing a responsive failure responds to every invocation, even though the response may be incorrect. In other words, the object remains wait-free even after if it fails. 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. Informally, an object that fails by R-crash behaves correctly until it fails, and once it fails, it returns a distinguished response $\perp$ to every invocation. This model is based on the premise that an object detects when it becomes faulty.

More precisely, an object $\mathcal{O}$ *fails in execution* E *by R-crash* iff it fails in E, and satisfies the following properties:

1. $\mathcal{O}$ is wait-free in E.
2. Every response from $\mathcal{O}$ in E is either $\perp$ or one of the responses allowed by the type of $\mathcal{O}$. An operation that returns $\perp$ is an *aborted* operation.
3. Let $\mathcal{H}$ be the history of $\mathcal{O}$ in E. Every operation in $\mathcal{H}$ that is preceded by an aborted operation is itself an aborted operation.
4. Removing the aborted operations from $\mathcal{H}$ results in a linearizable history with respect to the type of $\mathcal{O}$.

Property 3 is the "once $\perp$, everafter $\perp$" property of R-crash. Property 4 models the requirement that $\mathcal{O}$ should behave correctly until it fails.

3.1.2 R-omission

Consider an implementation $\mathcal{I}$, and a derived object $\mathcal{O}$ of $\mathcal{I}$. Even if the base objects of $\mathcal{O}$ can only fail by R-crash, $\mathcal{O}$ itself may experience a more severe failure than R-crash. To see this, suppose a base object b of $\mathcal{O}$ fails by R-crash. Consider a process P that invokes an operation op on $\mathcal{O}$ and executes $\texttt{Apply}(P, op, \mathcal{O})$. If $\texttt{Apply}(P, op, \mathcal{O})$ accesses b, b returns $\perp$ to P. This may cause P's invocation of op on $\mathcal{O}$ to terminate and return $\perp$. Now suppose that another process Q later invokes some operation op' on $\mathcal{O}$, and that $\texttt{Apply}(Q, op', \mathcal{O})$ is *not* required to access b. Then, process Q cannot notice the failure of b. So Q's invocation of op on $\mathcal{O}$ terminates "normally" and returns a non-$\perp$ response. Thus, $\mathcal{O}$'s behavior violates the "once $\perp$, everafter $\perp$" property of R-crash. Does this mean that $\mathcal{O}$'s failure is arbitrary? We now argue that this is not the case.

Recall that after P gets $\perp$, P refrains from accessing $\mathcal{O}$ again. To Q, this scenario is indistinguishable from one in which P had crashed in the middle of the procedure $\texttt{Apply}(P, op, \mathcal{O})$, while accessing b. Since the implementation $\mathcal{I}$ (from which $\mathcal{O}$ is derived) is wait-free, $\mathcal{O}$ tolerates the apparent crash of P. Thus, $\mathcal{O}$'s response to Q must be correct. So, the failure of $\mathcal{O}$ is more severe than R-crash, but is not completely arbitrary. The R-omission model captures such a failure[4].

More precisely, an object $\mathcal{O}$ *fails in execution E by R-omission* iff it fails in E, and satisfies the following properties:

1. $\mathcal{O}$ is wait-free in E.

2. Every response from $\mathcal{O}$ in E is either $\perp$ or one of the responses allowed by the type of $\mathcal{O}$.

3. Let $\mathcal{H}$ be the history of $\mathcal{O}$ in E. Replacing every aborted operation $(P, op, \perp, t_s, t_e)$ in $\mathcal{H}$ by an incomplete operation $(P, op, *, t_s, \infty)$ results in a linearizable history with respect to the type of $\mathcal{O}$.

3.1.3 R-arbitrary

An object $\mathcal{O}$ *fails in execution E by R-arbitrary*[5] iff it fails in E and is wait-free in E. In other words, $\mathcal{O}$ responds to every invocation in E, but the history of $\mathcal{O}$ is not linearizable with respect to the type of $\mathcal{O}$.

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 experiencing a non-responsive failure may also fail to respond to 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 (Property 1, below), and once it fails, it never responds to any invocations (Property 2, below). More precisely, an object $\mathcal{O}$ *fails in execution E by crash* iff it fails in E, and satisfies the following properties:

1. The history of $\mathcal{O}$ in E is linearizable with respect to the type of $\mathcal{O}$.

2. The total number of responses from $\mathcal{O}$ in E is finite.

[4] Formal justification for the R-omission model will be apparent in Section 7.

[5] For readability, we sometimes prefer writing "$\mathcal{O}$ experiences an R-arbitrary failure in E".

3.2.2 Omission

Omission failures are more severe than crash. An object $\mathcal{O}$ *fails in execution* E *by omission* iff it fails in E, and the history of $\mathcal{O}$ in E is linearizable with respect to the type of $\mathcal{O}$. In particular, an object that fails by omission does not necessarily satisfy Property 2 of crash model. Thus, an object that fails by omission may not respond to invocations from some processes, but respond to invocations from others forever.

3.2.3 Arbitrary

The behavior of an object that experiences an arbitrary failure is completely unrestricted. In particular, such an object may not respond to an invocation; even if it does, the response may be arbitrary. More precisely, an object $\mathcal{O}$ *fails in execution* E *by arbitrary* iff it fails in E.

4 Definition of fault-tolerant implementations

An implementation $\mathcal{I}$ of type T is *t-tolerant for failure model* $\mathcal{M}$ if every derived object $\mathcal{O}$ of $\mathcal{I}$ has the following property: In every execution, if at most t base objects fail, and they fail by $\mathcal{M}$, then $\mathcal{O}$ is correct.

An implementation $\mathcal{I}$ is *gracefully degrading for failure model* $\mathcal{M}$ if every derived object $\mathcal{O}$ of $\mathcal{I}$ has the following property: In every execution, if all base objects that fail, fail by $\mathcal{M}$, then either $\mathcal{O}$ is correct or it fails by $\mathcal{M}$.

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

Gracefully degrading implementations can be easily composed as shown in the following lemma. Given a list L of integers and an integer n, let $MinSum(n, L)$ be the sum of the n smallest integers in L.

Lemma 4.1 *If a type T has a t-tolerant gracefully degrading implementation $\mathcal{I}$ from the list $T_1, T_2, \ldots, T_n$ of types for failure model $\mathcal{M}$, and each T_i $(1 \le i \le n)$ has a t_i-tolerant gracefully degrading implementation $\mathcal{I}_i$ from $T_{i1}, T_{i2}, \ldots, T_{ij_i}$ for $\mathcal{M}$, then T has a t'-tolerant gracefully degrading implementation $\mathcal{I}'$ from $T_{11}, T_{12}, \ldots, T_{1j_1}, T_{21}, \ldots, T_{2j_2}, \ldots, T_{n1}, \ldots, T_{nj_n}$ for $\mathcal{M}$. In the above, $t' = MinSum(t+1, \langle t_1+1, t_2+1, \ldots, t_n+1\rangle) - 1$.*

Proof (sketch) Define $\mathcal{I}'(o_{11}, \ldots, o_{1j_1}, \ldots, o_{n1}, \ldots, o_{nj_n}) = \mathcal{I}(O_1, \ldots, O_n)$ where $O_1 = \mathcal{I}_1(o_{11}, o_{12}, \ldots, o_{1j_1}), \ldots, O_n = \mathcal{I}_n(o_{n1}, o_{n2}, \ldots, o_{nj_n})$. Assume that each o_{kl}, if it fails, only fails by $\mathcal{M}$. Since $\mathcal{I}_i$ is t_i-tolerant, O_i fails only if at least $t_i + 1$ objects among $o_{i1}, \ldots, o_{1j_i}$

fail; furthermore, since $\mathcal{I}_i$ is gracefully degrading, O_i fails only by $\mathcal{M}$. Similarly, since $\mathcal{I}$ is t-tolerant, $\mathcal{I}(O_1, \ldots, O_n)$ fails only if at least $t+1$ objects among $O_1, \ldots, O_n$ fail. Thus, for $\mathcal{I}(O_1, \ldots, O_n)$ to fail, at least $MinSum(t+1, \langle t_1+1, t_2+1, \ldots, t_n+1 \rangle)$ objects among $o_{11}, \ldots, o_{1j_1}, \ldots, o_{n1}, \ldots, o_{nj_n}$ must fail. In other words, $\mathcal{I}'$ is a t'-tolerant implementation of T from $T_{11}, \ldots T_{nj_n}$. $\mathcal{I}'$ is gracefully degrading for $\mathcal{M}$ because $\mathcal{I}$ and each $\mathcal{I}_i$ $(1 \leq i \leq n)$ are gracefully degrading for $\mathcal{M}$. □

The above lemma can be used to enhance the fault-tolerance of a self-implementation. This is the substance of the next corollary, obtained by setting $T_i = T$, $t_i = t$, $j_i = n$, and $\mathcal{I}_i = \mathcal{I}$ in the lemma.

Corollary 4.1 *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 $\mathcal{I}'$ of resource complexity n^2 for $\mathcal{M}$.*

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

Corollary 4.2 (Booster Lemma) *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 $O(t^{\log_2 k})$ for $\mathcal{M}$.*

In Section 5.1.4, we illustrate how this corollary can be applied to construct a t-tolerant self-implementation of `consensus` for R-arbitrary failures.

5 Tolerating responsive failures

Herlihy [Her91] and Plotkin [Plo89] 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 object 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 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 thus 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 $O(t \log t)$.

5.1.1 The object type N-consensus

N-consensus is an N-process object 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. The following two propositions follow directly from definitions:

Proposition 5.1 *An N-consensus object $\mathcal{O}$ is correct in execution E if and only if it is wait-free and satisfies the following three properties in E:*

- Validity: *If $\mathcal{O}$ returns a response v, and $v \in \{0,1\}$, then there was a prior invocation of* propose v *on $\mathcal{O}$.*
- 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: *Every response of $\mathcal{O}$ is either 0 or 1.*

An N-consensus object $\mathcal{O}$ satisfies *weak integrity* in an execution in E iff every response of $\mathcal{O}$ in E is either 0, 1, or $\perp$.

Proposition 5.2 *Let $\mathcal{O}$ be an N-consensus object that fails in execution E. Object $\mathcal{O}$ fails by R-omission in E if and only if it is wait-free, and satisfies validity, agreement, and weak integrity in E.*

In describing our implementations, we write *loc* := Propose$(p, v, \mathcal{O})$[6] to denote that process p invokes *propose v* on $\mathcal{O}$ and stores the response in its local variable *loc*.

5.1.2 Tolerating R-crash and R-omission failures

We present a t-tolerant self-implementation of **N-consensus** for R-omission failures. The resource complexity is $t+1$, and is therefore optimal. Since R-omission failures are strictly more severe than R-crash, this self-implementation also works for R-crash. However, it is *not* gracefully degrading either for R-crash or for R-omission. In fact, we will see in Section 7 that **N-consensus** has no t-tolerant gracefully degrading implementation for R-crash. For R-omission, however, we present a t-tolerant gracefully degrading self-implementation of resource complexity $2t+1$. We also prove that $2t+1$ is a lower bound on the resource complexity. In fact, this lower bound applies to every "non-trivial" deterministic object type, not just to **N-consensus**; furthermore, it is not restricted to self-implementations.

Theorem 5.1 *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.*

[6] Throughout this paper, we write Propose (with upper case "P") if the operation is on a derived object, and propose (with lower case "p") if it is on a base object.

$O_1, O_2, \ldots, O_{t+1}$: **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

Proof Let $\mathcal{O}$ be a derived **N-consensus** object of the implementation, and $O_1, O_2, \ldots, O_{t+1}$ be its base objects. Consider an execution E in which at most t base objects fail by R-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 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 $\perp$.

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 $O_{k+1}, \ldots, O_{t+1}$ is either correct or fails by R-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 R-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. □

The above (self) implementation is *not* gracefully degrading. For instance, suppose that $v_p = 0$ and $v_q = 1$, and all 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 5.2, 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[7]. The gracefully degrading self-implementation that requires $2t+1$ base objects is given in Figure 2.

```
O_1, O_2, ..., 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
        estimate_p := v_p
        for k := 1 to 2t+1 do
            w := propose(p, estimate_p, O_k)
            V_p[k] := w
            if (w ≠ ⊥)∧(w ≠ estimate_p) then
                estimate_p := w
                V_p[1...(k-1)] := (⊥, ⊥, ..., ⊥)
        if V_p has more than t ⊥'s then
            return(⊥)
        else return(estimate_p)
    end
```

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

---

Claim 5.1 *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 2, $estimate_p \in \{0,1\}$, and $V_p[1..k]$ contains only $\perp$'s and $estimate_p$'s.*

Proof By an easy induction on k. □

Theorem 5.2 *Figure 2 gives a t-tolerant gracefully degrading self-implementation of* **N-consensus** *for R-omission.*

Proof Let $\mathcal{O}$ be a derived **N-consensus** object of the implementation, and $O_1, O_2, \ldots, O_{t+1}$ be its base objects. Consider an execution E in which all base objects that fail, fail by R-omission.

1. $\mathcal{O}$ is wait-free: Obvious since base objects that fail by R-omission remain wait-free.
2. $\mathcal{O}$ satisfies validity: An easy induction on k 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 $\perp$.

[7] As will be shown later in Theorem 7.2, there is *no* t-tolerant gracefully degrading implementation of **N-consensus** for R-crash.

3. $\mathcal{O}$ satisfies agreement: Suppose, for a contradiction, there exist two processes p and q such that $\mathtt{Propose}(p, v_p, \mathcal{O})$ returns 0 and $\mathtt{Propose}(q, v_q, \mathcal{O})$ returns 1. From Claim 5.1, 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 $\mathtt{Propose}(p, v_p, \mathcal{O})$ and V_q has at least $t+1$ 1's at the end of the execution of $\mathtt{Propose}(q, v_q, \mathcal{O})$. This is possible only if there is a k ($1 \leq k \leq 2t+1$) such that $\mathtt{propose}(p, \mathit{estimate}_p, O_k)$ returned 0 and $\mathtt{propose}(q, \mathit{estimate}_q, O_k)$ returned 1. Thus O_k does not satisfy agreement. By Proposition 5.2, the failure of O_k in E is not by R-omission, a contradiction.

4. $\mathcal{O}$ satisfies weak integrity: Obvious.

5. $\mathcal{O}$ satisfies integrity if at most t base objects fail: Let $O_{k_1}, O_{k_2}, \ldots, O_{k_l}$ ($k_1 < k_2 < \ldots < k_l$) be all the correct base objects. Since at most t fail, we have $l \geq t+1$. By Proposition 5.1, O_{k_1} satisfies integrity and agreement. Thus, there is a $v \in \{0, 1\}$ such that for all p, $\mathtt{propose}(p, \mathit{estimate}_p, O_{k_1})$ returns v. Thus, for all p, $\mathit{estimate}_p = v$ at the end of k_1 iterations of the for-loop in $\mathtt{Propose}(p, v_p, \mathcal{O})$. Using this and Proposition 5.2, it is easy to verify that at the end of the execution of $\mathtt{Propose}(p, v_p, \mathcal{O})$, $V_p[k_i] = v$ and $\mathit{estimate}_p = v$ for all p and for all $1 \leq i \leq l$. This implies, by lines 8, 9 of the algorithm, that $\mathtt{Propose}(p, v_p, \mathcal{O})$ 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 R-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 2 is a t-tolerant gracefully degrading self-implementation of N-consensus for R-omission. □

We now prove a general lower bound on the resource complexity of gracefully degrading implementations for R-omission. Informally, a *type T is trivial* if it admits the following implementation: there is a function f such that every $\mathtt{Apply}(P, op, O)$ blindly returns $f(op)$. More precisely, T is trivial if there is a function $f : OP(T) \rightarrow RES(T)$ such that for every sequence $op_1, op_2, \ldots, op_k$ of operations, $(op_1, f(op_1))$, $(op_2, f(op_2))$, ..., $(op_k, f(op_k))$ is consistent with respect to T. An object type is *non-trivial* if it is not trivial. The following proposition is immediate from the definitions.

Proposition 5.3 *Let T be a deterministic non-trivial object type, and $f_0 : OP(T) \rightarrow RES(T)$ be the function such that for all op, $(op, f_0(op))$ is consistent with respect to T.*[8] *Then there exists a $k \geq 1$ and a sequence $op_1, op_2, \ldots, op_k, op_{k+1}$ of operations such that $(op_1, f_0(op_1)), (op_2, f_0(op_2)), \ldots, (op_k, f_0(op_k))$ is consistent with respect to T, but $(op_1, f_0(op_1))$, $(op_2, f_0(op_2))$, ..., $(op_k, f_0(op_k))$, $(op_{k+1}, f_0(op_{k+1}))$ is not.*

Theorem 5.3 *Let T be any deterministic non-trivial object type. The resource complexity of any t-tolerant gracefully degrading implementation of T for R-omission is at least $2t+1$.*

Proof Suppose T has a t-tolerant gracefully degrading implementation $\mathcal{I}$ from some list $T_1, T_2, \ldots, T_{2t}$ of object types for R-omission. Let $O_1, O_2, \ldots, O_{2t}$ be base objects of type

[8] Note that $f_0(op)$ is the response of an object of type T when op is the first operation applied to that object.

$T_1, T_2, \ldots, T_{2t}$, and let $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_{2t})$ be the corresponding derived object (of type T). Let f_0 and $op_1, op_2, \ldots, op_k, op_{k+1}$ be as in Proposition 5.3. Consider the following scenario in which two processes P and Q access the object $\mathcal{O}$. At the start of the scenario, object $\mathcal{O}$ is in the initial state, and all its base objects fail, as described below.

For objects O_i, $1 \leq i \leq t$: Whenever P invokes an operation on O_i, it returns a correct response to P and undergoes an appropriate change of state; but whenever Q invokes an operation on O_i, it returns $\perp$ and does not undergo any change of state. For objects O_j, $t+1 \leq j \leq 2t$: Whenever P invokes an operation on O_j, it returns $\perp$ and does not undergo any change of state; but whenever Q invokes an operation on O_j, it returns a correct response to Q and undergoes an appropriate change of state.

`Scenario S`

1. Process Q executes the sequence $op_1, op_2, \ldots, op_k$ of operations on $\mathcal{O}$. Let $v_1, v_2, \ldots, v_k$ be the corresponding responses.
2. Process P executes op_{k+1} on $\mathcal{O}$.

(All steps in Item 1 strictly precede every step in Item 2). Note that:

1. The failure of each base object is by R-omission.
2. The scenario S is indistinguishable to Q from a scenario S′ in which $O_1, O_2, \ldots, O_t$ fail as above, but $O_{t+1}, O_{t+2}, \ldots, O_{2t}$ are correct. Since $\mathcal{O}$ is derived from a t-tolerant implementation, the responses to $op_1, op_2, \ldots, op_k$ returned by Q in S′ must be correct. So the responses in S′ must be $f_0(op_1), f_0(op_2), \ldots, f_0(op_k)$, respectively. Since S and S′ are indistinguishable to Q, Q returns the same responses in S.
3. When P executes op on $\mathcal{O}$, the manner in which objects have failed makes it impossible for P to know whether Q previously executed any operations on $\mathcal{O}$. So, the scenario S is indistinguishable to P from a scenario S″ in which (i) it is the first process to invoke an operation on $\mathcal{O}$, and (ii) only t base objects, namely $O_{t+1}, O_{t+2}, \ldots, O_{2t}$, fail. Since $\mathcal{O}$ is derived from a t-tolerant implementation, P must return the correct response in S″. So P must return $f_0(op_{k+1})$ in S″. Since S is indistinguishable to P from S″, P also returns the response $f_0(op_{k+1})$ in S.

By Proposition 5.3, $(op_1, f_0(op_1)), (op_2, f_0(op_2)), \ldots, (op_k, f_0(op_k)), (op_{k+1}, f(op_{k+1}))$ is not consistent with respect to T. So, the history of object $\mathcal{O}$ in the above scenario is not linearizable with respect to its type T. Thus, $\mathcal{O}$ does not satisfy Property 3 of R-omission in Section 3.1.2. In other words, the failure of $\mathcal{O}$ is not by R-omission, even though the base objects of $\mathcal{O}$ have only failed by R-omission. This implies that $\mathcal{I}$, the implementation from which $\mathcal{O}$ is derived, is not gracefully degrading for R-omission. □

5.1.3 Translation from R-arbitrary to R-omission

A self-implementation $\mathcal{I}$ of object type T is a *t-tolerant translation from a failure model $\mathcal{M}$ to a failure model $\mathcal{M}'$ for T* if every derived object $\mathcal{O}$ of $\mathcal{I}$ satisfies the following property:

In every execution E, if at most t base objects of $\mathcal{O}$ fail, and fail by $\mathcal{M}$, then either $\mathcal{O}$ is correct or it fails by $\mathcal{M}'$. Note that if no base objects fail in E, then $\mathcal{O}$ does not fail either (this follows from the definition of implementation).

In this section, we present a t-tolerant translation from R-arbitrary to R-omission for **N-consensus**. We also show that its resource complexity, $3t+1$, is optimal. This translation can be used along with the t-tolerant self-implementation of **N-consensus** for R-omission (seen in Section 5.1.2) to obtain a t-tolerant self-implementation of **N-consensus** for R-arbitrary failures.

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

```
A[1...2t+1], B[1...t] : N-consensus objects

  Procedure Propose(p, v_p, O)
      count_p[0..1], w, i, belief_p : integer local to p
  begin
      Phase 1: count_p[0..1] := (0,0)
               for i := 1 to 2t+1 do
                   w := f-propose(p, v_p, A[i])
                   count_p[w] := count_p[w] + 1
      Phase 2: Choose belief_p such that
                   count_p[belief_p] > count_p[¬belief_p].
               for i := 1 to t do
                   if belief_p ≠ f-propose(p, belief_p, B[i]) then
                       return(⊥)
               return(belief_p)
  end
```

Figure 3: t-tolerant translation from R-arbitrary to R-omission for **N-consensus**

Let $\mathcal{O}$ be an N-consensus object derived from the translation in Figure 3. The base objects of $\mathcal{O}$ are $A[1 \ldots 2t+1]$, $B[1 \ldots t]$.

Claim 5.2 *$\mathcal{O}$ satisfies integrity in any execution in which all base objects of $\mathcal{O}$ are correct.*

Proof Clear from the algorithm. □

Claim 5.3 *$\mathcal{O}$ is wait-free in any execution in which all base objects of $\mathcal{O}$ are wait-free.*

Proof Clear from the algorithm. □

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

Claim 5.4 *$\mathcal{O}$ satisfies weak integrity in E.*

Proof Clear from the algorithm. □

Claim 5.5 *$\mathcal{O}$ satisfies validity in E.*

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 9), 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 Proposition 5.1, $A[i]$ satisfies validity. 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 5.6 *$\mathcal{O}$ satisfies agreement in E.*

Proof Suppose $\mathcal{O}$ fails to satisfy agreement by returning $v \in \{0, 1\}$ to some process p, and $\overline{v}$ to a different process q. $\mathcal{O}$ returns v to p implies $v = belief_p$. Similarly $\overline{v} = belief_q$. We thus 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 to p implies, for all $1 \leq i \leq t$, $\texttt{propose}(p, belief_p, B[i])$ returns $belief_p = v$ to p. Similarly, for all $1 \leq i \leq t$, $\texttt{propose}(q, belief_q, B[i])$ returns $belief_q = \overline{v}$ to q. Thus all t base objects $B[1 \ldots t]$ 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. □

From the above claims, 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 R-omission in every execution in which at most t base objects of $\mathcal{O}$ fail by R-arbitrary, and the remaining base objects are correct. Thus,

Theorem 5.4 *Figure 3 presents a t-tolerant translation from R-arbitrary failures to R-omission failures for* $\texttt{N-consensus}$*. The resource complexity of the translation is* $3t + 1$*.*

Theorem 5.5 *The resource complexity of any translation $\mathcal{I}$ from R-arbitrary to R-omission for* $\texttt{N-consensus}$ *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 *accesses a base object* o_i if during the execution of $\texttt{Propose}(p, v_p, \mathcal{O})$, p executes $\texttt{propose}(p, *, o_i)$.

Claim 5.7 *Suppose p executes* `Propose`$(p, 0, \mathcal{O})$ *to completion. If all base objects are correct, then p accesses at least* $t + 1$ *base objects.*

Proof Suppose the claim is false, and p accesses only $o_{i_1}, o_{i_2}, \ldots, o_{i_m}$ $(m \leq t)$ before completing `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.

<u>`Scenario S1`</u>

1. p executes `Propose`$(p, 0, \mathcal{O})$ to completion accessing 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.

<u>`Scenario S2`</u>

1. $o_{i_1}, o_{i_2}, \ldots, o_{i_m}$ fail and behave as though they are accessed 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`, $\mathcal{O}$ must be correct in `S1`. By Proposition 5.1, $\mathcal{O}$ 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 Propositions 5.1 and 5.2, $\mathcal{O}$ is neither correct nor does it fail by R-omission. Since at most t base objects fail in `S2`, and they fail by R-arbitrary, the translation $\mathcal{I}$ is incorrect, a contradiction. □

Claim 5.8 *Consider*

<u>`Scenario S3`</u>

1. *p executes* `Propose`$(p, 0, \mathcal{O})$ *up to the point where it has accessed exactly 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 $S = \{\text{base objects accessed by } q\} - \{o_{i_1}, o_{i_2}, \ldots, o_{i_t}\}$. Let $o_{j_1}, o_{j_2}, \ldots, o_{j_k}$ be all the base objects in S arranged in order of first invocation of q. Note that $k \leq n - t \leq 2t$.

Let `S2`$'$ represent scenario `S2` when $m = t$. Since at most t base objects fail in `S2`$'$, and they fail by R-arbitrary, $\mathcal{O}$ must either be correct or fail by R-omission. Hence, by Propositions 5.1 and 5.2, $\mathcal{O}$ 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`. Since no base object fails in `S3`, $\mathcal{O}$ must be correct. By Proposition 5.1, $\mathcal{O}$ satisfies integrity in `S3`. So `Propose`$(q, 1, \mathcal{O})$ returns either 0 or 1 in `S3`. Together with the above conclusion, this implies the claim. □

Claim 5.9 *Consider*

Scenario S4

1. *p executes* Propose$(p, 0, \mathcal{O})$ *up to the point where it has accessed exactly 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 accessed 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 accessed exactly 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 accessed by q exactly as in S4.
3. p completes the execution of Propose$(p, 0, \mathcal{O})$.

Since $k \leq 2t$, the number of base objects that fail in S5 $= k - t \leq t$. Since they fail by R-arbitrary in S5, either $\mathcal{O}$ is correct in S5, or $\mathcal{O}$ fails by R-omission in S5. Thus, by Propositions 5.1 and 5.2, $\mathcal{O}$ satisfies validity and weak integrity in S5. 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, $\mathcal{O}$ is correct in S4, and by Proposition 5.1, it satisfies integrity in S4. Thus Propose$(p, 0, \mathcal{O})$ returns 0 in S4. □

Claim 5.10 *Consider*

Scenario S6

1. *p executes* Propose$(p, 0, \mathcal{O})$ *up to the point where it has accessed exactly 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 5.8.*
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 accessed by q.*
4. *p completes the execution of* Propose$(p, 0, \mathcal{O})$.

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

Proof Note that S4 $\approx_p$ S6. By Claim 5.9, Propose$(p, 0, \mathcal{O})$ returns 0 in S4. So 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 Propositions 5.1 and 5.2, $\mathcal{O}$ fails in S6, but not by R-omission. Since at most t base objects fail in S6, and they fail by R-arbitrary, the translation $\mathcal{I}$ is incorrect, a contradiction. This completes the proof of Theorem 5.5. □

5.1.4 Tolerating R-arbitrary failures

Since N-consensus has a t-tolerant translation from R-arbitrary to R-omission (of resource complexity $3t + 1$), and has a t-tolerant self-implementation for R-omission failures (of resource complexity $t + 1$), it follows that N-consensus has a t-tolerant self-implementation for R-arbitrary failures. However the resulting self-implementation is expensive, requiring $(3t + 1)(t + 1)$ base objects. In this section, we present a t-tolerant self-implementation for R-arbitrary failures whose resource complexity is only $O(t \log t)$.[9] This self-implementation uses the divide-and-conquer strategy. In Figure 4, we present the base step: obtaining a 1-tolerant self-implementation of resource complexity 6. In Figure 6, we show the recursive step of obtaining a t-tolerant self-implementation from a $t/2$-tolerant self-implementation. Consider the 1-tolerant self-implementation of N-consensus given in Figure 4:

Claim 5.11 *Let i be either 1 or 4. If at most one object among O_i, O_{i+1}, and O_{i+2} fails, then* Majority$(p, O_i, O_{i+1}, O_{i+2}, v)$ *returns $\overline{v}$ only if there is a concurrent or preceding execution of* Majority$(q, O_i, O_{i+1}, O_{i+2}, \overline{v})$.

Proof Clear from the algorithm. □

Claim 5.12 *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,* Majority$(p, O_i, O_{i+1}, O_{i+2}, v_p)$ *returns the same value as* Majority$(q, O_i, O_{i+1}, O_{i+2}, v_q)$.

Proof Clear from the algorithm. □

Theorem 5.6 *Figure 4 gives a 1-tolerant self-implementation of* N-consensus *for R-arbitrary failures.*

Proof Consider an execution E in which at most one of $O_1, O_2, \ldots, O_6$ fails by R-arbitrary and the remaining are correct. Claim 5.11 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

[9] This implementation, and all other implementations for R-arbitrary failures in this paper, are gracefully degrading. Graceful degradation for R-arbitrary failures is, however, almost trivial to achieve: it only requires that, if all base objects are wait-free, then the derived object is also wait-free. For brevity, we omit references to graceful degradation in this section.

```
O_i : N-consensus objects (1 ≤ i ≤ 6)

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 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 := Majority(p, O_1, O_2, O_3, v)
    v := Majority(p, O_4, O_5, O_6, v)
    return(v)
end
```

Figure 4: 1-tolerant self-implementation of N-consensus for R-arbitrary failures

the latter case, Claim 5.12 implies that $\mathcal{O}$ satisfies agreement in E. In the former case, Claims 5.11 and 5.12 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. $\square$

Given this 1-tolerant self-implementation, by Booster lemma (Corollary 4.2) we 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.

A more efficient recursive algorithm is presented in Figure 6. This algorithm implements a t-tolerant N-consensus object $\mathcal{O}$ from O_1, a $\lceil\frac{t-1}{2}\rceil$-tolerant N-consensus object, O_2, a $\lfloor\frac{t-1}{2}\rfloor$-tolerant N-consensus object, and the following (0-tolerant) N-consensus objects: $A_0[1\ldots 3t+1], A_1[1\ldots 3t+1]$ and $B[1\ldots 4t+1]$. Figure 5 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 symmetrical).

Consider an execution E in which at most t base objects fail by R-arbitrary. Since O_1 is $\lceil\frac{t-1}{2}\rceil$-tolerant and O_2 is $\lfloor\frac{t-1}{2}\rfloor$-tolerant, either O_1 or O_2 is correct in E. The algorithm in Figure 6 is based on this key observation. We now sketch the intuition behind Figure 6.

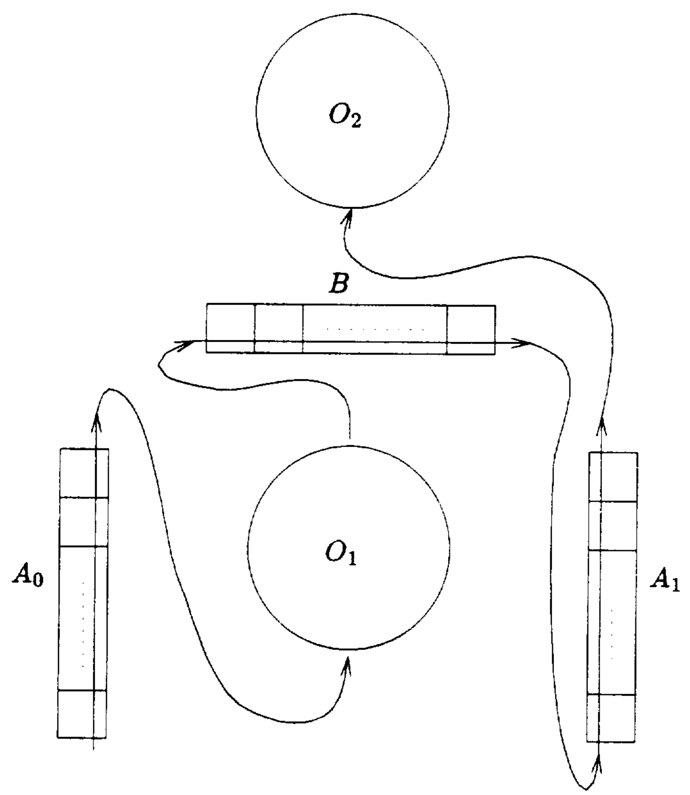

Figure 5: Execution trace of a process proposing 0 on $\mathcal{O}$

A process p executing $\texttt{Propose}(p, v_p, \mathcal{O})$ first executes $\texttt{f-propose}(p, v_p, O_1)$; if O_1 seems correct to p, p adopts the value returned by $\texttt{f-propose}(p, v_p, O_1)$ for $\texttt{Propose}(p, v_p, \mathcal{O})$. If p detects that O_1 failed, p uses O_2 to determine the response for $\texttt{Propose}(p, v_p, \mathcal{O})$.

Process p uses objects $A_0[1 \ldots 3t+1], A_1[1 \ldots 3t+1]$ and $B[1 \ldots 4t+1]$ to determine whether O_1 fails in E. O_1 can fail in one of the following 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 $\texttt{f-propose}$ as a "filter". The second and third cases are detected by using $A_v[1 \ldots 3t+1]$ and $B[1 \ldots 4t+1]$ respectively.

Note that the failure detection provided by $A_0[1 \ldots 3t+1], A_1[1 \ldots 3t+1]$ and $B[1 \ldots 4t+1]$ is not perfect. O_1 may seem correct to some processes, and these processes base their decision on O_1. Others processes may detect that O_1 failed and base their decision on O_2. The implementation in Figure 6 uses B to guarantee that both sets of processes decide on the same value. We describe the implementation in Figure 6 by sketching how it overcomes the different types of failures that O_1 may exhibit:

- O_1 returns a value that is not in $\{0,1\}$. As before, procedure $\texttt{f-propose}$ "filters" the response to eliminate this problem.

- O_1 returns a value that was not proposed by any process. $A_0[1 \ldots 3t+1]$ and $A_1[1 \ldots 3t+1]$ are used to detect that O_1 failed, as follows.

  Process p executes $\texttt{f-propose}(p, v_p, A_{v_p}[i])$, for $1 \leq i \leq 3t+1$, before executing $ans1_p := \texttt{f-propose}(p, v_p, O_1)$. It can be shown that if O_1 is correct in E, then all correct objects in $A_{ans1_p}[1 \ldots 3t+1]$ are "set" to $ans1_p$. Since a maximum of t objects in $A_{ans1_p}[1 \ldots 3t+1]$ may fail in E, p expects at least $2t+1$ objects to return $ans1_p$ when p accesses $A_{ans1_p}[1 \ldots 3t+1]$. If p gets fewer than $2t+1$ copies of $ans1_p$, p knows that O_1 failed in E. Thus, p uses O_2 to reach the decision value.

- O_1 may return 0 to some processes and 1 to others processes. $B[1 \ldots 4t+1]$ are used to detect that O_1 failed, as follows.

  Immediately after executing $ans1_p := \texttt{f-propose}(p, v_p, O_1)$, p executes $\texttt{f-propose}(p, ans1_p, B[i])$ for $1 \leq i \leq 4t+1$. If O_1 is correct in E, no process q will execute $\texttt{f-propose}(q, \overline{ans1_p}, B[i])$ for $1 \leq i \leq 4t+1$. Thus, all correct objects in $B[1 \ldots 4t+1]$ will be "set" to $ans1_p$. Since a maximum of t objects in $B[1 \ldots 4t+1]$ may fail in E, p expects at least $3t+1$ objects to return $ans1_p$ when p accesses $B[1 \ldots 4t+1]$. If p gets fewer than $3t+1$ copies of $ans1_p$, p knows that O_1 failed in E. Thus, p uses O_2 to reach the decision value.

If p detects that O_1 failed in E, p uses O_2 to reach a decision. Recall that it is possible that some other process q did not detect O_1's failure, hence $\texttt{Propose}(q, v_q, \mathcal{O})$ returned $ans1_q$. In this case, q gets at least $3t+1$ copies of $ans1_q$ from $B[1 \ldots 4t+1]$. To ensure that p agrees with q in this case, p proposes to O_2 the value v'_p, which is the majority value that it got from $B[1 \ldots 4t+1]$. Note that care is taken to ensure that v'_p is valid: p should have received at least $t+1$ copies of v'_p when p accessed $A_{v'_p}[1 \ldots 3t+1]$. We now prove:

$A_0[1 \ldots 3t+1]$, $A_1[1 \ldots 3t+1]$, $B[1 \ldots 4t+1]$: (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
    count_p[0..1], WitnessCount_p[0..1] := (0,0)

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

    Phase 2: ans1_p := f-propose(p, v_p, O_1)

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

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

    Phase 5: Choose belief_p such that WitnessCount_p[belief_p] > WitnessCount_p[\overline{belief_p}]
             if WitnessCount_p[belief_p] ≥ 3t + 1 and count_p[belief_p] ≥ 2t + 1 then
                 return(belief_p)
             if WitnessCount_p[belief_p] ≥ 2t + 1 and count_p[belief_p] ≥ t + 1 then
                 v'_p := belief_p
             else v'_p := v_p
             ans2_p := propose(p, v'_p, O_2)
             return(ans2_p)
end
```

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

Theorem 5.7 *Figure 6 gives a t-tolerant self-implementation of* N-consensus *for R-arbitrary failures of resource complexity $O(t \log t)$.*

Proof Consider an execution E in which at most t base objects fail by R-arbitrary, and the remaining are correct. We show below, through a series of claims, 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.

Claim 5.13 *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. □

Claim 5.14 *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 $B[1 \ldots 4t+1]$ 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 $A_v[1 \ldots 3t+1]$ 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. □

Claim 5.15 *If O_1 fails in E, $\mathcal{O}$ satisfies validity and agreement in E.*

Proof Suppose O_1 fails. Then by Claim 5.13, O_2 is correct, and thus, satisfies validity and agreement. We need to consider two cases.

<u>CASE 1</u> 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 $val = belief_p$. Since $WitnessCount_q[belief_q] \geq 2t+1$ and $count_q[belief_q] \geq t+1$, either q returns $belief_q = val$ by line 14 and we have agreement between p and q, or q sets v'_q to $belief_q$ by line 16, making v'_q equal to val. Thus every q, that does not return by line 14, proposes $v'_q = val$ on O_2. By the validity property of O_2, $ans2_q = val$, and q returns 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. By the agreement property of O_2, for all p, q, we have $ans2_p = ans2_q$. (Let $val = ans2_p$). Thus, $\mathcal{O}$ satisfies agreement.

By the validity property 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 $count_p[belief_p] \geq t+1$. Since at most t objects in $A_{belief_p}[1 \ldots 3t+1]$ fail, some process q proposed $v_q = belief_p$ on some object in $A_{belief_p}[1 \ldots 3t+1]$. Thus, process q proposed v_q on $\mathcal{O}$. Thus, $\mathcal{O}$ satisfies validity. □

Claim 5.16 *The resource complexity of the implementation in Figure 6 is* $O(t \log t)$.

Proof Denoting the resource complexity of the t-tolerant 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 $f(1) = 6$. □

It is obvious that $\mathcal{O}$ satisfies integrity and is wait-free in E. By Claims 5.14 and 5.15, $\mathcal{O}$ satisfies validity and agreement in E. Thus, by Proposition 5.1, $\mathcal{O}$ is correct in E. This completes the proof of Theorem 5.7. □

5.2 Fault-tolerant implementation of register

The `register` type supports two operations, *read* and *write v*. The sequential specification is simple: *read* returns the value most recently written.

In [Lam86], Lamport defined three types of registers: *safe, regular,* and *atomic*. Atomic register corresponds to `register` in our terminology. A safe register is not linearizable, but it satisfies the following: a read operation that does not overlap with a write, returns the latest value written into the register. A read that overlaps with a write may return an arbitrary value.

In the following, we first show how to build a fault-tolerant safe register from safe registers, some of which may experience R-arbitrary failures. We then resort to the register construction results in the literature to show that `register` has a t-tolerant self-implementation for R-arbitrary failures.

Lemma 5.1 *A t-tolerant 1-reader, 1-writer, n-valued* (resp. *unbounded*) *safe register can be implemented from* $2t+1$ *1-reader, 1-writer, n-valued* (resp. *unbounded*) *safe registers, at most t of which may experience R-arbitrary failures.*

Proof (sketch) The implementation is as follows. To read the derived safe register, the reader reads all $2t+1$ base registers, and returns the majority response. If there is no majority, it returns an arbitrary value. To write a value v into the derived register, the writer writes v to all $2t + 1$ base registers. It is easy to verify that the above scheme implements a safe register that is correct even if at most t base registers experience R-arbitrary failures. □

Since one can implement a multi-reader, multi-writer n-valued (resp. unbounded) atomic register using 1-reader, 1-writer, boolean (resp. unbounded) safe registers, we have:

Theorem 5.8 `boolean register` *and* `unbounded register` *have t-tolerant self-implementations for R-arbitrary failures.*

5.3 Universality results

We now describe how to implement fault-tolerant objects of a generic type. Let `N-consensus with reset` be an N-process object type informally defined as follows: In addition to *propose 0* and *propose 1* operations, `N-consensus with reset` supports a *reset* operation. The reset operation re-initializes the object so that it may be used for a fresh round of consensus (see Appendix D for a formal specification of this type).

Herlihy showed that *every* finite object type[10] has an implementation from (`N-consensus with reset, unbounded register`)[11] [Her91]. The use of unbounded registers can be replaced by boolean registers [Plo89, JT92]. Using this result, together with Theorems 5.7 and 5.8, we obtain the following corollary.

Corollary 5.1 *Let $\mathcal{M}$ be any responsive failure model, and T be any finite object type.*

- *T has a t-tolerant implementation from* (`N-consensus with reset, boolean register`) *for $\mathcal{M}$.*
- *If* `N-consensus with reset` *and* `boolean register` *have gracefully degrading implementations from T for $\mathcal{M}$, then T has a t-tolerant self-implementation for $\mathcal{M}$.*

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 5.1 holds even for an infinite object type T, provided that `boolean register` is replaced by `unbounded register` in the statement of the corollary.

The types `fetch&add, queue, stack, test&set` implement `2-consensus`, and `compare&swap, move, swap` implement `N-consensus` [Her91].

It is easy to show that `compare&swap, move, swap, test&set` implement `boolean register`, and `fetch&add, queue, stack` implement `unbounded register`. Furthermore, all these implementations are gracefully degrading for R-arbitrary failures. Thus,

[10] Notice that, by our definition of object type, every object type has a "sequential implementation".

[11] For this implementation, it suffices if the *reset* operation on an N-consensus object works in the absence of concurrent operations on that object.

Corollary 5.2 *The following object types have t-tolerant self-implementations for R-arbitrary failures:* (2-process) **fetch&add, queue, stack, test&set,** *and* (N-process) **compare&swap, move, swap.**

6 Tolerating non-responsive failures

So far we have considered base objects that remain responsive (*i.e.*, wait-free) even if they fail. Thus, a process can access a base object and afford to wait for a response before proceeding to access the next one. In other words, base objects can be accessed sequentially. With non-responsive failures, waiting on a base object that fails could block the process forever. Hence, to tolerate non-responsive failures, we allow a process to access base objects "in parallel"[12], so that it can complete its operation on the derived object even if some of the base objects fail and never respond.

As we will see, this ability to access base objects in parallel allows us to build t-tolerant implementations of **register**, even for arbitrary failures. In contrast, we show that **N-consensus** does not have a (deterministic) implementation that tolerates the crash of a single base object even if we do not restrict the number and the type of the base objects that can be used in the implementation. However, randomization circumvents this impossibility result. *Every* object type has a t-tolerant *randomized* implementation from **register**, even for arbitrary failures.

The impossibility results of this section are proved by reducing the consensus problem [FLP85] to the problem in question. The *consensus problem* for a system of N processes is defined as follows. Each process p_i has an initial binary input v_i. The consensus problem requires each correct process to reach the same (irrevocable) decision value d such that $d \in \{v_1, v_2, \ldots, v_N\}$.

Theorem 6.1 *There is no 1-tolerant implementation of* **2-consensus** *for crash failures.*

Proof Suppose, for contradiction, 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. We will use this implementation to solve the consensus problem among a set of $l+2$ processes, one of which may crash, in a system in which processes communicate only through registers.

Consider the concurrent system S consisting of $l+2$ processes named $\{p_1, p_2\} \cup \{q_j | 1 \leq j \leq l\}$, and $4l+1$ registers named $\{invocation(i,j),\ response(j,i)\ |1 \leq i \leq 2, 1 \leq j \leq l\} \cup \{decision\}$. We claim that the consensus problem is solvable in S even if one process crashes. 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:

[12] However, we do not allow a process to invoke an operation on a base object if its previous invocation on that object is still pending.

1. Use a set $\{o_1, o_2, \ldots, o_l\}$ of base objects of type $T_1, T_2, \ldots, T_l$, and the implementation $\mathcal{I}$, to construct a **2-consensus** object $\mathcal{O} = \mathcal{I}(o_1, \ldots, o_l)$ that tolerates the crash of one of its base objects.
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 $\texttt{Propose}(p_i, v_i, \mathcal{O})$ on the derived object $\mathcal{O}$.

The details are given below.

Initialize all $4l + 1$ registers to $\perp$. Process p_i simulates $\texttt{Propose}(p_i, v_i, \mathcal{O})$ as follows. If $\texttt{Propose}(p_i, v_i, \mathcal{O})$ requires p_i to invoke some operation *op* on o_j, p_i appends *op* to the contents of *invocation*(i, j). If $\texttt{Propose}(p_i, v_i, \mathcal{O})$ 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 $\texttt{Propose}(p_i, v_i, \mathcal{O})$ 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 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 crash of at most one (simulated) base object of the (simulated) derived object $\mathcal{O}$ in E'. Since $\mathcal{I}$, the **2-consensus** implementation from which $\mathcal{O}$ is derived, is 1-tolerant for crash, $\mathcal{O}$ is correct in E' (despite the 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, $\texttt{Propose}(p_i, v_i, \mathcal{O})$ eventually returns some value v (in E'). Since $\mathcal{O}$ satisfies integrity, v is a binary value. Since $\mathcal{O}$ satisfies validity, v is either v_1 or v_2. Since $\mathcal{O}$ satisfies agreement, $\texttt{Propose}(p_1, v_1, \mathcal{O})$ and $\texttt{Propose}(p_2, v_2, \mathcal{O})$ 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, 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 that 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 6.2 *There is no 1-tolerant implementation of* 2-consensus *for 1-unfairness to a known process.*

Proof (sketch) Suppose, for contradiction, 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 1-unfairness to, say, process p_1. Consider the concurrent system S, as defined in the proof of Theorem 6.1. 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 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 2-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 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 *invocation*$(2, j)$, as before, but now it also waits until a corresponding response is appended to *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[13]. Therefore, at worst, to process p_1, 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, $\mathcal{O}$ remains correct. Since p_1 does not crash (we assumed that only one process in S crashes, and this is q_k), Propose$(p_1, v_1, \mathcal{O})$ returns a value that p_1 writes into *decision*. The rest of the proof is as in Theorem 6.1.

Again, we have a contradiction to the impossibility result in [LAA87].

□

From the above two theorems we have:

[13] Of course, it also continues to read the *decision* register periodically, and decides if a non-$\perp$ value is found there.

Corollary 6.1 *If type T implements* `2-consensus`, *then there is no 1-tolerant implementation of T for crash or for 1-unfairness to a known process.*

From [Her91] and this corollary, we conclude that `compare&swap`, `fetch&add`, `move`, `queue`, `stack`, `sticky-bit`, `swap`, `test&set`, 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 6.3 `boolean register` *and* `unbounded register` *have t-tolerant self-implementations for arbitrary failures.*

This follows immediately from the following lemma and the fact that one can implement a multi-reader, multi-writer n-valued (resp. unbounded) atomic register using 1-reader, 1-writer, boolean (resp. unbounded) safe registers.

Lemma 6.1 *A t-tolerant 1-reader, 1-writer, n-valued* (resp. *unbounded*) *safe register can be implemented from* $5t+1$ *1-reader, 1-writer, n-valued* (resp. *unbounded*) *safe registers, at most t of which may experience arbitrary failures.*

Proof (sketch) Informally, the reader invokes a 'read' on each base register (the reader delays this read if its previous read on the base register is still pending). When it gets a response from $4t+1$ distinct registers, it returns the majority value. If there is no majority, it returns an arbitrary value. To write a value v, the writer invokes a 'write v' on each base register (again, this write is delayed if the previous write on the base register is still pending). The writing completes when $4t+1$ base registers return an "ack". It is easy to verify that the above scheme implements a safe register that is correct even if at most t base registers experience arbitrary failures. □

Randomized implementations of `N-consensus` from `register` are well known (for example, see [Asp90]). Together with Theorem 6.3, this implies that randomized t-tolerant implementations of `N-consensus` from `register` exist for arbitrary failures. Combining this with Theorem 6.3 and the universality results of [Her91, Plo89], we have

Theorem 6.4 *Every finite object type has a randomized t-tolerant implementation from* `boolean register` *for arbitrary failures, and every infinite object type has a randomized t-tolerant implementation from* `unbounded register` *for arbitrary failures.*

Thus, if a finite (resp. infinite) object type T implements `boolean register` (resp. `unbounded register`), then T has a *randomized* t-tolerant self-implementation for arbitrary failures. This implies that `compare&swap, fetch&add, queue, move, stack, swap, test&set` have t-tolerant randomized self-implementations, even for arbitrary failures!

Our next result concerns the nature of arbitrary failures. It states that the problem of tolerating arbitrary failures can be reduced to two strictly simpler problems: tolerating R-arbitrary failures and tolerating omission failures.

Lemma 6.2 (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 $\mathcal{I}_a$ for R-arbitrary failures, and $\mathcal{I}_o$ for omission failures.*

Proof (sketch) The "only if" direction is obvious. To prove the "if" direction, define $\mathcal{I}(o_1, o_2, \ldots, o_{nm}) = \mathcal{I}_o(\mathcal{I}_a(o_1, \ldots, o_m), \ldots, \mathcal{I}_a(o_{(n-1)m+1}, \ldots, o_{nm}))$. It can be verified that $\mathcal{I}$ is a t-tolerant self-implementation of T for arbitrary failures. □

7 Graceful degradation for benign failure models

We have seen that every object type has a t-tolerant implementation for R-crash and R-omission failures. But what if we also require the implementation to be gracefully degrading? The results are mostly negative for R-crash, but not so for R-omission.

7.1 R-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 R-crash. Suppose we wish to implement an object $\mathcal{O}$ of type T using only objects in S, and that we require $\mathcal{O}$ to function correctly only in the absence of failures. However, when objects in S fail by R-crash, we would like $\mathcal{O}$ to fail only by R-crash. This last requirement is desirable for two reasons:

- The benign failure semantics of R-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 all the objects in $S \cup \{\mathcal{O}\}$ in hardware.

In our terminology, we are seeking a gracefully degrading implementation of T for R-crash from the types (of the objects) in S. Unfortunately, as we show below, many object types do not have such implementations, even from very powerful object types. This negative result implies that, in many cases, the simple and desirable R-crash failure semantics cannot be achieved.

An object type T is *order-sensitive* if it is deterministic and the following holds: There exists a state S in $G(T)$, 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 consistent from the state S of T, and $u \neq v$ and $u' \neq v'$. Intuitively, when an object $\mathcal{O}$ is in the state S, and two processes p and q invoke operations op and op' concurrently on $\mathcal{O}$, they can, based on the return values, determine the order in which their operations are linearized. **queue** is an example of an order-sensitive object type. To see this, let S be the state in which there are two elements 5 and 10 in the queue (5 at the head), 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**, **N-consensus**, **stack**, **test&set** 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 7.1 *There is no gracefully degrading implementation of any order-sensitive object type for R-crash from any list of non-order-sensitive object types.*

Proof Suppose there are T, $\mathcal{L}$, and $\mathcal{I}$ such that T is an order-sensitive type, $\mathcal{L} = \{T_1, T_2, \ldots, T_n\}$ is a list of non-order-sensitive types, and $\mathcal{I}$ is a gracefully degrading implementation of T from $\mathcal{L}$ for R-crash. We arrive at a contradiction after a series of claims involving bivalency arguments [FLP85] and 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 an order-sensitive type. Consider the concurrent system consisting of two processes p and q, and the shared object $\mathcal{O}$ (implemented from $O_1, O_2, \ldots, O_n$). Define the *configuration* (at an instant t) as the tuple (S_p, S_q, S_o) where S_p, S_q, and S_o are the states of process p, process q, and object $\mathcal{O}$ respectively (at the instant t). Let C_0 denote the configuration in which $\mathcal{O}$ is in state S, and p, q are about to execute $\texttt{Apply}(p, op, \mathcal{O})$ and $\texttt{Apply}(q, op', \mathcal{O})$ respectively.

Claim 7.1 *Suppose all base objects are correct. For any interleaving of the steps in the complete executions of* $\texttt{Apply}(p, op, \mathcal{O})$ *and* $\texttt{Apply}(q, op', \mathcal{O})$, *either* $\texttt{Apply}(p, op, \mathcal{O})$ *returns* u *and* $\texttt{Apply}(q, op', \mathcal{O})$ *returns* u', *or* $\texttt{Apply}(p, op, \mathcal{O})$ *returns* v *and* $\texttt{Apply}(q, op', \mathcal{O})$ *returns* v'.

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

Let C denote a configuration reached from C_0 after some interleaving of (partial) executions of $\texttt{Apply}(p, op, \mathcal{O})$ and $\texttt{Apply}(q, op', \mathcal{O})$. We say C is *X-valent* if, in the absence of base object failures, $\texttt{Apply}(p, op, \mathcal{O})$ returns X, no matter how the steps of $\texttt{Apply}(p, op, \mathcal{O})$ and $\texttt{Apply}(q, op', \mathcal{O})$ interleave when execution resumes from C. By Claim 7.1, 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.

Claim 7.2 *C_0 is bivalent.*

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

Similarly, starting from C_0, if q completes all the steps of $\texttt{Apply}(q, op', \mathcal{O})$ before p starts $\texttt{Apply}(p, op, \mathcal{O})$, then $\texttt{Apply}(q, op', \mathcal{O})$ returns v'. Thus, by Claim 7.1, when $\texttt{Apply}(p, op, \mathcal{O})$ completes, it returns v. Thus C_0 is not u-valent.

Since C_0 is neither u-valent nor v-valent, it is bivalent. □

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 $\texttt{Apply}(p, op, \mathcal{O})$ from C. $C(q)$ is similarly defined.

Claim 7.3 *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 $\texttt{Apply}(p, op, \mathcal{O})$ and $\texttt{Apply}(q, op', \mathcal{O})$ as shown in Figure 7. 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 claim. □

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

Figure 7: 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.

Claim 7.4 *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 $u \neq v$. □

Assume that O_k is the base object mentioned in the above claim, and $\texttt{Apply}(p, oper, O_k)$, $\texttt{Apply}(q, oper', O_k)$ 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 $\texttt{Apply}(q, oper', O_k)$ returns the same value whether applied in C_{crit} or $C_{crit}(p)$, or $\texttt{Apply}(p, oper, O_k)$ 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.

Claim 7.5 *Consider*

<u>`Scenario S1`</u> (Starts from the configuration C_{crit})

1. *Process q takes the step* $\texttt{Apply}(q, oper', O_k)$.

2. *Process p completes the execution of* $\texttt{Apply}(p, op, \mathcal{O})$.

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

4. *Process q resumes and completes the execution of* $\texttt{Apply}(q, op', \mathcal{O})$.

Then $\texttt{Apply}(p, op, \mathcal{O})$ *returns* v *and* $\texttt{Apply}(q, op', \mathcal{O})$ *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 $\texttt{Apply}(p, op, \mathcal{O})$ in Item 2, $\texttt{Apply}(p, op, \mathcal{O})$ returns v in Item 2 of the scenario.

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

Suppose $\texttt{Apply}(q, op', \mathcal{O})$ returns w where $\perp \neq w \neq v'$. 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, v),(op', w) or (op', w),(op, v) is a sequential execution from S consistent with T. This is false since (op, u),(op', u') and (op', v'),(op, v) are the only sequences consistent from the state S of T, and $u \neq v$, $w \neq v' \neq v$.

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

Claim 7.6 *Consider*

<u>`Scenario S2`</u> (Starts from the configuration C_{crit})

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

2. *Process q takes the step* $\texttt{Apply}(q, oper', O_k)$.

3. *Process p resumes and completes the execution of* $\texttt{Apply}(p, op, \mathcal{O})$.

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

5. *Process q resumes and completes the execution of* $\texttt{Apply}(q, op', \mathcal{O})$.

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

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

Neither (op, u),(op', v') nor (op', v'),(op, u) is a sequence consistent from the state S of T. Hence the execution in Claim 7.6 is not linearizable. Thus the failure of $\mathcal{O}$ in S2 is not by R-crash. We conclude that $\mathcal{I}$ is not a gracefully degrading implementation for R-crash, a contradiction which concludes the proof of Theorem 7.1. □

Preserving the failures semantics of the underlying system is a desirable property of an implementation. For R-crash, the above theorem shows that this property is often not achievable: implementations necessarily amplify the R-crash failures of base objects. 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. Suppose we are given the following guarantee: if any of the given registers or sticky bits fail, they fail only by R-crash. Can we implement a queue that cannot fail more severely than 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 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 most objects of interest. This is shown by the theorem below.

Theorem 7.2 *There is no 1-tolerant gracefully degrading implementation of any order-sensitive object type for R-crash.*

Proof Suppose there are T, $\mathcal{L}$, and $\mathcal{I}$ such that T is an order-sensitive type, $\mathcal{L} = \{T_1, T_2, \ldots, T_n\}$ is a list of types, and $\mathcal{I}$ is a 1-tolerant gracefully degrading implementation of T from $\mathcal{L}$ for R-crash. We arrive at a contradiction after 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. Suppose $\mathcal{O}$ is in state S, and p, q are about to execute $\texttt{Apply}(p, op, \mathcal{O})$ and $\texttt{Apply}(q, op', \mathcal{O})$ respectively.

Claim 7.7 *Suppose all base objects are correct. For any interleaving of the steps in the complete executions of* $\texttt{Apply}(p, op, \mathcal{O})$ *and* $\texttt{Apply}(q, op', \mathcal{O})$*, either* $\texttt{Apply}(p, op, \mathcal{O})$ *returns* u *and* $\texttt{Apply}(q, op', \mathcal{O})$ *returns* u'*, or* $\texttt{Apply}(p, op, \mathcal{O})$ *returns* v *and* $\texttt{Apply}(q, op', \mathcal{O})$ *returns* v'*.*

Proof Same as Claim 7.1. □

Claim 7.8 *There exists a (possibly empty) sequence* α *of steps of* p *and a step* s *of* p *such that the following Scenarios* S1 *and* S2 *are possible.*

<u>**Scenario S1**</u> (scenario starts with $\mathcal{O}$ in state S)

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

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

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

<u>`Scenario S2`</u> (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* $\alpha \cdot s$.

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 7.7 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 7.9 *Consider*

<u>`Scenario S3`</u> (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, $\mathcal{O}$ must either be correct or fail by 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 is false 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 is false since (op, u),(op', u') and (op', v'),(op, v) are the only sequences consistent from the state S of T, and $w \neq u$, $u' \neq v'$.

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

Claim 7.10 *Consider*

<u>`Scenario S4`</u> (scenario starts with $\mathcal{O}$ in state S)

1. *p initiates and* (partially) *executes* `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) `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* `Apply`$(p, op, \mathcal{O})$.

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

Proof Clearly `S4`$\approx_p$`S3`. Therefore, as in `S3`, `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 `Apply`$(q, op', \mathcal{O})$, `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 7.11 *Consider*

<u>`Scenario S5`</u> (scenario starts with $\mathcal{O}$ in state S)

1. *p initiates and partially executes* `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) `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* `Apply`$(p, op, \mathcal{O})$.

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

Proof Clearly `S5`$\approx_q$`S4`. Therefore `Apply`$(q, op', \mathcal{O})$ returns u' as in `S4`. By similar arguments as in Claim 7.9, it can be shown that `Apply`$(p, op, \mathcal{O})$ returns u. □

Claim 7.12 *Consider*

<u>`Scenario S6`</u> (scenario starts with $\mathcal{O}$ in state S)

1. *p initiates and partially executes* `Apply`$(p, op, \mathcal{O})$ *by completing the steps in* α.
2. *q initiates and completes* (all the steps of) `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 Apply$(p, op, \mathcal{O})$.

Then Apply$(p, op, \mathcal{O})$ *returns* u, *and* 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 sequence consistent from the state S of T. Hence the execution in Claim 7.12 is not linearizable. Thus the failure of $\mathcal{O}$ in S6 is not by R-crash. We conclude that $\mathcal{I}$ is not a gracefully degrading implementation for R-crash, a contradiction which concludes the proof of Theorem 7.2. □

The above discussion raises some questions on the "practicality" of the R-crash model: Even if "hardware" objects fail by R-crash, "software" objects usually don't. The R-omission model defined in this paper does not have this serious limitation. In fact, for any $t \geq 0$, *every* N-process object type has a t-tolerant *gracefully degrading* implementation from any universal list of types. 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. These results are presented in the next section.

7.2 R-omission

The object type **N-consensus** is order-sensitive. By Theorem 7.2, **N-consensus** has no t-tolerant gracefully degrading implementation for R-crash. In contrast, **N-consensus** has such an implementation for R-omission (Theorem 5.2 in Section 5). Further, we can show

Theorem 7.3 **register** *has a t-tolerant gracefully degrading self-implementation for R-omission.*

Theorems 5.2 and 7.3 can be combined with the universal constructions in [Her91, JT92] to obtain the following result for R-omission.

A list $\mathcal{L}$ of object types is *N-universal* if every N-process object type has an implementation from $\mathcal{L}$. An example of a N-universal list is (**N-consensus with reset, register**).

Theorem 7.4 *Every N-process object type has a t-tolerant gracefully degrading implementation from any N-universal list of object types for R-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 models 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 model is obtained for $k = \infty$.

In our terminology, these models are responsive. The second one, with $k = \infty$, corresponds to our R-arbitrary failure model.

[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 model.
2. Amongst the responsive failure models, benign ones, such as R-crash and R-omission, are also not considered in [AGMT92].
3. This paper does not consider models 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 are fundamentally different. There is no direct way to model benign failures, such as R-crash and R-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 R-crash and R-omission failure models. For R-arbitrary failures, graceful degradation reduces to the "*strong wait-freedom*" concept considered in [AGMT92].
6. The concept of fault-tolerant *self-implementation*, is a central theme of this paper. Corollary 5.1 states sufficient conditions for their existence, and Corollary 5.2 lists several types that have such implementations. 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`** and

bounded register [AGMT92], with the implementation of **bounded register** from **test&set** [Jay93].

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** and **bounded register** shown in [AGMT92], does not follow from our results (when $N > 2$).

Acknowledgement

We thank Vassos Hadzilacos for many interesting discussions. His detailed comments on an earlier version helped improve the presentation.

References

[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 47–58, August 1992. A draft of a more complete version of this paper dated Aug 7, 1992 was also privately sent to us.

[Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, 1990.

[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.

[FLP85] M. Fischer, N. Lynch, and M. 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.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[Jay93] P. Jayanti. *Fault-Tolerant Wait-Free Shared Objects.* PhD thesis, Cornell University, 1993. Dept. of Computer Science, Cornell University, Ithaca, NY 14853 (In preparation).

[JCT92a] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. Technical Report TR 92-1281, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, April 1992.

[JCT92b] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992.

[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. (To appear in Lecture Notes in Computer Science, Springer-Verlag).

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

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

[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.

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

A Formal model

Our formal model is based on I/O Automata [LT88]. We use the model to make our definitions of failure models (Appendix B) and fault-tolerant implementations (Appendix C) precise. The implementations in the paper are described in the more intuitive Pascal-like style. In the following, we borrow several definitions from in [HW90, Her91]. There are however some differences between our model and Herlihy's [Her91]. Notable among these are: (i) our addition of an explicit "crash" state for a process, (ii) the definitions of wait-freedom, and implementation, (iii) the added assumption of fairness in our model, and (iv) the definition of clocked concurrent systems.

A.1 I/O Automata

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.

An *execution fragment* of an automaton A is a finite sequence $s_0, e_1, s_1, e_2, s_2, \ldots, e_n, s_n$ or an infinite sequence $s_0, e_1, s_1, e_2, s_2, \ldots$ of alternating states and events such that (s_i, e_{i+1}, s_{i+1}) is a step of A. An *execution* is an execution fragment in which s_0 is a starting state. A *history fragment* of an automaton is the subsequence of events in an execution fragment of the automaton. A *history* of an automaton is the subsequence of events in an execution. An execution fragment E is *fair* if either E is finite, or E is infinite and every internal event or an output event that is enabled in every state of a suffix of E occurs infinitely many times in E.[14]

A new automaton can be constructed by composing a set of compatible automata. A set of automata are *compatible* if, no two of them share any internal or output events. That

[14]Since this simple notion of fairness is adequate for our purpose, we do not need the general machinery described in [LT88] for formulating fairness.

is, for every A, B in the set, $(Int(A) \cup Out(A)) \cap (Int(B) \cup Out(B)) = \emptyset$. A state of the composed automaton S is a tuple of the components' states, and a starting state of S is the tuple of the components' starting states. The set of output events of S, $Out(S)$, is the union of the sets of output events of the component automata. The set of internal events of S, $Int(S)$, is the union of the sets of internal events of the component automata. The set of input events of S, $In(S)$, 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 in s and s' is the same.

If H is a history of a composed automaton and $A_1, A_2, \ldots, A_k$ are component automata, then $H|\{A_1, A_2, \ldots, A_k\}$ is the subhistory of H consisting of all events e, where e is an event of one of $A_1, A_2, \ldots, A_k$.

A.2 Object type

An *object type* T is a tuple (N, OP, RES, G), where N is an integer greater than one, OP, RES are sets of operations and responses respectively, and G is a directed finite or infinite graph in which each edge has a label of the form (op, res) where $op \in OP$ and $res \in RES$. Intuitively, if $\mathcal{O}$ is an object of type T, then $\mathcal{O}$ supports the operations in OP and may be shared by N processes (we say T is an *N-process type*). G specifies the expected behavior of $\mathcal{O}$ in the absence of concurrent operations on $\mathcal{O}$.

The vertices of G are the *states of* T. One state of T is the *initial* state. A state s of T is *reachable* if there is a path in G from the initial state to s. We assume that every state of T is reachable. A sequence $S = (op_1, res_1), (op_2, res_2), \ldots, (op_l, res_l)$ is *consistent from a state s of T* if there is a path labeled S in G from the state s. S is *consistent with respect to T* if it is consistent from the initial state of T.

An object type T is *total* if for every state s of T, and every operation $op \in OP$, there is a response res such that there is an edge labeled (op, res) from s in G. All object types studied in this paper are assumed to be total. T is *deterministic* if for every state s of T and every operation $op \in OP$, there is at most one edge from s labeled (op, res). T is *non-deterministic* otherwise. T is *finite* if G is finite; T is *infinite* otherwise.

A.3 Processes and objects

An *object* is an automaton with two attributes: a unique name and a type. A *process* is an automaton with a unique name. A process automaton P satisfies the following properties:

1. There is a distinguished state $CRASHED(P)$ in $States(P)$.

2. The event $crash(P)$ is in $In(P)$.

3. For every state $s \in States(P)$, $(s, crash(P), CRASHED(P))$ is in $Steps(P)$.

4. The event *crashed*(P) is in *Out*(P), and is enabled in the state *CRASHED*(P).

5. if (*CRASHED*(P)$, e, s)$ is in *Steps*(P), then either $e =$ *crashed*(P), or e is an input event of P, and $s =$ *CRASHED*(P).

The above conditions capture the notion that an adversary can crash a process at any time by generating the input event *crash*(P) (see 2 and 3); and once it crashes, a process remains crashed forever (see 5).

A.4 Clock

A *clock* is an automaton with a single state s, a single output event *tick*, and a single step $(s, tick, s)$. It has no input or internal events.

A.5 Concurrent system

A *concurrent system consisting of processes* $P_1, P_2, \ldots, P_n$, *and objects* $O_1, O_2, \ldots, O_m$, is an automaton composed from process automata $P_1, \ldots, P_n$, and object automata $O_1, \ldots, O_m$. We denote such a concurrent system by $(P_1, P_2, \ldots, P_n; O_1, O_2, \ldots, O_m)$. A *clocked concurrent system*[15] consisting of $P_1, \ldots, P_n$, and objects $O_1, \ldots, O_m$ has an additional component, the clock automaton $\mathcal{C}$, and is denoted by $(P_1, \ldots, P_n; O_1, \ldots, O_m; \mathcal{C})$. The output events of a process P_i include *invoke*(P_i, op, O_j), where *op* is an operation supported by the type of O_j, and the input events of P_i include *respond*(P_i, res, O_j), where *res* is a response. We refer to the events *invoke*(P_i, op, O_j) and *respond*(P_i, res, O_j) as invocations and responses respectively. An object O_j includes input events *invoke*(P_i, op, O_j), and output events *respond*(P_i, res, O_j). Process and object names are unique, and no two automata among processes and objects share any internal or output events. This ensures that the process and object automata are compatible, and therefore, can be composed.

Let σ be a sequence of events or a sequence of states and events (for example, σ can be a history or an execution). A response r *matches* an invocation i in σ if i is the latest event in σ that precedes r such that the process and object names of i and r agree. An *operation in* σ is a pair of events, an invocation and its matching response. A relation $<_\sigma$ reflecting the partial "real time" order of operations in $<_\sigma$ is defined as follows: $op <_\sigma op'$ if the response of *op* precedes the invocation of *op'* in σ. Two operations unrelated by $<_\sigma$ are said to be *concurrent* in σ. An invocation is *pending* in σ if it has no matching response. *Complete*(σ) denotes the maximal subsequence of σ in which there is no pending invocation.

A history H of a concurrent system $S = (P_1, P_2, \ldots, P_n; O_1, O_2, \ldots, O_m)$ is *k-well-formed* if, for each pair P_i, O_j, $(H|P_i)|O_j$ begins with an invocation, and alternates invocations and matching responses[16], and $H|P_i$ has at most k pending invocations in H. The

[15] Clock ensures that the system execution progresses, no matter how the other components in the system behave. This simplifies the definition of wait-free implementations, especially wait-free implementations that must tolerate non-responsive failures.

[16] With the exception of the last invocation which may not have a matching response

concurrent system $\mathcal{S}$ is k-well-formed if every history of $\mathcal{S}$ is k-well-formed. Intuitively, in a k-well-formed concurrent system, if an invocation of a process P on object O is pending, then P may not issue a new invocation on O; however, P may issue an invocation on a different object O' as long as the number of pending invocations from P does not exceed k. The need for a k-well-formed system, for $k > 1$, arises while designing implementations that tolerate non-responsive failures of the underlying objects. For example, it is easy to see that any implementation that has to be wait-free in spite of the crash of at most t underlying objects must be at least $(t+1)$-well-formed. We assume that a concurrent system is 1-well-formed unless specifically mentioned otherwise.

In this paper, we restrict our attention to only fair executions of concurrent systems. Thus, when we refer to infinite executions in this section and in Sections 3 and 4, we implicitly assume they are fair.

A.6 Linearizability

The *behavior of an object $\mathcal{O}$ in an execution E*, denoted by $B(\mathcal{O}, E)$, is the subsequence of invocation and response events of $\mathcal{O}$ in E.

A *behavior B is linearizable with respect to type T* if B can be extended to B' by appending zero or more responses, and there is a sequence $\sigma =$ *invoke*$(P_{i_1}, op_1, \mathcal{O})$, *respond*$(P_{i_1}, res_1, \mathcal{O})$, *invoke*$(P_{i_2}, op_2, \mathcal{O})$, *respond*$(P_{i_2}, res_2, \mathcal{O})$, ..., *invoke*$(P_{i_l}, op_l, \mathcal{O})$, *respond*$(P_{i_l}, res_l, \mathcal{O})$, such that:

1. σ is a permutation of the events in Complete(B').
2. $<_B \subseteq <_\sigma$.
3. $(op_1, res_1), (op_2, res_2), \ldots, (op_l, res_l)$ is consistent with respect to T.

Informally, extending B to B' captures the notion that some operations in B may have taken effect, although the responses have not appeared yet. The definition captures the notion that processes appear to interleave at the granularity of complete operations on O (as is evident from the form of σ and Condition 1), the notion that this apparent interleaving respects the real time order (Condition 2) and the semantics of the object type T (Condition 3).

An object $\mathcal{O}$ is *linearizable with respect to type T in a finite execution E* of a concurrent system if $B(\mathcal{O}, E)$ is linearizable with respect to T.

Object $\mathcal{O}$ is linearizable with respect to type T in an *infinite* execution E of a concurrent system if and only if it is linearizable with respect to T in every finite prefix of E.

A.7 Wait-freedom

Let E be an execution of a concurrent system. An object $\mathcal{O}$ is *wait-free* in E if either (i) E is finite, or (ii) every invocation on $\mathcal{O}$ by a process that does not crash in E has a matching response.

A.8 Correctness

An object $\mathcal{O}$ is *correct in an execution* E if one of the following holds:

- $\mathcal{O}$ is is wait-free in E, and $\mathcal{O}$ is linearizable with respect to its type in E.
- More than $N(T)$ distinct processes have invocations on $\mathcal{O}$ in E.

The latter condition captures the notion that an object need not exhibit any sane behavior if accessed by more processes than the object is intended for.

An object $\mathcal{O}$ *fails in an execution* E if it is not correct in E.

A.9 Implementations

Let *Obj*(T) denote the universe of objects whose type is T. Let $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ be a list of object types (T_i's are not necessarily distinct). A *wait-free implementation of* T *from* $\mathcal{L}$ *for processes* $P_1, P_2, \ldots, P_{N(T)}$ is a function $\mathcal{I} : \mathrm{Obj}(T_1) \times \mathrm{Obj}(T_2) \times \ldots \mathrm{Obj}(T_n) \rightarrow \mathrm{Obj}(T)$ satisfying the following conditions:

1. If $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$, the automaton of $\mathcal{O}$ has the structure of a concurrent system: $(F_1, F_2, \ldots, F_{N(T)}; O_1, O_2, \ldots, O_n)$, for some process automata $F_1, F_2, \ldots, F_{N(T)}$.
2. F_i and F_j $(i \neq j)$ have no common events.
3. If $\mathcal{O} = \mathcal{I}(O_1, \ldots, O_n)$, each input event *invoke*$(P_i, op, \mathcal{O})$ of $\mathcal{O}$ is an input event of F_i; each output event *respond*$(P_i, res, \mathcal{O})$ of $\mathcal{O}$ is an output event of F_i.
4. Each output event *crashed*(P_i) of P_i is matched with the input event *crash*(F_i) of F_i.
5. Let $O_1, O_2, \ldots, O_n$ be any distinct objects of type $T_1, T_2, \ldots, T_n$, respectively, and $\mathcal{O} = \mathcal{I}(O_1, \ldots, O_n)$. For every execution E of the clocked concurrent system $(P_1, P_2, \ldots, P_{N(T)}; \mathcal{O}; \mathcal{C})$, if $O_1, O_2, \ldots, O_n$ are correct in E, then $\mathcal{O}$ is also correct in E.

In the above, the F_i's are called the *front-ends*, $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$ is called a *derived object* of the implementation $\mathcal{I}$, and $O_1, O_2, \ldots, O_n$ are called the *base objects* of $\mathcal{O}$. The front-end F_i models the procedure `Apply` (called by process P_i to execute operations on a derived object) alluded to in the informal model of Section 2.

Condition 1 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 implementation by one process P_i cannot affect another process P_j.

Condition 3 captures the notion that (i) invoking an operation on $\mathcal{O}$, by process P_i causes the front-end F_i to be activated, and (ii) the value returned by the front-end F_i is the response of $\mathcal{O}$.

Condition 4 condition 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 behaves correctly when its base objects do.

All implementations studied in this paper are wait-free. Hereafter we write "implementation" as shorthand for "wait-free implementation". The implementation $\mathcal{I}$ is a *self-implementation* if $T_1 = T_2 = \ldots = T_n = T$. The *resource complexity* of $\mathcal{I}$ is n, the number of base objects that make up a derived object of the implementation.

B Models of failure

Failure models for objects were explained in Section 3 using the informal terminology of Section 2. We present here the formal definitions of these failure models based on the formal model developed in Appendix A.

The failure models fall into two broad classes: *responsive* and *non-responsive*. As we will see, in most models of failure, an object $\mathcal{O}$ of type T that fails may return a response that is not in *RES*(T). When a process P gets such a response from $\mathcal{O}$, it knows 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.

B.1 Responsive models of failure

Responsive failure models share the following property: even an object that fails in an execution E, is wait-free in E.

B.1.1 R-crash

An object $\mathcal{O}$ *fails by R-crash in an execution* E of a concurrent system iff it fails in E, and the following hold in E:

1. $\mathcal{O}$ is wait-free.
2. Every response from $\mathcal{O}$ either belongs to *RES*(T) or is $\perp$ (where $\perp$ is a distinguished value not in *RES*(T), T being the type of $\mathcal{O}$).
3. If $op <_E op'$ and the response for op is $\perp$, then the response for op' is also $\perp$. This is the "once $\perp$, everafter $\perp$" property of R-crash.
4. Recall $B(\mathcal{O}, E)$, the behavior of $\mathcal{O}$ in E. Let B' be obtained by removing all operations[17] in $B(\mathcal{O}, E)$ whose responses are $\perp$. B' is linearizable with respect to the type of $\mathcal{O}$. This property captures the notion that an object failing by R-crash behaves correctly until it fails.

[17] Removing an operation involves removing the invocation *and* the response of that operation.

B.1.2 R-omission

An informal motivation for this model can be found in Section 3.1.2, and a formal justification in Section 7.

An object $\mathcal{O}$ *fails by R-omission in an execution* E of a concurrent system iff it fails in E, and the following hold in E:

1. $\mathcal{O}$ is wait-free.
2. Every response from $\mathcal{O}$ either belongs to $RES(T)$ or is $\perp$ (where $\perp$ is a distinguished value not in $RES(T)$, T being the type of $\mathcal{O}$).
3. Let B' be obtained from $B(\mathcal{O}, E)$ by removing all response events that get $\perp$. Then B' is linearizable with respect to the type of $\mathcal{O}$.

Property 3 captures the notion that a failed operation of P appears like an incomplete operation. Also notice the subtle difference in the way we obtain B' from $B(\mathcal{O}, E)$ for R-crash and for R-omission. We urge the reader to understand its implications on the failure semantics of the two models.

B.1.3 R-arbitrary

An object *fails by R-arbitrary in an execution* E of a concurrent system iff it fails in E, and is wait-free in E.

B.2 Non-responsive models of failure

Each responsive model of failure has its non-responsive counter-part. The difference is that, with non-responsive failures, an object that fails in an execution E may not be wait-free in E.

B.2.1 Crash

An object $\mathcal{O}$ *fails by crash in an execution* E of a concurrent system iff it fails in E, and the following hold in E:

1. $B(\mathcal{O}, E)$ is linearizable with respect to the type of $\mathcal{O}$.
2. The total number of responses from $\mathcal{O}$ in E is finite.

Property 2 captures the notion that an object that fails by crash does so at some finite point in the execution. Hence the number of times it will have responded in that execution must be finite.

B.2.2 Omission

An object $\mathcal{O}$ *fails by omission in an execution* E of a concurrent system iff it fails in E, and $B(\mathcal{O}, E)$ is linearizable with respect to the type of $\mathcal{O}$.

B.2.3 Arbitrary

An object $\mathcal{O}$ *fails by arbitrary in an execution* E of a concurrent system iff it fails in E.

C Definition of fault-tolerant implementations

An implementation $\mathcal{I}$ of type T for processes $P_1, P_2, \ldots, P_{N(T)}$ is *t-tolerant for failure model* $\mathcal{M}$ if every derived object $\mathcal{O}$ of $\mathcal{I}$ has the following property: In every execution of the clocked concurrent system $(P_1, P_2, \ldots, P_{N(T)}; \mathcal{O}; \mathcal{C})$, if at most t base objects of $\mathcal{O}$ fail, and they fail by $\mathcal{M}$, then $\mathcal{O}$ is correct.

An implementation $\mathcal{I}$ of type T for processes $P_1, P_2, \ldots, P_{N(T)}$ is *gracefully degrading for failure model* $\mathcal{M}$ if every derived object $\mathcal{O}$ of $\mathcal{I}$ has the following property: In every execution of the clocked concurrent system $(P_1, P_2, \ldots, P_{N(T)}; \mathcal{O}; \mathcal{C})$, if all base objects of $\mathcal{O}$ that fail, fail by $\mathcal{M}$, then either $\mathcal{O}$ is correct or it fails by $\mathcal{M}$.

D Type definitions

Recall that an object type T is defined (Section 2) as a tuple (N, OP, RES, G), where N is the number of processes supported by an object O of type T, OP is a set of operations supported by O, RES is a set of result values, and G is a graph giving the sequential specification of O. In this appendix, we specify OP, RES and G for most object types that occur in the paper. The parameter N is unspecified: each choice of N results in a different type. Similarly, in most cases, the initial state of G is not specified. A new type results for each choice of an initial state.

$OP = \{\texttt{compare\&swap}(v_1, v_2) | v_1, v_2 \text{ are booleans}\}$
$RES = \{0, 1\}$
Object State:
X, a boolean

```
compare&swap(v1, v2)
    if X = v1 then
        X := v2
    return(X)
```

Figure 8: **Compare&swap**

$OP = \{\texttt{reset}()\} \cup \{\texttt{propose}(v) | v \in \{0, 1\}\}$
$RES = \{0, 1, ack\}$
Object State:
$X \in \{0, 1, \perp\}$, initially $\perp$

```
propose(v)
    if X = ⊥ then
        X := v
    return(X)

reset()
    X := ⊥
    return(ack)
```

Figure 9: **Consensus-with-reset**

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

$\texttt{fetch\&add}(v)$
  $X := X + v$
  return(X)

Figure 10: **Fetch&add**

$OP = \{\mathbf{enq}(v) | v \text{ is integer}\} \cup \{\mathbf{deq}()\}$
$RES = \{v | \ v \text{ is integer}\} \cup \{nil, \ ack\}$
Object State:
  X, a sequence of integers

enq(v)
  $X := X \cdot v$
  return(ack)

deq()
  if X *is empty* then
    return(nil)
  else if $X = v \cdot X'$ then
    $X := X'$
    return(v)

Figure 11: **Queue**

$OP = \{\mathtt{read}(i), \mathtt{write}(v,i), \mathtt{move}(i) | v, i \in \{0,1\}\}$
$RES = \{0, 1, ack\}$
Object State:
    $X_0, X_1 \in \{0,1\}$

read(i)
    if $i = 0$ then
        return(X_0)
    else return(X_1)

write(v, i)
    if $i = 0$ then
        $X_0 := v$
    else $X_1 := v$
    return(ack)

move(i)
    $X_{\bar{i}} := X_i$
    return(ack)

Figure 12: **Move**

$OP = \{\mathtt{write}(v) | \ v \text{ is integer}\} \cup \{\mathtt{read}()\}$
$RES = \{v | \ v \text{ is integer}\} \cup \{ack\}$
Object State:
    X, an integer

read()
    return(X)

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

Figure 13: **(Unbounded) Register**

$OP = \{\text{push}(v) | v \text{ is integer}\} \cup \{\text{pop}()\}$
$RES = \{v | \ v \text{ is integer}\} \cup \{nil, \ ack\}$
Object State:
  X, a sequence of integers

push(v)
  $X := X \cdot v$
  return(ack)

pop()
  if X *is empty* then
    return(nil)
  else if $X = X' \cdot v$ then
    $X := X'$
    return(v)

Figure 14: `Stack`

$OP = \{\texttt{write}(v) | v \in \{0,1\}\} \cup \{\texttt{read}()\}$
$RES = \{0, 1, ack\}$
Object State:
  $X \in \{0, 1, \perp\}$, initially $\perp$

`read`()
  return(X)

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

Figure 15: `Sticky-bit`

$OP = \{\texttt{read}(i), \texttt{write}(v,i), \texttt{swap}() | v, i \in \{0,1\}\}$
$RES = \{0, 1, ack\}$
Object State:
    $X_0, X_1 \in \{0,1\}$

```
read(i)
    if i = 0 then
        return(X_0)
    else return(X_1)

write(v, i)
    if i = 0 then
        X_0 := v
    else X_1 := v
    return(ack)

swap()
    temp = X_0
    X_0 := X_1
    X_1 := temp
    return(ack)
```

Figure 16: **Swap**

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

```
test&set()
    y := X
    X := 0
    return(y)

reset()
    X := 1
    return(ack)
```

Figure 17: **Test&set**