Fault-Tolerant Wait-Free Shared Objects

PRASAD JAYANTI

Dartmouth College, Hanover, New Hampshire

TUSHAR DEEPAK CHANDRA

IBM T. J. Watson Research Center, Hawthorne, New York

AND

SAM TOUEG

Cornell University, Ithaca, New York

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

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

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

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

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; B.4.3

A preliminary version of this paper appeared in the *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science* (Oct.). IEEE, New York.

The research of P. Jayanti was partially supported by National Science Foundation (NSF) grant CCR 91-02231 and NSF RIA grant CCR 94-10421.

The research of S. Toueg was partially supported by NSF grants CCR 94-02896 and CCR 97-11403 and by ARPA/ONR grant N00014-96-1-1014.

Authors' addresses: P. Jayanti, 6211 Sudikoff Lab for Computer Science, Dartmouth College, Hanover, NH 03755; T. D. Chandra, IBM T. J. Watson Research Center, Hawthorne, NY 10532; S. Toueg, Department of Computer Science, Cornell University, Ithaca, NY 14853.

[**Input/Output and Data Communications**]: Interconnections (Subsystems)—*asynchronous/synchronous operations*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors* (*MIMD*); D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing/multiprogramming/multitasking, synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Asynchronous computing, fault-tolerance, implementation, MIMD, shared memory, shared objects, synchronization

1. *Introduction*

1.1. PROBLEM ADDRESSED. We consider concurrent systems in which asynchronous processes communicate via typed linearizable shared objects. In such systems, complex (shared) objects, such as queues and stacks, are implemented in software from simple objects, such as registers and test&sets, which are often supported in hardware. Traditional implementations (for example, Courtois et al. [1971]) use lock-based techniques and are consequently not fault-tolerant: if any process crashes while holding the lock, the other processes are effectively prevented from accessing the implemented object. Wait-free implementations, which have been the focus of much recent research, were introduced to overcome this drawback [Lamport 1977]. An implementation is *wait-free* if every access by a nonfaulty process is guaranteed a response, regardless of whether the other processes are slow, fast, or have crashed.

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

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

We divide the responsive class into three failure modes: *crash*, *omission*, and *arbitrary*. An object that fails by crash behaves correctly until it fails and, once it fails, it returns a distinguished response $\perp$ to every operation. Clearly, crash is the most benign failure mode. The most severe responsive failure mode is the arbitrary mode. Objects experiencing arbitrary failures may "lie", that is, they may return arbitrary responses. In terms of severity, omission falls between crash and arbitrary. When an object fails by omission, it returns normal responses to some operations and $\perp$ to others, and satisfies the following property: the object would seem non-faulty if every operation that obtained the response $\perp$ were treated like an incomplete operation that never obtained a response. Our study of omission failures is motivated by the fact that implementations tolerating such failures can be composed, but implementations tolerating the simpler crash failures cannot be (this is explained in Section 7).

Similarly, we divide the nonresponsive class into *NR-crash*, *NR-omission*, and *NR-arbitrary* failure modes. An object that fails by NR-crash behaves correctly until it fails and, once it fails, it stops responding. An object that fails by

NR-omission may fail to respond to the operations of an arbitrary subset of processes, but continue to respond to the operations of the remaining processes (forever). The behavior of an object that fails by NR-arbitrary is completely unrestricted: it may not respond to an operation and, even if it does, the response may be arbitrary.

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

Consider a t-tolerant implementation for failure mode $\mathcal{F}$. By definition, a derived object of this implementation is guaranteed to behave correctly even if up to t base objects fail by $\mathcal{F}$. But what happens if more than t base objects fail by $\mathcal{F}$? In general, the derived object may experience a more severe failure than $\mathcal{F}$. In other words, implementations may "amplify" failures: derived objects may fail more severely than base objects. This undesirable behavior is prevented by implementations that are "gracefully degrading". An implementation is *gracefully degrading for failure mode* $\mathcal{F}$ if it has the following property: if base objects only fail by $\mathcal{F}$, then the derived object does not fail more "severely" than $\mathcal{F}$. Thus, if $\mathcal{F}$ is guaranteed to be the most severe failure mode that hardware objects may experience, the graceful degradation property of an implementation makes it possible to extend the same guarantee to software objects.

We study the problem of designing t-tolerant and/or gracefully degrading implementations for the various responsive and non-responsive failure modes. An independent work by Afek et al. [1992; 1995] has the same general goal, but differs in many respects. We present a comparison of the two works in Section 8.

1.2. SUMMARY OF RESULTS. The three main topics studied are: tolerating responsive failures, tolerating nonresponsive failures, and achieving graceful degradation.

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

It is known that every type has an implementation from {`consensus`, `register`} [Herlihy 1988; 1991b; Plotkin 1989].[1] Hence, if the types `consensus` and `register` have t-tolerant implementations, so does every type. We therefore focus on obtaining t-tolerant implementations of `consensus` and `register`.

1.2.1. *Tolerating Responsive Failures*. We give t-tolerant self-implementations of `consensus` for crash, omission, and arbitrary failures. For crash and omission failures, our self-implementation is optimal requiring only $t + 1$ base consensus objects. For arbitrary failures, our self-implementation is efficient requiring $O(t \log t)$ base consensus objects. We also give t-tolerant self-implementations of

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

register for crash, omission, and arbitrary failures. Combining the above results with the universality results in Herlihy [1991b] and Plotkin [1989], we conclude that every type T has a t-tolerant implementation (from {consensus, register}) for *all* responsive failure modes. Moreover, if T implements both consensus and register, then T has a t-tolerant *self*-implementation. This implies that familiar types such as (2-process) fetch&add, queue, stack, test&set, and (N-process) compare&swap, move, memory-to-memory swap have t-tolerant self-implementations even for arbitrary failures.

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

We ask whether randomization can be used to circumvent these impossibility results. The answer is yes. Specifically, we show that register has a t-tolerant (deterministic) *self*-implementation even for NR-arbitrary failures. Furthermore, randomized implementations of consensus from register are well-known (for example, see Aspnes [1990]). These two results, together with the universality results in Herlihy [1991b] and Plotkin [1989] imply that every type has a *randomized* t-tolerant implementation from register even for NR-arbitrary failures.

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

We investigate the feasibility of designing implementations that are gracefully degrading for crash and omission failure modes. We show that there is a large class of types that have no gracefully degrading implementations for crash. This class includes many common types, such as queue, stack, test&set, and

[2] The impossibility of implementing a fault-tolerant consensus *object* from any finite set of base *objects*, one of which may fail by NR-crash, is shown using the impossibility of solving the consensus problem among a finite number of *processes*, one of which may crash [Fischer et al. 1985; Loui and Abu-Amara 1987; Dolev et al. 1987].

`compare&swap`. Intuitively, crash is so benign that it is impossible to ensure that the implemented object does not fail more severely than crash even when base objects fail only by crash. In contrast, for omission failures, we prove the following universality result: *Every* type has a t-tolerant gracefully degrading implementation from {`consensus`, `register`} for omission.

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

1.3. ORGANIZATION. In Section 2, we describe the model. In Section 3, we define the responsive and non-responsive classes of failures, and the failure modes within each class. We define the concepts of t-tolerant implementation and graceful degradation in Section 4. The three main topics—tolerating responsive failures, tolerating non-responsive failures, and the feasibility of graceful degradation for crash and omission failure modes—are studied in Sections 5, 6, and 7, respectively. In Section 8, we present a comparison with the results in Afek et al. [1992; 1995].

2. *Model*

2.1. I/O AUTOMATA. Our description of I/O automata is brief. The reader is referred to the work of Lynch and Tuttle [1988] for details.

An *I/O Automaton* is a nondeterministic automaton with the following components: (i) a finite/infinite set of states, including a distinguished set of starting states, (ii) a set of input events, (iii) a set of output events, (iv) a set of internal events, and (v) a transition relation given by a set of tuples (s, e, s'), where s and s' are states, and e is an event.

Each triple (s, e, s') in the transition relation is called a *step*, and it means that, if the automaton is in state s, event e can occur and change the state to s'. We say e is *enabled* in state s. An *execution* 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_0 is a starting state and (s_i, e_{i+1}, s_{i+1}) is a step of A. In the former case, s_n is the *final state* of the execution. A *history* of an automaton is the subsequence of events in an execution.

A new automaton can be constructed by composing a set of "compatible" automata. Let E be an execution of an automaton composed from $A_1, A_2, \ldots, A_k$ and H be the corresponding history. The *history of a component A_i in E*, denoted by $H|A_i$, is the subsequence of H consisting only of the events of A_i.

2.2. OBJECT TYPE. Every object has a type. The type specifies the expected behavior of the object. More precisely, a *type T* is a tuple (OP, RES, G, τ) where OP and RES are sets of operations and responses respectively, G is a directed

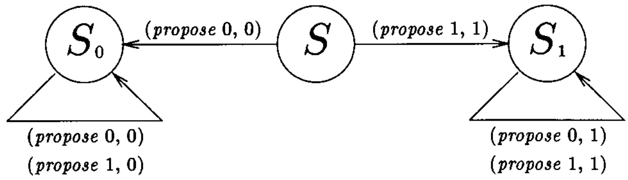

FIG. 1. Sequential specification of consensus.

finite or infinite multi-graph in which each edge has a label of the form (op, res) where $op \in OP$ and $res \in RES$, and τ is a history transformation function. We refer to G as the *sequential specification* of T and the vertices of G as the *states* of T. Intuitively, if there is an edge, labeled (op, res), from state s to state s', it means that applying the operation op to an object in state s may change the state to s' and return the response res. We explain the history transformation function τ later in Section 2.7.

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

The type consensus is central to this paper. Its sequential specification is in Figure 1.

2.3. OBJECTS AND PROCESSES. Objects and processes are modeled as automata. Each object O has two attributes: a type T and a state s of T to which O is initialized.

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

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

2.4. CONCURRENT SYSTEM. A *concurrent system* consisting of processes P_1, $P_2, \ldots, P_n$ and objects $O_1, \ldots, O_m$ is defined as the automaton composed from the process automata P_i, $1 \leq i \leq n$, and the object automata O_j, $1 \leq j \leq m$. We write $(P_1, P_2, \ldots, P_n; O_1, \ldots, O_m)$ to denote such a system.

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

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

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

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

2.5. Fairness. An execution E of a concurrent system is *fair* if the following conditions hold:

(1) If E is finite, then no internal or output event is enabled in the final state of E.
(2) If E is infinite, then for each internal or output event e, E contains either infinitely many occurrences of e or infinitely many states in which e is not enabled.

2.6. Linearizability. Linearizability requires that each operation, spanning over an interval of time from the invocation of the operation to its response, must appear to take effect at some instant in this interval [Herlihy and Wing 1990]. More precisely, let H be the history of some object in an execution of a

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

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

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

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

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

—Consider the type `safe register`, defined by Lamport [1986]. It supports read and write operations and has the same sequential specification as `register`: every read returns the value written by the most recent write. However, in the presence of concurrent operations, a safe register extends fewer guarantees than a (linearizable or "atomic") register. In particular, if a read operation on a safe register is concurrent with a write, then that read operation can return an arbitrary response. Thus, the history H of a safe register does not have to be linearizable. However, H satisfies the following weaker property [Lamport 1986]. If H' is the result of removing all read operations in H that are concurrent with a write, then H' is linearizable.

—Consider the type `consensus with safe-reset` [Herlihy 1991b]. It supports a *reset* operation in addition to *propose 0* and *propose 1*. Its sequential specification is the same as that of `consensus` (see Figure 1) with one addition: from any state, application of *reset* causes the state to change to S and return the response *ack*. In using an object of this type, if a reset operation is concurrent with a propose or another reset operation, then the object is allowed to return arbitrary responses to all operations thereafter. Thus, the history H of an object of type `consensus with safe-reset` does not have to be linearizable. However, H satisfies the following weaker property [Herlihy 1991b]: If H' is the maximal prefix of H in which no reset operation is concurrent with any other operation, then H' is linearizable.

—Consider the type `1-reader 1-writer register`. A history H of an object of this type does not have to be linearizable if either more than one process reads or more than one process writes. However, H satisfies the following weaker property: If H' is the maximal prefix of H in which no more than one process reads and no more than one process writes, then H' is linearizable.

—Consider the type `1-reader 1-writer safe register`. A history H of an object of this type satisfies the following property. Let H' be the maximal prefix of H in which no more than one process reads and no more than one process writes. Let H'' be the result of removing all read operations in H' that are concurrent with a write. Then, H'' is linearizable.

In all these examples, given a history H of an object of type T, we required that a transformation of H, not H itself, be linearizable with respect to T. This is the motivation for including a history transformation function τ as a component in the 4-tuple defining a type. We are now ready to define well-behavedness. Let O be an object of type $T = (OP, RES, G, \tau)$ which is initialized to state s of T. Let H be the history of O in an execution E of a concurrent system. We say that O is *well behaved in* E if $\tau(H)$ is linearizable with respect to (T, s).

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

2.8. Wait-Freedom and Correctness. Recall that every process automaton has a *FAIL* state. A *process* P *crashes in an execution* E of a concurrent system if P's state is $FAIL(P)$ in any state of E. P *is correct in* E if it does not crash in E. An *object* O *is wait-free in* E if the following condition holds: If E is fair, then every invocation on O by a correct process has a matching response. An *object* O *is correct in* E if O is wait-free and well-behaved in E. *Object* O *fails in* E if O is not correct in E.

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

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

concurrent system $(P_1, P_2, \cdots, P_N; \mathbb{O})$, if $O_1, O_2, \cdots$ are well-behaved in E, then $\mathbb{O}$ is well-behaved in E.

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

Condition (1) above states that a derived object is constituted by base objects and access procedures (front-ends). Condition (2) captures the notion that the execution of a step of the access procedure by one process P_i cannot affect the state of another process P_j. Condition (3) captures the notion that (i) invoking an operation on $\mathbb{O}$ by process P_i activates the front-end F_i or, equivalently, begins the execution of an access procedure, and (ii) the value returned by the front-end (access procedure) F_i is the response of $\mathbb{O}$. Condition (4) captures our intuition that when a process P_i crashes, the front end F_i of that process must stop executing. Condition (5) ensures that a derived object is well behaved whenever all its base objects are well behaved.

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

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

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

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

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

3. *Failure Modes*

A *failure mode* describes the manner in which a failed object departs from correct behavior. In this section, we define a spectrum of failure modes that fall into two broad classes: *responsive* and *nonresponsive.*

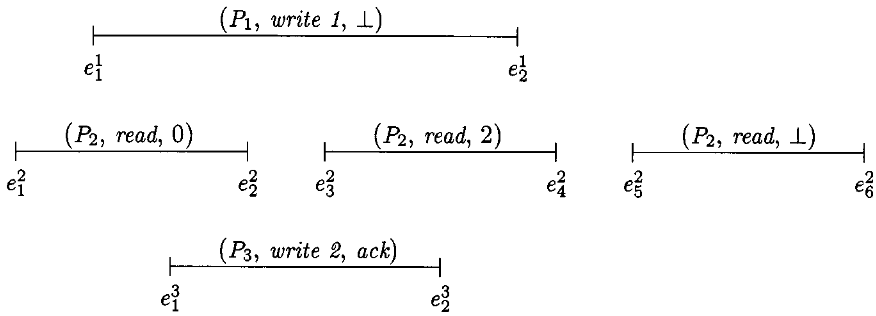

FIG. 2. History of a register, initialized to 0.

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

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

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

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

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

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

—Consider the history $\mathcal{H}$ of $\mathcal{R}$ in Figure 2. (In the figure, a line segment represents the duration of an operation, from invocation to response. A triple

(P_i, *op*, *res*) over the line segment denotes that P_i is the invoking process, *op* is the operation invoked, and *res* is the response from $\mathcal{R}$.) The failure of $\mathcal{R}$ is by crash, as verified below. Removing aborted operations in $\mathcal{H}$ results in $\mathcal{H}' = e_1^2$, e_1^3, e_2^2, e_3^2, e_2^3, e_4^2. (Event e_i^j denotes the ith event of process P_j.) Clearly, $\mathcal{H}'$ is linearizable with respect to (`register`, 0): e_1^2, e_2^2, e_1^3, e_2^3, e_3^2, e_4^2 is a linearization. The history transformation function τ for `register` is the identity function. Thus, $\tau(\mathcal{H}') = \mathcal{H}'$, and is linearizable with respect to (`register`, 0). Thus, Property (4) holds in $\mathcal{H}$. Other properties also hold and are trivial to verify.

—Let $\mathcal{G}$ be the same history as in Figure 2 with one modification: P_2's read returns 1 instead of 2. Clearly, $\mathcal{G}'$, obtained by removing the aborted operations in $\mathcal{G}$, does not result in a linearizable history. Thus, the failure of $\mathcal{R}$ in $\mathcal{G}$ is not by crash.

3.1.2. Omission. We begin with the motivation for the omission failure mode. Consider an implementation $\mathcal{I}$, and a derived object $\mathbb{O}$ of $\mathcal{I}$. Even if the base objects of $\mathbb{O}$ may only fail by crash, $\mathbb{O}$ itself may experience a more severe failure than crash. To see this, suppose that a base object o of $\mathbb{O}$ fails by crash. Consider a process P that invokes an operation *op* on $\mathbb{O}$ and executes `Apply`(P, *op*, $\mathbb{O}$). If, during the execution of `Apply`(P, *op*, $\mathbb{O}$), P accesses o, o returns $\perp$ to P. This may cause `Apply`(P, *op*, $\mathbb{O}$) to terminate and also return $\perp$. Strictly after this occurs, suppose that another process Q invokes some operation op' on $\mathbb{O}$, and that `Apply`(Q, op', $\mathbb{O}$) is not required to access o. Then, while executing `Apply`(Q, op', $\mathbb{O}$), Q does not notice the failure of o. So `Apply`(Q, op', $\mathbb{O}$) terminates "normally" and returns a non-$\perp$ response. Thus, $\mathbb{O}$'s behavior violates the "once $\perp$, everafter $\perp$" property: $\mathbb{O}$ returned $\perp$ to P's operation and a non-$\perp$ response to a strictly later operation by Q. We conclude that $\mathbb{O}$'s failure is more severe than crash. Does this mean that $\mathbb{O}$'s failure is arbitrary? We now argue that this is not the case.

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

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

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

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

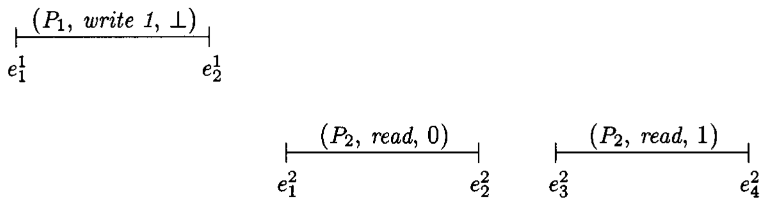

FIG. 3. History of a register, initialized to 0.

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

—Let $\mathcal{R}$ be an object of type `register`, initialized to 0. Consider the history $\mathcal{H}$ of $\mathcal{R}$ in Figure 3. The failure of $\mathcal{R}$ is by omission, as verified below. Removing the response event e_2^1 of the aborted operation results in $\mathcal{H}' = e_1^1, e_1^2, e_2^2, e_3^2, e_4^2$. $\mathcal{H}'$ (and hence, $\tau(\mathcal{H}')$) is linearizable with respect to (`register`, 0): $e_1^2, e_2^2, e_1^1, e, e_3^2, e_4^2$ is a linearization, where e is a response event returning ack. Thus, in the linearization of $\mathcal{H}'$, the first read by P_2 takes effect first, then the write by P_1 (which is aborted in $\mathcal{H}$ and incomplete in $\mathcal{H}'$) takes effect, and then the second read by P_2 takes effect. This example shows that an aborted operation may take effect a long time after it completed.

—Let $\mathcal{G}$ be the same history as in Figure 3 with one modification: the second read by P_2 returns 2 instead of 1. The failure of $\mathcal{R}$ in $\mathcal{G}$ is not by omission since the history $\mathcal{G}'$ obtained by removing e_2^1 is clearly not linearizable.

—Same as the previous example, but suppose that $\mathcal{R}$ is of type `safe register`. Recall that the function τ for `safe register` removes all read operations that overlap with a write. Thus, $\tau(\mathcal{H}') = e_1^1$, and is obviously linearizable with respect to (`safe register`, 0). (The empty sequence is a linearization of $\tau(\mathcal{H}')$.) Thus, Property (3) of omission holds. Other properties also hold and are trivial to verify. Thus, $\mathcal{R}$ fails by omission in $\mathcal{H}$.

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

3.2. NONRESPONSIVE FAILURE MODES. With responsive failure modes, a faulty object remains wait-free. Nonresponsive failure modes do not have this property.

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

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

(1) $\mathbb{O}$ is well behaved in E.
(2) The total number of responses from $\mathbb{O}$ in E is finite.

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

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

3.2.3. NR-ARBITRARY. An *object $\mathbb{O}$ fails in an execution E by NR-arbitrary* if it fails in E.

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

4. *Fault-tolerance and Graceful Degradation—Definitions and Properties*

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

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

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

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

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

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

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

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

LEMMA 4.1.1 (COMPOSITIONAL LEMMA). *Suppose that T has a t-tolerant implementation from $\mathcal{L}$ for failure mode $\mathcal{F}$, where $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ is a list of types. Furthermore, suppose that each T_i has a t_i-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$. Then we have*:

(1) *T has a t'-tolerant implementation from $\mathcal{L}'$ for failure mode $\mathcal{F}$, where $\mathcal{L}' = \mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \cdots \cdot \mathcal{L}_n$ and $t' = MinSum(t + 1, \langle t_1 + 1, t_2 + 1, \ldots, t_n + 1\rangle) - 1$.*

(2) *If the t-tolerant implementation of T from $\mathcal{L}$ is gracefully degrading for $\mathcal{F}$, then T has a t'-tolerant gracefully degrading implementation from $\mathcal{L}'$ for failure mode $\mathcal{F}$.*

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

Let $o_{11}, \ldots, o_{1j_1}, \ldots, o_{n1}, \ldots, o_{nj_n}$ be objects of types $T_{11}, \ldots, T_{1j_1}, \ldots, T_{n1}, \ldots, T_{nj_n}$, initialized to states $s_{11}, \ldots, s_{1j_1}, \ldots, s_{n1}, \ldots, s_{nj_n}$, respectively. Define an implementation $\mathcal{I}'$ as follows: $\mathcal{I}'(o_{11}, \ldots, o_{1j_1}, \ldots, o_{n1}, \ldots, o_{nj_n}) = \mathcal{I}(O_1, \ldots, O_n)$, where $O_i = \mathcal{I}_i(o_{i1}, o_{i2}, \ldots, o_{ij_i})$. Assume that each o_{kl}, if it fails, only fails by $\mathcal{F}$. Since $\mathcal{I}_i$ is t_i-tolerant, O_i fails only if at least $t_i + 1$ objects among $o_{i1}, \ldots, o_{ij_i}$ fail. Furthermore, since $\mathcal{I}_i$ is gracefully degrading, O_i can only fail by $\mathcal{F}$, no matter how many base objects of O_i fail. From this and the fact that $\mathcal{I}$ is t-tolerant for $\mathcal{F}$, it follows that $\mathcal{I}(O_1, \ldots, O_n)$ 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 $MinSum(t + 1, \langle t_1 + 1, t_2 + 1, \ldots, t_n + 1\rangle) = t' + 1$ 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, s) from $(\mathcal{L}', \Sigma')$, where $\Sigma' = \Sigma_1 \cdot \Sigma_2 \cdot \cdots \cdot \Sigma_n$. This completes the proof of the first part of the lemma.

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

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

COROLLARY 4.1.2. *Suppose that T has a* (0-*tolerant*) *implementation from* $(T_1, T_2, \ldots, T_n)$. *Furthermore, suppose that each T_i has a t-tolerant gracefully degrading implementation from $\mathcal{L}_i$ for failure mode $\mathcal{F}$, where $\mathcal{L}_i$ is some list of types. Then we have*:

(1) *T has a t-tolerant implementation from $\mathcal{L}_1 \cdot \mathcal{L}_2 \cdot \cdots \cdot \mathcal{L}_n$ for failure mode $\mathcal{F}$.*

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

The compositional lemma can be used to enhance the fault-tolerance of a self-implementation. This is the substance of the following corollary, obtained by setting $T_i = T$, $\mathcal{L}_i = \mathcal{L}$, and $t_i = t$ in Lemma 4.1.1. Below, we say that *T has an implementation of resource complexity n* if, for all states s of T, (T, s) has an implementation of resource complexity n.

COROLLARY 4.1.3. *If T has a t-tolerant gracefully degrading self-implementation $\mathcal{I}$ of resource complexity n for failure mode $\mathcal{F}$, then T has a $(t^2 + 2t)$-tolerant gracefully degrading self-implementation $\mathcal{I}'$ of resource complexity n^2 for $\mathcal{F}$.*

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

COROLLARY 4.1.4 (BOOSTER LEMMA). *If T has a 1-tolerant gracefully degrading self-implementation of resource complexity k for failure mode $\mathcal{F}$, then T has a t-tolerant gracefully degrading self-implementation of resource complexity $O(t^{\log_2 k})$ for $\mathcal{F}$.*

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

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

LEMMA 4.2.1. *If T has a t-tolerant k-bounded implementation from $\mathcal{L}$ for arbitrary failures, then T has a t-tolerant gracefully degrading k-bounded implementation from $\mathcal{L}$ for arbitrary failures.*

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

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

5. *Tolerating Responsive Failures*

In this section, we prove that it is feasible to design deterministic implementations that tolerate even the most severe of the responsive failures, viz., arbitrary failures.

Herlihy [1991b] and Plotkin [1989] showed that one can implement a (wait-free) object of any type using only consensus and register objects. Therefore, if `consensus` and `register` have t-tolerant implementations, then every type has a t-tolerant implementation. We therefore focus on fault-tolerant implementations of `consensus` and `register` in Sections 5.1 and 5.2, respectively. Combining these with the universal implementations of Herlihy and Plotkin, we obtain in Section 5.3 a deterministic *fault-tolerant* universal implementation that tolerates arbitrary failures.

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

We begin with a brief discussion of why the design of t-tolerant implementations of `consensus` is non-trivial. Achieving consensus among processes, some of which may fail, is a well-studied problem.[4] However, the existing solutions to this problem are for *synchronous message passing* systems in which a process can "wait" to hear from other correct processes. In contrast, we study the consensus problem for *asynchronous shared-memory* systems and we seek wait-free solutions. Because of these differences, the problem of t-tolerant implementation of `consensus` does not reduce to any previous problem considered in the literature.

The "State Machine" approach [Lamport 1978; Schneider 1990] of replicating objects, applying an operation to all objects, and returning the majority response is not useful in deriving t-tolerant implementations of `consensus`. For example, consider the following implementation which uses $2t + 1$ base consensus objects $(O_1, O_2, \ldots, O_{2t+1})$ to tolerate the crash failure of any t of them. A process p proposes a value v_p to the derived consensus object $\mathbb{O}$ by proposing v_p to each of $O_1, O_2, \ldots, O_{2t+1}$. At the end of this, p will have obtained the response 0 from, say, n_0 base objects, the response 1 from n_1 base objects, and the response $\perp$ from $2t + 1 - n_0 - n_1$ base objects. p returns 0 (as the response of $\mathbb{O}$) if $n_0 > n_1$. Otherwise, it returns 1. Unfortunately, as the following counterexample demonstrates, this implementation is not t-tolerant for crash. Let $t = 2$ and suppose that processes p and q wish to propose 0 and 1, respectively, to the derived consensus object $\mathbb{O}$. Consider the scenario in which events occur in the following order: p proposes 0 to O_1, O_2, and O_3; O_1 and O_2 fail by crash; q proposes 1 to all of $O_1, O_2, \ldots, O_5$; p resumes and proposes 0 to O_4 and O_5. In this scenario, p obtains three 0's (from O_1, O_2, and O_3), and two 1's (from O_4 and O_5). Process q obtains two $\perp$'s (from O_1 and O_2), one 0 from O_3, and two 1's (from O_4 and O_5). By the above implementation, p returns 0 and q returns 1. Thus, $\mathbb{O}$ does not satisfy the agreement property.

[4] See, for example, Pease et al. [1980], Lamport et al. [1982], Fischer et al. [1986], Coan [1987], Srikanth and Toueg [1987], Berman et al. [1989], Dolev et al. [1990], and Coan and Welch [1992].

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

5.1.1. *Properties of* `consensus`

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

—Object $\mathbb{O}$ satisfies *integrity* in E if and only if every response from $\mathbb{O}$ in E is either 0 or 1.

—Object $\mathbb{O}$ satisfies *weak integrity* in E if and only if every response from $\mathbb{O}$ in E is either 0, 1, or $\perp$.

—Object $\mathbb{O}$ satisfies *validity* in E if and only if the following holds in E. If there is a response of v from $\mathbb{O}$ and $v \in \{0, 1\}$, then there is an invocation of *propose* v on $\mathbb{O}$ preceding this response.

—Object $\mathbb{O}$ satisfies *agreement* in E if and only if the following holds in E. If $\mathbb{O}$ returns v_1, v_2 to two invocations, and $v_1, v_2 \in \{0, 1\}$, then $v_1 = v_2$. (By this definition, if $\mathbb{O}$ returns 0 to some processes and $\perp$ to all others, it still satisfies agreement.)

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

PROPOSITION 5.1.1.1. *Let $\mathbb{O}$ be an object of type* `consensus`, *initialized to the uncommitted state. Let E be an execution of* $(P_1, P_2, \ldots, P_N; \mathbb{O})$. *Object $\mathbb{O}$ is correct in E if and only if it is wait-free in E and satisfies integrity, validity, and agreement in E.*

PROPOSITION 5.1.1.2. *Let $\mathbb{O}$ be an object of type* `consensus`, *initialized to the uncommitted state. Let E be an execution of* $(P_1, P_2, \ldots, P_N; \mathbb{O})$ *in which $\mathbb{O}$ fails. Object $\mathbb{O}$ fails by omission in E if and only if it is wait-free in E and satisfies weak integrity, validity, and agreement in E.*

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

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

[5] Throughout this paper, we write `Propose` (with uppercase "P") if the operation is on a derived object, and `propose` (with lowercase "p") if it is on a base object.

```
O_1, O_2, ..., O_{t+1} : consensus objects, initialized to the uncommitted state

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
        w := propose(p, estimate_p, O_k)
        if w ≠ ⊥ then estimate_p := w
    return(estimate_p)
end
```

FIG. 4. t-tolerant self-implementation of consensus for omission.

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

Figure 4 presents a t-tolerant self-implementation of consensus for omission failures. (In all our algorithms, we use indentation to convey the scope of an **if** statement or a **for** statement.) This implementation uses $t + 1$ base objects. A process p proposes to the derived object $\mathbb{O}$ by accessing each of $O_1, O_2, \ldots, O_{t+1}$, in that order. At any point in the algorithm, p holds an estimate of the eventual return value in $estimate_p$. When p proposes its current estimate to a base object O_k, if O_k returns a non-$\perp$ response w different from p's current estimate, p changes its estimate to w. After accessing all $t + 1$ base objects, p returns its estimate as the response of the derived object $\mathbb{O}$.

THEOREM 5.1.2.1. *Figure* 4 *presents a t-tolerant self-implementation of* consensus *for omission failures*.[6] *The resource complexity of the implementation is t +* 1 *and is optimal*.

PROOF. Let $\mathbb{O}$ be a derived 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 omission, and the remaining objects are correct. We show that $\mathbb{O}$ is correct in E.

(1) $\mathbb{O}$ *satisfies validity*. An easy induction on k, the variable in Figure 4, 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, $\mathbb{O}$). The induction will use Proposition 5.2, and the fact that p does not change $estimate_p$ if a base object returns $\perp$.

(2) $\mathbb{O}$ *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 kth iteration of the loop. Since each base object in $O_{k+1}, \ldots, O_{t+1}$ is either correct or fails by omission in E, by Propositions 5.1.1.1 and 5.1.1.2, each of these base objects satisfies validity.

[6] Recall our convention that, if we do not mention the number of processes for which an implementation is intended, then the implementation is for N processes, where N is arbitrary.

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, $\mathbb{O}$ returns the same response w to every p.

(3) $\mathbb{O}$ *satisfies integrity*. Obvious.

Since a base object that fails by omission remains wait-free, it is clear that $\mathbb{O}$ is wait-free in E. By Proposition 5.1.1.1, $\mathbb{O}$ is correct in E. It is obvious that the resource complexity of $t + 1$ of our self-implementation is optimal. □

We remark that the above implementation is *not* gracefully degrading. To see this, suppose that $v_p = 0$ and $v_q = 1$, and all the $t + 1$ base objects fail by crash initially. It is easy to see that $\mathbb{O}$ returns 0 to p and 1 to q. Thus, $\mathbb{O}$ does not satisfy agreement and, by Proposition 5.1.1.2, the failure of $\mathbb{O}$ is more severe than omission. However, there is a t-tolerant self-implementation of `consensus` that is also gracefully degrading (for omission). This implementation uses $2t + 1$ base objects. In fact, $2t + 1$ is a lower bound on the resource complexity of any t-tolerant gracefully degrading implementation of `consensus` for omission. (The implementation and the lower bound can be found in Jayanti et al. [1996].) In contrast to omission, as we will prove later in Section 7, `consensus` has no t-tolerant gracefully degrading implementation for crash.

5.1.3. *Tolerating Arbitrary Failures*. In this section, we present a t-tolerant self-implementation for arbitrary failures whose resource complexity is $O(t \log t)$. This self-implementation, described in Figure 5, uses the divide-and-conquer strategy: it implements a t-tolerant consensus object $\mathbb{O}$ from O_1, a $\lceil (t - 1)/2 \rceil$-tolerant consensus object, O_2, a $\lfloor (t - 1)/2 \rfloor$-tolerant consensus object, and $10t + 3$ (0-tolerant) consensus objects—$A_0[1 \cdots 3t + 1]$, $A_1[1 \cdots 3t + 1]$, and $B[1 \cdots 4t + 1]$. Since a (base) consensus object that experiences an arbitrary failure may return nonbinary responses, we always "filter" responses to force them to be binary: procedure `f-propose`(p, v, O) returns `propose`(p, v, O) if it is 0 or 1, and returns 0 otherwise.

Figure 6 illustrates the order in which the base objects of $\mathbb{O}$ are accessed by a process proposing 0 on $\mathbb{O}$ (the access pattern for a process proposing 1 on $\mathbb{O}$ is symmetric). Before presenting a formal correctness proof, we provide some intuition for the implementation.

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

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

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

```
A0[1...3t+1], A1[1...3t+1], B[1...4t+1] : (0-tolerant) consensus objects,
        initialized to the uncommitted state
O1 : ⌈(t-1)/2⌉-tolerant consensus objects, initialized to the uncommitted state
O2 : ⌊(t-1)/2⌋-tolerant consensus objects, initialized to the uncommitted state

    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
                     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, O1)

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

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

        Phase 5: Choose belief_p such that WitnessCount_p[belief_p] > WitnessCount_p[\bar{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, O2)
                 return(ans2_p)
    end
```

FIG. 5. Efficient t-tolerant self-implementation of consensus for arbitrary failures.

second and third cases are detected with the help of $A_0[1 \cdots 3t + 1]$, $A_1[1 \cdots 3t + 1]$, and $B[1 \cdots 4t + 1]$.

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

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

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

If O_1 is correct and O_1 returns 0, then some process q proposed 0 to O_1 before any process got a response from O_1. It follows from our implementation that (i) process q had proposed 0 to each of $A_0[1 \cdots 3t + 1]$ before it proposed 0 to O_1, and (ii) no process proposed 1 to any of $A_0[1 \cdots 3t + 1]$ before q proposed 0 to O_1. Thus, when p accesses the objects $A_0[1 \cdots 3t +$

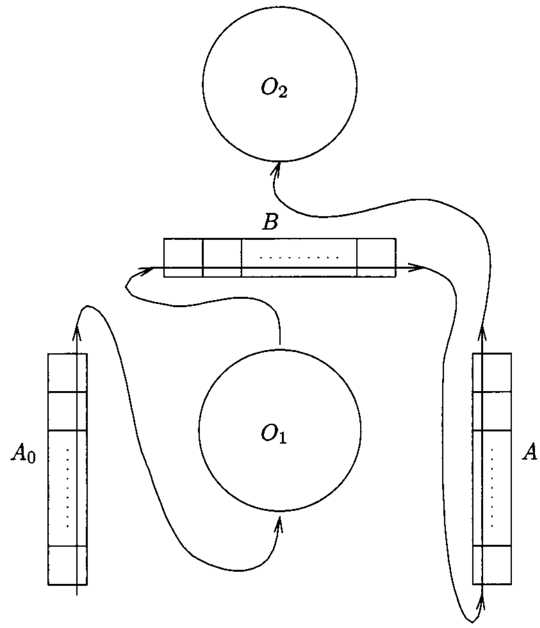

FIG. 6. Execution trace of a process proposing 0 on $\mathbb{O}$.

1], every correct object in $A_0[1 \cdots 3t + 1]$ returns 0. Since at least $2t + 1$ of the objects in $A_0[1 \cdots 3t + 1]$ are correct, we have $count_p[0] \geq 2t + 1$.

P2. If O_1 is correct and O_1 returns v, then, for all processes p, $WitnessCount_p[v] \geq 3t + 1$.

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

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

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

THEOREM 5.1.3.1. *Figure 5 presents a t-tolerant gracefully degrading self-implementation of* consensus *for arbitrary failures of resource complexity $O(t \log t)$.*

PROOF. Since the implementation is bounded, by Lemma 4.2.1, it is gracefully degrading for arbitrary failures. We now prove that the implementation is t-tolerant.

Consider an execution E in which at most t base objects fail by the arbitrary failure mode, and the remaining are correct. We show below, through a series of lemmas, that $\mathbb{O}$ is correct in E; or equivalently (by Proposition 5.1.1.1), that $\mathbb{O}$

satisfies validity, agreement, and integrity, and is wait-free in E. Proposition 5.1.1.1 is used very often in this proof. For brevity, we omit references to it.

LEMMA 5.1.3.2. *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 $\lceil (t-1)/2 \rceil$-tolerant implementation, at least $\lceil (t-1)/2 \rceil + 1$ base objects of O_1 must fail in E. Similarly, at least $\lfloor (t-1)/2 \rfloor + 1$ base objects of O_2 must fail in E. Thus, a total of $\lceil (t-1)/2 \rceil + \lfloor (t-1)/2 \rfloor + 2 > t$ base objects of $\mathbb{O}$ fail in E, a contradiction to the definition of E. □

LEMMA 5.1.3.3. *If O_1 is correct in E, $\mathbb{O}$ 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 $B[1 \cdots 4t + 1]$ 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 $A_v[1 \cdots 3t + 1]$ 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. □

LEMMA 5.1.3.4. *If O_1 fails in E, $\mathbb{O}$ satisfies validity and agreement in E.*

PROOF. Suppose O_1 fails. Then, by Lemma 5.1.3.2, O_2 is correct, and thus satisfies validity and agreement. We need to consider two cases.

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

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

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

By the validity property of O_2, some process p must have proposed ans to O_2. That is $v'_p = ans$. In the algorithm, v'_p equals either v_p or $belief_p$. If $v'_p = v_p$, then clearly $\mathbb{O}$ 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 $A_{belief_p}[1 \cdots$

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

FIG. 7. Reset procedure of the t-tolerant self-implementation of `consensus with safe-reset` for arbitrary failures.

$3t + 1]$ fail, some process q proposed $v_q = belief_p$ on some object in $A_{belief_p}[1 \cdots 3t + 1]$. Thus, process q proposed v_q on $\mathbb{O}$. Thus, $\mathbb{O}$ satisfies validity. □

LEMMA 5.1.3.5. *The resource complexity of the implementation in Figure 5 is $O(t \log t)$.*

PROOF. Denoting the resource complexity of the t-tolerant self-implementation of `consensus` for arbitrary failures by $f(t)$, we have the following recurrence: $f(t) = 2f(t/2) + 2(3t + 1) + (4t + 1)$. Furthermore, $f(1) = 15$ since the implementation in Figure 5 requires fifteen consensus objects to build a 1-tolerant consensus object.[7] The lemma follows from solving this recurrence. □

It is obvious that $\mathbb{O}$ satisfies integrity and is wait-free in E. By Lemmas 5.1.3.3 and 5.1.3.4, $\mathbb{O}$ satisfies validity and agreement in E. Thus, by Proposition 5.1.1.1, $\mathbb{O}$ is correct in E. This completes the proof of Theorem 5.1.3.1. □

As we will see later, to obtain fault-tolerant implementations of generic types, it is useful to have a fault-tolerant implementation of `consensus with safe-reset`, not just of `consensus`. Let us first recall the type `consensus with safe-reset`. Its sequential specification and its history transformation function are described in Section 2.7. Intuitively, an object of this type is like a consensus object, but it also supports the reset operation. Applying reset causes the object to move to the uncommitted state. Thus, the object can be used for multiple rounds of consensus by resetting it between rounds. However, the reset operation is guaranteed to work only if it is executed in "isolation": that is, if it is not concurrent with another reset operation or a propose operation. Otherwise the object may return arbitrary responses.

Figures 5 and 7, with the following modifications, present a t-tolerant gracefully degrading self-implementation of `consensus with safe-reset`. In Figure 5, assume that objects $A_0[1 \cdots 3t + 1]$, $A_1[1 \cdots 3t + 1]$, and $B[1 \cdots 4t + 1]$ are no longer just consensus objects, but are consensus-with-safe-reset objects, initialized to the uncommitted state. Also, assume that O_1 and O_2 are $\lceil (t - 1)/2 \rceil$-tolerant and $\lfloor (t - 1)/2 \rfloor$-tolerant consensus-with-safe-reset objects, initialized to the uncommitted state.

[7] See Jayanti [1996] for a 1-tolerant self-implementation of resource complexity 6.

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

```
Apply(P_r, read, R)
    val, i : integers, local to P_r
    S : multi-set of integers, local to P_r
begin
    S := ∅
    for i := 1 to 2t + 1
        val := apply(P_r, read, R_i)
        S := S ∪ {val}
    return mode(S)
end
```

```
Apply(P_w, write v, R)
    i : integer, local to P_w

begin
    for i := 1 to 2t + 1
        apply(P_w, write v, R_i)
    return ack
end
```

FIG. 8. t-tolerant self-implementation of `1-reader 1-writer safe register` for arbitrary failures.

THEOREM 5.1.3.6. *Figures* 5 *and* 7 *present a t-tolerant gracefully degrading self-implementation of* `consensus with safe-reset` *for arbitrary failures.*

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

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

First, we present a t-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register` in Figure 8.[8] The implementation uses $2t + 1$ base registers. To read the derived register, the reader process P_r reads all $2t + 1$ base registers and collects their responses in S. It then returns *mode*(S), a value that occurs at least as many times in S as any other value. To write a value v into the derived register, the writer process P_w simply writes v to all $2t + 1$ base registers.

LEMMA 5.2.1. *Figure* 8 *presents a t-tolerant gracefully degrading self-implementation of* `1-reader 1-writer safe register` *for arbitrary failures.*

[8] Recall that this type has the same sequential specification as `register`, but has a different history transformation function, as explained in Section 2.7. Intuitively, if a read operation on an object of this type overlaps with a write, then that read operation is allowed to return any value [Lamport 1986]. Furthermore, the object's behavior is unrestricted if either more than one process invokes read operations or more than one process invokes write operations.

PROOF SKETCH. Since the implementation is bounded, by Lemma 4.2.1, it is gracefully degrading for arbitrary failures. We now prove that the implementation is t-tolerant.

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

There are many results presenting bounded implementations of one type of register from another.[9] Some of them (e.g., Lamport [1986], Singh et al. [1987], and Schaffer [1988]), can be combined to implement a multi-reader, multi-writer, linearizable register using 1-reader, 1-writer, safe registers. In our terminology, this means that `register` has a bounded implementation from `1-reader 1-writer safe register`. This implies, by Lemma 4.2.1, that `register` has a 0-tolerant gracefully degrading implementation from `1-reader 1-writer safe register` for arbitrary failures. Using this result and Lemma 5.2.1, and applying Corollary 4.1.2, we conclude that `register` has a t-tolerant gracefully degrading implementation from `1-reader 1-writer safe register` for arbitrary failures. This trivially implies the following theorem.

THEOREM 5.2.2. `register` has a t-tolerant gracefully degrading self-implementation for arbitrary failures.

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

The theorems of this section depend on the universality results due to Herlihy [1991b] and Plotkin [1989]. These results are stated below.

THEOREM 5.3.1 (HERLIHY). *For all types T, there is a k such that T has a* (0-*tolerant*) *k-bounded implementation from* {`consensus with safe-reset`, `unbounded register`}.

Herlihy's universal construction requires unbounded registers even to implement finite types. Plotkin's construction, on the other hand, requires only boolean registers in such a situation [Plotkin 1989]. (Jayanti and Toueg [1992] achieve the same result as Plotkin, but with a more intuitive construction.)

[9] See, for example, Peterson [1983], Lamport [1986], Vitanyi and Awerbuch [1986], Bloom [1987], Burns Peterson [1987], Newman-Wolfe [1987], Peterson and Burns [1987], Singh et al. [1987], Schaffer [1988], Vidyasankar [1988; 1989], and Haldar and Vidyasankar [1991].

THEOREM 5.3.2 (PLOTKIN). *For all finite types T, there is a k such that T has a (0-tolerant) k-bounded implementation from* {`consensus with safe-reset`, `boolean register`}.

From Plotkin's theorem and Lemma 4.2.1, it follows that every finite type has a (0-tolerant) gracefully-degrading implementation from {`consensus with safe-reset`, `boolean register`} for arbitrary failures. Using this, together with Theorems 5.1.3.6, 5.2.2, and Lemma 4.1.1, we obtain:

COROLLARY 5.3.3. *Let T be any finite type.*

—*T has a t-tolerant gracefully degrading implementation from* {`consensus with safe-reset`, `boolean register`} *for arbitrary failures.*

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

From Theorem 5.3.1 and Lemma 4.2.1, it follows that every type has a (0-tolerant) gracefully-degrading implementation from {`consensus with safe-reset`, `unbounded register`} for arbitrary failures. Using this, together with Theorems 5.1.3.6, 5.2.2, and Lemma 4.1.1, we obtain:

COROLLARY 5.3.4. *Let T be any type.*

—*T has a t-tolerant gracefully degrading implementation from* {`consensus with safe-reset`, `unbounded register`} *for arbitrary failures.*

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

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

It is known that `fetch&add`, `queue`, `stack`, and `test&set` implement `consensus with safe-reset` for two processes, and that `compare&swap`, `memory-to-memory move` (henceforth, `move`), and `memory-to-memory swap` (henceforth, `m-m swap`) implement `consensus with safe-reset` for any number of processes [Herlihy 1991b; Kleinberg and Mullainathan 1993].[10] These are all bounded implementations and, by Lemma 4.2.1, are gracefully degrading for arbitrary failures.

We claim that `compare&swap`, `move`, `m-m swap`, and `test&set` implement `1-reader 1-writer boolean safe register`, and that `fetch&add`, `queue`, and `stack` implement `1-reader 1-writer unbounded safe register`. The implementations claimed above are bounded and are easy to obtain. We therefore omit their descriptions. (See Jayanti et al. [1996] for an implementation of `1-reader 1-writer boolean safe register` from `test&set`.)

[10] An object of type `move` consists of a pair of cells and supports operations with which either cell can be read or written, or the contents of one cell copied to the other. An object of type `m-m swap` is similar, but instead of a *move* operation, it supports a *swap* operation that swaps the contents of the two cells.

As already mentioned, it is known that `register` has a bounded implementation from `1-reader 1-writer safe register`. From these results, we conclude that `boolean register` has a bounded implementation from each of `compare&swap`, `move`, `m-m swap`, and `test&set`, and that `unbounded register` has a bounded implementation from each of `fetch&add`, `queue`, and `stack`. By Lemma 4.2.1, these implementations are gracefully degrading for arbitrary failures.

From the above, we have

COROLLARY 5.3.4. `compare&swap`, `move`, *and* `m-m swap` *have t-tolerant self-implementations for arbitrary failures.*

COROLLARY 5.3.5. `queue`, `stack`, `test&set`, *and* `fetch&add` *have t-tolerant self-implementations for arbitrary failures. These implementations are for two processes.*

6. *Tolerating Nonresponsive Failures*

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

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

6.1. IMPOSSIBILITY OF FAULT-TOLERANT IMPLEMENTATION OF CONSENSUS.

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

In each case, to prove that a certain implementation $\mathcal{I}$ does not exist, we show that if $\mathcal{I}$ exists, it would violate the well-known impossibility result due to Loui and Abu-Amara [1987] and Dolev et al. [1987]. This result is about the *consensus problem for n processes*, defined informally as follows. Each process P_i is initially given an input $v_i \in \{0, 1\}$. Each correct process P_i must eventually decide a value d_i such that (i) $d_i \in \{v_1, v_2, \cdots, v_n\}$, and (ii) for all processes P_i and P_j that decide, $d_i = d_j$.

THEOREM 6.1.1 (LOUI AND ABU-AMARA, DOLEV, DWORK, AND STOCKMEYER). *The consensus problem for n processes has no solution if processes may communicate only via registers and at most one process may crash.*

THEOREM 6.1.2. *There is no* 1*-tolerant implementation of* `consensus`, *even for two processes*, *for NR-crash*.

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

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

(1) Let $O_1, O_2, \ldots, O_l$ be objects of type $T_1, T_2, \ldots, T_l$, initialized to states $s_1, s_2, \ldots, s_l$, respectively. Let $\mathbb{O} = \mathcal{I}(O_1, \ldots, O_l)$. Thus, $\mathbb{O}$ is a consensus object that can be shared by two processes. Moreover, by definition of $\mathcal{I}$, $\mathbb{O}$ remains correct even if one of its base objects fails by NR-crash.

(2) In system S, process q_j $(1 \le j \le l)$ simulates the base object O_j, and process p_i $(i = 1, 2)$ simulates the execution of `Propose`$(p_i, v_i, \mathbb{O})$ on the derived object $\mathbb{O}$.

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

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

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

simulates the application of *op* to O_j (using the sequential specification of the type T_j) and appends the corresponding response to *response*(j, i). In addition to (and concurrently with) the above, q_j periodically checks if the register *decision* contains a non-$\perp$ value. If so, it decides that value.

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

We showed that we can use $\mathcal{I}$ to solve the consensus problem in system S. This contradicts Theorem 6.1.1. Thus, $\mathcal{I}$ cannot exist. □

We can strengthen the above result as follows: Suppose that *at most one* base object may fail and that it can only do so by being "unfair" (i.e., by not responding) to *at most one* process. Furthermore, suppose that the identity of this process is a priori "common knowledge" among all the processes. Even with this extremely weak failure mode, called *unfairness to a known process*, we can prove the following (the proof can be found in Jayanti et al. [1996]):

THEOREM 6.1.3. *There is no* 1-*tolerant implementation of* `consensus`, even for two processes, for unfairness to a known process.

From the above two theorems we have:

COROLLARY 6.1.4. *If a type T implements* `consensus` *for two processes*, *then T has no* 1-*tolerant implementation*, *for two processes*, *for NR-crash or for unfairness to a known process*.

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

COROLLARY 6.1.5. *None of the following types has a* 1-*tolerant implementation*, *for two processes*, *for NR-crash or for unfairness to a known process*: `compare&swap`, `fetch&add`, `move`, `queue`, `stack`, `sticky-bit`, `m-m swap`, *and* `test&set`.

$R_1, R_2, \cdots, R_{5t+1}$: 1-reader 1-writer safe registers, initialized to the initial value of the derived register

$Pending_r$: set, local to the reader process P_r, initialized to $\emptyset$
$Pending_w$: set, local to the writer process P_w, initialized to $\emptyset$

```
Apply(P_r, read, R)
  Invoked_r: set, local to P_r
  Responses_r: multi-set, local to P_r
  val, i : integers, local to P_r
begin
  Invoked_r := ∅
  Responses_r := ∅
  i := 0
  Loop
    i := (i mod 5t + 1) + 1
    if R_i ∈ Pending_r then
      Check if R_i responded
      if (yes) then
        Pending_r := Pending_r − {R_i}
        Let val be the response
        if R_i ∈ Invoked_r then
          Responses_r := Responses_r ∪ {val}
    if (R_i ∉ Pending_r) ∧ (R_i ∉ Invoked_r) then
      Invoke read on R_i
      Invoked_r := Invoked_r ∪ {R_i}
      Pending_r := Pending_r ∪ {R_i}
  Until |Responses_r| = 4t + 1
  return mode(Responses_r)
end
```

```
Apply(P_w, write v, R)
  Invoked_w: set, local to P_w
  Responses_w: multi-set, local to P_w
  val, i : integers, local to P_w
begin
  Invoked_w := ∅
  Responses_w := ∅
  i := 0
  Loop
    i := (i mod 5t + 1) + 1
    if R_i ∈ Pending_w then
      Check if R_i responded
      if (yes) then
        Pending_w := Pending_w − {R_i}
        Let val be the response
        if R_i ∈ Invoked_w then
          Responses_w := Responses_w ∪ {val}
    if (R_i ∉ Pending_w) ∧ (R_i ∉ Invoked_w) then
      Invoke write v on R_i
      Invoked_w := Invoked_w ∪ {R_i}
      Pending_w := Pending_w ∪ {R_i}
  Until |Responses_w| = 4t + 1
  return ack
end
```

FIG. 9. t-tolerant self-implementation of `1-reader 1-writer safe register` for NR-arbitrary failures.

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

First, we present a t-tolerant self-implementation of `1-reader 1-writer safe register` in Figure 9. The implementation uses $5t + 1$ base registers. To read the derived register, the reader process P_r invokes *read* on each base register (P_r delays this read if its previous read on the base register is still incomplete). When P_r gets responses from $4t + 1$ base registers, which are collected in the multi-set *Responses*, it returns *mode*(*Responses*). (Recall that *mode*(S) is a value that occurs at least as many times in S as any other value.) To write a value v into the derived register, the writer process P_w invokes *write* v on each base register (again, the writer delays invoking this write if its previous write on the base register is still incomplete). The writing of the derived register completes when the writer receives the response *ack* from $4t + 1$ base registers.

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

LEMMA 6.2.1. *Figure* 9 *presents a t-tolerant self-implementation of* `1-reader 1-writer safe register` *for NR-arbitrary failures*.

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

As mentioned in Section 5.2, it is known that `register` has an implementation from `1-reader 1-writer safe register`. Using this result and Lemma 6.2.1, and applying Corollary 4.1.2,[11] we conclude that `register` has a t-tolerant implementation from `1-reader 1-writer safe register` for NR-arbitrary failures. This implies the following theorem.

THEOREM 6.2.2. `register` has a t-tolerant self-implementation for NR-arbitrary failures.

6.3. RANDOMIZED FAULT-TOLERANT IMPLEMENTATIONS OF GENERIC TYPES. So far, we assumed that processes are deterministic. Suppose instead that processes have access to "fair coins". A process can toss a coin and, based on the outcome of the toss, choose its step. Furthermore, let us informally define a *randomized implementation* as an implementation in which every correct process completes its operation on the derived object in a finite expected number of operations on the base objects. Interestingly, every type has a randomized implementation from `register` [Herlihy 1991a], but most types have no (deterministic) implementations from `register` [Herlihy 1991b]. In the following, we present a generalization of the former result.

`consensus with safe-reset` has a randomized implementation from `register` [Aspnes 1990]. Together with Theorem 6.2.2, this implies that `consensus with safe-reset` has a t-tolerant randomized implementation from `register` for NR-arbitrary failures. Combining this with Theorem 6.2.2, and Theorems 5.5 and 5.6 of Herlihy and Plotkin, we have

[11] Observe that every implementation is automatically gracefully degrading for NR-arbitrary failures. Thus, we are able to apply Corollary 4.1.2.

THEOREM 6.3.1. *Every finite type has a t-tolerant randomized implementation from* `boolean register` *for NR-arbitrary failures. Every infinite type has a t-tolerant randomized implementation from* `unbounded register` *for NR-arbitrary failures.*

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

7. *Graceful Degradation for Benign Failure Modes*

Graceful degradation is a desirable property of implementations: it ensures that an implemented object never fails more severely than any of its components. Furthermore, if fault-tolerant implementations are gracefully degrading, then they can be composed (Lemma 4.1.1) and their degree of fault-tolerance can be automatically boosted (Corollary 4.1.4). In this section, we investigate the feasibility of achieving graceful degradation for the benign crash and omission failure modes. We identify a class of "order sensitive" types that includes many common types such as `queue`, `stack`, `test&set`, and `compare&swap`, and prove that no type in this class has a fault-tolerant gracefully degrading implementation for crash. In contrast, we show that graceful degradation for omission is achievable in a strong sense: For omission, every type has a t-tolerant gracefully degrading implementation from every universal set of types. (A set S of types is *universal* if every type has an implementation from S.) Thus, the message of this section is that gracefully degrading implementations are feasible for omission failures, but not for crash failures.

7.1. GRACEFUL DEGRADATION FOR CRASH.

In this section, we identify a class of "order-sensitive" types and present two negative results with respect to achieving gracefully degrading implementations of these types for crash.

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

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

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

We now present two theorems for crash and discuss their implications before the proofs.

THEOREM 7.1.1. *Let T be any order-sensitive type and $\mathcal{S}$ be any set of non-order-sensitive types. T has no gracefully degrading implementation from $\mathcal{S}$ for crash.*

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

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

THEOREM 7.1.2. *There is no* 1*-tolerant gracefully degrading implementation of any order-sensitive type for crash.*

The above two theorems raise serious concerns about the "practicality" of the crash mode: even if "hardware" objects are designed to fail only by crash, "software" objects usually don't. The omission mode does not have this severe limitation. In fact, we show in the next subsection that, for any $t \geq 0$, *every* type has a t-tolerant *gracefully degrading* implementation from every universal set of types for omission. In other words, implementations preserving the omission failure semantics of the underlying system always exist. This is a formal justification for adopting the omission failure mode.

We remark that there are no obvious ways to strengthen Theorem 7.1.2. For instance, consider the statement "There is no 1-tolerant gracefully degrading implementation of any type in *CONS2* for crash".[12] This statement is false. In fact, even the weaker version "There is no 1-tolerant gracefully degrading implementation of any type in *CONS2* from any set of non-order-sensitive types for crash" does not hold: We can show that `sticky-bit` has a t-tolerant gracefully degrading implementation from {`sticky-bit`, `register`} for crash.

Since `sticky-bit` belongs to *CONS2*, and both `sticky-bit` and `register` are non-order-sensitive, such an implementation is a counter-example to the above statement. The details of this implementation are long and tedious, and are therefore omitted.

We now prove Theorem 7.1.1. The proof of Theorem 7.1.2 is similar and can be found in Jayanti et al. [1996].

[12] This statement is stronger than Theorem 7.1.2 since, as remarked earlier, the set of order-sensitive types is a proper subset of *CONS2*.

PROOF OF THEOREM 7.1.1. Suppose that the theorem is false. Then, there is an order-sensitive type T which has a gracefully degrading implementation from some set of non-order-sensitive types for crash. For type T, let op, op', s, u, v, u', v' be as in the definition of an order-sensitive type. It follows that there is a list $\mathcal{L} = (T_1, T_2, \ldots, T_n)$ of nonorder-sensitive types and a list $\Sigma = (s_1, s_2, \ldots, s_n)$ of states (s_i is a state of T_i) such that (T, s) has a gracefully degrading implementation $\mathcal{I}$ from $(\mathcal{L}, \Sigma)$ for crash. We arrive at a contradiction after a series of lemmas involving bivalency arguments [Fischer et al. 1985] and indistinguishable scenarios.

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

LEMMA 7.1.3. *Suppose all base objects are correct. For any interleaving of the steps in the complete executions of* `Apply`(p, op, $\mathbb{O}$) *and* `Apply`(q, op', $\mathbb{O}$), *either* `Apply`(p, op, $\mathbb{O}$) *returns u and* `Apply`(q, op', $\mathbb{O}$) *returns u', or* `Apply`(p, op, $\mathbb{O}$) *returns v and* `Apply`(q, op', $\mathbb{O}$) *returns v'.*

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

Let C denote a configuration reached from C_0 after some interleaving of (partial) executions of `Apply`(p, op, $\mathbb{O}$) and `Apply`(q, op', $\mathbb{O}$). We say C is *X-valent* if, in the absence of base object failures, `Apply`(p, op, $\mathbb{O}$) returns X, no matter how the steps of `Apply`(p, op, $\mathbb{O}$) and `Apply`(q, op', $\mathbb{O}$) interleave when execution resumes from C. By Lemma 7.1.3, if C is X-valent, either $X = u$ or $X = v$. C is *monovalent* if C is either u-valent or v-valent. C is *bivalent* if it is neither u-valent nor v-valent.

LEMMA 7.1.4. *C_0 is bivalent.*

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

Similarly, starting from C_0, if q completes all the steps of `Apply`(q, op', $\mathbb{O}$) before p starts `Apply`(p, op, $\mathbb{O}$), then `Apply`(q, op', $\mathbb{O}$) returns v'. Thus, by Lemma 7.1.3, when `Apply`(p, op, $\mathbb{O}$) 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 `Apply`(p, op, $\mathbb{O}$) from C. $C(q)$ is similarly defined.

```
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)∧(C(q) is monovalent)
```

FIG. 10. Reaching a *critical* bivalent configuration.

LEMMA 7.1.5. *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 `Apply`(p, op, $\mathbb{O}$) and `Apply`(q, op', $\mathbb{O}$) as shown in Figure 10. Since $\mathbb{O}$ is wait-free, the **repeat** $\cdots$ **until** loop in the figure must terminate after a finite number of iterations. Let C_{crit} be the value of C just when the loop terminates. It is easy to verify that C_{crit} satisfies the properties required by the lemma. $\square$

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

LEMMA 7.1.6. *The enabled steps of p and q in C_{crit} access the same base object.*

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

Assume that O_k is the base object mentioned in the above lemma, and `Apply`(p, $oper$, O_k), `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 `Apply`(q, $oper'$, O_k) returns the same value whether applied in C_{crit} or $C_{crit}(p)$, or `Apply`(p, $oper$, O_k) returns the same value whether applied in C_{crit} or $C_{crit}(q)$. In the following, we will deal with the former case. The latter case can be handled similarly and is omitted.

LEMMA 7.1.7. *Consider*

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

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

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

Suppose `Apply`(q, op', $\mathbb{O}$) returns $\perp$. Since $\mathcal{I}$ is gracefully degrading, $\mathbb{O}$ must either be correct or fail by crash. Given that `Apply`(p, op, $\mathbb{O}$) returns a non-$\perp$

response, this requires that Apply(p, op, $\mathbb{O}$) precedes Apply(q, op', $\mathbb{O}$) in the linearization order. Doing so, however, implies that (op, v) is legal from state s of T. This is false since (op, u) is the only sequence legal from state s of T, and $v \neq u$. Thus Apply(q, op', $\mathbb{O}$) cannot return $\perp$.

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

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

LEMMA 7.1.8. *Consider*

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

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

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

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

Neither (op, u), (op', v') nor (op', v'), (op, u) is legal from state s of T. Hence, the execution in Lemma 7.1.8 is not linearizable. Thus, the failure of $\mathbb{O}$ in S2 is not by crash. We conclude that $\mathcal{I}$ is not a gracefully degrading implementation for crash, a contradiction. This concludes the proof of Theorem 7.1.1. □

7.2. GRACEFUL DEGRADATION FOR OMISSION. We now study the feasibility of achieving gracefully degrading implementations for omission. A *set* $\mathcal{S}$ *of types is universal* if every type has an implementation from $\mathcal{S}$. An example of such a set is {`consensus with safe-reset`, `register`} [Herlihy 1991b]. The main result of this section is the *graceful degradation theorem for omission*, stated as follows: Every type has a t-tolerant gracefully degrading implementation from every universal set of types for omission. We prove this result through three key lemmas. Below, we list these lemmas and explain how they are used in proving the main result.

—*Lemma 7.2.1.1.* Every 0-tolerant implementation can be transformed into a 0-tolerant implementation that is gracefully degrading for omission.

—*Lemma 7.2.2.4.* `register` has a t-tolerant gracefully degrading self-implementation for omission.

—*Lemma 7.2.3.3.* `consensus with safe-reset` has a t-tolerant gracefully degrading implementation from {`consensus with safe-reset`, `register`} for omission.

The steps involved in obtaining the graceful degradation theorem for omission are as follows (in the steps below, the failure mode is implicitly assumed to be omission):

Step (1) Every type has a 0-tolerant implementation from {`register`, `consensus with safe-reset`}. (This follows from Herlihy's universality result [Herlihy 1991b].)

Step (2) Every type has a 0-tolerant gracefully degrading implementation from {`register`, `consensus with safe-reset`}. (This follows from Step (1) and Lemma 7.2.1.1.)

Step (3) `register` has a t-tolerant gracefully degrading self-implementation. (This is Lemma 7.2.2.4.)

Step (4) `consensus with safe-reset` has a t-tolerant gracefully degrading implementation from {`register`, `consensus with safe-reset`}. (This is Lemma 7.2.3.3.)

From Steps (2), (3), and (4), and Corollary 4.1.2, we conclude that every type has a t-tolerant gracefully degrading implementation from {`register`, `consensus with safe-reset`} for omission. From this conclusion, Steps (5) and (6) below, and the compositional lemma (Lemma 4.1.1), we have the main theorem: Every type has a t-tolerant gracefully degrading implementation from every universal list of types for omission.

Step (5) `register` has a 0-tolerant gracefully degrading implementation from any universal set of types.
By definition of a universal set of types, `register` has a 0-tolerant implementation from such a set. This, together with Lemma 7.2.1.1, implies Step (5).

Step (6) `consensus with safe-reset` has a 0-tolerant gracefully degrading implementation from any universal set of types.
The reasoning is the same as for Step (5).

We now prove the three lemmas mentioned above.

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

LEMMA 7.2.1.1. *Let T be a type, s be a state of T, and $\mathcal{I}$ be a 0-tolerant implementation of (T, s) from $(\mathcal{L}, \Sigma)$, for processes $P_1, \ldots, P_N$. Then, $\mathcal{G}(\mathcal{I})$ is a 0-tolerant gracefully degrading implementation of (T, s) from $(\mathcal{L}, \Sigma)$, for processes $P_1, \ldots, P_N$, for omission.*

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

Let $\mathbb{O}$ be a derived object of $\mathcal{G}(\mathcal{I})$. Let E be an execution of $(P_1, \ldots, P_N; \mathbb{O})$ in which (i) one or more base objects of $\mathbb{O}$ fail, (ii) each base object that fails, fails by omission, and (iii) if a process gets the response $\perp$ from $\mathbb{O}$, that process does not subsequently invoke an operation on $\mathbb{O}$. We claim that if $\mathbb{O}$ fails in E, it fails by omission. This claim implies that $\mathcal{G}(\mathcal{I})$ is gracefully degrading for omission. To prove the claim, we must show that all three properties stated in the definition of omission hold for $\mathbb{O}$ in the execution E. Property (2), that every response of $\mathbb{O}$ is from $RES \cup \{\perp\}$, is obvious. We verify Properties 1 and 3 below.

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

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

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

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

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

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

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

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

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

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

7.2.2. *Graceful Degradation for* `register`. We show that `register` has a t-tolerant gracefully degrading self-implementation for omission. The following are the steps involved:

S1. We present a 1-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register`.

S2. As mentioned before, it is known that there is a 0-tolerant implementation of `register` from `1-reader 1-writer safe register`. It follows from Lemma 7.2.1.1 that there is a 0-tolerant gracefully degrading implementation of `register` from `1-reader 1-writer safe register`.

S3. Combining the results in Steps **S1** and **S2** with Corollary 4.1.2, we obtain a 1-tolerant gracefully degrading self-implementation of `register`. By Booster Lemma, this can be turned into a t-tolerant gracefully degrading self-implementation of `register`.

Figure 11 presents a 1-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register`. The implementation uses four base registers. The reader process P_r maintains a local variable $FAILED_r$ to remember the faulty base registers it has so far encountered. The writer process P_w similarly maintains $FAILED_w$. To read the derived register, P_r reads each base register that has so far not appeared faulty to it. It adds base registers that return $\perp$ to the set $FAILED_r$ and collects the responses from other base registers in the multi-set *ValuesRead*. If, at the end, P_r has detected two or more base registers to

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

```
Apply(P_r, read, R)

ValuesRead := ∅
for i := 1 to 4
   if R_i ∉ FAILED_r then
      resp := read(P_r, R_i)
      if resp = ⊥ then
         FAILED_r := FAILED_r ∪ {R_i}
      else ValuesRead := ValuesRead ∪ {resp}
if |FAILED_r| ≥ 2 then
   return ⊥
else return mode(ValuesRead)
```

```
Apply(P_w, write v, R)

for i := 1 to 4
   if R_i ∉ FAILED_w then
      resp := write(P_w, v, R_i)
      if resp = ⊥ then
         FAILED_w := FAILED_w ∪ {R_i}
if |FAILED_w| ≥ 2 then
   return ⊥
else return ack
```

FIG. 11. 1-tolerant gracefully degrading self-implementation of `1-reader 1-writer safe register` for omission

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

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

LEMMA 7.2.2.1. *Let E be any execution of $\mathcal{S}$ that satisfies the following*:

A1. *P_r invokes only* `Read` *operations on $\mathcal{R}$ and P_w invokes only* `Write` *operations on $\mathcal{R}$.*

A2. *If a process (P_r or P_w) gets the response $\perp$ from $\mathcal{R}$, it does not subsequently invoke an operation on $\mathcal{R}$.*

A3. *If a base object of $\mathcal{R}$ fails, it fails by omission.*

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

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

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

Let r be any complete read operation in H' that is not concurrent with a write operation in H'. Let V be the response returned by r. Let `Apply`(P_w, *write* V', $\mathcal{R}$), denoted by w, be the latest write operation in H' that precedes r. By construction of H' and the fact that r and w are complete operations in H', we have (i) $V \neq \perp$ and (ii) w returned ack (as opposed to $\perp$). Let $\mathbf{F_r}$ be the value of $FAILED_r$ at the end of the read operation r in E. Since r returned $V \neq \perp$, it follows from the implementation that $|\mathbf{F_r}| \leq 1$. Let $\mathbf{F_w}$ be the value of $FAILED_w$ at the end of w. Since w returned *ack*, it follows from the implementation that $|\mathbf{F_w}| \leq 1$. Let $S = \{R_1, R_2, R_3, R_4\} - (\mathbf{F_r} \cup \mathbf{F_w})$. The above implies that either $|S| > 2$ or $\mathbf{F_r} = 1$ and $|S| = 2$. Also, when the reader P_r reads a register $R \in S$ during the execution of r, it is obvious that R returns V'. Therefore, at the end of r, either V' occurs at least three times in *ValuesRead*, or V' occurs exactly twice in *ValuesRead* and $\mathbf{F_r} = 1$. In either case, at the end of r, *mode*(*ValuesRead*) $=$ V'. Hence, r returns V'. We conclude that $V = V'$. In other words, every complete read operation in H', which is not concurrent with a write operation in H', returns the most recent value written. This verifies that $\mathcal{R}$ satisfies Property 3 of omission in E. Hence, the lemma. □

LEMMA 7.2.2.2. *Let E be any execution of $\mathcal{S}$ which satisfies conditions* **A1**, **A2**, *and* **A3** *listed in the previous lemma. Additionally, assume that at most one base object of $\mathcal{R}$ fails in E. Then, $\mathcal{R}$ is correct in E.*

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

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

LEMMA 7.2.2.3. *Figure* 11 *presents a* 1*-tolerant gracefully degrading self-implementation of* `1-reader 1-writer safe register` *for omission.*

PROOF. Immediate from Lemmas 7.2.2.1 and 7.2.2.3. □

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

LEMMA 7.2.2.4. `register` *has a t-tolerant gracefully degrading self-implementation for omission.*

7.2.3. *Graceful Degradation for* `consensus with safe-reset`. We present a t-tolerant gracefully degrading implementation of `consensus with safe-reset` from {`consensus with safe-reset`, `register`} for omission. We begin by stating two propositions that characterize the type `consensus with safe-reset`. These propositions will be useful when we prove the correctness of our implementation. For ease of stating the propositions, we need some definitions.

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

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

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

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

—Object $\mathbb{O}$ satisfies *epoch-validity* in E if and only if the following holds. If $\mathbb{O}$ returns a response v to a propose operation in some clean epoch and $v \in \{0, 1\}$, then there is an invocation of propose v on $\mathbb{O}$, in the same epoch, preceding this response.

—Object $\mathbb{O}$ satisfies *epoch-agreement* in E if and only if the following holds. If $\mathbb{O}$ returns v_1, v_2 to two propose operations in some clean epoch and $v_1, v_2 \in \{0, 1\}$, then $v_1 = v_2$. (By this definition, if $\mathbb{O}$ returns 0 to some processes and $\perp$ to all others, it still satisfies epoch-agreement.)

Notice how these definitions generalize the ones in Section 5.1.1. The propositions below follow easily from the specification of `consensus with safe-reset`, and the definitions of linearizability and omission failures. These propositions are similar to Propositions 5.1.1.1 and 5.1.1.2.

PROPOSITION 7.2.3.1. *Let* $\mathbb{O}$ *be an object of type* `consensus with safe-reset` *and let E be an execution of* $(P_1, P_2, \ldots, P_N; \mathbb{O})$. *Object* $\mathbb{O}$ *is correct in E if and only if* $\mathbb{O}$ *is wait-free in E and satisfies integrity, epoch-validity, and epoch-agreement in E.*

PROPOSITION 7.2.3.2. *Let* $\mathbb{O}$ *be an object of type* `consensus with safe-reset` *and let E be an execution of* $(P_1, P_2, \ldots, P_N; \mathbb{O})$ *in which* $\mathbb{O}$ *fails. Object* $\mathbb{O}$

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

```
Procedure Propose(P_i, v_i, O)
    V_i[1...2t+1], estimate_i, resp, k,
    set-of-failed: local to P_i
begin
    estimate_i := v_i
    set-of-failed := ∅
    for k := 1 to 2t+1
        resp := Read(P_i, R_k)
        if resp = ⊥ then
            return ⊥
        else if resp = 1 then
            set-of-failed := set-of-failed ∪ {O_k}
    for k := 1 to 2t+1
        if O_k ∈ set-of-failed then
            V_i[k] := ⊥
        else
            resp := propose(P_i, estimate_i, O_k)
            if resp = ⊥ then
                resp := Write(P_i, 1, R_k)
                if resp = ⊥ then
                    return ⊥
            else if resp ≠ estimate_i then
                estimate_i := resp
                V_i[1...(k-1)] := (⊥, ⊥, ..., ⊥)
    if V_i has more than t ⊥'s then
        return ⊥
    else return estimate_i
end
```

```
Procedure Reset(P_i, O)
    set-of-failed, resp, k: local to P_i
begin
    set-of-failed := ∅
    for k := 1 to 2t+1
        resp := Read(P_i, R_k)
        if resp = ⊥ then
            return ⊥
        else if resp = 1 then
            set-of-failed := set-of-failed ∪ {O_k}
    for k := 1 to 2t+1
        if O_k ∉ set-of-failed then
            resp := reset(P_i, O_k)
            if resp = ⊥ then
                resp := Write(P_i, 1, R_k)
                if resp = ⊥ then
                    return ⊥
    return ack
end
```

FIG. 12. t-tolerant gracefully degrading implementation of `consensus with safe-reset` for omission

fails by omission in E if and only if it is wait-free in E and satisfies weak-integrity, epoch-validity, and epoch-agreement in E.

Figure 12 presents a t-tolerant gracefully degrading implementation of `consensus with safe-reset` from {`consensus with safe-reset`, `register`} for omission. The implementation uses $2t + 1$ consensus-with-safe-reset objects ($O_1, O_2, \ldots, O_{2t+1}$) and $2t + 1$ t-tolerant gracefully degrading boolean registers ($\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_{2t+1}$). (By Lemma 7.2.2.4, $\mathcal{R}_i$'s can be implemented from registers.) The register $\mathcal{R}_i$ is set to 1 if any process detects O_i to be faulty, that is, if any process obtains the response $\perp$ from O_i. The following is an important running feature of our implementation: If, during the execution of an operation on the derived object $\mathbb{O}$, a process P gets a response of $\perp$ from any $\mathcal{R}_i$, P returns $\perp$ as the response of $\mathbb{O}$. This is justified on the basis that $\mathcal{R}_i$ is t-tolerant, and thus, more than t base objects of $\mathcal{R}_i$ must have failed for $\mathcal{R}_i$ to fail. Since $\mathbb{O}$ needs to be only t-tolerant, $\mathbb{O}$ may fail and return $\perp$ if more than t base objects of $\mathbb{O}$ fail, or equivalently, if any $\mathcal{R}_i$ fails. We now describe the procedures `Reset`(P_i, $\mathbb{O}$) and `Propose`(P_i, v_i, $\mathbb{O}$).

To reset $\mathbb{O}$, a process P_i first reads all $\mathcal{R}_k$'s and collects the identities of the faulty objects among $\{O_1, O_2, \ldots, O_{2t+1}\}$. P_i then resets each nonfaulty object in $\{O_1, O_2, \ldots, O_{2t+1}\}$. If, during this resetting, an object O_k responds with $\perp$

to P_i, P_i writes 1 in $\mathcal{R}_k$ to record the fact that O_k is faulty. At the end of this, P_i returns with the response *ack*.

To propose v_i to $\mathbb{O}$, a process P_i first reads all $\mathcal{R}_k$'s and collects the identities of the faulty objects among $\{O_1, O_2, \ldots, O_{2t+1}\}$. At any point in the algorithm, P_i holds an estimate of the eventual return value in *estimate*$_i$. To start with, *estimate*$_i$ is set to v_i. P_i then goes through $O_1, O_2, \ldots, O_{2t+1}$, in that order, and performs the following steps on each of them. If O_k is known to be faulty, P_i does not access O_k; it simply pretends that O_k returned $\perp$. Otherwise, P_i proposes its current estimate to O_k. If O_k returns $\perp$, P_i writes 1 in $\mathcal{R}_k$ to record the fact that O_k is faulty. If O_k returns a non-$\perp$ response different from P_i's current estimate, P_i deduces that all of $O_1, O_2, \ldots, O_{k-1}$ have failed. Accordingly, P_i sets each location in its local vector $V_i[1 \cdots (k-1)]$ to $\perp$ and changes its estimate to the response it received from O_k. This deduction by P_i is an important step of the algorithm and is intuitively justified as follows: Suppose that some $O_l (1 \leq l \leq k-1)$ were correct. By the integrity and epoch-agreement properties of O_l, every process would receive the same non-$\perp$ response, call it *est*, in that epoch from O_l. Thus, every process will have the same estimate est, at the end of accessing O_l. Furthermore, since even objects that fail by omission satisfy epoch-validity and epoch-agreement, if a base object in $O_{l+1} \cdots O_{2t+1}$ returns a non-$\perp$ response in that epoch, the response must be *est*. Thus, we conclude that, if O_k returns a response in $\{0, 1\}$ which is different from P_i's current estimate, objects $O_1, O_2, \ldots, O_{k-1}$ are faulty. At the end of accessing all $2t+1$ base objects, if P_i believes that no more than t base objects failed, it returns its current estimate. Otherwise, it returns $\perp$.

LEMMA 7.2.3.3. *Figure* 12 *presents a t-tolerant gracefully degrading implementation of* `consensus with safe-reset` *from* {`consensus with safe-reset`, `register`} *for omission*.

PROOF. Let $\mathcal{R}_i$ $(1 \leq i \leq 2t+1)$ be a derived object of the t-tolerant gracefully degrading implementation of `register` (such an implementation exists by Lemma 7.2.2.4). Let $R_{i,1}, R_{i,2}, \ldots, R_{i,m}$ be the base registers of $\mathcal{R}_i$. Let $\mathbb{O}$ be derived from the implementation in Figure 12 using $O_1, O_2, \ldots, O_{2t+1}$ and $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_{2t+1}$. Thus, $O_1, O_2, \ldots, O_{2t+1}$ and $R_{i,j}$ $(1 \leq i \leq 2t+1$, $1 \leq j \leq m)$ are the base objects of $\mathbb{O}$. Consider an execution E of $(P_1, P_2, \ldots, P_N; \mathbb{O})$ in which all base objects that fail, fail by omission. Let $\mathcal{E}$ be a clean epoch of $\mathbb{O}$ in E. Let *FAILED*$(\mathcal{E})$ be the set of all O_j $(1 \leq j \leq 2t+1)$ such that some process had written 1 in $\mathcal{R}_j$ before epoch $\mathcal{E}$ started. Thus, *FAILED*$(\mathcal{E})$ is the subset of $\{O_1, O_2, \ldots, O_{2t+1}\}$ that failed before the start of $\mathcal{E}$. We make the following observations.

O1. For each base object $O \in \{O_1, O_2, \ldots, O_{2t+1}\} - \mathit{FAILED}(\mathcal{E})$, $\mathcal{E}$ is a clean epoch of O.

O2. In epoch $\mathcal{E}$, no process invokes an operation on a base object in *FAILED*$(\mathcal{E})$.

O3. In the execution of `Propose`$(P_i, v_i, \mathbb{O})$, at the end of the kth iteration of the **for-loop** $(1 \leq k \leq 2t+1)$, *estimate*$_i \in \{0, 1\}$, and $V_i[1 \cdot\cdot k]$ contains only $\perp$'s and *estimate*$_i$'s.

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

(1) $\mathbb{O}$ *is wait-free*. Recall that base objects that fail by omission remain wait-free. From this and the implementation, it is obvious that $\mathbb{O}$ is wait-free.

(2) $\mathbb{O}$ *satisfies epoch-validity*. Suppose that an execution of $\texttt{Propose}(P_i, v_i, \mathbb{O})$ in epoch $\mathscr{E}$ returns $v \in \{0, 1\}$. (Let e_{ret} denote the event of completion of this execution.) It follows that, during this execution, some base object O_j returns v to P_i when P_i performs $\texttt{propose}(P_i, \textit{estimate}_i, O_j)$. Let e_f denote the first response event in $\mathscr{E}$ in which a base object among $\{O_1, O_2, \ldots, O_{2t+1}\}$ returns the response v. Let O_f be the base object associated with the event e_f. By **O2**, $O_f \in \{O_1, O_2, \ldots, O_{2t+1}\} - \textit{FAILED}(\mathscr{E})$. By **O1**, $\mathscr{E}$ is a clean epoch of O_f. Since O_f either is correct of fails by omission, by Propositions 7.2.3.1 and 7.2.3.2, O_f satisfies epoch-validity. That is, there is an invocation of $\texttt{propose}(P_l, v, O_f)$ in $\mathscr{E}$ before the response event e_f. From the implementation and the definition of e_f, this invocation of $\texttt{propose}(P_l, v, O_f)$ is possible only during the execution of $\texttt{Propose}(P_l, v, \mathbb{O})$. Thus, the invocation of $\texttt{Propose}(P_l, v, \mathbb{O})$ precedes the invocation of $\texttt{propose}(P_l, v, O_f)$, which, in turn, precedes e_f. Furthermore, e_f precedes e_{ret}. This implies that the invocation of $\texttt{Propose}(P_l, v, \mathbb{O})$ precedes e_{ret}. We conclude that $\mathbb{O}$ satisfies epoch-validity in E.

(3) $\mathbb{O}$ *satisfies epoch-agreement*. Suppose that, in $\mathscr{E}$, there is an execution of $\texttt{Propose}(P_i, v_i, \mathbb{O})$ and one of $\texttt{Propose}(P_j, v_j, \mathbb{O})$, which return 0 and 1, respectively. We will refer to these executions as *exec1* and *exec2*. From **O3** and the implementation, it follows that V_i has at least $t + 1$ 0's at the end of *exec1*. Similarly, V_j has at least $t + 1$ 1's at the end of *exec2*. This implies that there is a k $(1 \le k \le 2t + 1)$ such that O_k returns 0 when P_i performs $\texttt{propose}(P_i, \textit{estimate}_i, O_k)$ in *exec1* and returns 1 when P_j performs $\texttt{propose}(P_j, \textit{estimate}_j, O_k)$ in *exec2*. By **O2**, $O_k \in \{O_1, O_2, \ldots, O_{2t+1}\} - \textit{FAILED}(\mathscr{E})$. It follows from **O1** that $\mathscr{E}$ is a clean epoch for O_k. Since O_k either is correct or fails by omission, by Propositions 7.2.3.1 and 7.2.3.2, O_k satisfies epoch-agreement. This contradicts the earlier conclusion that O_k returns 0 to P_i and 1 to P_j. We conclude that $\mathbb{O}$ satisfies epoch-agreement in E.

(4) $\mathbb{O}$ *satisfies weak integrity*. Obvious.

(5) $\mathbb{O}$ *satisfies integrity if at most t base objects fail*. Suppose that no more than t base objects of $\mathbb{O}$ fail. For all j, $1 \le j \le 2t + 1$, since $\mathscr{R}_j$ is t-tolerant, $\mathscr{R}_j$ will be correct. It follows from the implementation that every reset operation on $\mathbb{O}$ in E returns *ack*. We now make some observations to show that every propose operation on $\mathbb{O}$ in $\phi(E)$ returns either 0 or 1. In the following, let $\mathscr{E}$ be any (not necessarily clean) epoch of $\mathbb{O}$ in E.

   (a) Let $O_{k_1}, O_{k_2}, \ldots, O_{k_l}$ $(k_1 < k_2 < \cdots < k_l)$ be all the base objects among $\{O_1, O_2, \ldots, O_{2t+1}\}$, which are correct in E. Since at most t fail, we have $l \ge t + 1$.

   (b) From the fact that O_{k_1} is correct in E, it is easy to verify that $\mathscr{E}$ is a clean epoch for O_{k_1}. Since O_{k_1} is correct and $\mathscr{E}$ is a clean epoch for O_{k_1}, by Proposition 7.2.3.1, O_{k_1} satisfies integrity and epoch-agreement in epoch $\mathscr{E}$. Thus, there is a $v \in \{0, 1\}$ such that every propose operation on O_{k_1}

in epoch $\mathscr{E}$ returns v. This implies that, for every execution of `Propose`(P_i, v_i, $\mathbb{O}$) in $\mathscr{E}$, $estimate_i = v$ at the end of k_1 iterations of the for-loop.

(c) For all $1 \leq j \leq l$, O_{k_j} is correct in E. From this, it is easy to verify that $\mathscr{E}$ is a clean epoch for O_{k_j}. Since O_{k_j} is correct and $\mathscr{E}$ is a clean epoch for O_{k_j}, by Proposition 7.2.3.1, O_{k_j} satisfies integrity, epoch-validity, and epoch-agreement in epoch $\mathscr{E}$. In particular, if every process that proposes to O_{k_j} in epoch $\mathscr{E}$ proposes the value v, then O_{k_j} returns only v in $\mathscr{E}$.

(d) Let $O_j \in \{O_1, O_2, \ldots, O_{2t+1}\} - \{O_{k_1}, O_{k_2}, \ldots, O_{k_l}\}$. By definition, O_j fails by omission in E, returning $\perp$ to some process. Let P be the first process to receive $\perp$ from O_j and let *oper* denote the execution of P's operation on the derived object $\mathbb{O}$ during which P received $\perp$ from O_j. Consider the following two cases. In the first case, assume that O_j returned $\perp$ to P before epoch $\mathscr{E}$ started. Since *oper* is from an earlier epoch than $\mathscr{E}$, it follows that *oper* completed before $\mathscr{E}$ started. This implies that P wrote 1 in $\mathscr{R}_j$ before the start of epoch $\mathscr{E}$. It follows from the implementation that no process invokes an operation on O_j in epoch $\mathscr{E}$. In the second case, assume that O_j never returned $\perp$ to any process before the start of epoch $\mathscr{E}$. Then, it is easy to see that $\mathscr{E}$ is a clean epoch for O_j. Thus, by Proposition 7.2.3.2, if every process that proposes to O_j in epoch $\mathscr{E}$ proposes the value v, O_j returns either v or $\perp$ in $\mathscr{E}$.

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

From (1), (2), (3), and (4) above, and Proposition 7.2.3.2, we conclude that either $\mathbb{O}$ is correct in E or $\mathbb{O}$ fails by omission in E. Thus, the implementation is gracefully degrading for omission. From (1), (2), (3), and (5) above, and Proposition 7.2.3.1, we conclude that if at most t base objects of $\mathbb{O}$ fail in E, and they fail by omission, then $\mathbb{O}$ is correct in E. Thus, the implementation is t-tolerant for omission. This completes the proof of the lemma. □

7.2.4. *Graceful Degradation Theorem for Omission*. From the previous three lemmas, and the argument presented at the beginning of Section 7.2, we have

THEOREM 7.2.4.1. *Every type has a t-tolerant gracefully degrading implementation from every universal set of types for omission*.

8. *Related Work*

In an independent work, Afek et al. [1992; 1995] consider the problem of coping with shared memory subject to *memory failures*. Informally, each failure is modeled as a *faulty write*. The following failure modes are considered:

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

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

Afek et al. [1992] focus 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`). Afek et al. [1992] also give a universal fault-tolerant implementation from unbounded `RMW`, based on Herlihy's universal implementation. The main differences between Afek et al. [1992] and this paper are as follows:

(1) Afek et al. [1992] does not consider any non-responsive failure mode.
(2) Amongst the responsive failure modes, benign ones, such as crash and omission, are also not considered in Afek et al. [1992].
(3) This paper does not consider failure modes that bound the number of times faulty objects can fail (in Afek et al. [1992], each "faulty write" is counted as a failure).
(4) The two approaches to modeling failures appear to be fundamentally different. There is no direct way to model benign failures, such as crash and omission failures, with "faulty writes". On the other hand, our approach—defining how each faulty object deviates from its type—is not suited to handle Model A above.
(5) This paper introduces the concept of *graceful degradation*, and presents several related results, in particular, for crash and omission failure modes. For arbitrary failures, graceful degradation reduces to the "*strong wait-freedom*" concept introduced in Afek et al. [1992].
(6) In the Open Problems section of Afek et al. [1992], 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 5.3.3. For N-process `test&set`, one can combine the fault-tolerant implementation of `test&set` from {`test&set`, `bounded register`} [Afek et al. 1992], with the implementation of `bounded register` from `test&set` presented in Jayanti et al. [1996].

(7) The existence of a fault-tolerant *self*-implementation of `consensus`, shown in this paper, does not follow from the results in Afek et al. [1992].
(8) The fault-tolerant implementation of N-process `test&set` from {`test&set`, `bounded register`}, shown in Afek et al. [1992], does not follow from our results (when $N > 2$).

ACKNOWLEDGMENTS. We would like to thank Vassos Hadzilacos, Jon Kleinberg, Nancy Lynch, Sendhil Mullainathan, Gil Neiger, King Tan, and the anonymous referees. Their comments on earlier drafts helped us improve the presentation.

REFERENCES

AFEK, Y., GREENBERG, D., MERRITT, M., AND TAUBENFELD, G. 1992. Computing with faulty shared memory. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 47–58.

AFEK, Y., GREENBERG, D., MERRITT, M., AND TAUBENFELD, G. 1995. Computing with faulty shared objects. *J. ACM 42*, 6 (Nov.), 1231–1274.

ASPNES, J. 1990. Time- and space-efficient randomized consensus. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (Quebec City, Que., Canada, Aug. 22–24). ACM, New York, pp. 325–332.

BERMAN, P., GARAY, J., AND PERRY, K. J. 1989. Towards optimal distributed consensus. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. IEEE, New York, pp. 410–415.

BLOOM, B. 1987. Constructing two-writer atomic registers. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 249–259.

BURNS, J. E., AND PETERSON, G. L. 1987. Constructing multi-reader atomic values from nonatomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 222–231.

COAN, B. A. 1987. Achieving consensus in fault-tolerant distributed computer systems: Protocols, lower bounds, and simulations. Ph.D dissertation. Massachusetts Institute of Technology, Cambridge, Mass., June.

COAN, B. A., AND WELCH, J. L. 1992. Modular construction of a Byzantine agreement protocol with optimal message bit complexity. *Inf. Comput. 97*, 1, 61–85.

COURTOIS, P. J., HEYMANS, R., AND PARNAS, D. L. 1971. Concurrent control with "readers" and "writers." *Commun. ACM 14*, 10 (Oct.), 667–668.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan.), 77–97.

DOLEV, D., REISCHUK, R., AND STRONG, H. R. 1990. Early stopping in Byzantine agreement. *J. ACM 37*, 4 (Oct.), 720–741.

FISCHER, M., LYNCH, N., AND MERRITT, M. 1986. Easy impossibility proofs for distributed consensus problems. *Distr. Comput. 1*, 26–39.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr.), 374–382.

HALDAR, S., AND VIDYASANKAR, K. 1991. A simple construction of 1-writer multi-reader multi-valued atomic variable from regular variables. Tech Rep. No. 9108. Department of Computer Science, Memorial University of Newfoundland, St. John's, NF, Canada.

HERLIHY, M. P. 1988. Impossibility and universality results for wait-free synchonization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ont., Canada, Aug. 15–17). ACM, New York, pp. 276–290.

HERLIHY, M. P. 1991a. Randomized wait-free concurrent objects. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Que., Canada, Aug. 19–21). ACM, New York, pp. 11–21.

HERLIHY, M. P. 1991b. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst. 13*, 1 (Jan.), 124–149.

HERLIHY, M. P., AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst. 12*, 3 (July), 463–492.

JAYANTI, P., CHANDRA, T. D., AND TOUEG, S. 1996. Fault-tolerant wait-free shared objects. Tech. Rep. TR 96-1565. Department of Computer Science, Cornell Univ., Ithaca, New York, January.

JAYANTI, P., AND TOUEG, S. 1992. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms* (Haifa, Israel, Nov.). Lecture Notes in Computer Science, vol. 647. Springer-Verlag, New York.

KLEINBERG, J., AND MULLAINATHAN, S. 1993. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, N.Y., Aug. 15–18). ACM, New York, pp. 133–144.

LAMPORT, L. 1977. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov.), 806–811.
LAMPORT, L. 1978. The implementation of reliable distributed multi-process systems. *Comput. Netw. 2*, 95–114.
LAMPORT, L. 1986. On interprocess communication, Parts I and II. *Distr. Comput. 1*, 77–101.
LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July), 382–401.
LOUI, M. C., AND ABU-AMARA. 1987. Memory requirements for agreement among unreliable asynchonous processes. *Adv. Comput. Res. 4*, 163–183.
LYNCH, N., AND TUTTLE, M. 1988. An introduction to input/output automata. Tech. Rep. MIT/LCS/TM-373. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.
NEWMAN-WOLFE, R. 1987. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 232–248.
PEASE, M., SHOSTAK, R., AND LAMPORT, L. 1980. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr.), 228–234.
PETERSON, G. L. 1983. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Prog. Lang. Syst. 5*, 1 (Jan.), 56–65.
PETERSON, G., AND BURNS, J. 1987. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, New York.
PLOTKIN, S. A. 1989. Sticky bits and universality of consensus. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing* (Edmonton, Alb., Canada, Aug. 14–16). ACM, New York, pp. 159–175.
SCHAFFER, R. 1988. On the correctness of atomic multi-writer registers. Tech. Rep. TR No.: MIT/LCS/TM-364. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.
SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec.), 299–319.
SINGH, A. K., ANDERSON, J. H., AND GOUDA, M. G. 1987. The elusive atomic register, revisited. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 10–12). ACM, New York, pp. 206–221.
SRIKANTH, T. K., AND TOUEG, S. 1987. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comput. 2*, 2, 80–94.
VIDYASANKAR, K. 1988. Converting Lamport's regular register to atomic register. *IPL 28*, 287–290.
VIDYASANKAR, K. 1989. An elegant 1-writer multireader multivalued atomic register. *IPL 30*, 221–223.
VITANYI, P., AND AWERBUCH, B. 1986. Atomic shared register access by asynchonous hardware. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. IEEE, New York.

RECEIVED DECEMBER 1993; REVISED JUNE 1997; ACCEPTED FEBRUARY 1998