Computing with Faulty Shared Objects

Yehuda Afek
Tel-Aviv University and AT&T Bell Laboratories
and
David S. Greenberg
Sandia National Laboratories
and
Michael Merritt
AT&T Bell Laboratories
and
Gadi Taubenfeld
AT&T Bell Laboratories

This paper investigates the effects of the failure of shared objects on distributed systems. First the notion of a faulty shared object is introduced. Then upper and lower bounds on the space complexity of implementing reliable shared objects are provided.

Shared object failures are modeled as instantaneous and arbitrary changes to the state of the object. Several constructions of non-faulty wait-free shared objects from a set of shared objects, some of which may suffer any number of faults, are presented. Three of these constructions are: (1) A reliable atomic read/write register from $20f+8$ atomic read/write registers f of which may be faulty, (2) a reliable test&set register for n processes from $n+10$ primitive test&set registers, one of which may be faulty, and $3n+13$ reliable atomic registers, and (3) a reliable consensus object from $2f+1$ read-modify-write registers when f of these may be faulty. Using these constructions a universal construction of any linearizable shared object from a set of either n-processor consensus objects or n-processor read-modify-write registers, some of which may be faulty, is presented.

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

General terms: Fault-tolerance, Shared Memory, Algorithms, Synchronization.

Additional Key Words and Phrases: Atomic operations.

---

1. INTRODUCTION

The desire for increased performance, higher reliability, and decreased cost is causing many real-world applications to be moved from mainframes to distributed systems. Previous to this work, research on the reliability of distributed systems has concentrated on tolerating the failure of individual processors. It has been shown that, by using shared objects to provide communication and coordination between processors, it is possible to tolerate processor failures. Distributed systems can be built which achieve the expected high performance when no processors fail and which are able to continue (at a lower performance) when even all but one processor fails.

David Greenberg was supported in part by the U.S. Department of Energy under contract DE-AC04-76DP00789. A preliminary version of the results presented in this paper appear in [AGMT92].

These fault tolerant systems depend critically on strong shared objects, yet little research has been directed to studying how to tolerate faults in the shared objects. This paper explicitly defines the problem of faulty shared objects and shows how they can be made fault tolerant.

Before considering *faulty* shared objects we must define a *shared object*. Intuitively, as the name implies, a shared object allows two or more processors to share information (a formal definition is given in Section 2). The simplest shared object is a shared memory word (or register), into which one processor writes and any other may read [BP87, Lam86, SAG94, VA87]. The necessity of having more complex shared objects was recognized by IBM and other computer manufacturers in the early 1970's. The inability of concurrent systems which used only read/write memory cells to maintain a concurrent queue or stack or to reach consensus led to the definition (and construction) of stronger primitives such as *test-and-set* and *semaphores* [Dij65, PS85]. Most of this paper is concerned with these stronger types of objects (one section, Section 4 is devoted to read/write registers).

Herlihy has defined a hierarchy of progressively stronger shared objects [Her91]. Objects at each level are able to perform tasks which are impossible for objects at the lower levels. The question which is asked in this paper is whether it is possible to use shared objects which sometimes fail to meet their specifications. It is shown, here, that it is indeed possible to use a set of objects some of which are faulty (at each of three levels of the hierarchy: read/write, test-and-set, and read-modify-write) to produce compound objects which are fault tolerant. The highest level of the hierarchy (exemplified by our read-modify-write) is universal, i.e., allows the construction of any other shared object. A result of our constructions is that any shared memory object Y has a fault tolerant construction from any object type X in the highest level, that is, a construction from a set of objects of type X some of which may be faulty.

It should be clear that a shared object is very different from local processor memory. However, the distinction between shared object and shared memory is less straightforward. A shared memory typically consists of a set of memory locations which can be used for storing and retrieving data by several different processors, and is not used for synchronization. A shared object includes a set of semantic rules which restricts the behavior of the object which thus can be used for synchronization. A shared memory can be used to build shared objects by adding either hardware or software protocols to enforce the timing/coordination features of the shared object.

Since a shared object is defined by its behavior it can be implemented in many ways, either through hardware or through the combination of software and hardware. Thus the *failure* of a shared object can occur in many ways. Our intent is that the model of shared object failure should cover all possible failures in implementation. Hardware implementations can fail in several ways: the data contained in the object can be corrupted, the control logic can incorrectly order events, or requests can be lost due to switching failures. Software protocols can fail in several ways: a buggy protocol entering an infinite loop, a protocol mistakenly allowing a process to affect memory not assigned to it, or a protocol receiving data in an order for which no contingency had been planned.

One might attempt to apply common techniques for dealing with faulty local memory to faulty shared objects. These techniques include keeping many copies of each datum or, in general, using some type of redundant coding which allows errors to be detected and/or corrected. Unfortunately a shared object is much more complex to implement than a memory cell of a uni-processor; i.e., common primitives such as read-modify-write or even simple `read`s and `write`s require a memory which is much more than a passive repository of data (see, e.g., [Smi82]). The use of standard redundancy techniques within the memory associated with the object will not, for example, prevent a loss of access to the object through protocol failure.

If redundancy is spread across objects, then the encoding is subject to the timing inconsistencies which are the bane of all distributed algorithms. Techniques to rectify faults in shared memory objects will necessarily have to incorporate distributed coordination techniques. It is precisely how to supply redundancy while maintaining distributed coordination which is the subject of this paper.

After a discussion of shared object failures and the specification of faulty memory primitives (in Section 2), the body of this paper presents a sequence of constructions that use faulty shared memory objects. Section 3 presents elementary relations among general fault-tolerant constructions, showing how different constructions can be composed. Section 4 begins with constructions of reliable read/write memories from faulty read/write memories. Following Lamport [Lam86], we present simple constructions of safe and regular registers and conclude with a construction of a reliable atomic register from faulty atomic registers. In Section 5 reliable *test&set* objects from similar objects, some of which might be faulty, are constructed.

In Section 6, implementations of consensus are studied, using unreliable *read-modify-write* primitives. The constructions in this section demonstrate that faults do not qualitatively decrease the power of these primitives, in that they retain their positions in the memory hierarchy of Herlihy [Her91]. Moreover, in combination with the earlier register results, our consensus constructions can be used to implement the universal construction of Herlihy [Her91]. Hence, for example, faulty read-modify-write primitives can be used to implement any shared object. The paper closes with a discussion of open problems.

1.1 Related work

Theoretical research on fault-tolerance in shared memory systems typically studies process failures and assumes that the shared objects are reliable [Abr88, AAD+93, Blo87, BP87, CIL87, Her91, Lam86, Plo88, Rab82, SAG94, VA87]. This paper describes the first study of the tolerance of distributed systems to general faults in shared objects.[1] Memory failures are restricted only either in total number, or in the number of data objects that may be affected, without any restriction on the timing of the faults. Some previous research has explored failures which are restricted to occur during specific periods of time. For example, such constrained memory faults are studied in work on *self-stabilizing* systems defined by Dijkstra [Dij74]. Self-stabilizing systems are required to recover once the final memory fault has occurred, and the system is in an arbitrary state. That is, self-stabilizing protocols may start at any erroneous, globally inconsistent state and must always reach a correct global state satisfying particular safety requirements.

[1]Concurrently and independently of our work Jayanti, Chandra, and Toueg have addressed similar problems [JCT92] (see Section 7 for more on this paper).

Three previous papers investigated *initialization failures*, a special case of self-stabilization [FMRT90, FMT93, MTY92]. In that model, only the initial state of shared objects may be corrupted, while the initial state of the processors is not corrupted (e.g. the program counters). In this paper the corruption may repeat at any time during the run.

In Section 6 of this paper we use implementations of consensus to explore properties of faulty shared memory. The consensus problem is fundamental in distributed computing and is at the core of many algorithms for fault-tolerant distributed applications. Much is known about the consensus problem in other fault models (see, e.g., [Abr88, AH90, CIL87, Fis83, FLM85, FLP85, LAA87, SSW91].) Sections 4 and 5 investigate the question of constructing reliable registers in an unreliable environment. This relates to the fundamental problem of implementing one type of shared object from another. Previous work on shared object implementations includes [Blo87, BP87, Her91, Lam86, LTV89, Plo88, SAG94, VA87].

As noted above there is a relationship between implementing a parallel processor shared memory and the construction of shared objects. Implementing shared objects of any type in a network-based machine is difficult. The task of including coordination protocols and fault tolerance could not begin without the basic mechanisms for sharing. One approach to providing sharing is to use local caches to hold pieces of global data required by each processor. Special operating system functions and hardware remove from the user the burden of coordinating the accesses to the shared data. One such system has been proposed, implemented and analyzed in [LH89]. The KSR, DASH, and Alewife machines are a few examples of machines that use such distributed shared memory [Bel92]. Another, more software-oriented, approach is implemented in the the Linda system [CG89]. In Linda, an abstract tuple space is shared (instead of cache lines or pages), and operations are available to insert and delete tuples. Obviously, the choice of one of these methods of implementing sharing (or any of many other clever techniques) will affect the types of errors expected from shared objects. We expect that our ideas on fault tolerance will be specialized to various implementations in order to increase efficiency.

2. SPECIFYING FAULTY SHARED MEMORY OBJECTS

We consider a collection of *asynchronous* processes which communicate via a collection of shared memory objects. The shared memory may consist of a variety of shared data objects. These shared objects are subject to *faults*. Each fault of a shared object is modeled as a state change that appears to be atomic with respect to the processes' operations. The local (non-shared) memory for each process is assumed to be reliable.

An object $\mathcal{O}$ shared by n processes can be specified via a state machine in which the state transitions are labeled by the invocations and responses of operations performed by the processes.

Definition 1. *A* sequential specification of an object type *is a quintuple* $\{Q, S, I, R, \delta\}$ *where:*

. Q *is a (finite or infinite) set of* states.

. $S \subseteq Q$ *is a set of* initial states,

. $I = (Inv_1, ..., Inv_n)$ *is an n-tuple of sets, where each* Inv_i *is a set of symbols denoting the* operation invocations *by process i. Let* $Inv = \cup_i Inv_i$.

. *$R = (Res_1, ..., Res_n)$ is an n-tuple of sets, where each Res_i is a set of symbols denoting the* operation responses *for process i. Let $Res = \cup_i Res_i$.*

. *Define the set of operations by process i on $\mathcal{O}$ to be $Op_i = Inv_i \times Res_i$, all the two-character strings of invocations and responses by i, and let $Op = \cup_i Op_i$. Then $\delta \subset Q \times Op \times Q$ is the* transition relation*.*

This state machine denotes a set of (finite and infinite) strings, obtained by concatenating the edge labels along paths in the state transition graph. The *sequential runs* of $\mathcal{O}$ are the (finite and infinite) prefixes of these strings. (Taking prefixes allows runs to end with a pending invocation, that is, an inv_i by some process i with no succeeding response, res_i, by i).

Operations on shared objects are required to be total: for every state $s \in Q$, every process i, and every $inv_i \in Inv_i$, there exists $res_i \in Res_i$ and $s' \in Q$ such that $(s, inv_i res_i, s') \in \delta$.

The sequential specification allows operations to be specified as atomic state transitions. However, in asynchronous concurrent systems operations have duration and can overlap in time. This is modeled by allowing the interleaving of invocations and responses by different processes, so that between the invocation and response by one process may be any number of invocations or responses by other processes. Thus, concurrent runs of the object are modeled as elements of $(Inv + Res)^\infty$ (i.e., finite and infinite runs). Specific correctness conditions constrain these runs by relating them to those in the sequential specification. One such correctness condition is linearizability [HW87].

We next define the notion of linearizable runs of an object $\mathcal{O}$ [HW87]. Given a string of *events* $\alpha \in (Inv + Res)^\infty$, define a partial order, $\prec_\alpha$ on the events in α as follows: $a \prec_\alpha b$ if and only if either (1) both a and b are invocations or responses of the same process (have the same subscript) and a appears before b in α, or (2) a is a response that precedes the invocation b, in α. Then α is atomic if there is a sequential run β of $\mathcal{O}$, containing exactly the events in α and such that the total order $\prec_\beta$ is an extension of $\prec_\alpha$.

Some widely-studied objects are not atomic—the regular and safe registers defined by Lamport are the best-known examples [Lam86]. In Section 4, we will define safe and regular runs for register objects.

A fault in an object run is modeled as a transition to an arbitrary state. Hence, a *faulty object* extends the set of operations of $\mathcal{O}$ with a set of `fault` operations, $Op_F = \{F(s')ack : s' \in Q\}$. The transition relation δ is augmented to include $(s, F(s')ack, s')$, for every s, $s' \in Q$. Occurrences of these operations in a run are *faults* of the object. Hence, the sequential specification of the faulty object now includes faults, and atomic runs of the faulty object are defined as before.

If all the shared objects in a shared memory may fail then obviously the only constructions possible are those in which processes work in isolation and do not communicate. Therefore, some constraints are imposed on the occurrence of memory faults:

. We use m to denote the total number of memory faults (`fault` operations) in a run of a system.

. We use f to denote the total number of objects in a system that may be affected by memory faults in a run (or collection of runs) of a system.

. A data object is *k-faulty* in a run or set of runs if it suffers at most k faults. That is, at most k `fault` operations, in which this object is involved, are invoked during each run. A data object is *∞-faulty* if there is no finite bound on the number of faults it suffers.

2.1 Implementations

Whatever the types of fault, we are concerned with the construction of high-level objects from potentially-faulty primitive objects. For the purposes of this paper, it suffices to consider an implementation of an n-process high-level object $\mathcal{O}$ from a set of primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$ as a set of procedures for each process. Invocations and responses of these procedures are identified with those in the specification of $\mathcal{O}$. The procedures are allowed to do local computation, and communicate only by making invocations and receiving responses from the primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$. Hence, runs of the implementation of $\mathcal{O}$ consist of sequences containing invocations and responses to the high-level object $\mathcal{O}$, local steps of the procedures, and invocations and responses to the primitive objects $\{\mathcal{O}_1, ..., \mathcal{O}_k\}$.

The runs are constrained in the obvious way to respect the control flow of the individual procedures, allowing arbitrary, asynchronous interleaving between the threads of distinct processes. Moreover, the subsequences of invocations and responses to each primitive object must satisfy the specification of that object. Given these constraints, the subsequences of high-level invocations and responses to $\mathcal{O}$ must in turn satisfy the specification of $\mathcal{O}$.

To provide fault-tolerance against process failure, most of the implementations are also required to be *wait-free*; that is, any single high-level operation *op* (procedure invocation) must terminate, regardless of the steps taken by any other high-level operations, provided the local actions and low-level operations of *op* are allowed to progress [Lam86].

This informal notion of implementation may be made precise using any of several formalisms [HW87, Her91, Lam86, LT87].

In this paper we require the constructions to tolerate a certain number of memory faults. But we further require the constructions to be *strongly wait-free*; i.e., operations must return (perhaps with an incorrect value), even if too many faults occur. That is, any high-level operation by a process must terminate its execution, regardless of the number of shared memory faults and independent of the steps taken by other processes. Thus, a strongly wait-free construction may correctly implement a shared object for only a bounded number of memory faults, but each high-level operation by a non-faulty process must still terminate, even if the bound on the number of memory faults is exceeded in a given run. This property is important in composing implementations, as in Theorem 2. (Strong wait-freedom is a special case of *gracefully degrading* constructions [JCT92]. See also the discussion in Section 7.)

In order to quantify the cost of having a certain number and type of memory faults, we define a function, **CONST**, which represents the number of copies of one type of object, some of which are faulty, that are necessary in any strongly wait-free construction of a reliable n-processor object (of the same or other type):

DEFINITION 2.

. $\mathbf{CONST}(X, m, Y) \stackrel{\text{def}}{=}$ *the number of n-processor objects of type X required in* *any strongly wait-free construction of one n-processor object of type Y, which is reliable in the presence of at most m memory faults among the type X objects.*

. $\mathbf{CONST}(X, (f, k), Y) \stackrel{\text{def}}{=}$ *the number of n-processor objects of type X required in any strongly wait-free construction of one n-processor object of type Y, which is reliable if at most f of the type X objects may be k-faulty (k can be ∞).*

Our results will typically be of the form $\mathbf{CONST}$(RMW, m, consensus) $\leq 2m+1$, an upper bound on the number of read-modify-write registers sufficient for a strongly wait-free implementation of consensus when m memory faults may occur, or $\mathbf{CONST}(\text{RMW}, m, \text{consensus}) \geq 2m+1$, a lower bound on the number of read-modify-write registers necessary to implement consensus if m memory faults may occur.[2]

2.2 Alternative fault models

The fault model in the previous section assumes that faulty objects send signals that are syntactically correct, if semantically absurd. A seemingly more malicious fault model might allow faulty objects to behave arbitrarily, never responding to requests, sending syntactically incorrect replies, or sending replies for which there has been no request. Objects which may not respond to requests are studied in [JCT92], and are shown to be too weak to be useful in fault-tolerant constructions.

Consider objects which respond with syntactically senseless replies (a `write` acknowledgment for a `read` operation, for example), or which spontaneously send replies with no associated request. It is simple to simulate the model considered in this paper, by interposing code locally at each process. This code substitutes arbitrary syntactically correct responses for syntactically incorrect responses (substituting a `read` of any value for the `write` acknowledgment) and discards unanticipated responses. Once the request/response pattern is syntactically correct, faulty `write` operations can be used to explain the semantic absurdities.

Other less general types of faulty runs might include objects which never change their value (after becoming faulty all `read`s return the same value regardless of any write command), objects which occasionally miss a `write` (the written value is never available to a `read`), and objects which occasionally return the wrong value (some `read`s return arbitrary values or return values which are not consistent with any ordering of the `write`s). Alternatively, the timing of the faults might be restricted; for example, all memory faults might occur before any process takes a step. Jayanti *et al.* [JCT92] define additional, weaker (i.e., more constrained) fault models, which they call crash and omission. (A discussion of this paper appears in Section 7.)

3. GENERAL CONSTRUCTIONS

Recall the notation $\mathbf{CONST}(X, kf, Y)$ indicates the minimum number of objects of type X needed to construct an object of type Y for runs in which the total number of faults is at most kf. Since these runs include those in which at most f faults occur among at most k of the components of type X, we have the first part of the theorem below.

[2]The lower bounds for specific problems in this paper hold for constructions that are reliable but are not necessarily strongly wait-free. That is, the strongly wait-free assumption is not needed.

THEOREM 1.

. $\mathbf{CONST}(X,(f,k),Y) \le \mathbf{CONST}(X,kf,Y)$.

. $\mathbf{CONST}(X,f,Y) \le \mathbf{CONST}(X,(f,f),Y)$.

The second part of the theorem says it is no harder to survive a total of f faults than to survive at most f faults each, among at most f components.

The next theorem demonstrates the importance of self-implementations: if objects of type X can be composed to create an object of the same type that is tolerant of $t \ge 1$ faults, then that construction can be bootstrapped to overcome any larger number of faults. (Throughout, logarithms are base 2 unless otherwise noted.)

THEOREM 2. *For any f and t, $f > t > 0$, let $L = \log_{t+1} \mathbf{CONST}(X,(t,\infty),X)$. Then:*

$$\mathbf{CONST}(X,(f,\infty),X) \le ((t+1)f)^L.$$

PROOF. As illustrated in the left side of Figure 1, assume that there is a base construction of a reliable object of type X, using $C = \mathbf{CONST}(X,(t,\infty),X)$ strongly wait-free objects of type X, t of which may be ∞-faulty.

The goal is to construct a strongly wait-free object X that is reliable, even if f of the primitive objects are ∞-faulty. Consider the following idea: take the base construction, which survives up to t faults among the C primitives of type X. Call each of these C primitives "embedded objects", and now implement each of the C embedded objects using the same base construction. The result, as illustrated on the right side of Figure 1, is a construction of X from C^2 primitive objects of type X. Moreover, each of the embedded objects is resilient to t faults of the true primitives, and the single high-level object is resilient to t faults of the embedded objects.

Consider next how many primitive faults are necessary to cause the high-level object to fail: at least $t+1$ of the embedded objects must fail, and each of these embedded objects will be correct unless at least $t+1$ of its primitive objects fail. Hence, the high-level object is resilient to $(t+1)^2 - 1$ primitive faults. The strong wait-freedom of the base construction ensures that the embedded objects return some value, even if $t+1$ or more primitive objects fail.

The theorem follows by applying this idea recursively. First, recursively construct C strongly wait-free objects of type X, each of which is resilient to $\left\lfloor \frac{f}{t+1} \right\rfloor$ faulty objects, then use these C objects to construct a single object X, using the t fault construction. The result is an object that can tolerate the fault of t of the embedded $\left\lfloor \frac{f}{t+1} \right\rfloor$ resilient objects.

By recursion, each of the C embedded objects, c, will behave in a fault-free manner, provided no more than $\left\lfloor \frac{f}{t+1} \right\rfloor$ of the primitive objects used to construct c are faulty. If more than $\left\lfloor \frac{f}{t+1} \right\rfloor$ primitives in c fail, then c may no longer behave correctly, and may appear to be faulty, to the higher level construction. But by the strong wait-freedom of the construction, calls to c will at least return.

In order to cause the highest level of the recursion to exhibit faulty behavior, $f+1$ of the C embedded objects must exhibit faulty behavior. For one of the embedded objects to fail, there must be at least $\left\lceil \frac{f}{t+1} \right\rceil$ memory faults in it. Since the total

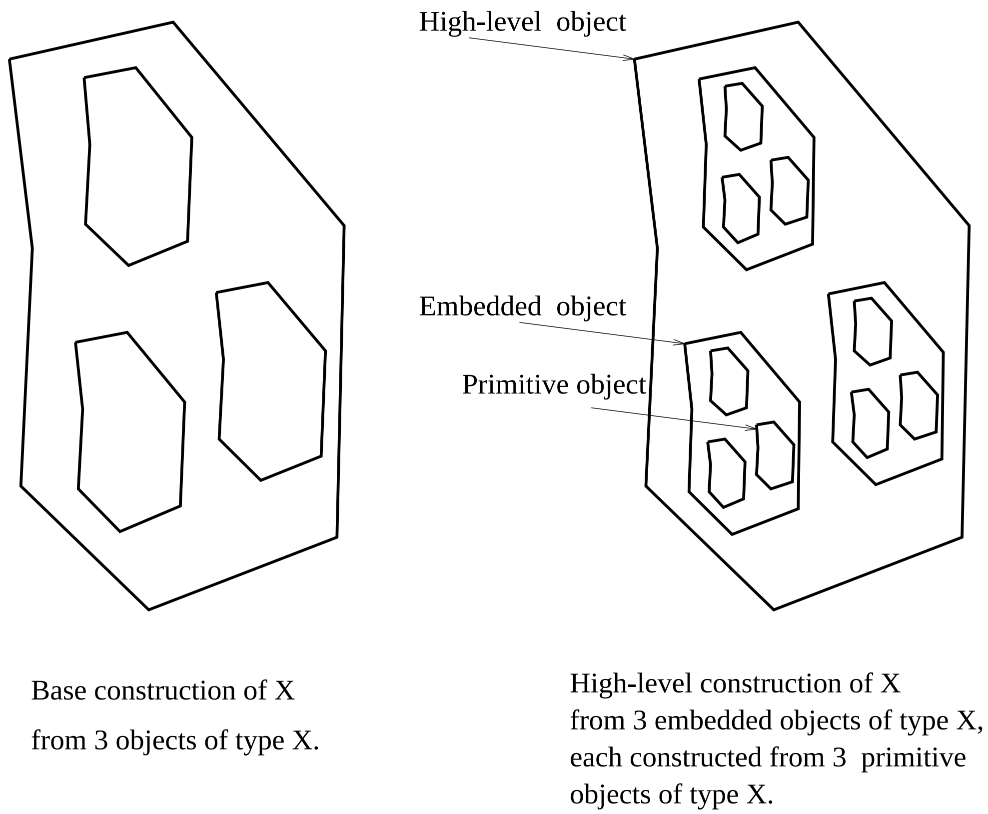

Fig. 1. Constructions in the proof of Theorem 2.

number of faults is f and $f - t\left\lceil\frac{f}{t+1}\right\rceil \leq \frac{f}{t+1}$, none of the $C - t$ other embedded objects is faulty. Hence, the final construction is tolerant of at least f memory faults.

The total number of X objects used in this construction is:

$$\begin{aligned}
\mathbf{CONST}(X,(f,\infty),X) &= \mathbf{CONST}(X,(t,\infty),X)\cdot\mathbf{CONST}(X,(\left\lfloor\frac{f}{t+1}\right\rfloor,\infty),X)\\
&\leq \mathbf{CONST}(X,(t,\infty),X)^{\lfloor\log_{t+1} f\rfloor+1}\\
&\leq \mathbf{CONST}(X,(t,\infty),X)^{\log_{t+1} f+1}\\
&= ((t+1)f)^L.
\end{aligned}$$

Time complexity grows similarly: suppose in the base construction of X, that T is an upper bound on the number of primitive operations needed to implement a single high-level operation on the constructed object. Then in the recursive construction to overcome f faults, each high-level operation requires at most $T^{\log_{t+1} f+1} = ((t+1)f)^{\log_{t+1} T}$ operations on primitive objects. $\square$

Fault-tolerant constructions can be composed with fault-intolerant constructions in obvious ways:

THEOREM 3. *For any $f, m \geq 0$, and for all $k \in \{1, ...\} \cup \{\infty\}$,*

$$\mathbf{CONST}(X, (f,k), Z) \leq \mathbf{CONST}(X, (f,k), Y) \cdot \mathbf{CONST}(Y, (0,k), Z);$$
$$\mathbf{CONST}(X, m, Z) \leq \mathbf{CONST}(X, m, Y) \cdot \mathbf{CONST}(Y, 0, Z).$$

PROOF. The theorem follows by taking a fault-intolerant construction of an object of type Z from objects of type Y, but constructing each object of type Y in a fault-tolerant way from potentially-faulty components of type X. □

A less obvious composition allows faulty objects of type X to be used in a strongly wait-free, but otherwise fault-intolerant construction of objects of type Y. If the X objects are in fact faulty, the result will be an object of type Y that may be ∞-faulty, but can then be used in fault-tolerant constructions. Note, that in the following theorem $\mathbf{CONST}(X, (0, \infty), Y)$ is the usual space complexity of wait-free constructions of Y from X;

THEOREM 4. *For any $f > 1$,*

$$\mathbf{CONST}(X, (f, \infty), Z) \leq \mathbf{CONST}(X, (0, \infty), Y) \cdot \mathbf{CONST}(Y, (f, \infty), Z).$$

4. READ/WRITE REGISTERS

One approach to tolerating faulty read/write registers is to add a software layer between the faulty memory and the user which looks to the user like fault-free memory. In this section it is shown that this is possible for safe, regular, and atomic read/write registers. That is, we construct safe, regular, and atomic read/write registers from a collection of the corresponding primitives, f of which may be ∞-faulty. (Henceforth, we use "atomic register," "read/write register," and "atomic read/write register" interchangeably.)

Atomic registers can be specified using sequential specifications, as described in Section 2. Safe registers, as defined by Lamport [Lam86], have behavior that is sensitive to concurrency. Hence, we directly specify the legal runs of a safe register (over a data domain V): A safe register may be accessed by two processes. One performs `read` operations, which return values from V, and the other performs `write` operations, which take values as arguments. A sequence of invocations and responses of these operations is safe if and only if the invocations and responses by each process alternate appropriately, and each `read` operation that is not concurrent with a `write` operation returns the value written by the last `write` operation that precedes the `read`, or returns the initial value, if there is no such `write`. (Hence, `read` operations which are concurrent with `write`s may return any value.)

Regular registers are also sensitive to concurrency, but are more constrained than safe registers. They are defined as safe registers, except that `read` operations must return a value of a `write` that is either concurrent with the `read`, or is the last `write` (or initial value) that precedes the `read` [Lam86].

A pair of read and write procedures is safe (or regular) if all concurrent executions give rise to safe (or regular) sequences of invocations and responses.

The notion of a fault in a safe or regular register is introduced, in the spirit of that of atomic objects, as an atomic `write` operation (by some outside agent). A sequence of invocations and responses for a safe register is called *m-faulty* if, m is the smallest number such that it is possible to produce a legal sequence of a safe register from the original sequence by adding m instantaneous `write` operations (strings of the form $W(v_i)ack$) (Note that m may be infinite.) Faults of regular registers are defined analogously.

Next we prove that, one reliable, strongly wait-free, multi-reader/single-writer safe register can be constructed from $2f+1$ similar registers, f of which may be ∞-faulty, but cannot be constructed from $2f$ such registers, f of which may be 1-faulty.

THEOREM 5.

. $\mathbf{CONST}(\mathit{safe}, (f, \infty), \mathit{safe}) \leq 2f+1$.

. $\mathbf{CONST}(\mathit{safe}, (f, 1), \mathit{safe}) > 2f$.

PROOF. For the upper bound, a simple construction works: the writer writes the $2f+1$ registers, and the reader reads them. If the reader sees a majority value ($f+1$ with the same value), then it returns that value; otherwise it returns any value. Each high-level operation requires $2f+1$ operations on primitive registers.

For the lower bound, assume to the contrary that there is a solution using $2f$ registers. Without loss of generality, let the initial value in the high-level object be 0. Let the initial values of the primitive registers $r_1, ..., r_{2f}$ be $u_1, ..., u_{2f}$, respectively.

Let α be a run in which process p_1 runs alone and performs a single `write` of 1 to the high-level register. Let the final values of the registers $r_1, ..., r_{2f}$ in α be $v_1, ..., v_{2f}$, respectively.

Now consider a run β in which p_2 runs alone and performs a single high-level `read` operation, after f faults have changed the initial values of $r_{f+1}, ..., r_{2f}$ to $v_{f+1}, ..., v_{2f}$, respectively. Hence, p_2 finds the registers holding values $u_1, ..., u_f, v_{f+1}, ..., v_{2f}$. Since p_1 has taken no steps in β, the `read` by p_2 must return 0. But β is indistinguishable by p_2 from a run γ that extends the run α in which p_1 wrote 1, by f faults that change the values of $r_1, ..., r_f$ back to $u_1, ..., u_f$, respectively, followed by a single high-level `read` operation by p_2. Hence, the `read` by p_2 in γ (and so in the indistinguishable β) must return the 1 written by p_1, a contradiction. □

In addition to bounding the number of primitive objects, the last argument in the above proof also shows that at least $2f+1$ primitive operations are necessary in order to implement high-level `read` or `write` operations, thus matching the upper bound in the theorem. Note that the lower bound also holds for constructions that are not strongly wait-free.

Since the upper bound holds for an infinite number of faults per object, and the lower bound holds for one fault per object, the first part of the corollary below follows. The second part is a consequence of the first part and Theorem 1, which implies $\mathbf{CONST}(\mathit{safe}, (m, 1), \mathit{safe}) \leq \mathbf{CONST}(\mathit{safe}, m, \mathit{safe}) \leq \mathbf{CONST}(\mathit{safe}, (m, m), \mathit{safe})$.

COROLLARY 6.

. *For all* $k \in \{1, ...\} \cup \{\infty\}$, $\mathbf{CONST}(\mathit{safe}, (f, k), \mathit{safe}) = 2f+1$.

. $\mathbf{CONST}(\mathit{safe}, m, \mathit{safe}) = 2m+1$.

In what follows, registers are assumed to be single-reader/single-writer. Since a single (reliable) safe bit is sufficient to implement a regular bit [Lam86], and regular registers are stronger than safe registers, it follows from Theorems 3 and 5 that $\mathbf{CONST}(\mathit{binary\_safe}, (f, \infty), \mathit{binary\_regular}) = 2f + 1$. Moreover, given the fault-tolerant construction in Theorem 5, one can construct any reliable (multi-reader/multi-writer, arbitrary value) atomic register by composing with fault-intolerant constructions from safe bits [Blo87, BP87, Lam86, LTV89, Pet83, PB87, SAG94, Tro89], appealing to Theorem 3. For example, a construction due to Tromp [Tro89] produces a binary atomic register from 3 safe bits (3 are necessary [Lam86]); thus,

$$\begin{aligned}
&\mathbf{CONST}(binary_safe, (f, \infty), binary_atomic) \leq \\
&\mathbf{CONST}(binary_safe, (f, \infty), binary_safe) \\
&\cdot \mathbf{CONST}(binary_safe, (0, \infty), binary_atomic) \\
&= 6f + 3.
\end{aligned}$$

Define a V-register to be a read/write register on (arbitrary) value domain V.

A fault-intolerant construction by Peterson [Pet83] produces an atomic V-register from 3 safe V-registers and 4 atomic binary registers. Appealing as above to Theorems 3 and 5, each of the 3 safe V-register can be implemented reliably from $2f+1$ unreliable safe V-registers, and as above, each of the 4 atomic binary registers can be implemented from $6f+3$ unreliable safe bits.

Another fault-intolerant construction due to Tromp [Tro89] produces an atomic V-register from 4 safe V-registers and 8 safe binary registers. As above, each of the 4 safe V-registers can be implemented reliably from $2f+1$ unreliable safe V-registers, and the $2f+1$ unreliable safe binary registers suffice to reliably implement each of the 8 safe binary registers.

In an earlier version of this paper [AGMT92], a construction is given (Figure 1 in that paper) that constructs a reliable atomic V-register from $8f+2$ atomic V-registers, f of which may fail, and 2 reliable atomic binary registers. As above, the 2 atomic bits can each be implemented from $6f+3$ unreliable safe bits.

We conclude:

THEOREM 7. *One reliable, strongly wait-free, atomic V-register can be constructed from the combinations of primitive registers below, where in each case, up to f of the primitive registers may be ∞-faulty:*

. *$6f+3$ safe V-registers and $24f+12$ safe binary registers (*[Pet83] *and Theorem 5).*

. *$8f+4$ safe V-registers and $16f+8$ safe binary registers (*[Tro89] *and Theorem 5).*

. *$8f+2$ atomic V-registers and $12f+6$ safe binary registers (*[AGMT92] *and Theorem 5).*

If we consider the construction of atomic registers *only* from faulty atomic registers of the same type, the construction from [AGMT92] dominates (using V-registers to implement safe bits):

COROLLARY 8. $\mathbf{CONST}(atomic, (f, \infty), atomic) \leq 20f + 8$.

Theorem 5 results in other $\mathbf{CONST}(safe, (f, \infty), Z)$ theorems for any object Z that is constructed from single-writer/multi-reader safe registers. Shared memory algorithms such as Lamport's "Bakery" mutual exclusion algorithm [Lam74], which originally uses $2n$ registers, can now be implemented from faulty memory using $4nf + 2n$ safe registers, f of which may be faulty (following Theorem 3 and 5).

Fault-tolerant constructions of read/write registers can also be used modularly in randomized constructions of higher-level objects, such as consensus objects [AH90, CIL87, Her91, SSW91]. (Since deterministic construction of consensus objects from read/write shared memory is impossible [LAA87, Her91], the latter constructions rely on randomization to reach consensus within finite expected time.) These randomized constructions will behave correctly so long as the assumed fault bound is not exceeded in any of the modular read/write register components.

However, replacing the read/write registers in such randomized algorithms to produce reliable implementations of higher-level objects will not, in general, result in strongly wait-free implementations: To be strongly wait-free, such constructions must at least terminate if the fault bound is exceeded. But when the number of faulty read/write registers in the randomized implementation of a consensus object exceeds the assumed bound, the runs of the algorithm may no longer satisfy the required probabilistic properties. For example, although calls to the no-longer reliable read/write registers must still return (since the component constructions are strongly wait-free), they could return erroneous values so as to *force* an infinite, undecided run, which in the fault-free case must have probability 0. (A shared coin could always appear to return "heads".) Then calls to the randomized consensus object would remain pending forever, exhibiting a fault mode not exhibited by the components. (The processes cannot use a counter to detect such failures, because every randomized construction must have low-probability runs of any length.)

5. TEST&SET OBJECTS

Atomic registers do not provide a very strong memory primitive – even the simple task of two-process consensus is impossible with just atomic registers and requires a stronger primitive such as test&set [Her91, LAA87]. In this section we provide constructions of test&set objects from similar objects, some of which may be faulty.

A *test&set register* is a concurrent object accessible by the processes sharing it through the operations `test&set` and `reset`. The sequential specification of the object is most simply understood as operations on a binary register, initialized to 0. The `test&set` operation atomically reads the register, writes 1 into it, and returns the value read. The `reset` operation writes 0. If the object is faulty, the failure operations `fault(0)` and `fault(1)` have the obvious effect (of writing 0 and 1 respectively into it). The processes are constrained in their use of the `reset` operation—a process should invoke the `reset` operation only if the object has been set; that is, since the last `reset` was invoked, a `test&set` has returned 0. If the processes violate this *well-formedness condition*, then the object may exhibit arbitrary behavior. (This restriction is consistent with applications of test&set in operating systems. See for example [PS85], Section 5.2.2. In this paper, we do not investigate fault-tolerant constructions of test&set objects in the absence of this restriction.)

The main construction of this section, given in Figure 5, implements an *n-process test&set* object from similar registers, one of which may be faulty. By Theorem 2, this construction can be extended to a construction that tolerates f faulty registers. This construction is built upon a *single-use test&set object*, defined as follows:

DEFINITION 3. *A single-use test&set object, is an object in which each process is allowed to perform only one* `test&set` *operation and which has no* `reset` *operation.*

That is, the single-use sequential specification is that the first `test&set` *operation returns* 0 *and all others return* 1*.*

Terminology: Throughout this section the following distinction between the *primitive objects* and *high-level objects* is made: The unreliable primitive test&set registers which are the elementary building blocks in all the constructions, are referred to as *primitive registers* and the operations on them are `p-test&set` and `p-reset`. All the constructed objects will be referred to as *high-level objects.* We also refer to the test&set object as a *multi-use* test&set, to distinguish it from the single-use primitives. Throughout, we say that a `test&set` operation has *won* the test&set, or is *successful*, if the response to the operation is 0 (i.e., the process succeeded in changing the value from 0 to 1); if the response is 1, the operation is said to have *lost*.

5.1 Single-use two-process test&set

Consider the following simple strategy to construct a single-use, 2-process test&set object. Each process executes the test&set operation on all the registers in an array, and considers itself successful if and only if it succeeds on the majority of the registers. This strategy does not tolerate even a single faulty register. Each of two processes might win exactly half of the correct registers, and then the faulty register could cause both processes to decide they succeeded or both to decide they have lost.

On the other hand, the majority strategy on an array of just three test&set registers, at most one of which is faulty, has the following two properties:

(1) If there are no faults, then the majority strategy works correctly, and
(2) If only one process at a time is attempting to perform a test&set, then again it works correctly.

The construction in Figure 2 (a schematic is part of Figure 3) makes repeated use of these properties.[3]

THEOREM 9. *There is a strongly wait-free, single-use, two-process test&set construction from 7 test&set registers, 1 of which may be* ∞*-faulty (Figures 2 and 3):*

$$\mathbf{CONST}(single\_use\_2\_process\_T\&S, (1,\infty), single\_use\_2\_process\_T\&S) \leq 7.$$

PROOF. The sequential specification of single-use, two-process test&set is very simple. The first `test&set` operation returns 0 (winning) and the second returns 1 (losing). We will demonstrate that every run of our protocol can be serialized to meet the specification. We start by proving three properties on all runs of the protocol. Then these properties are used to specify a correct serialization.

Denote the first action of each process operation by `request`, and the last action `return(0)` (if it wins in the construction) or `return(1)` (if it loses in the construction). Throughout this proof we denote by *p-win* (*p-lose*) an operation that wins (loses) on one of the seven primitive test&set registers. The two processes are denoted P_1 and P_2. The existence of a serialization follows from the following three properties:

[3] A construction which reduces the number of primitive objects needed from seven to six primitive registers is devised in [MO93].

```
shared A[1..3], B, C[1..3]: primitive test&set registers, initially 0

function single_use_2_process-test&set                    % return 0 (win) or 1 (lose)
    sum := 0
    for i := 1 to 3 do sum := sum + p-test&set(A[i]) od
    if sum ≥ 2 then if p-test&set(B)= 0 then return 1 fi fi   % lost A and won B: lost
    sum := 0                                    % won A, or lost A and B: continues to C
    for i := 1 to 3 do sum := sum + p-test&set(C[i]) od
    if sum ≥ 2 then return 1 else return 0 fi
end_function
```

Fig. 2. Single-use two-process test&set using seven test&set's, one of which may be faulty.

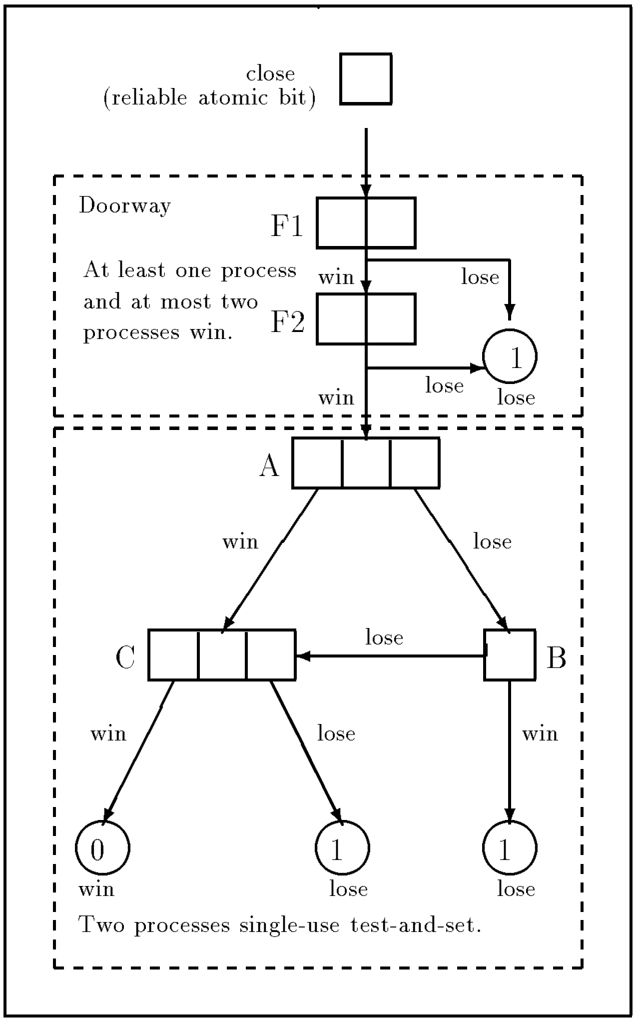

Fig. 3. Schematic of single-use test&set for n processes.

1. *There is at most one* `return(0)` *action*: A single_use_2_process-test&set operation returns 0 only if it read 0 in at least two of the three test&set registers in C (Lines 5 and 6). If there is no fault in C, then it cannot be that both P_1 and P_2 have read 0 in at least two registers of C. If on the other hand there is a fault in C, then both registers A and register B are correct. In this case at most one process, say P_1, p-wins in A. Therefore, the other process, here P_2, p-wins at B and loses without touching any primitive test&set in C (Lines 2 and 3). Thus, it cannot happen that both processes win and perform a `return(0)` action.

2. *There is at most one* `return(1)` *action:* Consider the first `return(1)` action and assume w.l.o.g. that it is by process P_1. If P_1 lost by p-losing two out of the three primitive registers in competition C, then it must be that both P_1 and P_2 have reached C. This could happen only if there was a fault in a register in A or in B, and thus the faulty primitive test&set is not in C. But if there is no fault in C, then P_2 must have p-won or will p-win at least two of the primitive test&sets in C, and terminate as a winner in the overall construction. If, on the other hand, P_1 lost by p-winning at B, then P_2 has visited or will visit C alone and terminates as a winner in the overall construction.

3. *Any* `return(1)` *action is preceded by a* `request` *action by the other process:* A losing process must have p-lost two out of the three primitive registers in at least one of the competitions A and C. But at most one of these two p-losses can be attributed to a faulty primitive test&set—the other p-loss must be because another process has p-won in that primitive. Thus, the other process must have taken at least one step.

By these properties any run is serialized as follows: The losing process, say P_1 is serialized at the time of its `return(1)` action and P_2 as winning at the time of its `request` action. If there is only one process active, it wins and the operation is serialized at the time of its `request` action. These serializations obey the requirements of the serial specification.

The construction is obviously strongly wait-free, as each of the two processes performs at most 7 operations on the primitive test&set registers before terminating, and these are assumed to be wait-free. □

5.2 Single-use n-process test&set

The construction in the last section works correctly only in a system with at most two processes. In this subsection we introduce a mechanism that selects at most two out of n processes, for any $n > 1$. This new mechanism is used as a "doorway" to the construction of the previous subsection, which selects exactly one winner out of the two.[4]

If there are no faults, then two primitive test&set registers can be used to ensure

[4] The mechanism introduced in this subsection is very similar in its behavior to a 2-exclusion algorithm. (2-exclusion [ADG+94, DGS88, FLBB79, FLBB85] is a generalization of mutual exclusion in which up to two processes may be in the critical section at the same time, but no more.) The difference between 2-exclusion and the mechanism presented here is as follows. In a 2-exclusion algorithm, if two or more processes attempt to enter their critical sections, then eventually at least two processes, out of n that try, enter the critical section. In contrast, in the "doorway" construct, if two or more processes try to enter then at most two and at least one enter the critical section. (In this application the critical section is the seven object construction from the previous subsection.)

that at most two processes pass the doorway, by having each `p-test&set` the two registers, and pass if successful in at least one. However, if one of the two registers is faulty, then at least one process and as many as n may enter (because the faulty one may let everybody win). To overcome this problem we require the processes to go through two such gateways. That is, we maintain two pairs, F_1 and F_2, of primitive test&set registers (see function `doorway` in Figure 4). To pass the doorway each process must successfully `p-test&set` one register in each of the two pairs, otherwise the process returns 1 for its test&set operation. (In the OR operations in the function `doorway` (Lines 1 and 2) only one successful clause needs to be executed. That is, if the first clause is true, then the second need not be executed—however, the construction is correct with either interpretation.)

The doorway implementation is given in Figure 4.[5] The `doorway` guarantees that at most two processes enter the two process construction; however, it could be that one process is blocked in the doorway, while a faulty primitive allows a later arriving process to pass it into the two process construction (left as an exercise). This enables a scenario in which one process loses, and only after it exits, the eventual winner starts. This violates the linearizability requirement (operations must be serialized within the time interval in which they are active), and can be resolved in several ways. The simplest solution is to add a reliable multi-writer/multi-reader atomic bit, *close* as is suggested in [AGTV92] (which can be constructed from unreliable primitive objects as described in Section 4). Any process first reads the *close* bit, and if *close* = 1 it returns 1 and exits; otherwise it sets *close* to 1 and continues into the doorway. This ensures that once a process has lost, no other process can later start and win.

```
shared F1[1..2], F2[1..2]: T&S registers, initially 0
        close: reliable atomic on {0,1}, initially 0

function s-test&set                                  %return 0 (win) or 1 (lose)
1:  if close = 1 then return(1) fi                   %exit if construct is closed
2:  close := 1                                       %close the construct and continue
3:  if doorway then return(single_use_2_process-test&set) else return(1) fi
                                                                    %from Figure 2
end_function

function doorway                            %return true (passed) or false (lost)
1:  if (p-test&set(F1[1]) = 0) OR (p-test&set(F1[2]) = 0)           %see note a below.
2:  then_if (p-test&set(F2[1]) = 0) OR (p-test&set(F2[2]) = 0)
        then return(true) fi fi             %won at least one in F1 and one in F2: pass
3:  return(false)                           %won at neither F1 nor at F2: do not pass
end_function
```

Fig. 4: Single-use n-process test&set using eleven test&set's, one of which may be faulty, and a reliable, atomic read/write bit (see also Figure 2).

[a] In the OR operations only one successful clause needs to be executed. That is, if the first clause is true, then the second need not be executed—however, the construction is correct with either interpretation.)

[5] In the code, the execution of a return command terminates (exits) the operation.

THEOREM 10. *There is a strongly wait-free, n-process, single-use, test&set construction from* 11 *test&set registers,* 1 *of which may be* ∞*-faulty, and a single reliable read/write bit (Figures 2, 3 and 4).*

PROOF. It is argued above that at least one and at most two processes leave the doorway and enter the single-use, two-process test&set. By Theorem 9, exactly one of these will be the winner.

We serialize the winner with it's high-level request, and the losers with their returns. The winner's `read` of *close* $= 0$ in line 1 of `s-test&set` must precede either a loser's `read` of *close* $= 1$ in line 1 of `s-test&set`, or (if the loser also `read` *close* $= 0$ in line 1 of `s-test&set`), the loser's `write` of *close* $:= 1$ in line 2 of `s-test&set`. Regardless, the winner is serialized before all the losers.

As in the previous construction, this one is obviously strongly wait-free, as each of the n processes performs at most 2 operations on the primitive read/write register and 11 operations on the primitive test&set registers before terminating, and these are assumed to be wait-free. □

5.3 Multi-use n process test&set

This section extends the ideas of the previous construction, resulting in a construction of a multi-use, n process test&set object. A series of lemmas lead to the proof of the following theorem, in Section 5.3.4.

THEOREM 11. *There is a strongly wait-free n-process multi-use test&set construction from* $n + 10$ *test&set registers one of which may be* ∞*-faulty and* $3n + 13$ *reliable atomic registers (Figures 5 and 6).*

5.3.1 *Overview of the construction.* The single-use test&set provides a simple lock: whichever process wins the test&set (reads 0 while setting to 1) holds the lock. However, without the `reset` operation the lock cannot be released. The construction in Figure 5 implements a multiple-use test&set, that is, one that can be `test&set` and `reset`, and the picture in Figure 6 describes the shared data structure.

A simple way to add a reset operation to a high-level implementation of a test&set object would be to simply reset the constituent primitive registers. Unfortunately, in the constructions given above, resetting the primitive registers would interfere with the high-level `test&set` operations of other processes and could cause incorrect behavior of the high-level test&set object.

The first observation towards an implementation of a multi-use test&set object is that in the n process single-use construction of the previous subsection, if a process tries to perform a high-level `test&set` an arbitrary number of times, it is guaranteed that it will lose in all its repeated attempts (unlike the two process single-use construction where a process may try only one time). Given this observation, a simple solution that uses an unbounded number of primitive test&set registers and read/write bits would work as follows: divide the collection of test&set registers into bundles of 11 test&set registers and one bit, and enumerate the bundles $1, 2, \ldots$. Use one atomic read/write register as a pointer to the bundle that is currently in use. Once the high-level test&set is set, it can be reset by simply incrementing the pointer. Processes that have started the high-level `test&set` operation before the reset would lose, since by the above observation they are blocked in the old copies, where a winner was already declared.

This unbounded solution can be modified to use $11n$ primitive `test&set` registers (one of which might be faulty) and $O(n)$ reliable atomic registers, by the following observation: All the bundles, except n of them, could be recycled, since no process will ever access them again. This solution will require that each process declare which copy it is using and that the resetter find a copy not in use.

A still more efficient solution carries the recycling idea one more step. Instead of duplicating and recycling the bundles, we recycle the primitive test&set registers (see Figure 5). This is possible by requiring each process to declare which primitive test&set *register* it is using at each step that it is taking (Line 9 of function `test&set`). The resetter will now search for 11 *primitive registers* that are not in use (Lines 7-10 of function `reset`) and will compose them into a new 11-piece n-process single-use object. Thus, rather than giving a pointer, it will have now to provide 11 pointers ($current[1..11]$ in the code) for the newly composed 11-piece object. (The read/write bits are replicated and reused similarly.) In this way a solution is obtained that uses a total of $n + 10$ primitive `test&set` registers (one of which might be faulty) and $O(n)$ reliable atomic registers.

To ensure that while the resetter is composing a new 11-piece copy no process is switching between registers, the resetter will set a multi-writer/multi-reader *stop* bit telling all other processes to lose and exit (Line 1 of `reset`). Every process tests the *stop* bit before each primitive operation (Lines 3 and 10 of `test&set`). If the bit is set, the process aborts its high-level `test&set` operation and returns with 1. The *stop* bit by itself is not sufficient because a process performing a high-level `test&set` could be suspended during the reset period, and not observe the *stop* bit being set. Such a process could later access a primitive test&set register that it should not. Therefore another bit, called $restart[p]$, is associated with each process p to detect if a `reset` was taking place while it was not checking the *stop* bit. The restart bit works as follows; when starting a high-level `test&set` operation each process assigns 0 to its restart bit (Line 1 of `test&set`), and before finishing a high-level `reset` a resetter assigns 1 to the restart bits of all processes (Line 14 of `reset`). Now, before performing each primitive operation a process checks whether its restart bit is still 0 (Lines 4 and 11 of `test&set`). If not, it aborts its high-level `test&set` operation and returns with 1. The combination of the stop bit and the restart bits ensures that the primitive objects chosen by the resetter are not in use, and that they will not be accessed until they are themselves reset and the resetter assigns 0 to the *stop* bit.

The high-level `test&set` operation is now performed on the chosen 12 primitives by simulating the `s-test&set` operation (from Figure 4). The simulation proceeds in iterations, such that in each iteration one primitive test&set register is `p-test&set` (see Figure 5). A local state variable, *state*, records the history of the `s-test&set` simulation. This variable contains eleven fields, each corresponding to one of the eleven primitive test&set registers in `s-test&set`, as in Figure 4. The values of the fields are $\perp$, 0 or 1, and are initialized to $\perp$, indicating that no calls to the primitive registers have been made. Each call to a primitive test&set register is made to simulate a call in `s-test&set`, and the return value is assigned to the corresponding entry in *state*.

Two functions are defined on *state*. The function `result` returns $\perp$ if the values in *state* indicate that the simulation of `s-test&set` is incomplete. It returns 0 (or 1) if the values indicate that the simulation is complete and returned 0 (or 1). The function `next` examines the values in *state* and returns the index (between 1 and 11) of the primitive register which should be called next in the simulation of `s-test&set`, given the results currently recorded.

5.3.2 *Well-formed executions.* The use of the primitive test&set registers in the construction is designed to work correctly only in constrained environments, in which a `p-reset` operation is invoked only if a `p-test&set` has returned successfully since the last invocation of `p-reset`. The constructions' calls to these primitives depend, in turn, on their behaving properly. This complicates the proof, which must avoid circular reasoning.

Accordingly, a system execution is *well-formed* for a multi-use test&set register (either a faulty primitive or high-level reliable construction) if the number of calls to the `reset` operation is either equal to or one less than the number of successful returns for calls to the `test&set` operation, in every prefix of the execution. That is, the sequence of successful returns from `test&set` and of calls to `reset` operations strictly alternate, beginning with a successful return from a `test&set` operation. Hence, in well-formed executions the `reset` and `test&set` operations can all be serialized so that successful `test&set`'s alternate with `reset`'s, and every unsuccessful `test&set` is ordered after a successful `test&set` and before the next `reset`. In environments which violate well-formedness for a test&set register, calls may not terminate and can be serialized arbitrarily. (That is, anywhere within their interval, without regard for semantics of the operations.)

Hence, regardless of whether an execution is well-formed, we have defined a serialization and can consider calls to primitive test&set registers as single atomic events. Given an execution of the high-level reliable multi-use construction, we reason about this serialization, and argue that the execution is well-formed for each primitive. Then, we conclude that the atomic events have the appropriate semantics (are ordered as described in the preceding paragraph).

The fact that well-formedness is maintained for each primitive test&set object is proven by induction on an execution α of the high-level construction. Given a prefix, α', of the execution that is well-formed for each primitive, we will want to reason that an extension of the prefix is also well-formed for each primitive. To do so, we need to use the semantics of (well-formed) executions of these primitives. The semantics are in turn encoded in the serialization of each primitive. Hence, we need to relate serializations of α and of α'. That is, we want to reason, not about the execution, α, but a serialization, β, of the execution. In the induction step, we have a well-formed prefix α' of the execution. Any serialization β' of α' obeys the semantics of the test&set primitives. To use this fact to reason about β and hence α, we choose the serialization β so that it has β' as a prefix. The following technical lemma assures us this is possible:

LEMMA 5.1. *Let α be an execution of a system containing a test&set register, r. Let α' be the longest prefix of α that is well-formed for r. Then there is a serialization of α' which is a prefix of a serialization of α.*

PROOF. If α is well-formed then $\alpha = \alpha'$ and we are done. Otherwise, let β be a serialization of α and β' be a serialization of α'. The serialization we want is $\beta'(\beta - \beta')$. (That is, append to β' the subsequence of events in β that are not in β'. This may move the serialization of some operations that are pending in α', but

```
Protocol for process p:
shared    PTST: array[1..(n + 10)] of primitive t&s with
              p-test&set and p-reset operations
          stop: reliable atomic on {0, 1}                        %written by winner, initially 0
          close[1..n]: reliable atomic on {0, 1}, initially 0    %written and read by everybody
          my_next[1..n]: reliable atomic on {1, ..., n + 10}     %written by owner
          restart[1..n]: reliable atomic on {0, 1}               %initially 1
          current[0..11]: reliable atomic on {1, ..., n + 10}    %initially 1, 2, ..., 11
local     state[1..11]: local on {⊥, 0, 1}, initially ⊥
          inuse[1..(n + 10)]: local on {0, 1}

function test&set
    restart[p] := 0                                  %start by ensuring that no one already won
    my_next[p] := current[0]                         %indicates which (close) bit is used
    if stop = 1 then return(1) fi                    %check global stop bit,
    if restart[p] = 1 then return(1) fi              %personal restart bit,
    if close[my_next[p]] = 1 then return(1) fi       %and simulation's close bit

    close[my_next[p]] := 1                           %close the construction and continue
    for i = 1 to 11 do state[i] := ⊥ od
    while result(state) = ⊥ do                       %do s-test&set step by step
        my_next[p] := current[next(state)]           %indicates which t&s register is used next
        if stop = 1 then return(1) fi                %check that no one already won
        if restart[p] = 1 then return(1) fi
        state[next(state)] := p-test&set(PTST[my_next[p]])
    od
    return(result(state))
end_function

function reset
    stop := 1                                        %prevent anyone from winning
    for i = 1 to n + 10 do inuse[i] := 0 od          %mark registers which
    for i = 1 to n do if i ≠ p then inuse[my_next[i]] := 1 fi od   %might be corrupted
    j := 1
    while inuse[j] = 1 do j := j + 1 od              %is register j used?
    current[0] := j
    j := 1
    for i = 1 to 11 do                               % finds 11 PTSRs
        while inuse[j] = 1 do j := j + 1 od
        current[i] := j
    od
    close[current[0]] := 0                           %re-open the construct
    for i = 1 to 11 do p-test&set(PTST[current[i]]);
        p-reset(PTST[current[i]]) od                 %reset the t&s registers
    for i = 1 to n do restart[i] := 1 od             % make everyone start again
    stop := 0
end_function
```

function `result`: returns ⊥ if the values in *state* indicate that the simulation of `s-test&set` is incomplete. Otherwise it returns the value that is returned by the simulation.

function `next`: examines the values in *state* and returns the index (between 1 and 11) of the primitive register which should be accessed next in the simulation of `s-test&set`.

Fig. 5. Multi-use test&set.

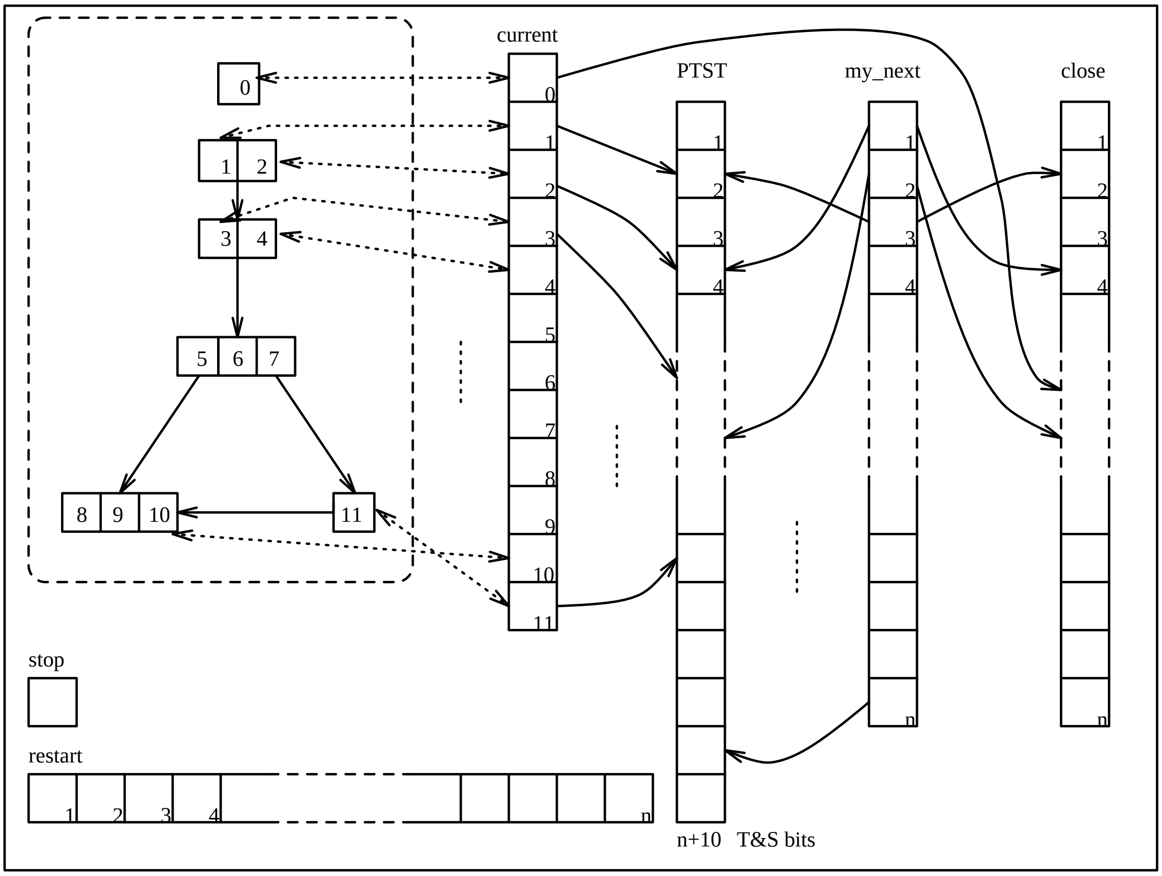

Fig. 6. Shared data structure for the multiple-use test&set construction.

serialized early, until after β'.) $\square$

By restricting the class of serializations as described in Lemma 5.1, we can consider calls to primitive test&set objects, within executions of the reliable multi-use construction, as consisting of single atomic events. (Because linearizability is a local property [HW87], it suffices to consider each primitive separately in Lemma 5.1.) Moreover, in any well-formed prefix of the execution, these atomic events obey the test&set semantics.

5.3.3 *Properties of the construction.* Each call to the `reset` function begins with an assignment, "$stop := 1$" (`reset` Line 1), and shortly before terminating, assigns 1's to the restart bits (`reset` Line 14).

Definition 4. *For a process p, execution α, and call to the* `reset` *function within α, we say that the* `reset` *function call is* active *for p within the subinterval of α that begins with the initial assignment of the* `reset` *function, $\langle stop := 1\rangle$ (*`reset` *Line 1), and ends with the assignment $\langle restart[p] := 1\rangle$ (*`reset` *Line 14).*

We say that calls to `reset` are *sequential* in an execution α if each call executes its final assignment to *stop* (`reset` Line 15) before the next call begins. For now we state lemmas for executions which are assumed to satisfy this property. Note, in particular, that when calls to `reset` are sequential, $stop = 1$ is invariant during `reset` function calls, up to the final atomic assignment $stop := 0$ (`reset` Line 15).

LEMMA 5.2. *Let α be an execution in which calls to* `reset` *are sequential, such that the two steps* $\langle \text{restart}[p] := 0\rangle$ *(*`test&set` *Line 1), and* $\langle \textbf{if}\, \text{stop} = 1\rangle$ *(*`test&set` *Line 3 or 10), are a subsequence of α from a single call to* `test&set` *by process p. If a call to the* `reset` *function is active for p at any point between the* $\langle \text{restart}[p] := 0\rangle$ *step and the* $\langle \textbf{if}\, \text{stop} = 1\rangle$ *step, then either* $\text{stop} = 1$ *in the* **if** *test (*`test&set` *Line 3 or 10), or* $\text{restart}[p] = 1$ *in the next step of p which is* $\langle \textbf{if}\, \text{restart}[p] = 1\rangle$ *(*`test&set` *Line 4 or 11).*

PROOF. Since a call to the `reset` function is active for p between the $\langle restart[p] := 0\rangle$ step and the $\langle \textbf{if}\ stop = 1\rangle$ step, at that point $stop = 1$. If this is still true during the $\langle \textbf{if}\ stop = 1\rangle$ step, we are done. So assume that $stop = 0$ during the $\langle \textbf{if}\ stop = 1\rangle$ step. It follows that some assignment to $stop$ occurs after the point in which the `reset` function is active for p, and before the $\langle \textbf{if}\ stop = 1\rangle$ step. Since only calls to `reset` assign to $stop$, as their last step (`reset` Line 14), and `reset`'s are sequential, the `reset` which is active for p must assign $stop := 0$ between the $\langle restart[p] := 0\rangle$ step of `test&set` and the $\langle \textbf{if}\ stop = 1\rangle$ step. This in turn implies the assignment $\langle restart[p] := 1\rangle$ occurs between the $\langle restart[p] := 0\rangle$ step and the $\langle \textbf{if}\ stop = 1\rangle$ step. Only p assigns 0 to $restart[p]$ (`test&set` Line 1); it follows that $restart[p] = 1$ during the $\langle \textbf{if}\ restart[p] = 1\rangle$ step. □

LEMMA 5.3. *Let α be an execution in which calls to* `reset` *are sequential.*

(1) *Let* $\langle \text{restart}[p] := 0\rangle$, $\langle$`p-test&set`$(\text{PTST}[\text{ind}])\rangle$, *where* $1 \le \text{ind} \le n + 10$, *be a subsequence of α from a single call to* `test&set`*, by process p (Function* `test&set` *Lines 1 and 12). No call to* `p-reset`$(\text{PTST}[\text{ind}])$ *occurs after* $\langle \text{restart}[p] := 0\rangle$ *and before* $\langle$ `p-test&set`$(\text{PTST}[\text{ind}])\rangle$.

(2) *Let* $\langle \text{restart}[p] := 0\rangle$, $\langle$`read`$(\text{close}[\text{ind}]) = 1\rangle$ *be a subsequence of α from a single call to* `test&set`*, by process p (Function* `test&set` *Lines 1 and 5). No assignment of* 0 *to* close[ind] *(Function* `reset` *Line 11) occurs after* $\langle \text{restart}[p] := 0\rangle$ *and before* $\langle$`read`$(\text{close}[\text{ind}]) = 1\rangle$.

(3) *Let* $\langle \text{restart}[p] := 0\rangle$, $\langle$`read`$\text{close}[\text{ind}] = 0\rangle$, $\langle \text{close}[\text{ind}] := 1\rangle$ *be a subsequence of α from a single call to* `test&set`*, by process p (Function* `test&set` *Lines 1, 5, and 6). No assignment of 0 to* close[ind] *occurs after* $\langle \text{restart}[p] := 0\rangle$ *and before* $\langle \text{close}[\text{ind}] := 1\rangle$.

PROOF. We prove the first case—the others are similar. The call to the `test&set` function by p includes the following sequence of steps:

$restart[p] := 0$	(`test&set` Line 1)
$my\_next[p] := ind\ (= current[$`next`$(state)])$	(`test&set` Line 9)
$stop = 0$	(`test&set` Line 10)
$restart[p] = 0$	(`test&set` Line 11)
`p-test&set`$(PTST[ind])$	(`test&set` Line 12)

By the previous lemma, no call to `reset` is active for p at any point between $\langle restart[p] := 0\rangle$ and $\langle stop = 0\rangle$.

Consider a call to `reset` which is active after $\langle stop = 0\rangle$ and which calls `p-reset` (`reset` Line 13) before the end of `p-test&set`$(PTST[ind])$. This call to `reset` reads $my\_next[p] = ind$ (`reset` Line 3) between $\langle stop = 0\rangle$ and $\langle$`p-test&set`$(PTST[ind])\rangle$. Hence, it will mark this register in use, and will not run `p-reset` on $PTST[ind]$.

It follows that no `reset` which runs `p-reset`$(PTST[ind])$ is active for p after $\langle stop = 0\rangle$ and before the end of `p-test&set`$(PTST[ind])$. The first part of the lemma follows. □

LEMMA 5.4. *Let α be an execution in which calls to* `reset` *are sequential. In α, no primitive test&set register,* PTST[ind]*, is* `p-reset` *concurrently with any other call to register* PTST[ind].

PROOF. Since calls to `reset` are sequential, any two `p-reset` operations are sequential. The previous lemma implies that calls to `p-test&set`$(PTST[ind])$ cannot be concurrent with calls to `p-reset`$(PTST[ind])$. □

LEMMA 5.5. *Let α be an execution in which calls to* `reset` *are sequential. Then for every primitive test&set register,* PTST[ind]*, α is well-formed for* PTST[ind].

PROOF. A simple induction on the prefixes of α, using Lemma 5.4.

That is, let $\alpha'\pi$ be a prefix of α, where α' is well-formed for $PTST[ind]$. By induction and the semantics of the register $PTST[ind]$, $\alpha'\pi$ is well-formed if π is any event other than the request for a reset of $PTST[ind]$. This request occurs within a call to `p-reset`$(PTST[ind])$, (Line 13) which has just run a `p-test&set`$(PTST[ind])$ operation (Line 12). By Lemma 5.4, all other operations on $PTST[ind]$ in α' terminated before that `p-test&set`$(PTST[ind])$ began. Hence, the `p-test&set`$(PTST[ind])$ returned successfully if the number of other successful `p-test&set`$(PTST[ind])$ operations in α' is equal to the number of calls to `p-reset`$(PTST[ind])$, and returned unsuccessfully if the number of other successful `p-test&set`$(PTST[ind])$ operations in α' is one more than the number of calls to `p-reset`$(PTST[ind])$. Hence, the new request does not violate well-formedness. □

DEFINITION 5. *For any execution α and any call to* `test&set` *by process p in α, the* `reset` preceding that call to `test&set` *is the call to* `reset` *which executes the step* $\langle \text{restart}[p] := 1\rangle$ *(Line 14) most recently before the first step of the call to* `test&set` *(that is, Line 1* $\langle \text{restart}[p] := 0\rangle$*). If no such* `reset` *exists, we say the call to* `test&set` *has no preceding* `reset`.

LEMMA 5.6. *Let α be an execution in which calls to* `reset` *are sequential.*

(1) Suppose `p-test&set`(PTST[ind]) *is called during a call to* `test&set` *by process p in α. Then* ind *was written to* current *by the* `reset` *preceding the call to* `test&set`*, (or was in the initial state of* current*, if there is no such* `reset`*) and no later* `reset` *assigns* ind *to* current *until after the call to* `p-test&set`*(*PTST[ind]*).*

(2) Suppose close[ind] *is read during a call to* `test&set` *by process p in α. Then* ind *was written to* current[0] *by the* `reset` *preceding the call to* `test&set`*, (or was in the initial state of* current[0]*, if there is no such* `reset`*) and no later* `reset` *assigns* ind *to* current[0] *until after the* `read` *and subsequent* `write` *of* close[ind].

PROOF. We prove the first case—the second is analogous. The call to `test&set` includes the following sequence of steps:

$restart[p] := 0$	(`test&set` Line 1)
$my\_next[p] := ind\ (= current[\texttt{next}(state)])$	(`test&set` Line 9)
$stop = 0$	(`test&set` Line 10)
$restart[p] = 0$	(`test&set` Line 11)
`p-test&set`$(PTST[ind])$	(`test&set` Line 12)

By Lemma 5.2, no `reset` is active for p at any point between the steps $\langle restart[p] := 0\rangle$ and $\langle stop = 0\rangle$. As in the previous lemma, any `reset` which is active for p after $stop = 0$ either sees $my\_next[p] = ind$, and will not write ind to $current$, or reads $my\_next[p]$ after the call to `p-test&set`$(PTST[ind])$, and so will not assign anything to $current$ until after this call. □

Lemma 5.7. *Let α be an execution in which calls to* `reset` *are sequential. Let P and Q be the corresponding sets of primitive registers (test&set and read/write) accessed in two different calls to* `test&set`. *Either the two calls to* `test&set` *have the same preceding* `reset`, *or every primitive register (test&set or read/write) in $P \cap Q$ is either* `p-reset` *or assigned 0 between the first call and the second.*

Proof. Suppose $x \in P \cap Q$ and that the two calls to `test&set` have different preceding `reset` operations. We assume that the primitive is a test&set register, $x = PTST[ind]$, —the proof for the read/write primitives is analogous. Denote the two calls to `p-test&set(`$PTST[ind]$`)` by π and ϕ, with π occurring before ϕ in α, where π is an operation of process i, and ϕ is an operation of process j. We have the following two sequences of operations in α:

1_i)	$restart[i] := 0$	(`test&set` Line 1)
2_i)	$my\_next[i] := ind$	(`test&set` Line 9)
3_i)	$stop = 0$	(`test&set` Line 10)
4_i)	$restart[i] = 0$	(`test&set` Line 11)
5_i)	$\langle$`p-test&set`$(PTST[ind])\rangle \equiv \pi$	(`test&set` Line 12)
1_j)	$stop := 1$	(`reset` Line 1)
2_j)	$my\_next[i] \neq ind$	(`reset` Line 3)
3_j)	`p-reset`$(PTST[ind])$	(`reset` Line 13)
4_j)	$restart[i] := 1$	(`reset` Line 14)
5_j)	$\langle$`p-test&set`$(PTST[ind])\rangle \equiv \phi$	(`test&set` Line 12)

Here, $1_i - 5_i$ are steps from the call to `test&set` that includes π, and steps $1_j - 4_j$ are steps from the `reset` operation that precedes the call to `test&set` that contains ϕ (there must be such by the assumption).

To establish the lemma, it suffices to show that line 3_j, $\langle$`p-reset`$(PTST[ind])\rangle$, comes after π. Suppose it came before π (5_i). Since line 1_i, $\langle restart[i] := 0\rangle$, precedes the assignment to $restart[i]$ in line 4_j, and the `reads` in lines 3_i and 4_i return 0, Lemma 5.2 implies that line 1_j is ordered after line 3_i. Since in turn line 3_j is ordered before line 5_i, the `read` of $my\_next[i]$ in line 2_j should return ind, a contradiction. □

Definition 6. *Given an execution α of the construction, any set of calls to* `test&set` *which have the same preceding call to* `reset` *(or which all have no such preceding call) are called a* collection *of calls to* `test&set`.

Lemma 5.8. *Let α be an execution of the construction. Then α satisfies the following properties:*

(1) *α is well-formed for the reliable high-level multi-use test&set object. Moreover, each* `reset` *operation finishes its final assignment to the* stop *bit before the next successful* `test&set` *operation returns.*

(2) *calls to* `reset` *are sequential,*

(3) *the* `reads` *and* `writes` *of the* close *bits and calls to* `p-test&set` *made by each collection of calls to* `test&set` *are a prefix of a run of the* `su-test&set` *construction.*

PROOF. Note first that the first invariant implies the second.

The proof is by induction on the length of the execution, with a trivial basis. Consider a finite execution $\alpha'\pi$, where α' satisfies the conditions of the lemma, and π is the next step by the construction. We must show that the first and third invariants are maintained in $\alpha'\pi$.

—Suppose the next step is a call to `reset`. By induction, α' is well-formed for the reliable high-level multi-use test&set object. Thus, the environment is required to preserve this property. The first invariant follows, and the third is immediate.

—Suppose the next step is a step of a primitive register. The first invariant is immediate. By induction, calls to `reset` are sequential in $\alpha'\pi$. Lemma 5.5 implies that $\alpha'\pi$ is well-formed for the primitive, and Lemma 5.1 implies the sequence of steps of the primitive in $\alpha'\pi$ obey test&set semantics. Lemmas 5.6 and 5.7 imply the second invariant.

—Suppose the next step is the successful return, π, of a call to `test&set` by process p. We must show there is a `reset` call which runs through its final assignment to *stop*, since the last such successful return, ϕ, of a call to `test&set`. By induction, the properties of `su-test&set` and the last invariant, each collection of calls to `test&set` can have at most one successful call. Hence, the two calls to `test&set`, which return with ϕ and π, are not from the same collection. (That is, they have different preceding calls to `reset`.) Let r_π and r_ϕ be the corresponding calls to `reset` preceding π and ϕ. (That is, r_π and r_ϕ consist of the subsequences of events of α for the two calls.) Since calls to `reset` are sequential, ϕ precedes π, and well-formedness holds for calls to `reset`, it follows that the beginning of r_π follows ϕ.

  Since π is the successful return of a call to `test&set`, the third invariant implies this call simulated a successful call to `su-test&set`, and hence must read $stop =$ 0 at some point. By Lemma 5.2, r_π is not active for p during some interval beginning with the first step of the call to `test&set`. It follows that the final assignment to *stop* in r_π must precede π. □

5.3.4 *Proof of Theorem 11.* Let α be an arbitrary, finite run of the construction. Without loss of generality, assume all operations of α are complete. (Recall the construction is strongly wait-free, if calls to primitives are strongly wait-free. By Lemmas 5.8 and 5.5, α is well-formed for each primitive, and so calls to primitives are strongly wait-free.)

We must show that each `test&set` and `reset` operation can be appropriately serialized. We proceed as follows. Each successful `test&set` operation is serialized at the point of its assignment to *restart* (`test&set` Line 1), which is obviously within its interval. Each `reset` is serialized either at its final assignment to *stop* (Line 15), or just before the serialization of the successful `test&set` operation which returns next, whichever is earlier. Lemmas 5.8 and 5.2 imply this is within the interval of the `reset`, indeed, no earlier than its first assignment to the array *restart* (Line 14 of the `reset` function). By Lemma 5.8, the sequence of serializations is of the appropriate form: "successful `test&set`, `reset`, successful `test&set`, `reset`, ..."

It remains to show that given this partial serialization, the unsuccessful `test&set` operations can all be serialized, within their interval, after a successful `test&set` and before the next `reset`.

We consider the unsuccessful calls to `test&set` depending upon how they return. It suffices to show that these calls cannot begin and end between the serialization of a `reset` and before the serialization of the next successful `test&set`.

Suppose an unsuccessful call returns because of losing the test on *stop* (Line 3 or 10 of `test&set`). We argue that the call can be serialized at the point of this `read` of $stop = 1$. A call to `reset` is in progress when this `read` occurs. If that call to `reset` is serialized at the point where it assigns $stop := 0$ (Line 15), then the `read` of $stop = 1$ falls between the previous successful return to `test&set` and the serialization of the `reset`. Alternatively, suppose the call to `reset` was serialized just before the serialization of the next successful call to `test&set`. Then these two serialization points are adjacent, and the `read` comes either before or after both. By Lemma 5.8, the next `reset` cannot have begun, and the `read` of $stop = 1$ is a correct serialization point.

Suppose an unsuccessful call returns because of losing the test on $restart[p]$ (Line 4 or 11 of `test&set`). Then a call to `reset` assigns to $restart[p]$ (Line 14) after the `test&set` begins and before the `read` of *restart*. If the `reset` operation is serialized during the unsuccessful call, we can serialize the latter just before the `reset`. If it is serialized after the unsuccessful call, then since the `reset` does not begin until after the previous successful call to `test&set`, there is an interval during the unsuccessful call, between the previous successful call to `test&set` and the `reset`. Last, suppose the `reset` is serialized before the unsuccessful call begins. Then it has been serialized early because the following successful `test&set` was also serialized before the unsuccessful call begins (at its assignment to *restart*). The return of that successful `test&set` follows the `reset`'s final assignment to *stop* (Line 15), which in turn must come after the unsuccessful call to `test&set`, and the next `reset` must begin even later. Hence, there is a period after the serialization of the successful `test&set` and before the next `reset`.

Finally, consider the unsuccessful calls to `test&set` which lost because they are unsuccessful in the simulated calls to `su-test&set`. By Lemma 5.8 and the semantics of the single-use test&set, this can happen only if there is a successful simulated call to `su-test&set`, by a call to `test&set` in the same collection. Moreover, this simulated call to `su-test&set`, (by the serialization of `su-test&set`), begins before the unsuccessful simulated call ends, since it can be serialized before the serialization point of the simulated unsuccessful call. That successful simulated call to `su-test&set` is within a successful call to `test&set` which has been serialized even earlier, when it assigns to *restart*, before its simulation started.

It remains to argue that no `reset` is serialized after the serialization point of the successful `test&set` and before the beginning of the unsuccessful call to `test&set`. Because the unsuccessful `test&set` progressed to run at least one step of the simulation, Lemma 5.2 implies no `reset` is active for this process during an interval of its operation. Hence, any `reset` serialized before the unsuccessful call must be the call to `reset` which precedes the unsuccessful call, or is even earlier. But these calls to `reset` are also serialized before the successful call to `test&set`.

Note that each high-level `test&set` operation requires at most a constant number of operations on the primitive shared objects, and each high-level `reset` operation requires at most $O(n)$ operations on the primitive shared objects. □

6. RMW REGISTERS AND CONSENSUS

To complete our investigation of memory fault-tolerance we turn to the read-modify-write register. Herlihy showed that *reliable* read-modify-write (RMW) is a universal shared memory primitive [Her91]; any other shared object can be simulated using RMW. This follows because RMW registers can be used to solve n-process consensus, which is itself universal.

Briefly, RMW registers enable a process to atomically read a register and, *based on the value read,* to write a new value. Alternatively, a consensus object is accessed by n processes, each with an input value, $input_p \in \{-1, +1\}$. A process *decides* on a value *output* if it writes *output* to its write-once output register. The requirements of the consensus problem are that there exists a *decision value* v such that each non-faulty process eventually decides on v and that v is the input value of some process.

In this section several different constructions of fault-tolerant consensus objects are presented. These constructions use a variety of read-modify-write and atomic registers, and tolerate different types of faults. The following table summarizes these constructions.

Number of registers used to reach consensus in the presence of faults				
Number and type of faults	# RMW Regs. used	# atomic bits used	# bits in RMW reg.'s	Algorithm
f ∞-faulty	$(f+1)^2$	0	4	[AGMT92]
f ∞-faulty	$f+1$	$4f^2+6f+2$	2	§ 6.1
f ∞-faulty	$2f+1$	0	$4+2\log f$	§ 6.1
m-faults	$2m+1$	0	4	§ 6.2

6.1 Unbounded failures per register

We start by presenting a construction (Theorem 12) of a consensus object from $f+1$ ternary RMW registers and a quadratic number of atomic bits, such that f of them may be ∞-faulty. We then show in Theorem 14, how to encode all of the atomic bits and the ternary RMW registers into $2f+1$, $O(\log f)$-bit wide, RMW registers. We support these upper bound with two lower bounds, (Theorem 13) that $f+1$ RMW registers are always necessary, and (Theorem 14) that a total of $2f+1$ registers (RMW and/or read/write) are necessary.

THEOREM 12. *For any $f \geq 0$, there is a strongly wait-free consensus construction using $f+1$ ternary RMW registers and $2(2f+1)(f+1) = 4f^2+6f+2$ atomic bits, at most f of which are ∞-faulty.*

PROOF. The construction is shown in Figure 7. This figure uses the notation **lock**(r) and **unlock** to mark the beginning and end of atomic, exclusive access to shared RMW register r. That is, it is assumed that a process can lock only

Protocol for process p, $input_p \in \{-1,+1\}$**:**

```
shared r: array[1..f+1] of ternary RMW registers              %initially each register = 0
shared v: array[1..f+1][1..2f+1][-1,1] of atomic bits          % initially each bit = 0
local decide_p ,x: ranges over {-1,0,+1}, initially 0

1:  x := input_p                                               %Set initial value of x
2:  for i = 1 to f+1 do
3:      for j = 1 to 2f+1 do v(i,j,x) := 1 od %Establish validity of current value at this stage
4:      lock(r(i))
5:          if r(i) = 0 then r(i) := x fi                      %Vote for x at this stage
6:          tmp := r(i)
7:      unlock
8:      count := 0                              %Adopt this stage's vote if it is valid
9:      for j = 1 to 2f+1 do if v(i,j,tmp) = 1 then count := count + 1 fi od
10:     if count ≥ f+1 then x := tmp fi
11: od
12: decide_p := x
```

Fig. 7. Consensus in the presence of f, ∞-faulty registers
The construction uses $f+1$ RMW registers and $4f^2 + 6f + 2$ atomic read/write bits.

one register at a time, that a process does not fail between pairs of **lock**(r) and **unlock** statements, and that any non-faulty process that reaches a **lock** instruction eventually executes it.

The construction uses two ideas. The first is that consensus is trivially achieved with one non-faulty, ternary RMW register. The register is initially set to 0, the first process to do a RMW sets the register to its input value, and all processes agree on the value written. The second idea is to use validation bits to ensure that an invalid value cannot be chosen. Before proposing a value in a RMW register, a process establishes it as valid by writing to many validity bits. Before adopting a value from a RMW register a process confirms its validity by reading the validity bits. Faults in a RMW register therefore cannot convince a process to choose an invalid value.

The two ideas are combined as follows. The construction proceeds in $f+1$ stages. Each stage is assigned a RMW register and two sets of $2f+1$ atomic bits. One set of bits is called the *validity bits* for value $+1$ and the other set the validity bits for value -1. In each stage the processes validate their current value using the appropriate validity bits, attempt to write their value in a RMW register reserved for this stage, and adopt the value written in this register if it is valid.

More specifically, the value x is marked valid in stage i by ensuring that all $2f+1$ stage-i validity bits for value x are set to 1. (In Figure 7 Line 3 these bits are denoted $v(i,*,x)$.) The attempt to write a value at stage i is made to the ith RMW register (Lines 4 to 7). A process writes its value if and only if the register was 0 prior to the RMW. The value x is considered valid by a process checking its validity at stage i if and only if at least $f+1$ stage-i validity bits for value x are set to 1 (Lines 8 to 10). (Note there is a straightforward generalization of this construction, to solve consensus for any bounded set of input values, by adding additional validity registers for each value, and appropriate RMW values.)

LEMMA 6.1. **(Validity)** *In the protocol of Figure 7 processes decide only on valid values.*

PROOF. A stronger statement is proved: the value of the local variable x is always valid after Line 1 has been executed.

For each process the variable x is modified only in Line 1 and possibly $f+1$ times at Line 10. We inductively prove that no variable x can be changed to an invalid value. Suppose all changes numbered less than i were to valid values (the assignment in Line 1 for some process serves as base cases). Next it is shown that the change numbered i is to a valid value.

If change number i is an execution of Line 1 then it is to an input value which is by definition valid and the induction holds. If, on the other hand, the change is due to an execution of Line 10 then it must be the case that the **for** loop at Line 9 read at least $f{+}1$ validity bits set to 1 for the new value of x. These bits are initially 0 (meaning not valid for all stages) and at least one is a field of a non-faulty register. This non-faulty bit could only be non-zero (valid) if it were written by some process at an earlier time. But processes only write these bits (at Line 3) when they correspond to their current value of x. By induction the earlier values of x were valid and thus the non-faulty bit corresponds to a valid value of x and the new, adopted value of x is valid. □

LEMMA 6.2. **(Agreement)** *Every process which writes a value in* decide$_p$ *writes the same value.*

PROOF. In any execution there is at least one stage in which the RMW register which is written is non-faulty. Call the first such stage g. The first process to attempt a read-modify-write at g will succeed in writing its value, call it x_g. Furthermore, it will have first established the validity of x_g for stage g. Since the non-faulty validity bits are set to 1 but never set to 0, the $f{+}1$ non-faulty validity bits at stage g will always show x_g as valid for stage g. All subsequent processes will thus be able to successfully confirm the validity of x_g for stage g. Since register g is non-faulty all subsequent processes will also agree that its value is x_g. Thus all processes will adopt the value x_g at stage g.

Using stage g as a base case we now show that if all processes agree on value x_g at the end of stage $i \geq g$, then all processes agree on value x_g at the end of stage $i+1$. Since all processes agree on value x_g at the end of stage i they will all set validity bits for the value x_g at stage $i+1$. None will set validity bits for the value $-x_g$ at stage $i+1$. Thus, regardless of the value of the $i+1$st RMW register no process can switch its value to $-x_g$. □

Note that each process performs $4f^2 + 6f + 2$ primitive operations on the read/write registers, and $f{+}1$ primitive RMW operations. Hence, the construction is strongly wait-free. This fact, together with Lemmas 6.1 and 6.2, concludes the proof of Theorem 12. □

Complementing Theorem 12, we have a simple lower bound that shows that the number of RMW registers used in the proof of Theorem 12, $f{+}1$, is optimal.

THEOREM 13. *There does not exist a strongly wait-free consensus construction which tolerates f ∞-faulty registers and uses fewer than $f{+}1$ RMW registers and any number of read/write registers.*

PROOF. Since a fault could erase the effect of every operation in an ∞-faulty RMW register, this object can be trivially implemented by an object that always returns the initial value and uses no shared primitive objects. Thus, f ∞-faulty RMW registers and any number of reliable read/write registers are no more powerful for solving consensus than the read/write registers alone. The theorem follows from the impossibility of consensus from read/write registers [Her91, LAA87]. □

If only RMW registers are used, we have the following tight bound.

THEOREM 14. *If only primitive ∞-faulty RMW registers are used to solve consensus, and if as many as f of the registers may be faulty, then $2f+1$ (logarithmic-size) RMW registers are both necessary and sufficient:*

$$\mathbf{CONST}(RMW, (f, \infty), consensus) = 2f+1.$$

PROOF. The lower bound, $\mathbf{CONST}(RMW, (f, \infty), consensus) > 2f$, is a corollary of a stronger lower bound (for a weaker fault model), Lemma 6.3, in the next subsection.

The upper bound, $\mathbf{CONST}(RMW, (f, \infty), consensus) \leq 2f+1$, is proved by modifying the construction used in the proof of Theorem 12. The modification encodes the $f+1$ ternary RMW registers and the $O(f^2)$ validity atomic bits into $2f+1$ RMW registers of size $4 + (2\lceil \log f \rceil)$-bits.

Each of the $2f+1$ RMW registers in the new construction is divided into three fields: *decide, vplus,* and *vminus*. The $f+1$ ternary RMW registers are simulated by the *decide* field of the first f+1 new RMW registers. This is possible since RMW registers can be read-modify-written in one field without affecting the values in any other field.

The encoding of the validity bits relies on the following two observations about the construction. First, *a value is valid in stage k if and only if its validity was established in all stages i, $i < k$*, and second, *once valid in stage k a value will never be invalidated in stage k*. Thus, instead of recording the validation for each stage separately, it is enough to record for each value the maximum stage in which it is valid. Of course, since f of the RMW registers could be faulty, we repeatedly record this information in all the $2f$+1 RMW registers. Each of the fields *vplus* and *vminus* can encode the values 0 through $f+1$ and is initially 0. The setting of bit $v(i, j, 1)$ in the previous construction is now performed by setting the *vplus* field of register j to be equal to i if it is found to be less than i. The bit $v(i, j, 1)$ is read as 1 if the value of the *vplus* field of register j is greater than or equal to i and as 0 otherwise. The bit $v(i, j, -1)$ is handled in the same way.

In this construction, each process now performs $4f^2 + 7f + 3$ primitive RMW operations. □

6.2 Bounded failures per register

For a bounded number of faults per register we are able to fully characterize the number of RMW registers required for consensus:

THEOREM 15.

. $\mathbf{CONST}(RMW, (f, 1), consensus) = 2f+1$.

. $\mathbf{CONST}(RMW, m, consensus) = 2m+1$.

PROOF. Lemma 6.3 (below) implies the lower bound: $\mathbf{CONST}(RMW, (f,1), consensus) > 2f$ while Lemma 6.4 implies the upper bound: $\mathbf{CONST}(RMW, m, consensus) \leq 2m+1$. Together with Theorem 1, these imply the theorem. □

LEMMA 6.3. *There is no n-process consensus construction using fewer than $2f+1$ primitive objects, at most f of which may be 1-faulty, and which survives $\lceil n/2 \rceil$ process failures.*

PROOF. The proof is by contradiction. Assume to the contrary that there is a solution using $2f$ primitive objects. Let the initial internal states of the registers $r_1, ..., r_{2f}$, be $u_1, ..., u_{2f}$, respectively.

The process-failure assumption requires that in any run in which only half the processes, $p_1, ..., p_{\lceil n/2 \rceil}$ take steps, they must eventually decide. Also, if their inputs are identical, the validity condition requires that they must decide on that value, as they do not know whether the other decision value is an input of some process. Thus, there is a failure-free, finite run α where half the processes run alone with input $+1$ and decide on $+1$. Let the final internal states of the primitives $r_1, ..., r_{2f}$, in α be $v_1, ..., v_{2f}$, respectively.

Now consider a run in which the other processes, $p_{\lceil n/2 \rceil+1}, ..., p_n$ take steps, run alone with input -1, but find the primitives initially are in states $u_1, ..., u_f, v_{f+1}, ..., v_{2f}$. This is consistent with a run in which $p_1, ..., p_{\lceil n/2 \rceil}$ have taken no steps and f faults have changed the internal states of $r_{f+1}, ..., r_{2f}$ to $v_{f+1}, ..., v_{2f}$, respectively. Hence, $p_{\lceil n/2 \rceil+1}, ..., p_n$ must all decide -1. But this run is also consistent with a run in which $p_1, ..., p_{\lceil n/2 \rceil}$ have already decided $+1$, and f faults have changed the internal states of $r_1, ..., r_f$ back to $u_1, ..., u_f$, respectively. Hence, $p_{\lceil n/2 \rceil+1}, ..., p_n$ must all decide $+1$, a contradiction. □

Note that this lower bound depends on a weaker progress assumption than wait-freedom, which requires each process to make progress regardless of whether the others take steps. By $\lceil n/2 \rceil$ process failures, we mean that the construction need only be correct in runs in which at least $\lfloor n/2 \rfloor$ processes are guaranteed to make progress. (A process may wait for $\lfloor n/2 \rfloor - 1$ other processes to take steps.)

Moreover, the bound also does not depend on the behavior of the construction when the object fault bound is exceeded. (It does not depend on a requirement that the construction be strongly wait-free.)

LEMMA 6.4. *For any $m \geq 0$, there is a strongly wait-free consensus construction using $2m+1$ faulty RMW registers, provided the total number of memory failures is at most m:*

$$\mathbf{CONST}(RMW, m, consensus) \leq 2m+1.$$

PROOF. The proof of this Lemma is based on the construction in Figure 8, which solves strongly wait-free consensus using $2m+1$ faulty RMW registers, provided the total number of memory failures is at most m. The correctness of this construction is a consequence of the series of lemmas in the next subsection.

6.3 Proof of the upper bound

In Figure 8, the input to process p is initially assigned to register $input_p$, local to p, and an appropriate output values must be assigned to register $decide_p$. As in

Figure 7, Figure 8 uses the notation **lock**(r) and **unlock** to mark the beginning and end of atomic, exclusive access to shared RMW register r.

In the construction, there are $2m+1$ shared registers $r[i]$, $1 \leq i \leq 2m+1$. Each register contains three fields: *vote*, *plus*, and *minus*. The boolean *plus* and *minus* fields are used to check that the associated value (from $\{-1,+1\}$) is the input to some process, and hence is *valid*. This is assured by a discipline in which every process initially indicates that its input is valid by setting the appropriate $2m+1$ fields to 1 in lines $1-3$. This guarantees that henceforth, any other process reading these fields will find at least $m+1$ of them 1, and conclude the value is valid. In turn, any process finding at least $m+1$ of the *plus* (respectively, *minus*) fields 1, sets all of them to 1 (in lines $12-16$ of function *valid*) before otherwise acting on the knowledge that $+1$ (respectively, -1) is valid. This is necessary to ensure the stability of the validity. Otherwise some process with input u could write it in lines $1-3$ to only $m+1$ registers and stop, then some process would confirm u valid, but yet another would not be able to confirm because some of the $m+1$ registers have failed.

For $i = 1, 2, \ldots 2m+1$ each process scans the *vote* fields of the registers $r[1] \cdots r[i-1]$ and tries to write the sign of the sum of the previous registers to $r[i].vote$, or its own value if the sum is either 0 or is not valid. At the end it scans once again all the registers, and decides on the sign of this final sum.

The proof of correctness proceeds as follows. Note first (from lines 5–9 in the main protocol body) that each process's protocol performs $O(m^2)$ primitive wait-free operations before deciding and terminating. Hence, the construction is strongly wait-free, and it suffices to consider only *complete runs*, those in which every process has terminated. Thus, the construction is correct if its complete runs satisfy the validity and agreement conditions. The construction is analyzed under a stronger fault model which allows m independent faults to occur to each of the *vote*, *plus* and *minus* fields of the shared registers, up to $3m$ faults in all. These faults are modeled as assignments to the appropriate register fields. The validity condition is proven in a straightforward manner (Lemma 6.5). Subsequently we argue that the construction is correct if and only if each of a constrained set of executions is correct (Lemmas 6.6–6.8). These executions are shown to satisfy an invariant that implies the agreement condition (Lemmas 6.9–6.11).

Note that nowhere in any process's code is a shared register field ever set to 0.

Lemma 6.5. **(Validity)**

(1) No process p writes to a plus *(respectively,* minus*) field unless either* $\mathrm{input_p} = +1$ *(respectively,* $\mathrm{input_p} = -1$*), or the process has previously observed* 1 *as the value in* $m+1$ *of the* plus *(respectively,* minus*) fields.*

(2) No process writes to a plus *(respectively,* minus*) field unless* $+1$ *(respectively,* -1*) is the input of some process.*

(3) No process decides $+1$ *(respectively,* -1*) unless the process has previously observed* 1 *as the value in* $m+1$ *of the* plus *(respectively,* minus*) fields.*

(4) No process writes $+1$ *(respectively,* -1*) to a* vote *field without first assigning* 1 *to the* plus *(respectively,* minus*) fields of all* $2m+1$ *registers.*

Two *complete* runs α and β are *similar* if each process has the same input value in α as in β, and each decides on the same value in α as in β.

```
Protocol for process p, input_p ∈ {−1, +1}:

type     reg = record [minus, plus: in {0,1} vote: in {−1,0,+1}]
shared   r: array[1..(2m + 1)] of reg, initially all (0,0,0)
local    decide_p: in {−1,0,+1}, initially 0

begin Main
    for i = 1 to 2m+1 do                                     %indicate that input_p is valid
        if input_p = +1 then RMW(r[i], plus, 1)
        else RMW(r[i], minus, 1) fi od
    d := input_p
    for i = 1 to 2m+1 do                                     %push sum away from 0
        RMW(r[i], vote, d)
        sum := 0
        for j = 1 to i do sum := sum + RMW(r[j], vote, d) od   %sum of r[1]···r[i]
        d := valid(sum) od
    decide_p := d                                            %make final decision
    end_Main


    function RMW(reg, field, t): integer    %set reg.field from 0 to t if reg.field was 0
    lock(reg)
        if reg.field = 0 then reg.field := t fi
        tmp := reg.field
    unlock
    return(tmp)
    end_function

    function valid(v): integer        %return sign(v) if valid, otherwise return input_p
    if v = 0 or sign(v) = sign(input_p)
    then return(input_p)
    else sm := 0
        for j = 1 to 2m+1 do
            if sign(v) = +1 then sm := sm + r[j].plus
            else sm := sm + r[j].minus fi
        od
        if sm ≤ m
        then return(input_p)                                 %v is not valid
        else                                                 %v is valid
            for j = 1 to 2m+1 do                             %ensure that v's validity is stable.
                if sign(v) = +1
                then RMW(r[j], plus, 1)
                else RMW(r[j], minus, 1) fi
            od
            return(sign(v))
        fi
    fi
    end_function

    function sign(x): integer
    case x of:
        0:    return(0)
        < 0:  return(−1)
        > 0:  return(+1)
    end_function
```

Fig. 8. Consensus in the presence of m memory failures.

Next, note that the read-modify-write in line 8 modifies $r[j].vote$ only if $r[j].vote$ is first observed to be 0. Since the same process will have either set or observed $r[j].vote \neq 0$ in an earlier scan, this observation of 0 and resulting modification (in Line 8) is due to a memory fault on $r[j].vote$.

LEMMA 6.6. *For any complete run β there is a similar run α, such that α contains no more memory faults than β, and such that in α no memory fault assigns 0 to any* vote *field.*

PROOF. Let β be a complete run of the form $\beta_1\langle r[j].vote := 0\rangle\beta_2$, where $\langle r[j].vote := 0\rangle$ is a memory fault. There are several cases:

. No operation in β_2 references $r[j].vote$.
Then $\beta_1\beta_2$ is a complete run that is similar to β, has no more memory faults than β, and has one fewer (faulty) assignment of 0 to a $vote$ field.

. The first reference to $r[j].vote$ in β_2 is a memory fault. That is, β_2 can be written as $\beta_3\langle r[j].vote := v\rangle\beta_4$, where β_3 contains no reference to $r[j].vote$, $r[j].vote = v$ is a memory fault and $v \in \{-1, 0, 1\}$. Then $\beta_1\beta_3\langle r[j].vote := v\rangle\beta_4$ is a run of the construction that is similar to β, contains no more memory faults than β, and has one fewer (faulty) assignment of 0 to a $vote$ field.

. The first reference to $r[j].vote$ in β_2 is a read-modify-write.
That is, β_2 can be written as $\beta_3\langle r[j].vote = 0; r[j].vote := v\rangle\beta_4$, where $\langle r[j].vote = 0; r[j].vote := v\rangle$ is the read-modify-write by some process p, and β_3 contains no explicit reference to $r[j].vote$. Note that the read-modify-write operations to the $vote$ fields change the value only when it is zero. Then $\beta_1\langle r[j].vote := v\rangle\beta_3\langle r[j].vote = v\rangle\beta_4$, where $\langle r[j].vote := v\rangle$ is a fault, is a run of the construction that is similar to β, contains no more memory faults than β, and has one fewer (faulty) assignment of 0 to a $vote$ field. This run is similar to β since in the construction a process does not distinguish between the case where its read-modify-write operation actually modifies the value of $r[j].vote$ to v and the case in which v was found in $r[j].vote$ by the read-modify-write operation that attempts to write v in it.

In each case the number of faulty assignments of 0 to a $vote$ field decreases by one. The lemma follows by induction. □

LEMMA 6.7. *Any complete run has a similar run, with no more memory faults, in which no memory fault occurs at* r[j].vote *when the value is* 0.

PROOF. Let β be a complete run of the form $\beta_1\langle r[j].vote := v\rangle\beta_2$, where $\langle r[j].vote := v\rangle$ is a memory fault and the value of $r[j].vote$ after β_1 is 0. Moreover, let this be the first such memory fault in β. By the previous lemma, it suffices to assume that no memory fault assigns 0 to a $vote$ field in β. Since no process ever writes 0 to any $vote$ field, it follows that the value of $r[j].vote$ is 0 throughout β_1. There are several cases:

. Either $v = 0$ or no operation in β_2 references $r[j].vote$.
Then $\beta_1\beta_2$ is a complete run that is similar to β, has one fewer memory fault than β, and has one fewer (faulty) assignment to $r[j].vote$ when the value is 0.

. $v \neq 0$ and the first reference to $r[j].vote$ in β_2 is a memory fault. That is β_2 can be written as $\beta_3\langle r[j].vote := v'\rangle\beta_4$, where β_3 contains no reference to $r[j].vote$ and $v' \in \{-1, 0, 1\}$. Then $\beta_1\beta_3\langle r[j].vote := v'\rangle\beta_4$ is a run of the construction that is similar to β, has one fewer memory fault than β and the same number of (faulty) assignments to $r[j].vote$ when the value is 0.

. $v \neq 0$ and the first reference to $r[j].vote$ in β_2 is a read-modify-write. That is, β_2 can be written as $\beta_3\langle r[j].vote = v\rangle\beta_4$, where $\langle r[j].vote = v\rangle$ is the conditional test of the read-modify-write by some process p, and β_3 contains no explicit reference to $r[j].vote$. Since the value of $r[j].vote$ is 0 throughout β_1, this read-modify-write operation is from line 6 of Main, and the value returned is discarded by the executing process. Hence, $\beta_1\beta_3\langle r[j].vote = 0; r[j].vote := v'\rangle\langle r[j].vote := v\rangle\beta_4$ is a run of the construction that is similar to β, has the same number of memory faults as β, and one fewer (faulty) assignment to $r[j].vote$ when the value is 0.

In each case the total number of memory faults is either reduced, or the total number remains the same, with one fewer (faulty) assignment to $r[j].vote$ when the value is 0. The lemma follows by induction. □

Definition 7. *A* legal *run, is a run in which any memory fault to a* vote *field either changes its value from* $+1$ *to* -1 *or vice-versa.*

Lemma 6.8. *Any complete run has a similar legal run with no more memory faults.*

Proof. By the previous two lemmas, it suffices to consider complete runs in which no memory fault assigns 0 to a *vote* field, or over-writes a 0 in a *vote* field. The remaining alternatives are memory faults which write $+1$ or -1, but do not change the value. These faults can be trivially deleted, resulting in a run satisfying the conditions of the lemma. □

Call read-modify-write operations to *vote* fields that actually change the value *successful read-modify-writes.* Call the $i-1$ `read`s by p immediately preceding a successful read-modify-write to $r[i].vote$ by p in line 6 of Main, the `collect` for that `write`.

Lemma 6.9. *In any legal run α, there are exactly $2m{+}1$ successful read-modify-writes, one to each* vote *field. Furthermore, the* `collect`*s for any two such successful read-modify-writes are not concurrent: if $i < j$, the* `collect` *for the successful read-modify-write to* r[i].vote *precedes the successful read-modify-write to* r[i].vote*, which in turn precedes the* `collect` *for the successful* `write` *to* r[j].vote.

Proof. By definition, every legal run is complete and the memory faults to *vote* fields change the value from $+1$ to -1, or -1 to $+1$. In complete runs, every process executes a read-modify-write on each *vote* field, so each is changed from 0 to $+1$ or -1 at least once. Once set, being non-zero is stable, so each *vote* field has exactly one successful read-modify-write.

The condition on `collect`s holds trivially if both successful read-modify-writes and their `collect`s are by the same process. Suppose the read-modify-writes are by different process, p and q, to $r[i].vote$ and $r[j].vote$, respectively. Note that q does an unsuccessful read-modify-write to $r[i].vote$ before the `collect` for $r[j].vote$ begins. Hence, the successful read-modify-write to $r[i].vote$ by p precedes this. The condition follows. □

Let s_k be a state of the system in a legal run of k atomic operations, and let RF_k be the remaining unexecuted faults to *vote* fields in a run, that is, m minus the number of such faults so far. Let ZS_k be the number of 0's in the vote fields in registers in s_k, define Σ_k to be the sum of the *vote* fields in s_k, $\Sigma_{i=1}^{2m+1} r[i].vote$, and finally, define Δ_k to be $|\Sigma_k| + ZS_k - 2RF_k$.

LEMMA 6.10. *$\Delta_k > 0$, for any $k \geq 0$.*

PROOF. We consider the changes to these parameters that can result from any single step of the construction. That is, let π be a step in a legal run that changes the state from s_k to s_{k+1}.

(1) π is a memory fault.
That is, π is $r[i].vote := v$, where the value of $r[i].vote$ is $-v$ in s_k. Note that $ZS_{k+1} = ZS_k$ and $RF_{k+1} = RF_k - 1$. Here there are four key sub-cases.
(a) $\Sigma_k = 0$. Then $|\Sigma_{k+1}| = 2$, and $\Delta_{k+1} = \Delta_k + 4$.
(b) $0 < |\Sigma_k| < |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| + 2$, and again $\Delta_{k+1} = \Delta_k + 4$.
(c) $0 < |\Sigma_k| = |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| = 1$, and $\Delta_{k+1} = \Delta_k + 2$.
(d) $|\Sigma_{k+1}| < |\Sigma_k|$. Then $|\Sigma_{k+1}| = |\Sigma_k| - 2$, and $\Delta_{k+1} = \Delta_k$.

(2) π is a successful read-modify-write.
That is, π is $r[i].vote = 0; r[i].vote := v$. Then $ZS_{k+1} = ZS_k - 1$, $RF_{k+1} = RF_k$, and $\Sigma_{k+1} = \Sigma_k + v$. There are two key subcases:
(a) $|\Sigma_k| < |\Sigma_{k+1}|$. Then $|\Sigma_{k+1}| = |\Sigma_k| + 1$, and $\Delta_{k+1} = \Delta_k$.
(b) $|\Sigma_{k+1}| < |\Sigma_k|$. Then $|\Sigma_{k+1}| = |\Sigma_k| - 1$, and $\Delta_{k+1} = \Delta_k - 2$.

(3) π is any other atomic step.
Then $ZS_{k+1} = ZS_k$, $RF_{k+1} = RF_k$, and $|\Sigma_{k+1}| = |\Sigma_k|$. Hence $\Delta_k = \Delta_{k+1}$.

In every (sub)case but one (2b), the value Δ_{k+1} is greater than or equal to Δ_k. The problematic case is the occurrence, then, of read-modify-write operations that decrease the value of $|\Sigma|$. Intuitively, such operations occur because faults have occurred so as to cause a process to inadvertently "move" the value of $|\Sigma|$ in the wrong direction. We claim that, for each such read-modify-write operation that decreases $|\Sigma|$ by 1, there must be an earlier matching fault of type 1a, 1b or 1c that increases $|\Sigma|$ by at least 2. This claim is proved below by induction, thus completing the proof of the invariant.

The proof of the claim proceeds by induction on the prefixes of the run. Clearly the invariant holds for the empty run ($RF_0 = m$; $ZS_0 = 2m+1$; and $|\Sigma_0| = 0$). Let $\alpha\pi$ be a prefix of the run, where π is the $(k+1)$st atomic operation and the invariant holds for every state $s_0, ..., s_k$. By the analysis above, no atomic step of the construction can falsify the invariant unless case 2b applies. In this case, π is a successful read-modify-write by some process p, $\langle r[i].vote = 0; r[i].vote := v\rangle$, and $|\Sigma_{k+1}| = |\Sigma_k| - 1$. Note that $\Sigma_k \neq 0$.

Since, in case (2b) $\Sigma_k \neq 0$ and this is a legal run (in which no memory fault overwrites a 0 in a *vote* field), $i > 1$. Moreover, by Lemma 6.9, $\alpha = \alpha_1 \langle r[i-1].vote = 0; r[i-1] := v'\rangle\ \alpha_2\pi$, where $\langle r[i-1].vote = 0;\ r[i-1] := v'\rangle$ is the successful read-modify-write to $r[i-1].vote$ and there are no successful read-modify-write operations in α_2. Let s_j be the state at the beginning of α_2, just after the successful read-modify-write to $r[i-1].vote$. By induction, $\Delta_j > 0$. Also by Lemma 6.9, α_2 contains the $i-1$ `reads` in the `collect` by p that precedes π. In addition, since none of the operations in α_2 are successful read-modify-writes, by the analysis above $\Delta_j \leq ... \leq \Delta_k$.

Next, consider the sequence of values $\Sigma_j, ..., \Sigma_k$. Observe that since the run is legal, and because there is no successful read-modify-write operation in α_2, the value of Σ never changes by only 1 in α_2. We examine cases depending on the sign of the sum collected by p, and show that a fault described in case 1a, 1b or 1c must occur in α_2, implying $\Delta_k \geq 3$, and so $\Delta_{k+1} \geq 1$.

. The `collect` sums to a valid value. There are two subcases.

. Some Σ_x in $\Sigma_j, ..., \Sigma_k$ has the same sign as the sum collected:
Since $|\Sigma_{k+1}| = |\Sigma_k| - 1$, it must be that the sign of the value written by π (v) is different than the sign of Σ_k. Thus, the sign of Σ_x (= sign of sum collected = sign of v) is different than the sign of Σ_k. Hence, there exist Σ_r and Σ_{r+1} in this sequence such that either $\Sigma_r = 0$ and $|\Sigma_{r+1}| = 2$, or $|\Sigma_r| = |\Sigma_{r+1}| = 1$ and $\Sigma_r = -\Sigma_{r+1}$. Then either $\Delta_{r+1} = \Delta_r + 4 \geq 5$ or $\Delta_{r+1} = \Delta_r + 2 \geq 3$, respectively. (This must be due to a fault of type 1b or 1c, above.) Thus, $\Delta_k \geq \Delta_{r+1} \geq 3$ and hence, $\Delta_{k+1} \geq 1$.

. No Σ in $\Sigma_j, ..., \Sigma_k$ has the same sign as the sum collected.
By definition the sign of the sum collected must be the sign of a majority of the $i-1$ registers read in the `collect`. Thus, a majority of the $i-1$ registers each have the same sign as the `collect` and as v at some point in the interval, but not at the end of the interval. Then a fault in the interval must change the value of at least one of these, from v to $-v$. That is, there exist Σ_r and Σ_{r+1} in this sequence such that $|\Sigma_{r+1}| = |\Sigma_r| + 2$, and $\Delta_{r+1} = \Delta_r + 4 \geq 5$. Hence, $\Delta_{k+1} \geq 3$.

. The `collect` sums to 0. There are two subcases.

. Some Σ_x in $\Sigma_j, ..., \Sigma_k$ has value 0.
Since $\Sigma_k \neq 0$, some fault must move the sum from 0. That is, there exist Σ_r and Σ_{r+1} in this sequence such that $\Sigma_r = 0$ and $|\Sigma_{r+1}| = 2$. Hence, $\Delta_{r+1} = \Delta_r + 4 \geq 5$ and $\Delta_{k+1} \geq 3$.

. No Σ in $\Sigma_j, ..., \Sigma_k$ has value 0.
That is, half the registers are read as positive, and half as negative. Suppose first that there exist Σ_r and Σ_{r+1} in the sequence $\Sigma_j, ..., \Sigma_k$ that have different sign: that $|\Sigma_r| = |\Sigma_{r+1}| = 1$ and $\Sigma_r = -\Sigma_{r+1}$. Then $\Delta_{r+1} = \Delta_r + 2 \geq 3$ and $\Delta_k \geq 3$. Hence, $\Delta_{k+1} \geq 1$.
Suppose next that all of $\Sigma_j, ..., \Sigma_k$ have the same sign. Since $|\Sigma_{k+1}| = |\Sigma_k + v| < |\Sigma_k|$, their sign is different than v's. The `collect` read half the registers with $vote = -v$ and half with $vote = v$. Since all the Σ have sign different than v, some fault changes a value from v to $-v$ in α_2. That is, there exist Σ_r and Σ_{r+1} in the sequence such that $|\Sigma_{r+1}| = |\Sigma_r| + 2$, and $\Delta_{r+1} = \Delta_r + 4 \geq 4 \geq 5$. Hence, $\Delta_{k+1} \geq 3$.

. The `collect` is nonzero and invalid.
By Lemma 6.5, all $i-1$ of the earlier successful read-modify-writes wrote the single valid value, v, yet the sum of the `collect` had opposite sign. Hence, in α at least $\lceil i/2 \rceil$ registers had faults changing the value from v to $-v$. Recall that RF_{k+1} is m minus the number of faults in $\alpha\pi$; hence, $RF_{k+1} \leq m - \lceil i/2 \rceil$, and $2RF_{k+1} \leq 2m - 2\lceil i/2 \rceil \leq 2m - i$. Hence, we have

$$\begin{aligned}
\Delta_{k+1} &= |\Sigma_{k+1}| + ZS_{k+1} - 2RF_{k+1} \\
&= |\Sigma_{k+1}| + (2m + 1 - i) - 2RF_{k+1} \\
&\geq |\Sigma_{k+1}| + 2m + 1 - i - (2m - i) \\
&\geq |\Sigma_{k+1}| + 1 \\
&\geq 1 \qquad \square
\end{aligned}$$

LEMMA 6.11. **(Agreement)** *All processes decide on the same value.*

PROOF. Consider Σ_k in the system state s_k, immediately after the last register, $r[2m{+}1]$, has been written. By Lemma 6.10, $2RF_k < |\Sigma_k|$. Henceforth, the number of remaining faults is insufficient to change the sign of Σ_k, or reduce it to 0. Since all the `reads` in any final `collect` (upon which any decision is based) are made after s_k, all processes decide on the same value. □

Recall that a register is k-faulty if it can change its value spontaneously, without any process writing into it, at most k times. Lemma 6.4 and Theorem 1 imply the following:

COROLLARY 16. *For any $1 \leq k \leq m$, there is a strongly wait-free consensus construction using $2m{+}1$ RMW registers where at most $\lfloor \frac{m}{k} \rfloor$ registers are k-faulty:*

$$\mathbf{CONST}(RMW, (\left\lfloor \frac{m}{k} \right\rfloor, k), consensus) \leq 2m{+}1.$$

6.4 Universal constructions

Herlihy defined the notion of a *universal* object, as an object that can be used to construct a wait-free implementation of any other object [Her91]. He showed that consensus and other objects for n processes are universal for systems with at most n processes. Herlihy's construction required atomic registers over an unbounded domain. Recently, Jayanti and Toueg showed how to bound the register size in Herlihy's construction [JT92].

THEOREM 17. **(Herlihy)** *Consensus objects together with atomic registers are universal, provided none of them are faulty: For all objects X,*

$$\mathbf{CONST}((consensus, atomic), (0, \infty), X) < \infty.$$

PROOF. For any number of processes n, Herlihy gives an explicit construction from n-processor consensus and atomic registers that uses $O(n^3)$ reliable atomic read/write registers of unbounded size and $O(n^3)$ reliable consensus objects over a bounded domain. Simple extensions of known constructions can be used to implement multi-valued consensus from binary consensus and n read/write registers [Plo88]. (Each process p writes its input to a read/write register r_p, initialized to some invalid value, then processes use binary consensus on $\log n$ consensus objects to agree on the index q of a register r_q that contains a valid value.)

Although the construction does not tolerate faults, it must be strongly wait-free. Herlihy's construction includes a loop with an exit condition dependent on values read from shared memory [Her91]. If a fault occurs, the construction may loop forever, so this precise construction is not strongly wait-free. However, there is a bound on the number of low-level operations on shared data that are required to implement a single high-level operation. Hence, this construction can be made strongly wait-free by adding an exit condition when the bound on the number of low-level steps is exceeded. Other constructions such as [Plo88] can be modified similarly. □

THEOREM 18.

. RMW registers are universal objects, if a bounded number of them are ∞-faulty: For all objects X, and for all $f < \infty$,

$$\mathbf{CONST}(RMW, (f, \infty), X) < \infty.$$

. *Consensus objects together with safe registers are universal objects, if a bounded number of them are ∞-faulty: For all objects X, and for all $f < \infty$,*

$$\mathbf{CONST}((\mathit{consensus}, \mathit{safe}), (f, \infty), X) < \infty.$$

PROOF. Note first that it is trivial to use a RMW register as a strongly wait-free implementation of an atomic register: $\mathbf{CONST}(\mathit{RMW}, (0, \infty), \mathit{atomic}) = 1$. ¿From this observation, Corollary 8 and Theorem 4, we have $\mathbf{CONST}(\mathit{RMW}, (f, \infty), \mathit{atomic}) \leq 20f + 8$ and from Theorem 14 we have $\mathbf{CONST}(\mathit{RMW}, (f, \infty), \mathit{consensus}) = 2f+1$. Composing these two facts with the construction in Theorem 17, and appealing to Theorem 3, proves the first part of the theorem.

To prove the second part of the theorem, known fault-intolerant constructions of atomic registers from safe registers ([Blo87, BP87, Lam86, LTV89, Pet83, PB87, SAG94, Tro89]) can be made strongly wait-free as was done above with Herlihy's construction: $\mathbf{CONST}(\mathit{safe}, (0, \infty), \mathit{atomic}) \leq \infty$. Composing these constructions and Theorem 17 we obtain $\mathbf{CONST}((\mathit{consensus}, \mathit{safe}), (0, \infty), \mathit{RMW}) \leq \infty$. The result follows by the first part of the theorem and Theorem 4. $\square$

The proof of the second part of this theorem can be readily generalized:

THEOREM 19. *Suppose X is a universal object, in the sense that for all objects Y, there is a strongly wait-free construction of Y from X. Then there is a fault-tolerant construction of Y from X: For all objects Y, and for all $f < \infty$,*

$$\mathbf{CONST}(X, (0, \infty), Y) < \infty \Rightarrow \mathbf{CONST}(X, (f, \infty), Y) < \infty.$$

In their recent paper on memory faults, Jayanti, Chandra, and Toueg give a direct fault-tolerant, strongly wait-free construction of consensus from consensus, which they use in an alternative proof of the second part of the previous theorem: consensus objects and registers can be used in fault-tolerant, strongly wait-free universal constructions [JCT92].

7. DISCUSSION, OPEN PROBLEMS, AND MORE ABOUT RELATED WORK

There remain many unresolved issues related to shared memory failures in distributed systems. Faulty versions of other shared data objects, such as multi-valued test&set registers, m-registers, and compare&swap registers, are of interest. Based on the constructions presented in this paper, fault tolerant construction of *fetch-and-add* and *swap* shared objects are presented in [AWW93]. We have tight bounds on only a few problems; more efficient constructions and corresponding lower bounds would also be interesting.

It would be particularly interesting to implement memory-fault tolerant data objects directly from similar, faulty objects, such as test&set from test&set, without using atomic registers, or read-modify-write from read-modify-write, without the overhead of a universal construction.

All our solutions are deterministic. It would be interesting to explore the use of randomization to tolerate memory failures. Also, there is much work to be done in exploring the effect of memory failures in other models, such as synchronous or semi-synchronous models.

Earlier, we referred several times to a paper by Jayanti, Chandra, and Toueg that explores other interesting issues related to memory failures [JCT92], as well as independently proving some of the same results presented here. Our fault models allow the study of individual faults within an object, setting the stage for an investigation of transient fault behavior. The work of Jayanti *et al.* assumes an object is either permanently faulty or completely reliable, but studies a range of fault types, from extremely malicious non-responsive faults, to benign crash behavior.

These definitions coincide in our notion of ∞-faulty objects, which correspond to *responsive, arbitrary* failures in the work of Jayanti *et al.* Specific results by Jayanti *et al.*, that are identical or closely related to ours include:

. A version of Theorem 2 that states general conditions on self-implementation, in any fault model, that suffice to apply the same recursive construction used in our proof.

. Composition results similar to Theorems 3 and 4.

. Register constructions analogous to Theorems 5, 7 and Corollary 8.

. A direct, fault-tolerant and strongly-wait free construction of consensus from consensus, which is used to prove universality results analogous to the second part of Theorem 18 and to Theorem 19.

The generalization of Theorem 2 depends on a notion of *gracefully degrading* construction that generalizes our notion of strongly wait-free construction. Gracefully degrading constructions from potentially faulty low-level primitives are required to exhibit the same type of fault behavior, when too many primitives fail, as the low-level primitives themselves suffer. Gracefully degrading constructions in different fault models can be composed in general ways, just as we compose strongly wait-free constructions in the ∞-fault model. We believe this generalization of strong wait-freedom is an important contribution.

Jayanti *et al.* also investigate an interesting range of fault models distinct from those we study, focusing on computability and universality issues. They show that unresponsive faults (in which invocations to objects may not evoke replies) are essentially impossible to overcome. In addition to the arbitrary fault model, they study two weaker responsive fault models, omission faults and crash faults. They show universal gracefully degrading constructions for omission faults, and argue that crash faults are too benign. That is, even if the low-level primitive objects fail by crashing, it is essentially impossible to create abstract components that gracefully degrade (or exhibit only crash faults when too many primitives crash).

Following on this work, three of us have defined a benign fault model intermediate in power between the omission and crash fault models, and shown that it supports universal, gracefully degrading constructions [AMT93].

8. ACKNOWLEDGEMENT

The authors benefited substantially from the efforts of outstanding referees. They caught several errors and omissions, and the presentation is much improved, in clarity and style, from our original draft.

From afek@research.att.com Tue Jul 25 18:35:17 1995 Return-Path: afek@research.att.com Received: from research.att.com (research.att.com [192.20.225.3]) by taurus.math.tau.ac.il (8.6.10/math) with SMTP id SAA02333 for ¡afek@math.tau.ac.il¿; Tue, 25 Jul 1995 18:33:12 +0300 Received: by research.att.com; Tue Jul 25 06:58 EDT 1995 Received: from ergo.research.att.com by hunny.research.att.com with smtp (Smail3.1.29.1 2) id m0sahhg-000d8LC; Tue, 25 Jul 95 06:59 EDT Received: by ergo.research.att.com (Smail3.1.29.1 2) id m0sahhG-000smeC; Tue, 25 Jul 95 06:58 EDT Message-Id: ¡m0sahhG-000smeC@ergo.research.att.com¿ Date: Tue, 25 Jul 95 06:58 EDT From: afek@research.att.com (Yehuda Afek) To: afek@math.tau.ac.il Status: RO

REFERENCES

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[Abr88] K. Abrahamson. On achieving consensus using a shared memory. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[ADG+94] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Trans. on Programming Languages and Systems*, 16(3):939–953, May 1994.

[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proc. 11th Ann. ACM Symp. on Principles of Distributed Computing*, pages 47–58, August 1992.

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. M. B. Vitányi. Wait-free test-and-set. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 85–94. Springer Verlag, November 1992.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, pages 281–294, September 1990.

[AMT93] Y. Afek, M. Merritt, and G. Taubenfeld. Benign failure models for shared memory. In *Proc. 7th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 725*, pages 69–83. Springer Verlag, September 1993.

[AWW93] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 159–170, August 1993.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *CACM*, 35(8):27–47, August 1992.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[BP87] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 222–231, 1987.

[CG89] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, April 1989.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 86–97, 1987.

[DGS88] D. Dolev, E. Gafni, and N. Shavit. Towards a non-atomic era: ℓ-exclusion as a test case. In *Proc. of the 20th Ann. ACM Symp. on Theory of Computing*, pages 78–92, 1988.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8(9):569, September 1965.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17:643–644, November 1974.

[Fis83] M. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In M. Karpinsky, editor, *Foundations of Computation Theory*, pages 127–140. Springer-Verlag, 1983. Lecture Notes in Computer Science, Vol. 158. Also, Yale University YALEU/DCS/RR-273.

[FLBB79] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proc. of the 20th IEEE Ann. Symp. on Foundation of Computer Science*, pages 234–254, 1979.

[FLBB85] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. Technical Report MIT/LCS/TM-290, M.I.T., Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA 02139, October 1985.

[FLM85] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1985.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[FMRT90] M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. In *Proc. of the 22nd ACM Ann. Symp. on Theory of Computing*, pages 106–116, October 1990. Also, SIAM Journal on Computing, to appear.

[FMT93] M. J. Fischer, S. Moran, and G. Taubenfeld. Space-efficient asynchronous consensus without shared memory initialization. *Information Processing Letters*, 45:101–105, February 1993.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HW87] M. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 13–26, January 1987.

[JCT92] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. In *Proc. of the 33rd IEEE Ann. Symp. on Foundation of Computer Science*. IEEE Computer Society Press, October 1992.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 69–84. Springer Verlag, November 1992.

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

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications Of The ACM*, 17:453–455, 1974.

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

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *TOPLAS*, 7(4):321–359, January 1989.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, April 1987.

[LTV89] M. Li, J. Tromp, and P. M. B. Vitányi. How to construct concurrent wait-free variables. Technical Report CS-8916, CWI, Amsterdam, April 1989. See also: pp. 488-505 in: *Proc. Int. Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

[MO93] M. Merritt and A. Orda. Efficient test&set algorithms for faulty shared memory. Unpublished Note, 1993.

[MTY92] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, August 1992.

[PB87] G. L. Peterson and J. E. Burns. Concurrent reading while writing II : The multi-writer case. In *Proc. of the 28th IEEE Ann. Symp. on Foundation of Computer Science*, pages 383–392, October 1987.

[Pet83] G. L. Peterson. Concurrent reading while writing. *ACM Trans. on Programming Languages and Systems*, 5(1):46–55, 1983.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus.* PhD thesis, M.I.T., August 1988.

[PS85] J. L. Peterson and Abraham Silberschatz. *Operating System Concepts.* (Third edition). Addison-Wesley Publishing Company, 1985.

[Rab82] M. O. Rabin. N-process mutual exclusion with bounded waiting by $4 \log n$ shared variables. *Journal of Computer and Systems Sciences*, 25:66–75, 1982.

[SAG94] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311, 1994.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14:473–540, 1982.

[SSW91] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proc. of 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 351–362, January 1991.

[Tro89] J. Tromp. How to construct an atomic variable. In J.C. Bermond and M. Raynal, editors, *Proc. of the 3rd Int. Workshop on Distributed Algorithms: Springer-Verlag LNCS 392*, pages 292–302, September 1989.

[VA87] P. M. B. Vitányi and B. Awerbuch. Shared register access by asynchronous hardware. In *Proc. of the 27th IEEE Ann. Symp. on Foundation of Computer Science*, pages 233–243, 1987.