Distrib. Comput. (2003) 16: 37–48
Digital Object Identifier (DOI) 10.1007/s00446-002-0075-3

DISTRIBUTED COMPUTING
© Springer-Verlag 2003

# Objects shared by Byzantine processes*

**Dahlia Malkhi**[1], **Michael Merritt**[2], **Michael K. Reiter**[3], **Gadi Taubenfeld**[4]

[1] School of Computer Science and Engineering, The Hebrew University of Jerusalem, Israel (e-mail: dalia@cs.huji.ac.il)
[2] AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971, USA (e-mail: mischu@research.att.com)
[3] Department of Electrical and Computer Engineering and Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA (e-mail: reiter@cmu.edu)
[4] The Open University and the Interdisciplinary Center, Israel (e-mail: gadi@cs.openu.ac.il)

Received: December 2000 / Accepted: July 2002

**Abstract.** Work to date on algorithms for message-passing systems has explored a wide variety of types of faults, but corresponding work on shared memory systems has usually assumed that only crash faults are possible. In this work, we explore situations in which processes accessing shared objects can fail arbitrarily (Byzantine faults).

**Keywords:** Byzantine faults, shared memory, emulation, consensus.

## 1 Introduction

### *1.1 Motivation*

It is commonly believed that message-passing systems are more difficult to program than systems that enable processes to communicate via shared memory. Many experimental and commercial processors provide direct support for shared memory abstractions, and increasing attention is being paid to implementing shared memory systems either in hardware or in software [Bel92,CG89,LH89,TKB92]. Moreover, several middleware systems have been built to implement shared memory abstractions in a message-passing environment. Of primary interest here are those that employ replication to provide fault-tolerant shared memory abstractions, particularly those designed to mask the arbitrary (Byzantine) failure of processes implementing these abstractions (e.g., see [PG89,SE+92,Rei96,KMM98,CL99,MR00]). These middleware systems generally guarantee that shared objects themselves do not "fail", and hence, that their integrity, safety properties, and access interfaces and restrictions, are preserved. Nevertheless, since legitimate clients accessing these objects might fail arbitrarily, they could corrupt the states of these objects in any way allowed by the object interfaces.

The question we address in this paper is: What power do shared memory objects have in such environments, in achieving coordination among distributed processes that access these objects? This question is daunting, as Byzantine faulty processes can configure objects in any way allowed by the object interfaces. Thus, seemingly even very strong shared objects such as consensus objects (which are universal for crash failures) might not be very useful in such a Byzantine environment, as faulty processes erroneously set their decision values. Surprisingly, although work to date on algorithms for message-passing systems has explored a wide variety of types of faults, corresponding work on shared memory systems has usually assumed that only crash faults are possible. Our work is the first study of the power of objects shared by Byzantine processes.

### *1.2 Summary of results*

We generalize the asynchronous crash-fault model of shared memory to accommodate Byzantine faults. We show how a variety of techniques can be used to cooperate reliably in the presence of Byzantine faults, including adding redundancy, *access control lists* that constrain faulty processes from accessing specific objects, and utilizing *persistent* objects (such as sticky bits [Plo89]) which cannot be overwritten. (We call objects that are not persistent, such as read/write registers, *ephemeral*.) We define a notion of shared object that is appropriate for this fault model, in which waiting between concurrent operations is permitted. We explore the power of some specific shared objects in this model, proving both universality and impossibility results, and finally identify some non-trivial problems that can be solved in the presence of Byzantine faults even when using only ephemeral objects.

The notions of consensus objects and sticky bits (a persistent, readable consensus object) in the Byzantine model, are formally defined in Section 2. Our results are:

*Universality result:* Our main result shows that sticky bits can be used to construct any other object (i.e., they are universal), assuming that the number of (Byzantine) faults, $t$, is bounded by $(t+1)(2t+1) \leq n$, where $n$ is the total number of processes.

* A preliminary version of the results presented in this paper appeared in *Proceedings of the 14th International Symposium on Distributed Computing*, Toledo, Spain, October 2000.

To prove this result, a universal construction is presented that works as follows: First, sticky bits are used to construct a *strong* consensus object, i.e., a consensus object whose decision is a value proposed by some *correct* process. Equipped with strong consensus objects, we proceed to emulate *any* shared memory object. Our emulation borrows closely from Herlihy's universal construction for crash faults [Her91], but differs in significant ways to cope with Byzantine failures.

*Bounds on faults:* We observe that strong consensus objects, used to prove the universality result, cannot be constructed when the possible number of faults is $t \geq n/3$. In Theorem 3.1, we observe that there exists a simple bounded-space universal object (the sticky bit), assuming $n \geq (t+1)(2t+1)$, and a trivial unbounded-space universal object assuming any number of faults. We prove that when a majority of the processes may be faulty, even *weak* consensus (i.e., a consensus object whose decision is a value proposed by some *correct* or *faulty* process) cannot be solved using many familiar ephemeral objects.

*Constructions using ephemeral objects:* While the universality result uses sticky bits, the impossibility result shows that consensus cannot be implemented using specific ephemeral objects. This raises the question of what can be done with such ephemeral objects. We show how various objects, such as $k$-set consensus and $k$-pairwise consensus, can be implemented in a Byzantine environment using only atomic registers. Then we show that familiar objects such as test&set, swap, compare&swap, and read-modify-write, can be used to implement election objects for any number of processes and under any number of Byzantine faults. (Election cannot be implemented using only atomic registers [MW87,TM96].)

We note that our work deals formally with the question of possibility/impossibility of implementing objects from others; investigating efficient implementation of such object types in practice is beyond the scope of this paper.

### *1.3 Related work*

The power of various shared objects has been studied extensively in shared memory environments where processes may fail benignly, and where every operation is wait-free. (An operation is *wait-free* if it is guaranteed to return within a finite number of steps.) Objects that can be used (together with atomic registers) to build wait-free implementations of any other object are called *universal objects*. Previous work on shared objects provided methods (called *universal constructions*) to transform sequential specifications of arbitrary shared objects into wait-free concurrent implementations that use universal objects [Her91,Plo89,JT92]. In particular, Plotkin showed that sticky bits are universal [Plo89], and independently, Herlihy proved that consensus objects are universal [Her91]. Herlihy also classified shared objects by their consensus number: that is, the maximum number of processes that can reach consensus using multiple instances of the object and read/write registers [Her91]. Attie investigates the power of shared objects accessed by Byzantine processes for achieving wait-free Byzantine agreement. He proves that strong agreement is impossible to achieve using *resettable* objects, i.e., objects that can be reset back to their initial setting, and constructs weak agreement using sticky bits [Att00].

Assume that at some point in a computation a shared register is set to some unexpected value. There are two complementary ways to explain how this may happen. One is to assume that the register's value was set by a Byzantine process (as may happen in the model of this paper). The other way is to assume that the processes are correct, but that the register itself is faulty. The subject of memory faults (as opposed to process faults) has been investigated in several papers [AGMT95, JCT98]. These papers assume any number of process crash failures, but bound the number of faulty objects, whereas we bound the number of (Byzantine) faulty processes, but each might sabotage all the objects to which it has access.

As described in the introduction, our focus on a shared memory Byzantine environment is driven by previous work on message-passing systems that emulate shared memory abstractions tolerant of Byzantine failures (e.g., [PG89,SE+92, Rei96,KMM98,CL99,MR00]). Though these systems guarantee the correctness of the emulated shared objects themselves, the question is what power do these objects provide to the correct processes that use them, in the face of corrupt processes accessing them.

## 2 Model and definitions

Our model of computation consists of an asynchronous collection of $n$ processes, denoted $p_1, \ldots, p_n$, that communicate via shared objects. In any run any process may be either correct or faulty. Correct processes are constrained to obey their specifications, while faulty processes can deviate arbitrarily from their specifications (Byzantine failures) limited only by the assumptions stated below. We denote by $t$ the maximum number of faulty processes.

### *2.1 Shared objects with access control lists*

Each shared object presents a set of operations. For example, $x.\mathsf{op}$ denotes operation $\mathsf{op}$ on object $x$. For each such operation on $x$, there is an associated access control list (ACL) that names the processes allowed to *invoke* that operation. Each operation execution begins with an invocation by a process in the operation's ACL, and remains pending until a response is received by the invoking process. The ACLs for two different operations on the same object can differ, as can the ACLs for the same operation on two different objects. The ACLs for an object do not change. For any operation $x.\mathsf{op}$, we say that $x$ is $k$-$\mathsf{op}$ if the ACL for $x.\mathsf{op}$ lists $k$ processes. We assume that a process not on the ACL for $x.\mathsf{op}$ cannot invoke $x.\mathsf{op}$, regardless of whether the process is correct or Byzantine (faulty). That is, a (correct or faulty) process cannot access an object in any way except via the operations for which it appears on the associated ACLs.

We note that the systems that motivated our study typically employ replicated servers to fault-tolerantly emulate shared memory abstractions. Therefore, ACLs can be implemented, e.g., by storing a copy of the ACL with each replica-server and filtering out disallowed operations before applying them to the replica. In this way, only operations allowed by the ACLs will be applied at correct replicas.

### 2.2 Fault tolerance and termination conditions

In wait-free fault models, no bound is assumed on the number of potentially faulty processes. (Hence, no process may wait upon an action by another.) Any operation by a process $p$ on a shared object must terminate, regardless of the concurrent actions of other processes. This model supports a natural and powerful notion of abstraction, which allows complex implementations to be viewed as atomic [HW90]. We extend this model in two ways: first, we make the more pessimistic assumption that process faults are Byzantine. To overcome Byzantine failures, it is necessary to use redundancy to overcome failures of peers. Second, we make the more optimistic assumption that the number of faults is bounded by $t$, where $t$ is less than the total number of processes, $n$. With the number of failures bounded away from $n$, it becomes possible for processes to coordinate with each other, meaning that processes may need to wait for each other within individual operation implementations.

An example that may provide some intuition is a sticky bit object emulated by an ensemble of data servers, such that the value written to it must reflect a value written by some *correct* process. A distributed emulation may implement this object by having servers set the object's value only when $t+1$ different processes write to it the same value. Of course, this object will be useful only when any value written to the object is indeed written by at least $t+1$ processes, and so an application must guarantee that $t+1$ correct processes write identical values. Below, we will see examples of such constructions.

Such an implementation is not wait-free, and raises the question of appropriate termination conditions for object invocations in a Byzantine environment. To address such concerns, we introduce two object properties, $t$-threshold and $t$-resilience. The first captures termination conditions appropriate for an object on which each client should invoke a single operation, and which functions correctly once enough correct processes access it. The second is appropriate when processes perform multiple operations on an object, each of which may require support from a collection of correct processes.

*$t$-threshold:* For any operation $x.\mathsf{op}$, we say that $x.\mathsf{op}$ is $t$-threshold if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run $\rho$ in which at least $n-t$ correct processes invoke $x.\mathsf{op}$.

*$t$-resilience:* For any operation $x.\mathsf{op}$, we say that $x.\mathsf{op}$ is $t$-resilient if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run $\rho$ in which each of at least $n-t$ correct processes infinitely often has a pending invocation of $x.\mathsf{op}$.

An object is $t$-threshold ($t$-resilient) if all the operations it supports are $t$-threshold ($t$-resilient). Notice that $t$-threshold implies $t$-resilience, but not vice versa.

In these definitions, it may seem odd that termination is guaranteed only when correct processes access the object using the *same* operation. On the surface, it seems more natural to require termination in runs where at least $n-t$ correct processes access the object via *any* operation. Our definitions are actually more general, since one could encode different operations to be invocations of a single operation with different operands.

The emphasis in this paper is on issues of computability and universality, so the specific objects studied, such as consensus, registers, or test&set, are abstractions with little immediate practical interest. From a practical perspective, more interesting fault-tolerant objects would include shared databases (and e-commerce applications they support). A $t$-threshold database guarantees termination of updates once enough other updates are invoked, and a $t$-resilient database guarantees termination of updates if enough sustained update activity takes place.

### 2.3 Object definitions

Below we specify some of the objects used in this paper.

*Atomic registers:* An atomic register $x$ is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{write}(v)$ where $v \neq \perp$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{write}()$ returns the value written in the last preceding $x.\mathsf{write}()$ operation. Throughout this paper we employ wait-free atomic registers, i.e., $x.\mathsf{read}$ or $x.\mathsf{write}()$ operations by correct processes eventually return (regardless of the behavior of other processes).

*Sticky bits:* A sticky bit $x$ is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{write}(v)$ where $v \in \{0,1\}$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{write}()$ returns the value written in the first $x.\mathsf{write}()$ operation. We will be concerned with wait-free sticky bits.

*Weak consensus objects:* A weak (binary) consensus object $x$ is an object with one operation: $x.\mathsf{propose}(v)$, where $v \in \{0,1\}$, satisfying: (1) In any run, the $x.\mathsf{propose}()$ operation returns the same value, called the *consensus value*, to every correct process that invokes it. (2) In any finite run in which all participating processes are correct (no Byzantine faults), if the consensus value is $v$, then some process invoked $x.\mathsf{propose}(v)$.

*Strong consensus objects:* A strong (binary) consensus object $x$ strengthens the second condition above to read: (2) If the consensus value is $v$, then some *correct* process invoked $x.\mathsf{propose}(v)$.

Observe that one sticky bit does not trivially implement a strong consensus object, where each process first writes this bit and then reads it and decides on the value returned. The first process to write the bit might be a faulty one, violating the requirement that the consensus value must be proposed by some *correct* process. (In Lemmas 3.1 and 3.2 we describe more complex implementations of strong consensus from sticky bits.) Indeed, strong consensus objects do not have sequential runs: the additional condition, using redundancy to mask failures, requires at least $t+1$ processes to invoke $x.\mathsf{propose}()$ before any correct process returns from this operation. (In addition, Theorem 3.2 in Section 3.4 shows that $t$-resilient strong consensus objects are ill-defined when $t \geq n/3$.)

Throughout the paper, unless otherwise stated, by a consensus object we mean a *strong* consensus object. Also, atomic registers and sticky bits are always assumed to be *wait-free*.

## 3 Implementing any shared object

This section contains the main result of this paper, the construction of any $t$-resilient object from wait-free sticky bits. That is, we show that sticky bits are universal when the number of faults is small enough.

### *3.1 Specifying fault-tolerant objects*

We assume any fault-tolerant object, $o$, is specified by two relations:

$$apply \subset INVOKE \times STATE \times STATE,$$

$$\text{and } reply \subset INVOKE \times STATE \times RESPONSE,$$

where INVOKE is the object's domain of invocations, STATE is its domain of states (with a designated set of start states), and RESPONSE is its domain of responses. Moreover, we assume INVOKE is the union of disjoint sets of invocations, $\text{INVOKE}_p$, for each process $p$, and similarly that RESPONSE is the union of disjoint sets of responses, $\text{RESPONSE}_p$, for each process $p$. The $apply$ relation denotes a nondeterministic state change based on the specific pending invocation and the current state (invocations do not block: we require a target state for every invocation and current state), and the $reply$ relation nondeterministically determines the calculated response, based on the pending invocation and the updated state.[1] It is necessary to define two relations because in fault-tolerant objects (such as strong consensus), the response may depend on later invocations. The $apply$ relation allows the state to be updated once the invocation occurs, without yet determining the response. The $reply$ relation may only allow a response to be determined when other pending invocations update the state.

The $apply$ and $response$ relations specify the runs of object $o$ as follows. Consider a (finite or infinite) sequence $\rho$ of invocations and responses. The sequence $\rho$ is *well-formed* if for each process $p$, the subsequence of $\rho$ consisting of invocations and responses of $p$, $\rho_p$, begins with an invocation, and alternates between invocations and responses (either forever, or ending in either an invocation or response).

Runs of the object $o$ are (finite or infinite) well-formed sequences of invocations and responses $\rho$ of $o$ that can be annotated with the elements of $apply$ and $reply$ in the natural way: First, for each process $p$, somewhere between every consecutive pair of invocations and responses by $p$ in $\rho$, $invoke_p$ and $response_p$, insert an element $(invoke_p, s_1, s_2)$ of $apply$ followed (not necessarily immediately) by an element $(invoke_p, s_3, reply_p)$ of $reply$. If $\rho_p$ ends in an invocation $invoke_p$, optionally insert an element of $apply$, $(invoke_p, s_1, s_2)$ sometime after $invoke_p$. The sequence of elements of $apply$ and $reply$ must satisfy the natural state machine semantics: the state $s_1$ of the first element $(invoke_p, s_1, s_2)$ of $apply$ must be a start state, for every two consecutive elements of $apply$, $(invoke_p, s_1, s_2)$ and $(invoke'_p, s'_1, s'_2)$, $s_2$ must equal $s'_1$, and for every element of $reply$, $(invoke_p, s, reply_p)$, the last preceding element of $apply$ must have the form $(invoke'_{p'}, s', s)$.

Intuitively, the object $o$ begins in any of its start states, (hence processes cannot choose which start state), and pending invocations by process $p$ enable the later occurrence of an $apply$ event, which updates the state. When enabled, a $reply$ event does not update the object state, but enables the later response. One can easily derive the $t$-resilience and $t$-threshold conditions as requirements on the $apply$ and $response$ relations, as exemplified for a particular object below.

For example, a $t$-threshold strong consensus object can be specified as follows: STATE is the set containing all pairs $(X, Y) \in \mathcal{P} \times \mathcal{P}$ where $\mathcal{P} = \{S \subseteq \{p_1, \ldots, p_n\} : |S| \leq t + 1\}$. The pair $(\emptyset, \emptyset)$ is the single start state. For $v \in \{0, 1\}$ and for all $X, Y \in \mathcal{P}$, the $apply$ relation is the smallest relation satisfying:

$$|X| \leq t \wedge |Y| \leq t \Rightarrow (\mathsf{propose}(0)_p, (X, Y), (X \cup \{p\}, Y)) \in apply$$

$$|X| \leq t \wedge |Y| \leq t \Rightarrow (\mathsf{propose}(1)_p, (X, Y), (X, Y \cup \{p\})) \in apply$$

$$|X| > t \vee |Y| > t \Rightarrow (\mathsf{propose}(v)_p, (X, Y), (X, Y)) \in apply$$

and the $reply$ relation is the smallest relation satisfying:

$$|X| > t \Rightarrow (\mathsf{propose}(v)_p, (X, Y), return(0)) \in reply$$

$$|Y| > t \Rightarrow (\mathsf{propose}(v)_p, (X, Y), return(1)) \in reply.$$

Hence, each invocation of a $\mathsf{propose}(0)$ operation (respectively, $\mathsf{propose}(1)$ operation) enables $apply$ to record the invoking process as having proposed 0 (respectively, 1). Concurrent invocations introduce race conditions (as to which application of $apply$ occurs first.) Once $t + 1$ processes propose the same value, the state is committed to that binary value, and the responses of pending invocations are enabled. (Note that $2t + 1$ invocations may be required before $t + 1$ have the same value, and so this object is neither $t$-threshold nor $t$-resilient unless $n - t > 2t$, or $n > 3t$. We show below, in Theorem 3.2, that this restriction is implied by the definition of strong consensus.)

### *3.2 The universal construction*

For the purposes of the universal construction below, we resolve any nondeterminism, and assume that there is a single start state, that the $apply$ relation is a function from $INVOKE \times STATE$ to $STATE$, and that the $reply$ relation is a partial function from $INVOKE \times STATE$ to $RESPONSE$. Given these restrictions, we may assume, without loss of generality, that the object's domain of states is the set of strings of invocations, and that the function from $INVOKE \times STATE$ to $STATE$, simply appends the pending invocation to the current state.

**Theorem 3.1** *Any $t$-resilient object can be implemented using:*

[1] This formulation generalizes Herlihy's specification of wait-free objects by a single relation $apply \subset INVOKE \times STATE \times STATE \times RESPONSE$, restricted (by the wait-free condition) to have at least one target state and response defined for any pair $INVOKE \times STATE$ [Her91]. This formulation is insufficient to define fault-tolerant objects such as strong consensus.

*1.* $(t+1)$-write(), $n$-read *sticky bits and* 1-write(), $n$-read *sticky bits, provided that* $n \geq (t+1)(2t+1)$*; or*

*2.* $(2t+1)$-write(), $(2t+1)$-read *sticky bits and* 1-write(), $n$-read *sticky bits, provided that* $n \geq (2t+1)^2$.

Figure 2 describes a universal implementation. In the lemmas, we provide two constructions of (strong) binary consensus objects using sticky bits. The two constructions differ in the access restrictions imposed on the sticky bits.

**Lemma 3.1** *If* $n \geq (t+1)(2t+1)$*, then an* $n$-propose() $t$-*threshold consensus object can be implemented using* $(t+1)$-write(), $n$-read *sticky bits.*

*Proof.* Let $o$ be the consensus object that is being implemented. Figure 1 depicts a construction of consensus as follows: Partition the $n$ processes $p_1, \ldots, p_n$, into blocks $B_1, \ldots, B_{2t+1}$, each of size at least $t+1$, and let $x_1, \ldots, x_{2t+1}$ be sticky bits with the property that the ACL for $x_i$.write() is $B_i$ (or a $(t+1)$-subset thereof) and the ACL for $x_i$.read is $\{p_1, \ldots, p_n\}$. For a correct process $p \in B_i$ to emulate $o$.propose($v$), it executes $x_i$.write($v$) (or does nothing if $p$ is not in the ACL for $x_i$) and, once that completes, repeatedly executes $x_j$.read for all $1 \leq j \leq 2t+1$ until none return $\perp$. Process $p$ chooses the return value from $o$.propose($v$) to be the value that is returned from the read operations on a majority of the $x_j$'s. All correct processes obtain the same return value from their $o$.propose() emulations because the $x_i$'s are sticky. If no correct process emulates $o$.propose($v$), then $v$ will not be returned from the reads on a majority of the $x_j$'s and thus will not be the consensus value. Because each correct process reads $x_j$, $1 \leq j \leq 2t+1$, until none return $\perp$, termination is guaranteed provided that each sticky bit is set. Since each $x_j$ has $t+1$ processes writing to it, it follows that $o$.propose() is guaranteed to return when at least $n-t$ perform propose() operations.

**Lemma 3.2** *If* $n \geq (2t+1)^2$*, then an* $n$-propose() $t$-*threshold consensus object can be implemented using* $(2t+1)$-write(), $(2t+1)$-read *sticky bits and* 1-write(), $n$-read *sticky bits.*

*Proof.* The construction here is similar in nature to the previous lemma (see also Fig. 1 for intuition). Let $o$ be the consensus object that is being implemented. Let $r_1, \ldots, r_n$ be 1-write(), $n$-read sticky bits such that the ACL for $r_i$.write() is $\{p_i\}$. Partition the $n$ processes $p_1, \ldots, p_n$ into blocks $B_1, \ldots, B_{2t+1}$, each of size at least $2t+1$, and let $x_1, \ldots, x_{2t+1}$ be sticky bits such that for each $i$, $1 \leq i \leq 2t+1$, the ACL's for each $x_i$.write() and $x_i$.read is some $(2t+1)$-sized subset of $B_i$. For a correct process $p_j \in B_i$ to emulate $o$.propose($v$), if it is in the ACL for $x_i$ it executes $x_i$.write($v$) and then executes $r_j$.write($x_i$.read). Regardless of whether $p_j$ is in the ACL for $x_i$, process $p_j$ then repeatedly reads the (single-writer) bits of all processes until for each $B_k$, it observes the same value $V_k$ in the bits of $t+1$ processes in $B_k$; note that $V_k$ must be the value returned by $x_k$.read (to a process allowed to execute $x_k$.read). The value that occurs as $t+1$ such $V_k$'s is selected as the return value from $o$.propose($v$). Because $x_i$ is sticky and $B_i$ contains at most $t$ faulty processes, $V_i$ is unique; thus, all correct processes obtain the same return value from their $o$.propose() emulations. If no correct process emulates $o$.propose($v$), then $v$ cannot occur in $t+1$ distinct $V_j$'s.

*3.3 Proof of Theorem 3.1*

For simplicity, we initially describe a universal construction of objects for which the domain of invocations is finite. Subsequently, we explain how to modify the construction to implement objects with (countably) infinite invocation domains.

The construction conceptually mimics Herlihy's construction showing that consensus is universal for wait-free objects in the fail-stop model [Her91]. Due to the possibility of arbitrarily faulty processes in our system model, however, the construction below differs in significant ways. First, we replace the objects in the original construction with objects (sticky bits and strong consensus) that cannot be corrupted by Byzantine processes. This ensures that each operation by Byzantine processes either has no impact, or appears as (the same) valid operation to the correct processes. The new construction also labors to ensure that operations by correct processes eventually complete – the "helping" mechanism in the original construction is inadequate. The new helping mechanism requires the cooperation of $n-t$ processes for each successful operation, and so is $t$-resilient rather than wait-free. Finally, we generalize the object specification to allow the implementation of objects that are not themselves wait-free.

There are two principal shared data structures:

1. For each process $p_i$ there is an unbounded array, Announce$[i][1...]$, each element of which is a "cell", where a cell is an array of $\lceil \log(|INVOKE|) \rceil$ sticky bits. The Announce$[i][j]$ cell describes the $j$-th invocation (operation name and arguments) by $p_i$ on $o$. Accordingly, the ACL for the write() operation of each sticky bit in each cell of Announce$[i]$ names $p_i$: all processes may read all Announce[][] cells.

2. The object itself is represented as an unbounded array Sequence$[1...]$ of process-id's, where each Sequence$[k]$ is a $\lceil \log(n) \rceil$ string of $t$-threshold, strong binary consensus objects, accessible by all processes. We refer to the value represented by the string of bits in Sequence$[k]$ simply as Sequence$[k]$. Intuitively, if Sequence$[k] = i$ and Sequence$[1], \ldots,$ Sequence$[k-1]$ contains the value $i$ in exactly $j-1$ positions, then the $k$-th invocation on $o$ is described by Announce$[i][j]$. In this case, we say that Announce$[i][j]$ has been *threaded*.

The universal construction of object $o$ is described in Fig. 2 as the code process $p_i$ executes to implement an operation $o$.op. In outline, the emulation works as follows: process $p_i$ first announces its next invocation, and then threads unthreaded, announced invocations onto the end of Sequence. It continues until it sees that its own operation has been threaded, and that enough additional invocations (if any) have been threaded, that it can compute a response and return. To assure that each announced invocation is eventually threaded, the correct processes first try to thread any announced, unthreaded cell of process $p_{\ell+1}$ into entry Sequence$[k]$, where $\ell = k \pmod{n}$. (Once process $p_{\ell+1}$ announces an operation, at most $n$ other operations can be threaded before $p_{\ell+1}$'s.)

In more detail, process $p_i$ keeps track of the first index of Announce$[i]$ that is vacant in a variable denoted MyNextAnnounce, and first (line 1) writes the invocation, bit by bit, into Announce$[i]$[MyNextAnnounce], and (line 2) increments MyNextAnnounce. To keep track of which cells it

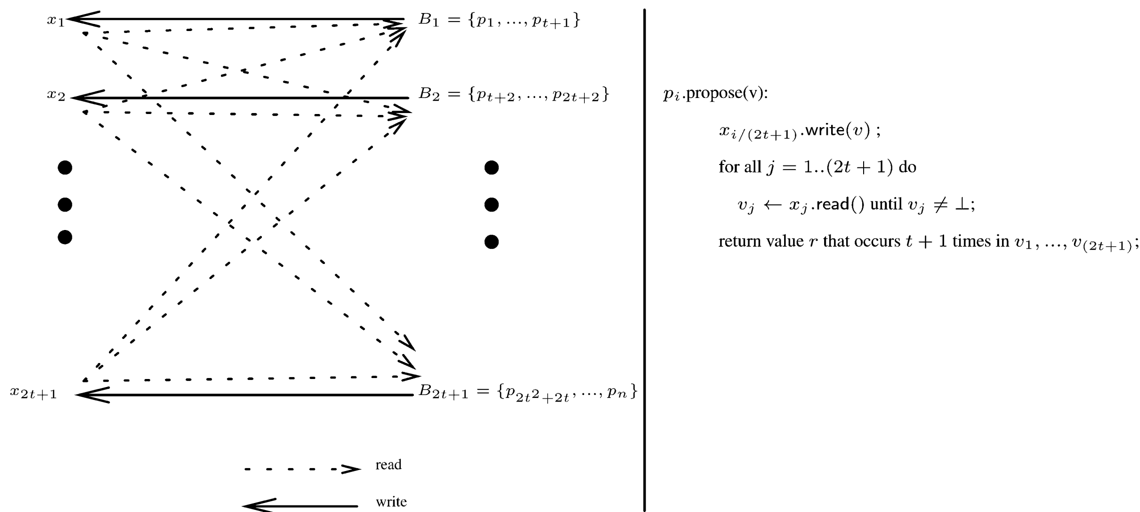


**Fig. 1.** A construction of consensus using $2t + 1$ sticky bits

has seen threaded (including its own), $p_i$ keeps $n$ counters in an array NextAnnounce$[1..n]$, where each NextAnnounce$[j]$ is one plus the number of times $i$ has read cells of $j$ in Sequence, and hence the index of Announce$[j]$ where $i$ looks to find the next operation announced by $j$. Hence, having incremented MyNextAnnounce, NextAnnounce$[i]$ = MyNextAnnounce$-$1 until the current operation of $p_i$ has been threaded.

This inequality is thus one disjunct (line 3) in the loop (lines 4-10) in which $p_i$ threads cells. Once $p_i$'s cell is threaded, (and NextAnnounce$[i]$ = MyNextAnnounce, see the first invariant in Lemma 3.3), the next conjunct (again line 3) keeps $p_i$ helping to thread cells until it sees that enough progress has been made that a response to its own threaded operation can be computed. (At which time it exits the loop and returns the associated value (line 15).) Notice that in some cases, this may require any finite number of additional operations to be threaded after $invoke$, but by the $t$-resilient properties of the abstract object being implemented, as long as $o$.op operations of correct processes are eventually threaded, eventually $invoke$ can return. For example, if $invoke$ is an invocation of propose() for a strong consensus object, then it can return once at least $t + 1$ propose() invocations with identical values occur. Process $p_i$ keeps an index NextSeq which points to the next entry in Sequence[1...] whose element it has not yet accessed.

Figure 3 portrays the shared data structures and some structures local to process $p_i$ at a point when process $p_1$ has threaded two operations (and is beginning to write the invocation of a third), process $p_n$ has threaded one operation, and a fourth operation is in process of being threaded. (This might be the operation process $p_2$ has begun.)

To thread cells, process $p_i$ proposes (line 9) the binary encoding of a process id, $\ell+1$, bit by bit, to Sequence[NextSeq]. In choosing $p_{\ell+1}$, process $p_i$ first checks (first disjunct, line 7) that Announce$[\ell + 1]$[NextAnnounce$[\ell + 1]$] contains a valid encoding of an operation invocation. (In particular, none of the sticky bits are still set to $\perp$.) As discussed above, $p_i$ gives preference (line 4) to a different process for each cell in Sequence. (All active, correct processes will eventually agree to give preference to any pending invocation, assuring it will eventually be threaded.)

Starting (line 5) with the *emptystring*, $p_i$ accumulates (line 9) the bit-by-bit encoding of the $id$ being recorded in Sequence[NextSeq] into a local variable, NameSuffix. If a bit being proposed by $p_i$ is not the result returned (second disjunct, line 7), then $p_i$ searches (line 8) for another process to help, whose $id$ matches the bits accumulated in NameSuffix. (The properties of strong consensus assure that such a process exists, as summarized in the second invariant below:)

**Lemma 3.3** *Let $\alpha$ be a run of the universal construction in Fig. 2. For all $i$, $1 \leq i \leq n$, the following are invariant properties (of the shared variables and those local to process $p_i$) in $\alpha$:*

*1.* NextAnnounce$[i] \in$ {MyNextAnnounce $-1$, MyNextAnnounce},

*2.* NameSuffix *is the suffix of the binary encoding of some $j$, $1 \leq j \leq n$, such that* Announce$[j]$[NextAnnounce$[j]$] *is valid.*

*Proof.* The invariants follow directly from the code in Fig. 2 as follows:

1. Initially, this holds because both MyNextAnnounce and NextAnnounce$[i]$ are set to 1. During an invocation of $o$.op at $p_i$, first MyNextAnnounce is incremeted by 1 (line 2), so the invariant still holds. Then in the 'while' loop, for any NameSuffix the value in NextAnnounce[NameSuffix] may increment (in line 12) on condition that Announce[NameSuffix][NextAnnounce[NameSuffix]] is defined. For NameSuffix $= i$, this can occur when NextAnnounce$[i]$ = MyNextAnnounce $-$ 1, i.e., only once. After it is incremented, the condition cannot hold again until the next invocation of $o$.op.

2. NameSuffix is determined bit-by-bit by the NextSeq'th entry in the Sequence array. The latter is set using strong consensus bits that are each proposed by some correct process. Since each correct process stipulates that the suffix

type: $ID$: array of $\lceil \log(n) \rceil$ strong consensus objects
$CELL$: array of $\lceil \log(|INVOKE|) \rceil$ sticky bits
global variables:
Announce$[1..n][1...]$, array of $CELL$: for all $j$, $1 \leq j \leq n$, and $k$, elements of Announce$[j][k]$ are writable by $p_i$
Sequence$[1...]$, infinite array of $ID$s: each accessible by all processes
variables local to process $p_i$:
MyNextAnnounce, index of next vacant cell in Announce$[i]$, initially 1
NextAnnounce$[1..n]$, for each $1 \leq j \leq n$, index in Announce$[j][]$ of next operation of $p_j$ to be read by $p_i$, initially 1
CurrentState $\in STATE$, $p_i$'s view of the state of $o$, initially the initial state of $o$
NextSeq, next position to be threaded in Sequence$[]$ as seen by $p_i$, initially 1
NameSuffix, $\lceil \log(n) \rceil + 1$ bit string
$o$.op:
(1) write, bit by bit, the invocation $invoke$ into Announce$[i]$[MyNextAnnounce]
(2) MyNextAnnounce++
(3) **while** ((NextAnnounce$[i]$ $<$ MyNextAnnounce) or ((NextAnnounce$[i]$ $\geq$ MyNextAnnounce) and ($reply(invoke$, CurrentState) is not defined))) **do** // Apply operations until $invoke$ is applied and $p_i$ can return, // each while loop iteration applies exactly one operation.
(4) $\ell \leftarrow$ NextSeq $(\bmod\ n)$ // Select preferred process to help.
(5) NameSuffix $\leftarrow$ *emptystring*
(6) **for** $k = 0$ to $\lceil \log(n) \rceil$ **do** // Loop applies the operation one bit per iteration.
(7) **while** ((Announce$[\ell + 1]$[NextAnnounce$[\ell + 1]$] is invalid) or (NameSuffix is not a suffix of the bit encoding of $\ell + 1$)) **do** // Search for a valid process index to propose.
(8) $\ell \leftarrow \ell + 1\ (\bmod\ n)$ **od**
(9) $prepend$(NameSuffix, Sequence[NextSeq]$[k]$.propose$((\ell + 1) \& (2^k))$) // Propose the $k$'th bit $((\ell + 1)\&(2^k))$ of $\ell + 1$.
(10) **od** // A new cell has been threaded by NameSuffix in Sequence[NextSeq].
(11) CurrentState $\leftarrow apply$(Announce[NameSuffix][NextAnnounce[NameSuffix]], CurrentState)
(12) NextAnnounce[NameSuffix] $++$
(13) NextSeq++
(14) **od**
(15) $return(reply(invoke$, CurrentState$))$

**Fig. 2.** Universal implementation of $o$.op at $p_i$

set in Sequence[NextSeq] indeed belongs to a process $j$ whose corresponding Announce entry is valid, and since NextAnnounce$[j]$ is determined by the preceding number of Sequence entries containing $j$'th name, the invariant follows.

Once process $p_i$ accumulates all the bits of the threaded cell into NameSuffix (the termination condition (line 6) of the **for** loop (lines 7-10)), it can update (line 11) its view of the object's state with this invocation, and increment its records of (line 12) process NameSuffix's successfully threaded cells and (line 13) the next unread cell in Sequence. Having successfully threaded a cell, $p_i$ returns to the top of the **while** loop (line 3).

The result is that each process' $k$-th iteration through the loop of lines 3-14 corresponds to the threading of a unique $k$-th operation on the object being emulated, as stated in the following lemma:

**Lemma 3.4** *Let $\alpha$ be a run of the universal construction in Fig. 2. For any $i, j$, $1 \leq i, j \leq n$, if processes $p_i$ and $p_j$ both execute the* **while** *loop (lines 3-14) at least $m \geq 0$ times in $\alpha$ (counting over successive executions of $o$.op), then the local variables* NextAnnounce$[1..n]$, CurrentState, NextSeq, *and* NameSuffix *local to process $p_i$ upon $p_i$'s $m$'th execution of line 14 have the same values as the values of the corresponding local variables of process $p_j$ upon $p_j$'s $m$'th execution of line 14.*

This lemma, which implies the sequencing and correct semantics of each operation, follows easily from the sequential ordering of invocations in Sequence and the application of the *apply* and *reply* functions. The proper termination of all correct operations follow as argued above from the $t$-threshold property of the embedded consensus objects, the preference mechanism, and from the $t$-resilience properties of the specification for object $o$.

The construction and this argument address objects with finite domains of invocation. We next briefly outline the modifications necessary to accommodate objects with (countably) infinite domains of invocation. The quandary here is that the representations of invocations using sticky bits are unbounded. Suppose we naively change the type $CELL$ to (unbounded) sequence of sticky bits.

When process $p_i$ attempts to read (line 7) an invocation in Announce$[\ell + 1]$[NextAnnounce$[\ell + 1]$], a faulty process might cause $p_i$ to read forever, by itself writing forever, in such a way that each finite prefix is a valid but incomplete encoding of an invocation. (For any encoding, such a sequence exists by König's lemma.) This problem can be avoided by looping over all $i$, $1 \leq i \leq n$, reading successive bits from each Announce$[\ell + 1]$[NextAnnounce$[i]$] entry, starting as before with the next bit of NextAnnounce$[\ell + 1]$, until one of the accumulated strings validly encodes an invocation. Details of the bookkeeping required, and the argument that correct invocations are eventually threaded, are left to the reader. (Though note that the number of invocations that may be threaded before a correct process's announcement is now dependent on the relative lengths of different encodings.)

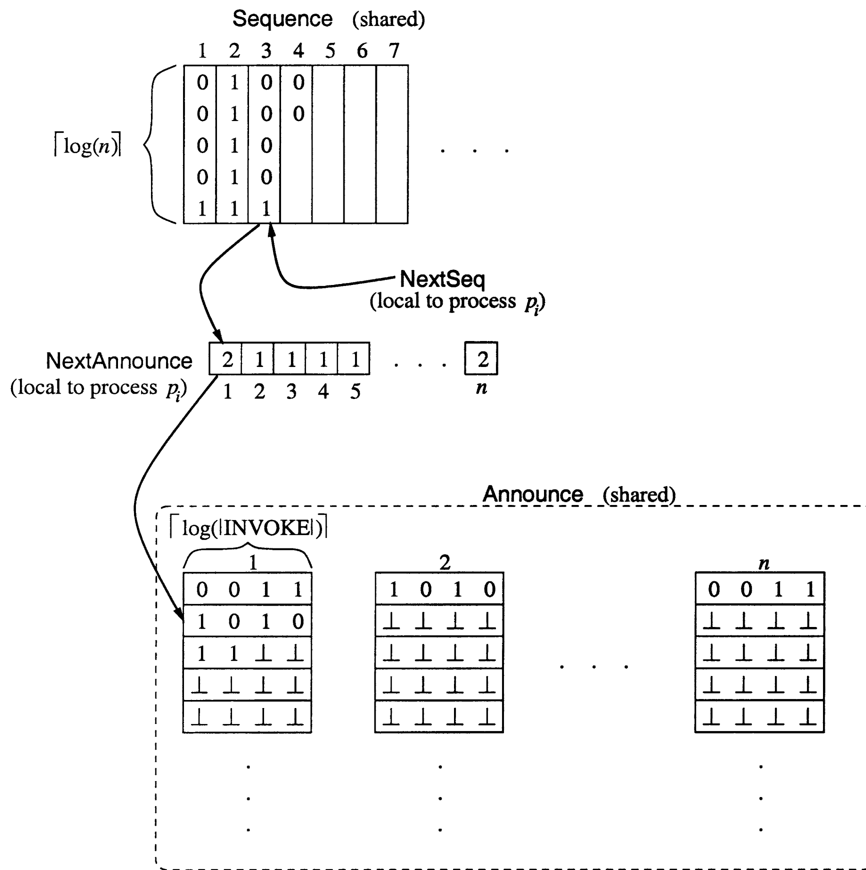


**Fig. 3.** Partial state of a run of the universal implementation. (See text.)

*3.4 Resilience and impossibility*

The proof of Theorem 3.1 presents a universal construction of $t$-resilient objects, where $(t+1)(2t+1) \leq n$. Naturally, one would like to know whether there are more fault-tolerant universal constructions, indeed whether wait-free universal constructions exist (in which $t = n$). Focusing on improving the bound $(t+1)(2t+1) \leq n$ in Theorem 3.1, that is, finding a universal construction or impossibility proofs for larger values of $t$, we note that the construction in Fig. 2 builds modularly on $t$-threshold strong consensus. The $(t+1)(2t+1) \leq n$ bound of Theorem 3.1 follows from the constructions of strong consensus from sticky bits, in Lemmata 3.1 and 3.2. Constructions of strong consensus from sticky bits for larger values of $t$ would imply a more resilient universality result. The theorem below demonstrates that such a search is bounded by $t < n/3$ even for $t$-resilience (and hence, a fortiori for $t$-threshold).

**Theorem 3.2** *For $t \geq n/3$, there are no $t$-threshold (or $t$-resilient) $n$-*propose() *(strong) consensus objects.*

*Proof.* Let $t \geq n/3$ and assume to the contrary that there exists a $t$-threshold $n$-propose() (strong) consensus object. Let $P_0$ and $P_1$ be two sets of processes such that for each $P_i$ (where $i \in \{0,1\}$) the size of $P_i$ is $\lceil n/3 \rceil$ and all processes in $P_i$ propose the value $i$ (i.e., have input $i$). Run these two groups as if all the $2\lceil n/3 \rceil$ processes are correct until they all return a consensus value. (They must all do so, since $n-t$ processes have invoked propose().) Without loss of generality, let this value be 0. Next, let all the remaining processes propose 1 and run until all return 0. We can now assume that all the processes in $P_0$ are faulty and reach a contradiction.

The extension to the $t$-resilient case is straightforward.

We point out that this result does not imply that universal constructions are impossible for this (or higher) failure thresholds. In fact, it is easy to define objects that are universal for any number of faults. An example is the wait-free *append-queue* object, which supports two operations. The first appends a value onto the queue, and the second reads the entire contents of the queue. By directly appending invocations onto the queue, the entire history of the object can be read.

## 4 Ephemeral objects

Sticky bits are examples of very simple persistent objects. Theorem 3.1 demonstrates that the combination of access control and persistence is extremely powerful, supporting the $t$-resilient implementation of any object. In this section, we explore the power of ephemeral objects. We prove an impossibility result for a class of *erasable* objects, and give several fault-tolerant constructions.

*Erasable objects:* An erasable object is an object in which (1) each pair of operations $\mathsf{op}_0$ and $\mathsf{op}_1$, when invoked by different processes commute, i.e., $apply(\mathsf{op}_0.\mathsf{op}_1, s) = apply(\mathsf{op}_1.\mathsf{op}_0, s)$, for every state $s$, or (2) for every pair of states $s_0$ and $s_1$, there exist invocation sequences $invoke_0$ and $invoke_1$ allowed by any two processes such that

$apply(invoke_0, s_0) = apply(invoke_1, s_1)$. Such familiar objects as registers, test&set, swap, read-modify-write, but also seemingly stronger objects such as compare-and-swap, are erasable. (This definition generalizes the notion of *interfering* commutative/overwriting operations [Her91].)

**Theorem 4.1** *For any $t > n/2$, there is no implementation of a $t$-resilient $n$-$\mathsf{propose}()$ weak consensus object using any set of erasable objects.*

*Proof.* Assume to the contrary that such an implementation, called $A$, is possible. We divide the $n$ processes into three disjoint groups: $P_0$ and $P_1$ each of size at least $\lfloor (n-1)/2 \rfloor$, and a singleton which includes process $p$. Consider the following finite runs of algorithm $A$:

1. $\rho_0$ is a run in which only processes in $P_0$ participate with input 0 and halt once they have decided. They must all decide on 0. Let $O_0$ be the (finite) set of objects that were accessed in this run, let $s_i^0$ be the state of object $o_i \in O_0$ at the end of this run, and let $\rho_0|o_i$ be the sequence of operations on $o_i$ invoked in $\rho_0$.

2. $\rho_1$ is a run in which only processes in $P_1$ participate with input 1 and halt once they have decided. They must all decide on 1. Let $O_1$ be the (finite) set of objects that were accessed in this run, let $s_i^1$ be the state of object $o_i \in O_1$ at the end of this run, and let $\rho_1|o_i$ be the sequence of operations on $o_i$ invoked in $\rho_1$.

3. $\rho_0'$ is a run in which processes from $P_0$ are correct and start with input 0, and processes from $P_1$ are faulty. It is constructed as follows. First the processes from $P_0$ run exactly as in $\rho_0$ until they all decide on 0. Then, the processes from $P_1$ set all the shared objects in $(O_1 - O_0)$ to the states that these objects have at (the end of) $\rho_1$, and set the states of the objects in $(O_1 \cap O_0)$ to hide the order of previous accesses.

That is, for objects in $(O_1 - O_0)$ and for objects in $(O_1 \cap O_0)$ in which all operations $\rho_0|o_i$ and $\rho_1|o_i$ commute, $P_1$ runs $\rho_1|o_i$, the same operations as in $\rho_1$. For each remaining object $o_i$, there exist invocation sequences $invoke_0$ and $invoke_1$, by processes in $P_0$ and $P_1$, respectively, such that $apply(invoke_1, s_i^0) = apply(invoke_0, s_i^1)$. (This follows from part (2) of the definition of erasable object.) The invocation sequence $invoke_1$ is applied to $o_i$ by the appropriate process in $P_1$.

4. $\rho_1'$ is a run in which processes from $P_1$ are correct and start with input 1, and processes from $P_0$ are faulty. It is constructed symmetrically to $\rho_0'$: First the process from $P_1$ run exactly as in $\rho_1$ until they all decide on 1. Then, as above, the processes from $P_0$ set all the shared objects in $(O_0 - O_1)$ to the states that these objects have at (the end of) $\rho_0$.

That is, for objects in $(O_0 - O_1)$ and for objects in $(O_1 \cap O_0)$ in which all operations $\rho_0|o_i$ and $\rho_1|o_i$ commute, $P_0$ runs $\rho_0|o_i$, the same operations as in $\rho_0$. For each remaining object $o_i$, $P_0$ invokes the operation sequence $invoke_0$ defined above.

By construction, every object is in the same state after $\rho_0'$ and $\rho_1'$. But if we activate process $p$ alone at the end of $\rho_0'$, it cannot yet decide, because it would decide the same value if we activate process $p$ alone at the end of $\rho_1'$. So $p$ must wait for help from the correct processes (which the $t$-resilience condition allows it to do) to disambiguate these identical states.

Having allowed $p$ to take some (ineffectual) steps, we can repeat the construction again, scheduling $P_0$ and $P_1$ to take additional steps in each run, but bringing the two runs again to identical states. By repeating this indefinitely, we create two infinite runs, in each of which the correct processes, including $p$, take an infinite number of steps, but in which $p$ never decides, a contradiction.

### *4.1 Atomic registers*

While the universality result involves sticky bits, the impossibility result shows that even weak consensus cannot be implemented using common ephemeral objects when a majority of the processes are faulty. This raises the question of what can be done with ephemeral objects. Next we provide some examples of implementations using (ephemeral) atomic registers. The first such object is $t$-resilient $k$-set consensus.

The definition of $k$-set consensus is due to Chaudhuri [Cha93]:
*$k$-set consensus objects:* A $k$-set consensus object $x$ is an object with one operation: $x.\mathsf{propose}(v)$ where $v$ is some number. The $x.\mathsf{propose}()$ operation returns a value such that (1) each value returned is proposed by some process, and (2) the set of values returned is of size at most $k$.

We adapt the definition to the $t$-resilient, Byzantine case by assigning to each faulty process a value which it can be presumed to have proposed. That is, an implementation of $\mathsf{propose}()$ is correct, if for every $t$-resilient execution, the faulty processes can be assigned (possibly distinct) values so that conditions (1) and (2) above hold on outputs of correct processes. (Other adaptations of the $k$-set consensus problem definition to a Byzantine setting were explored in [dPMR01].)

**Theorem 4.2** *For any $t < n/3$, if $t < k$ then there is an implementation of a $t$-resilient $n$-$\mathsf{propose}()$ $k$-set consensus object using atomic registers.*

*Proof.* Each process $p_i$, $1 \le i \le t+1$, announces its input value by writing it into a 1-writer register $announce[i]$, whose value is initially $\perp$. All processes $p_k$, $1 \le i \le n$, repeatedly read the $announce[1..t+1]$ registers, and echo the first non-$\perp$ value seen in each $announce[j]$ entry by copying it into a 1-writer register $echo[k, j]$. Interleaved with this activity, process $p_k$ also reads all the $echo[1..n, 1..t+1]$ registers, and returns the value it first finds echoed $n - t$ times in some column $echo[1..n, \ell]$. In subsequent operations, it returns the same value, but first examines the $announce[1..t+1]$ array, and as above copies into $echo[k, j]$ any entry $announce[j]$ that was $\perp$ last time $p_k$ read it.

The $t$-resilience property and the ACL for the $announce$ array assures that at least one entry, say $announce[i]$, will be written exactly once, and will eventually be echoed in at least $n - t$ $echo[k, i]$ entries. This guarantees that every operation by a correct process will eventually terminate. Since for each process $p_j$, $1 \le j \le t+1$, no correct process $p_k$ ever writes to $echo[k, j]$ more than once, and by the assumption that $t < n/3$, no column of $echo$ can have two values for which $n - t$ values are ever read. Hence, the operations by correct processes return one of at most $t+1$ different values.

The implementation above of $k$-set-consensus constructs a $t$-resilient object. The next result shows that registers can be used to implement the stronger $t$-threshold condition, though for a different object.

*$k$-pairwise set-consensus objects:* A $k$-pairwise set-consensus object $x$ is an object with one operation: $x$.propose($v$) where $v$ is some number. The $x$.propose() operation returns a set of at most $k$ values such that:
(1) in any finite run in which all participating processes are correct (no Byzantine faults), each value in the set returned by a process is proposed by some process, and
(2) the intersection of any two sets returned by correct processes is non-empty.

**Theorem 4.3** *For any $t < n/3$, there is an implementation of a $t$-threshold $n$-propose $(2t+1)$-pairwise set-consensus object using atomic registers.*

*Proof.* The implementation uses 1-write() registers which are initially $\perp$. Each process writes its proposed value to its 1-write() register, and repeatedly reads the $3t+1$ registers of processes $p_1$ through $p_{3t+1}$. It returns the set of the first $2t+1$ values (of distinct registers) that are not $\perp$. Termination follows from the fact that at least $2t+1$ process of the first $3t+1$ are correct. Since each two processes read at least one value written by the same correct process, the intersection of any two sets is not empty.

### *4.2 Fault-tolerant constructions using objects other than registers*

Even in the presence of only one crash failure, it is not possible to implement election objects [MW87,TM96] or consensus objects [LA87,FLP85] using only atomic registers. Next we show that many other familiar objects, such as 2-process weak consensus, test&set, swap, compare&swap, and read-modify-write, can be used to implement election objects for any number of processes and under any number of Byzantine faults.

In order to define election objects, we first introduce the notion of a clean run. A finite run $r$ is *clean* if (1) at least one process has participated in $r$ (i.e., $r$ is not the null run), (2) all the the processes that participate in $r$ are correct, and (3) all the processes that participated in $r$ have terminated (and hence, there are no pending operations in $r$).

*Election objects:* An election object $x$ is an object with one operation: $x$.elect. The $x$.elect operation returns a value, either 0 or 1, such that for any (finite) run $r$: (1) At most one correct process returns 1 in $r$, and (2) if $r$ is clean, then exactly one of the participating processes returns 1 (that process is called the *leader*).

Notice that it is not required for all the processes to "know" the identity of the leader. The following observation – which intuitively claims that a process that invokes elect strictly after another set of correct processes has terminated, can never be elected – follows immediately from the definition of an election object.

**Observation 4.1** Let $r$ be a clean run of an election protocol, and let $p$ be a correct process that has not participated in $r$. Then, in any extension of $r$ process $p$ never returns 1.

Next, we prove the following result.

**Theorem 4.4** *There is an implementation of*
*(1) $n$-threshold $n$-*elect *election from 2-threshold, two-process versions of weak consensus, test&set, swap, compare&swap, or read-modify-write, and*
*(2)* 2*-threshold* 2-propose() *weak consensus from*
2*-threshold* 2-elect *election.*

*Proof.*

1. We prove only the implementation from 2-propose() weak consensus objects–the other constructions are essentially the same. We implement $n$-elect election as follows: With every two processes $p_i$ and $p_j$, $i < j$, we associate a 2-propose weak consensus object, $cons[i, j]$, accessed only by $p_i$ and $p_j$. We say that $p_i$ and $p_j$ *contend* with each other by performing propose(0) or propose(1), respectively, on $cons[i, j]$. If the consensus value returned by any operation on $cons[i, j]$ is 0, we say that $p_i$ *won* $cons[i, j]$, otherwise $p_j$ won and $p_i$ *lost* $cons[i, j]$.

To implement the $n$-elect operation, process $p_i$ contends with each other process, from smallest index to biggest, until it first loses, at which point it returns 0. If it wins against all other processes then it returns 1.

2. The implementation of 2-threshold 2-propose() weak consensus for two processes $p_0$ and $p_1$ uses three 2-elect election objects, $Val[0..1]$ and $Leader$. Both processes run elect on $Leader$, and return their input value if they are elected. They use the $Val[0..1]$ objects to signal their input values to the other process in the case they are elected in $Leader$. That is, if the input of $p_i$ is 0, it first runs elect on $Val[i]$ before accessing $Leader$. If the input of $p_i$ is 1, $p_i$ never accesses $Val[i]$. If $p_i$ is not elected in $Leader$, it returns the value returned by elect on $Val[1-i]$. The code for process $p_i$, $i \in \{0, 1\}$, is below. (Assuming the elect invocation returns 1 to the leader, 0 to the loser.)

type: $Leader$, $Val[0]$, $Val[1]$: 2-elect election objects
propose($input$)
(1) **if** $input = 0$ **then** $Val[i]$.elect **fi**
(2) **if** ($Leader$.elect) **then** $return(input)$
(3) **else** $return(Val[1-i]$.elect$)$ **fi**

To prove that the implementation satisfies the requirements of weak consensus, let us assume that process $p_i$ is elected in $Leader$. The requirement that the consensus value is the input of one of the processes is trivially satisfied since $p_i$ returns its own input value. Next, we need to show that $p_{1-i}$ will also return $p_i$'s input value. To see this we consider two cases: (1) The input value of $p_i$ is 1. In this case $p_i$ never accesses $Val[i]$, and hence, by Observation 4.1, the operation $Val[i]$.elect by $p_{1-i}$ must return 1 and both processes will return 1. (2) The input value of $p_i$ is 0. In this case $p_i$ will accesses $Val[i]$ strictly before $p_{1-i}$ will access $Val[i]$ (and $p_i$ will get elected), and hence, by Observation 4.1, the operation $Val[i]$.elect by $p_{1-i}$ must return 0 and both processes will return 0.

## 5 Discussion

This paper is an initial exploration of fault-tolerant shared memory algorithms in the presence of Byzantine processes.

Since shared objects are vulnerable to misuse by faulty processes, known constructions for more benign failure environments generally will not tolerate Byzantine failures. Hence, much of the work that has been done in the crash-model for shared memory must be revisited. We briefly review the contributions of this paper and suggest directions for further work.

Since processes can overwrite gibberish, classical objects such as read/write or read-modify-write registers that are writable by all processes are useless in a Byzantine environment. We show how constructions can use access control lists, persistent objects, bounds on the numbers of faulty processes, and redundancy to overcome this drawback.

*Termination conditions.* To exploit redundancy, some form of waiting between processes is necessary. Hence, we explore weaker conditions than wait-free termination (in which an operation is required to terminate despite any number of crash failures by other processes). We define the abstractions of $t$-resilience and $t$-threshold, as natural termination conditions in Byzantine environments.

The first of these termination conditions, $t$-resilience, is appropriate for constructions in which each process requires the active participation of other processes in order to complete its operation – hence, it assures termination only when other (correct) processes continue to access the implemented object with the same operation. This requirement to access the implemented object infinitely often is a strong obligation to place on correct processes. In particular, it is a barrier to composing fault-tolerant constructions; either in utilizing multiple objects within the same algorithm (different correct processes must not wait for help in different objects), or in abstracting a complex implementation to use as a primitive object at another layer of abstraction (termination of operations on the abstracted object depends on repeated invocations of components of the implementation). But note that the universal construction of Theorem 3.1 provides one way to robustly combine multiple $t$-resilient objects within a single algorithm, by considering them as a single object with parameterized invocations and responses. (Thus, a collection of shared registers can be thought of as a single shared memory object.)

The $t$-threshold property is a stronger condition, requiring each operation by a correct process to terminate once at least $n - t$ correct processes have invoked that operation. This condition makes the most sense for "one-shot" objects, such as consensus or election. As illustrated in the proof of Theorem 3.1, this stronger condition is helpful in abstracting components in complex constructions.

Fault-tolerant termination conditions other than $t$-resilience and $t$-threshold are possible. For instance, one could require a $t$-resilient construction to terminate in failure-free runs: that is, in $t$-resilient implementations of one-shot objects, processes might signal the satisfactory termination of their initial operation (by setting a bit, for example). Once a process executing additional operations to "help" the others sees that all the bits are set, it can stop accessing the object, knowing all initial operations have terminated. (This assumes that the implementations of later operations are wait-free.)

*Specific question about resilience and composition.* The main positive result in this paper shows that there is a $t$-resilient universal construction out of wait-free sticky bits, in a Byzantine shared memory environment, when the number of failures $t$ is limited. This leaves open the specific questions of whether it is possible to weaken the wait-freedom assumption (assuming sticky bits which are $t$-threshold or $t$-resilient) and/or to implement a $t$-threshold object (instead of a $t$-resilient one).

*Specific open questions.* We have also presented several impossibility and positive results for implementing fault-tolerant objects. There are further natural questions concerning the power of objects in this environment, such as: Is the resilience bound in our universality construction tight for sticky bits? What is the resilience bound for universality using other types of objects? What type of objects can be implemented by others? The few observations regarding these questions in Sections 3.4 and 4 only begin to explore these questions.

## References


[Att00] P.C. Attie Wait-free Byzantine Agreement. Technical Report NU-CCS-00-02, College of Computer Science, Northeastern University, May 2000

[AGMT95] Y. Afek, D. Greenberg, M. Merritt, G. Taubenfeld. Computing with faulty shared memory. *Journal of the ACM* 42(6): 1231–1274, 1995

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM* 35(8): 27–47, 1992

[CL99] M. Castro, B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation – OSDI'99*, February, 1999, New Orleans, LA

[Cha93] S. Chaudhuri. More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation* 105(1): 132–158, 1993

[CG89] N. Carriero, D. Gelernter. Linda in context. *Communications of the ACM* 32(4): 444–458, 1989

[dPMR01] R. de Prisco, D. Malkhi, M. Reiter. *On $k$-set Consensus Problems in Asynchronous Systems*. IEEE Transactions on Parallel and Distributed Systems 12(1), January 2001

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

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 11(1): 124–149, 1991

[JCT98] P. Jayanti, T. Chandra, S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM* 45(3): 451–500, 1998

[JT92] P. Jayanti, S. Toueg. Some results on the impossibility, universality, and decidability of consensus. *Proc. of the 6th Int. Workshop on Distributed Algorithms: LNCS, 647*, pp. 69–84. Berlin Heidelberg New York: Springer 1992

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

[KMM98] K. P. Kihlstrom, L. E. Moser, P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Hawaii Int. Conf. on System Sciences*, pp. 317–326, 1998

[LA87] M. C. Loui, H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research, JAI Press* 4: 163–183, 1987

[LH89] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Programming Languages and Systems* 7(4): 321–359, 1989

[MR00] D. Malkhi, M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2): 187–202, 2000

[MW87] S. Moran, Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Info. Processing Letters* 26: 141–151, 1987

[PG89] F. M. Pittelli, H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems* 7(1): 25–60, 1989

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pp. 159–175, 1989

[Rei96] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM* 39(4): 71–74, 1996

[SE+92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, A. Tully. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Trans. on Computers* 41(5): 542–549, 1992

[TKB92] A. S. Tannenbaum, M. F. Kaashoek, H. E. Balvrije. Parallel programming using shared objects. *IEEE Computer*, pp. 10–19, 1992

[TM96] G. Taubenfeld, S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica* 33(1): 1–20, 1996

**Dahlia Malkhi** received her Ph.D. and M.Sc. degrees in computer science and a B.Sc. degree in mathematics and computer science in 1994, 1988, 1985, respectively, from the Hebrew University of Jerusalem, Israel. She was a member of the Secure Systems Research Department at AT&T Labs–Research in Florham Park, New Jersey from 1995 to 1999. She is currently a faculty member of the School of Computer Science and Engineering at the Hebrew University of Jerusalem, Jerusalem, Israel, where she heads the Secure Systems Research Laboratory. Her research interests include all areas of distributed systems and security.

**Michael Merritt** is head of the Network Optimization and Analysis Research Department of AT&T Labs–Research in Florham Park, New Jersey. He received the B.Sc. degree in philosophy and computer science from Yale University in 1979, and the M.Sc. and Ph.D. degrees in computer science from the Georgia Institute of Technology in 1980 and 1983, respectively. He joined AT&T Bell Labs in 1983 and became a founding member of AT&T Labs–Research in 1996. His research interests include distributed computing, security, and network management.

**Michael K. Reiter** is a Professor of Electrical and Computer Engineering and Computer Science at Carnegie Mellon University in Pittsburgh, Pennsylvania, USA. He received the B.Sc. degree in mathematical sciences from the University of North Carolina in 1989, and the M.Sc. and Ph.D. degrees in computer science from Cornell University in 1991 and 1993, respectively. He joined AT&T Bell Labs in 1993 and became a founding member of AT&T Labs–Research when NCR and Lucent Technologies (including Bell Labs) were split away from AT&T in 1996. He returned to Bell Labs in 1998 as Director of Secure Systems Research, and then joined Carnegie Mellon University in 2001. His research interests include all areas of computer and communications security and distributed computing.

**Gadi Taubenfeld** received the B.A., M.Sc. and Ph.D. degrees in computer science from the Technion (Israel Institute of Technology), in 1982, 1984 and 1988, respectively. From 1988 to 1990 he was a research scientist at Yale University. From 1991 to 1995 he was a member of technical staff at AT&T Bell Laboratories. Since 1995 he has been with Israel Open University. He is currently (2002) on sabbatical at the Interdisciplinary Center, Israel. His primary research interests are in concurrent and distributed computing.