Objects Shared by Byzantine Processes*

(Extended Abstract)

Dahlia Malkhi** Michael Merritt*** Michael Reiter† Gadi Taubenfeld‡

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).

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 any form of coordination among

* Paper to appear in Proceedings of the 14th International Symposium on DIStributed Computing (DISC 2000).

** School of Comuter Science and Engineering, The Hebrew University of Jerusalem, Israel. dalia@cs.huji.ac.il

*** AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971. mischu@research.att.com

† Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. reiter@research.bell-labs.com

‡ The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs. gadi@cs.openu.ac.il

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. Hence, our work is the first study of the power of objects shared by Byzantine processes.

1.2 Summary of results

We generalize the 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 bounds on the numbers of faulty processes, redundancy, *access control lists* that constrain faulty processes from accessing specific objects, and *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. The results are:

1. **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 is bounded by $(\sqrt{n}-1)/2$, where n is the total number of processes.
   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 object. Our emulation borrows closely from Herlihy's universal construction for crash faults [Her91], but differs in significant ways due to the need to cope with Byzantine failures.
2. **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 $t \geq n/3$. We observe that there exists a simple bounded-space universal object assuming $t < n/3$, and a trivial unbounded-space universal object assuming any number of $t \leq n$ 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 any of the familiar non-sticky objects.

3. **Constructions using ephemeral objects:** While the universality result involves sticky bits, the impossibility result shows that consensus cannot be implemented using known objects that are not persistent. 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.

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: the operation is guaranteed to return within a finite number of steps. Objects that can be used (together with atomic registers) to give a wait-free implementation of any other objects are called *universal objects*. Previous work on wait-free (and non-blocking) shared objects provided methods (called *universal constructions*) to transform sequential implementations of arbitrary shared objects into wait-free concurrent implementations, assuming the existence of a universal object [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 showed that shared objects can be classified according to their consensus number: that is, the maximum number of processes that can reach consensus using the object [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 the register itself is faulty. The subject of memory faults (as opposed to process faults) has been investigated recently 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. e.g., $x.\mathsf{op}$ denotes operation 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-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 replication to fault-tolerantly emulate shared memory abstractions. Therefore, ACLs can be implemented, e.g., by storing a copy of the ACL with each replica 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 safely 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, and 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 numbers of failures bounded away from n, it becomes possible (and indeed necessary) for processes to coordinate with each other, using redundancy to overcome the Byzantine failures of their peers. This means 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 one-shot objects, which function correctly once enough correct processes access them. 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 ρ in which $n-t$ 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 ρ 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.

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 $v \neq \bot$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\bot$. 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 $\bot$. 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) The $x.\mathsf{propose}()$ operation returns the same value, called the consensus value, to every process that invokes it. (2) 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 2 and 3 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 4 in Section 3.2 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 A universal construction

This section contains the main result of this paper, the construction of a universal 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.

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

$$apply \subset \text{INVOKE} \times \text{STATE} \times \text{STATE},$$

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

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. 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.[6] 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.

For example, a t-threshold strong consensus object can be specified as follows: STATE is the set of integer pairs, (x, y), $0 \leq x, y \leq t$, or the singletons 0 and 1, with $(0,0)$ as the single start state. For all integers x, y and u, v in $\{0,1\}$ (constrained as shown), the *apply* relation is, $\{(\textsc{propose}(0),(x < t, y),(x+1, y)\}) \cup \{(\textsc{propose}(1),(x, y < t),(x, y+1))\} \cup \{(\textsc{propose}(0),(t, y), 0)\} \cup \{(\textsc{propose}(1),(x, t), 1\}) \cup \{(\textsc{propose}(u), v \in \{0,1\}, v)\}$, and the *reply* relation

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

is $\{(\text{PROPOSE}(u), v \in \{0,1\}, \text{RETURN}(v))\}$. Hence, each invocation of a propose operation enables *apply* to increment the appropriate counter in the state. Concurrent invocations introduce race conditions (as to which application of *apply* occurs first. Once $t+1$ applications of the same value occur, the state is committed to that binary value, and the responses of pending invocations are enabled.

For the purposes of the universal construction below, we resolve any non-determinism, and assume that the first relation is a function from INVOKE $\times$ STATE to STATE, and that the second 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 1. *Any t-resilient object can be implemented using:*

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 1 describes a universal implementation. In the lemmas, we provide two constructions of (strong) binary consensus objects using sticky bits, which differ in the access restrictions.

Lemma 2. *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. Let $m = \lfloor \frac{n}{t+1} \rfloor$. Partition the n processes into blocks $B_1, \ldots, B_m$, each of size at least $t+1$, and let $x_1, \ldots, x_m$ 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 *skip* if p is not in the ACL for x_i) and, once that completes, repeatedly executes x_j.read for all $1 \leq j \leq m$ until none return $\perp$. 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.[7] 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 since $m \geq 2t+1$, 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, $1 \leq j \leq m$, until none return $\perp$, termination is guaranteed provided that each sticky bit is set. Since each x_j has $t+1$ processes proposing to it, it follows that o.propose() is guaranteed to return when at least $n-t$ perform propose() operations. □

[7] In case m is even and the number of 1's equals the number of 0's, the majority value is defined to be 1.

Lemma 3. *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: 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 $\{p_i\}$. Let $m = \lfloor \frac{n}{2t+1} \rfloor$. Partition the n processes into blocks $B_1, \ldots, B_m$, each of size at least $2t+1$, and let $x_1, \ldots, x_m$ be sticky bits with the property that the ACLs for x_i.write() and x_i.read are both B_i (or a $(2t+1)$-subset thereof). For a correct process $p_j \in B_i$ to emulate o.propose(v), it executes x_i.write(v) (or *skip* if p is not in the ACL for x_i) and, once that completes, it executes $r_j \leftarrow x_i$.read. 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 since $m \geq 2t+1$, v cannot occur in the majority of the V_j's. □

3.1 Proof of Theorem 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, construction below differs in significant ways.

The construction labors to ensure that operations by correct processes eventually complete, and that each operation by Byzantine processes either has no impact, or appears as (the same) valid operation to the correct processes. There are two principal 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 $\lceil \log(|\mathrm{INVOKE}|) \rceil$ 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.
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. 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], ..., 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*.

```
type:   ID: array of ⌈log(n)⌉ strong consensus objects
        CELL: array of ⌈log(|INVOKE|)⌉ sticky bits
global variables:
        Announce[1..n][1...], array of CELL
            for all i, 1 ≤ i ≤ n, and j, elements of Announce[i][j] are writable by p_i
        Sequence[1...], infinite array of IDs, each accessible by all processes
variables private to process p_i:
        MyNextAnnounce, index of next vacant cell in Announce[i], initially 1
        NextAnnounce[1..n], for each 1 ≤ j ≤ n, index in Announce[j][]
                                  of next operation of p_j to be read by p_i, initially 1
        CurrentState ∈ 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, ⌈log(n)⌉ bit string
o.op:
(1)     write, bit by bit, the invocation,
                                  o.op.invoke of o.op into Announce[i][MyNextAnnounce]
(2)     MyNextAnnounce++
                        ; Apply operations until o.op is applied and p_i can return.
                        ; Each while loop iteration applies exactly one operation.
(3)     while ((NextAnnounce[i] < MyNextAnnounce) or
                ((NextAnnounce[i] ≥ MyNextAnnounce)
                and (reply(o.op.invoke, CurrentState) is not defined))) do
(4)         ℓ ← NextSeq (mod n)                 ; Select preferred process to help.
(5)         NameSuffix ← emptystring
(6)         for k = 0 to ⌈log(n)⌉ do ; Loop applies the operation one bit per iteration.
                                          Search for a valid process index to propose
(7)             while ((Announce[ℓ + 1][NextAnnounce[ℓ + 1]] is invalid)
                        or (NameSuffix is not a suffix of the bit encoding of ℓ + 1)) do
(8)                 ℓ ← ℓ + 1 (mod n) od
                                          ; Propose the k'th bit (right to left) of ℓ + 1
(9)             prepend(NameSuffix, Sequence[NextSeq][k].propose((ℓ + 1)&(2^k))
(10)        od      ; A new cell has been threaded by NameSuffix in Sequence[NextSeq]
(11)        CurrentState ←
                apply(Announce[NameSuffix][NextAnnounce[NameSuffix]], CurrentState)
(12)        NextAnnounce[NameSuffix] + +
(13)        NextSeq++
(14)    od
(15)    return(reply(o.op.invoke, CurrentState))
```

Figure 1: Universal implementation of o.op at p_i.

The universal construction of object o is described in Figure 1 as the code process p_i executes to implement an operation o.op, with invocation o.op.$invoke$. 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 $\mathsf{Sequence}[k]$, where $\ell = k (\mathrm{mod})\ n$. (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 $\mathsf{Announce}[i]$ that is vacant in a variable denoted $\mathsf{MyNextAnnounce}$, and first (line 1) writes the invocation, bit by bit, into $\mathsf{Announce}[i][\mathsf{MyNextAnnounce}]$, and (line 2) increments $\mathsf{MyNextAnnounce}$. To keep track of which cells it has seen threaded (including its own), p_i keeps n counters in an array $\mathsf{NextAnnounce}[1..n]$, where each $\mathsf{NextAnnounce}[j]$ is one plus the number of times i has read cells of j in $\mathsf{Sequence}$, and hence the index of $\mathsf{Announce}[j]$ where i looks to find the next operation announced by j. Hence, having incremented $\mathsf{MyNextAnnounce}$, $\mathsf{NextAnnounce}[i] = \mathsf{MyNextAnnounce} - 1$ 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 $\mathsf{NextAnnounce}[i] = \mathsf{MyNextAnnounce}$), the next conjunct (again line 3) keeps p_i threading cells until a response to the 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 $o.\mathsf{op}$, but by the t-resilient condition, as long as operations of correct processes are eventually threaded, eventually $o.\mathsf{op}$ can return. For example, if $o.\mathsf{op}$ is the $\mathsf{propose}()$ operation of a strong consensus object, then it can return once at least $t+1$ $\mathsf{propose}()$ invocations with identical values occur. Process p_i keeps an index $\mathsf{NextSeq}$ which points to the next entry in $\mathsf{Sequence}[1, ...]$ whose cells it has not yet accessed.

To thread cells, process p_i proposes (line 9) the binary encoding of a process id, $\ell + 1$, bit by bit, to $\mathsf{Sequence}[\mathsf{NextSeq}]$. In choosing $p_{\ell+1}$, process p_i first checks (first disjunct, line 7) that $\mathsf{Announce}[\ell+1][\mathsf{NextAnnounce}[\ell+1]]$ contains a valid encoding of an operation invocation. (And, as discussed above, p_i gives preference (line 4) to a different process for each cell in $\mathsf{Sequence}$.)

Starting (line 5) with the *emptystring*, p_i accumulates (line 9) the bit-by-bit encoding of the id being recorded in $\mathsf{Sequence}[\mathsf{NextSeq}]$ into a local variable, $\mathsf{NameSuffix}$. 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 $\mathsf{NameSuffix}$. (The properties of strong consensus assure that such a process exists.)

Once process p_i accumulates all the bits of the threaded cell into $\mathsf{NameSuffix}$ (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 $\mathsf{NameSuffix}$'s successfully threaded cells and (line 13) the next unread cell in $\mathsf{Sequence}$. Having successfully threaded a cell, p_i returns to the top of the **while** loop (line 3).

The sequencing and correct semantics of each operation follow trivially from the sequential ordering of invocations in $\mathsf{Sequence}$ 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 and from the t-resilience of the 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 $\mathsf{Announce}[\ell + 1][\mathsf{NextAnnounce}[\ell + 1]]$, 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 interleaving reads of the bits of each entry in $\mathsf{Announce}[\ell+1][\mathsf{NextAnnounce}[1..n]]$, starting as before with the next bit of $\mathsf{NextAnnounce}[\ell + 1]$, 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.) □

3.2 Resilience and impossibility

The proof of Theorem 1 presents a universal construction of t-resilient objects, where $t \leq (\sqrt{n}-1)/2$. Naturally, one would like to know whether there are more fault-tolerant universal constructions, and in the limit, whether wait-free universal constructions exist. Focusing on improving the the bound $t \leq (\sqrt{n}-1)/2$ in Theorem 1, that is, finding a universal construction or impossibility proofs $t > (\sqrt{n}-1)/2$, we note that the construction in Figure 1 builds modularly on t-resilient strong consensus. The $t \leq (\sqrt{n}-1)/2$ bound of Theorem 1 follows from the constructions of strong consensus from sticky bits, in Lemmas 2 and 3. 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 $t < n/3$.

Theorem 4. *For $t \geq n/3$, there is no t-resilient n-*`propose`*() (strong) consensus object.*

Proof. Assume to the contrary that there exists such an 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 commit to a consensus value. Without loss of generality, let this value be 0. Next, we let all the remaining processes propose 1 and run until all commit to 0. We can now assume that all the processes in P_0 are faulty and reach a contradiction. □

We point out that it is easy to define objects that are universal for any number of faults. An example is the *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

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 each pair of operations $\mathtt{op}_1$ and $\mathtt{op}_2$, when invoked by different processes, either (1) commute (such as a read and any other operation) or (2) for every pair of states s_1 and s_2, have invocations $invoke_1$ and $invoke_2$ such that $apply(invoke_1, s_1) = apply(invoke_2, s_2)$. Such familiar objects as registers, test&set, swap, read-modify-write are erasable. (This definition generalizes the notion of commutative and overwriting operations [Her91].)

Theorem 5. *For any $t > n/2$, there is no implementation of a t-resilient n-*$\mathtt{propose}()$ *weak consensus object using any set of erasable objects.*

Proof. Assume to the contrary the 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. ρ_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. and let s_0^i be the state of object o_i at the end of this run.
2. ρ_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, and let s_1^i be the state of object o_i at the end of this run.
3. ρ_0' is a run in which processes from P_0 are correct and start with input 0, and processes from P_1 are faulty and start with input 1. It is constructed as follows. First the process from P_0 run exactly as in ρ_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 values that these objects have at (the end of) ρ_1, and set the values of the objects in $(O_1 \cap O_0)$ to hide the order of previous accesses That is, for objects in which all operations accessible by P_0 and P_1 commute, P_1 runs the same operations as in run ρ_0. For each remaining object o_i, P_0 invokes an operation $invoke_0$ such that P_1 has access to an operation $invoke_1$ where $apply(invoke_1, s_0^i) = apply(invoke_1, s_2^i)$.
4. ρ_1' is a run in which processes from P_1 are correct and start with input 1, and processes from P_0 are faulty and start with input 0. It is constructed symmetrically to ρ_2: First the process from P_1 run exactly as in ρ_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 values that these objects have at (the end of) ρ_0. For objects in which all operations accessible by P_0 and P_1 commute, P_0 runs the same operations as in run ρ_0. For each remaining object o_i, P_0 invokes the operation $invoke_1$ defined in ρ_0'.

By construction, every object is in the same state after ρ_0' and ρ_1'. But if we activate process p alone at the end of ρ_0', it cannot yet decide, because it would decide the same value if we activate process p alone at the end of ρ_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

Next we provide some examples of implementations using (ephemeral) atomic registers. The first such object is t-resilient k-set consensus [Cha93].

k-set consensus objects: A k-set consensus object x is an object with one operation: x.`propose`(v) where v is some number. The x.`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.

Theorem 6. *For any $t < n/3$, if $t < k$ then there is an implementation of a t-resilient n-*`propose`$()$ *k-set consensus object using atomic registers.*

Proof. Processes p_1 through p_{t+1} announce their input value by writing it into a register *announce*$[i]$, whose value is initially $\perp$. Each process repeatedly reads the *announce*$[1..t+1]$ registers, and echoes the first non-$\perp$ value it sees in any *announce*$[j]$ entry by copying it into a 1-writer register *echo*$[i,j]$. Interleaved with this process, p_i also reads all the *echo*$[1..n, 1..t+1]$ registers, and returns the value it first finds echoed the super-majority of $2n/3+1$ times in some column *echo*$[1..n,k]$. In subsequent operations, it returns the same value, but first examines *announce*$[1..t+1]$ array and writes any new values to *echo*$[i, 1..t+1]$.

Using this construction, no process can have two values for which a super-majority of echos are ever read. Moreover, any correct process among p_1 through p_{t+1} will eventually have its value echoed by a super-majority. Hence, every operation by a correct process will eventually 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. (The proof is omitted from this extended abstract.)

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) each value in the set returned is proposed by some process, and (2) the intersection of any two sets returned is non-empty.

Theorem 7. *For any $t < n/3$, there is an implementation of a t-threshold n-*`propose`$()$ $(2t+1)$*-pairwise set-consensus object using atomic registers.*

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 [TM96, MW87] 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.

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 at most one correct process returns 1, and if only correct processes participate then exactly one process gets 1 (that process is called the *leader*). Notice that it is not required for all the processes to "know" the identity of the leader. We have the following result. (Proof omitted from this extended abstract.)

Theorem 8. *There is an implementation of (1) n-threshold n-*`elect`() *election from 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-elect`() *election.*

5 Discussion

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).

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 Section 3.2 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, and G. Taubenfeld. Computing with faulty shared memory. *Journal of the ACM*, 42(6):1231-1274, November 1995.

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

[CL99] M. Castro and 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, July 1993.

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

[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.

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

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

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

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

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

[LA87] 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.

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

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

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

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

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

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

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

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

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

This article was processed using the LaTeX macro package with LLNCS style