Relationships Between Broadcast and Shared Memory in Reliable Anonymous Distributed Systems

James Aspnes[1], Faith Fich[2], and Eric Ruppert[3]

[1] Yale University
[2] University of Toronto
[3] York University

Abstract. We study the power of reliable anonymous distributed systems, where processes do not fail, do not have identifiers, and run identical programmes. We are interested specifically in the relative powers of systems with different communication mechanisms: anonymous broadcast, read-write registers, or registers supplemented with additional shared-memory objects. We show that a system with anonymous broadcast can simulate a system of shared-memory objects if and only if the objects satisfy a property we call *idemdicence*; this result holds regardless of whether either system is synchronous or asynchronous. Conversely, the key to simulating anonymous broadcast in anonymous shared memory is the ability to count: broadcast can be simulated by an asynchronous shared-memory system that uses only counters, but registers by themselves are not enough. We further examine the relative power of different types and sizes of bounded counters and conclude with a non-robustness result.

1 Introduction

Consider a minimal reliable distributed system, perhaps a collection of particularly cheap wireless sensor nodes. The processes execute the same code, because it is too costly to program them individually. They lack identities, because identities require customization beyond the capabilities of mass production. And they communicate only by broadcast, because broadcast presupposes no infrastructure. Where fancier systems provide specialized roles, randomization, point-to-point routing, or sophisticated synchronization primitives, this system is just a big bag of deterministic clones shouting at one another. The processes' only saving grace is that their uniformity makes them absolutely reliable—no misplaced sense of individuality will tempt any of them to Byzantine behaviour, no obscure undebugged path through their common code will cause a crash, and no glitch in their nonexistent network will lose any messages. The processes may also have distinct inputs, which saves them from complete solipsism, even though processes with the same input cannot tell themselves apart. What can such a system do?

Although anonymous systems have been studied before (see Section 1.1), much of the work has focused on systems where processes communicate with one another by passing point-to-point messages or by accessing shared read-write registers. In this paper, we start with simple broadcast systems, where processes transmit messages to all of the processes (including themselves), which are delivered serially but with no return addresses. We characterize the power of such systems by showing what classes of shared-memory objects they can simulate. In Section 3, we show that such a system can simulate a shared object if and only if the object can always return the same response whenever it is accessed twice in a row by identical operations, a property we call *idemdicence*; examples of such idemdicent objects include read-write registers, counters (with separate increment and read operations), and any object for which any operation that modifies the state returns only *ack*.

This characterization does not depend on whether either the underlying broadcast system or the simulated shared-memory system is synchronous or asynchronous. The equivalence of synchrony and asynchrony is partially the result of the lack of failures, because in an asynchronous system we can just wait until every process has taken a step before moving on to the next simulated synchonous round, but it also depends on the model's primitives providing enough power that detecting this condition is possible.

Characterizing the power of broadcast systems in terms of what shared-memory objects they can simulate leads us to consider the closely related question of what power is provided by different kinds of shared-memory objects. We show in Section 4 that counters of sufficient size relative to the number of processes, n, are enough to simulate an anonymous broadcast model, even if no other objects are available, which in turn means that they can simulate any reliable anonymous shared-memory system with idemdicent objects. In contrast, read-write registers by themselves cannot simulate broadcast, because they cannot distinguish between different numbers of processes with the same input value, while broadcasts can.

This leads us to consider further the relative power of different sizes of counters in an anonymous system. We show in Section 5 that mod-m counters are inherently limited if the number of processes, n, exceeds m by more than two, although they become more powerful when $n = m + 1$ if read-write registers are also available. Although these results hint at a hierarchy of increasingly powerful anonymous shared-memory objects, any such hierarchy is not *robust* [11, 12]; we show in Section 6 that mod-m counters with different values of m can simulate more objects together than they can alone. Previous non-robustness results typically use rather unusual object types, designed specifically for the non-robustness proofs (see [8] for a survey). The result given here is the first to use natural objects.

1.1 Related Work

Some early impossibility results in message-passing systems assumed that processes were anonymous [1]. This assumption makes symmetry-based arguments possible: all processes behave identically, so they cannot solve problems that require symmetry to be broken. A wide range of these results are surveyed in [8]. Typically, they assume that the underlying communication network is symmetric (often a ring or regular graph); our broadcast model is a complete graph. Some work has been done on characterizing the problems that are solvable in anonymous message-passing systems, depending on the initial knowledge of processes (see, for example, [4, 5, 15]).

With randomization, processes can choose identities at random from a large range, which solves the *naming problem* with high probability. Unfortunately, Buhrman *et al.* [6] show that no protocol allows processes to *detect* whether they have solved this problem in an asynchronous shared-memory model providing only read-write registers. Surprisingly, they show that wait-free consensus can nonetheless be solved in this model. The upper bound for consensus has since been extended to an anonymous model with infinitely many processes by Aspnes, Shah, and Shah [2].

Attiya, Gorbach and Moran [3] give a systematic study of the power of asynchronous, failure-free anonymous shared-memory systems that are equipped with read-write registers only. They characterize the agreement tasks that can be solved in this model if the number of processes is unknown. Drulă has shown the characterization is the same if the number of processes is known [7]. In particular, consensus is solvable, assuming the shared registers can be initialized. If they cannot, consensus is unsolvable [13]. Attiya *et al.* also give complexity lower bounds for solving consensus in their model.

The robustness question has been extensively studied in non-anonymous systems. It was first addressed by Jayanti [11, 12]. See [8] for a discussion of previous work on robustness.

2 Models

We consider *anonymous* models of distributed systems, where each process executes the same algorithm. Processes do not have identifiers, but they may begin with input values (depending on the problem being solved). We assume algorithms are deterministic and that systems are reliable (i.e. failures do not occur). Let $n \geq 2$ denote the number of processes in the system.

We assume throughout that the value of n is known to all processes. This assumption can be relaxed in some models even if new processes can join the system. In a shared-memory model with unbounded counters, it is easy to maintain the number of processes by having a process increment a size counter when it first joins the system. In a broadcast model, a new process can start by broadcasting an arrival message. Processes keep track of the number of arrival messages they have received and respond to each one by broadcasting this number. Processes use the largest number they have received as their current value for size. Algorithms will work correctly when started after this number has stabilized.

In the *asynchronous broadcast* model, each process may execute a *broadcast*(*msg*) command at any time. This command sends a copy of the message *msg* to each process in the system. The message is eventually delivered to all processes (including the process that sent it), but the delivery time may be different for different recipients and can be arbitrarily large. Thus, broadcasted messages are not globally ordered: they may arrive in different orders at different recipients.

A *synchronous broadcast model* is similar, but assumes that every process broadcasts one message per round, and that this message is received by all processes before the next round begins.

We also consider an anonymous shared-memory model, where processes can communicate with one another by accessing shared data structures, called *objects*. A process may invoke an operation on an object, and at some later time it will receive a response from the object. We assume that objects are linearizable [10], so that each operation performed on an object appears to take place instantaneously at some time between the operation's invocation and response (even though the object may in fact be accessed concurrently by several processes). The *type* of an object specifies what operations may be performed on it. Each object type has a set of possible states. We assume that a programmer may initialize a shared object to any state. An operation may change the state of the object and then return a result to the invoking process that may depend on the old state of the object. A *step* of an execution specifies an operation, the process that performs this operation, the object on which it is performed, and the result returned by the operation.

A *read-write register* is an example of an object. It has a *read* operation that returns the state of the object without changing the state. It also supports *write* operations that return *ack* and set the state of the object to a specified value.

Another example of an object is an *(unbounded) counter*. It has state set $\mathbf{N}$ and supports two operations: *read*, which returns the current state without changing it, and *increment*, which adds 1 to the current state and returns *ack*.

There are many ways to define a *bounded* counter, i.e., one that uses a bounded amount of space. For any positive integer m, a *mod-m counter* has state set $\{0, 1, 2, \ldots, m-1\}$ and an increment changes the state from x to $(x+1) \bmod m$. A *threshold-m counter* has state set $\{0, 1, 2, \ldots, m\}$. An increment adds 1 to the current state provided it is less than m and otherwise leaves it unchanged. A read of a mod-m or threshold-m counter returns the current state without changing it. An *m-valued counter* also has state set $\{0, 1, 2, \ldots, m\}$ and increment behaves the same as for a threshold-m counter. However, the behaviour of the read operation becomes unpredictable after m increments: in state m, a read operation may nondeterministically return any of the values $0, 1, 2, \ldots, m-1$, or may even fail to terminate. Note that both mod-m counters and threshold-$(m-1)$ counters are implementations of m-valued counters. Also, for $m' > m$, an m'-valued counter is an implementation of an m-valued counter.

In an *asynchronous* shared-memory system, processes run at arbitrarily varying speeds, and each operation on an object is completed in a finite but unbounded time. The scheduler is required to be fair in that it allocates an opportunity for each process to take a step infinitely often. In a *synchronous* shared-memory system, processes run at the same speed; the computation proceeds in rounds. During each round, each process can perform one access to shared memory. Several processes may access the same object during a round, but the order in which those accesses are linearized is determined by an adversarial scheduler.

An example of a synchronous shared-memory system is the anonymous ARBITRARY PRAM. This is a CRCW PRAM model where all processes run the same code and the adversary chooses which of any set of simultaneous writes succeeds. It is equivalent to a synchronous shared-memory model in which all writes in a given round are linearized before all reads. PRAM models have been studied extensively for non-anonymous systems (e.g. [14]).

3 When Broadcast Can Simulate Shared Memory

In this section, we characterize the types of shared-memory systems that can be simulated by the broadcast model in the setting of failure-free, anonymous systems. We also consider the functions that can be computed in this model.

Definition 1. *An operation on a shared object is called* idemdicent[4] *if, for every starting state, two consecutive invocations of the operation on an object (with the same arguments) can return identical answers to the processes that invoked the operations.*

It follows by induction that any number of repetitions of the same idemdicent operation on an object can all return the same result. *Idempotent* operations are idemdicent operations that leave the object in the same state whether they are applied once or many times consecutively. Reads and writes are idempotent operations. Increment operations for the various counters defined in Section 2 are idemdicent, but are not idempotent. In fact, any operation that always returns *ack* is idemdicent.

An object is called idemdicent if every operation that can be performed on the object is idemdicent. Similarly, an object is called idempotent if every operation that can be performed on the object is idempotent. Examples of idempotent objects include registers, sticky bits, snapshots and resettable consensus objects.

Theorem 2. *A n-process asynchronous broadcast system can simulate an n-process synchronous shared-memory system that uses only idemdicent objects.*

Proof. Each process simulates a different process and maintains a local copy of the state of each simulated shared object. We now describe how a process P simulates the execution of the rth round of the shared-memory computation. Suppose P wants to perform an operation op on object X. It broadcasts the message (r, X, op). (If a process does not wish to perform a shared-memory operation during the round, it can broadcast the message (r, nil, nil) instead.) Then, P waits until it has received n messages of the form $(r, *, *)$, including the message it broadcast.

Process P orders all of the messages in lexicographic order and uses this as the order in which the round's shared-memory operations are linearized. Process

[4] From Latin *idem* same + *dicens -entis* present participle of *dicere* say, by analogy with idempotent.

P simulates this sequence of operations on its local copies of the shared objects to update the states of the objects and to determine the result of its own operation during that round. All identical operations on an object are grouped together in the lexicographic ordering, so they all return the same result, since the objects are idemdicent. This is the property that allows P to determine the result of its own operation if several processes perform the same operation during the round. $\square$

Since a synchronous execution is possible in an asynchronous system, an asynchronous shared-memory system with only idemdicent objects can be simulated by a synchronous system with the same set of objects and, hence, by the asynchronous broadcast model. However, even an asynchronous system with one non-idemdicent object cannot be simulated by a synchronous broadcast system nor, hence, by an asynchronous broadcast system. The difficulty is that a non-idemdicent object can be used to break symmetry.

Theorem 3. *A synchronous broadcast system cannot simulate an asynchronous shared-memory system if any of the shared objects are non-idemdicent.*

Proof. Let X be an object that is not idemdicent. Consider the *k-election problem*, where processes receive no input, and exactly k processes must output 1 while the remaining processes output 0. We shall show that it is possible to solve k-election for some k, where $0 < k < n$, using X, but that there is no such election algorithm using broadcasts. The claim follows from these two facts.

Initialize object X to a state q where the next two invocations of some operation *op* will return different results r and r'. The election algorithm requires each process to perform *op* on X. Those processes that receive the result r output 1, and all others output 0. Let k be the number of operations that return r if *op* is performed n times on X, starting from state q. Clearly, this algorithm solves k-election. Furthermore, $k > 0$ since the first operation will return r, and $k < n$ since the second operation will return $r' \neq r$.

In a synchronous broadcast system, where processes receive no input, all processes will execute the same sequence of steps and be in identical states at the end of each round. Thus, k-election is impossible in such a system if $0 < k < n$. $\square$

A function of n inputs is called *symmetric* if the function value does not change when the n inputs are permuted. We say an anonymous system *computes* a symmetric function of n inputs if each process begins with one input and eventually outputs the value of the function evaluated at those inputs. (It does not make sense to talk about computing non-symmetric functions in an anonymous system, since there is no ordering of the processes.)

Proposition 4. *Every symmetric function can be computed in the asynchronous broadcast model.*

Proof. Any symmetric function can be computed as follows. Each process broadcasts its input value. When a process has received n messages, it orders the n input values arbitrarily and computes the function at those values. $\square$

4 When Counters Can Simulate Broadcast

In this section, we consider conditions under which unbounded and various types of bounded counters can be used to simulate broadcast. In the following section, we consider when different types of bounded counters can be used to simulate one another, which will show that some cannot be used to simulate broadcast.

We begin by proving that an asynchronous shared-memory system with mod-n counters can be used to simulate a synchronous broadcast system. Unbounded counters can simulate mod-n counters (and hence synchronous broadcast). Moreover, since counters are idemdicent, an asynchronous shared-memory system with counters can be simulated by an asynchronous broadcast system. Hence, shared-memory systems with mod-n counters, shared-memory systems with unbounded counters, and atomic broadcast systems are equivalent in power.

Theorem 5. *An n-process asynchronous shared-memory system with mod-n counters or unbounded counters can simulate the n-process synchronous broadcast system.*

Proof. The idea is to have each of the n asynchronous processes simulate a different synchronous process and have a mod-n counter for each possible message that can be sent in a given round. Each process that wants to send a message that round increments the corresponding counter. Another mod-n counter, $WriteCounter$, is used to keep track of how many processes have finished this first phase. After all processes have finished the first phase, they all read the message counters to find out which messages were sent that round. One additional mod-n counter, $ReadCounter$, is used to keep track of how many processes have finished the second phase. Simulation of the next round can begin when all processes have finished the second phase.

Let d denote the number of different possible messages that can be sent. The shared counter $M[i]$ corresponds to the i-th possible message (say, in lexicographic order) that can be sent in the current round. For $i = 1, \ldots, d$, the local variables $x_{0,i}$ and $x_{1,i}$, are used by a process to store the value of $M[i]$ in the most recent even- and odd-numbered round, respectively. The variables $x_{0,1}, \ldots, x_{0,d}$ are initialized to 0 at the beginning of the simulation. $WriteCounter$ and $ReadCounter$ are also initialized to 0.

The simulation of each round is carried out in two phases. In phase 1, a process that wants to broadcast the i-th possible message increments $M[i]$ and then increments $WriteCounter$. A process that does not want to broadcast in this round just increments $WriteCounter$. In either case, each process repeatedly reads $WriteCounter$ until it has value 0, at which point it begins phase 2. Note that $WriteCounter$ will have value 0 whenever a new round begins, because each of the n processes will increment it exactly once each round.

In phase 2, each process reads the values from $M[1], \ldots, M[d]$ and stores them in its local variables $x_{r,1}, \ldots, x_{r,d}$, where r is the parity of the current round. From these values and the values of $x_{1-r,1}, \ldots, x_{1-r,d}$, the process can determine the number of occurrences of each possible message that were supposed to be sent during that round. Specifically, the number of occurrences of the i-th possible message is $x_{r,i} - x_{1-r,i} \bmod n$, except when this is the message that the process sent and $x_{r,i} = x_{1-r,i}$. In this one exceptional case, the number of occurrences of the i-th possible message is n rather than 0. Once the process knows the set of messages that were sent that round, it can simulate the rest of the round by doing any necessary local computation. Finally, the process increments $ReadCounter$ and then repeatedly reads it until it has value 0. At this point, the process can begin simulation of the next phase.

Since an unbounded counter can be used to directly simulate a mod-n counter by taking the result of every read modulo n, the simulation will also work for unbounded counters. However, the values in the counters can get increasing large as the simulation of the execution proceeds.

The number of counters used by this algorithm is $\Theta(d)$. If d is very large (or unbounded), the space complexity of this algorithm is poor. The number of counters can be improved to $\Theta(n)$ by simulating the construction of a trie data structure [9] over the messages transmitted by the processes; up to $4n$ counters are used to transmit the trie level by level, with each group of 4 used to count the number of 0 children and 1 children of each node constructed so far.

In terms of messages, processes broadcast their messages one bit per round and wait until all other messages are finished before proceeding to their next message. However, it does not suffice to count the number of 0's and 1's sent during each of the rounds. For example, it is necessary to distinguish between when messages 00 and 11 are sent and when messages 01 and 10 are sent.

Each process uses the basic algorithm described above to broadcast the first bit of its message. Once processes know the first k bits of all messages that are k or more bits in length, they determine the next bit of each message or whether the message is complete. Four counters ($M[0]$, $M[1]$, *WriteCounter*, and *Read-Counter*) are allocated for each distinct k-bit prefix that has been seen. Since all processes have seen the same k-bit prefixes, they can agree on this allocation without any communication with one another. If a process has a message $s_1 s_2 \dots s_\ell$, where $\ell > k$, it participates in an execution of the basic algorithm described above to broadcast s_{k+1}, using the four counters assigned to the prefix $s_1 s_2 \dots s_k$. Each process also participates in executions of the basic algorithms for the other k-bit prefixes that have been seen, but does not send messages in them. This procedure to broadcast one more bit of the input of a message is continued until no process has these k bits as a proper prefix of its message. Because counters are reused for different bits of the messages, the number of counters needed is at most $4n$. □

A similar algorithm can be used to simulate the n-process synchronous broadcast system using threshold-n or $(n+1)$-valued counters, provided the counters support a decrement operation or a reset operation. Decrement changes the state of such a counter from $x \in \{1, \dots, n\}$ to $x - 1$. Decrement leaves a threshold-n counter in state 0 unchanged. In an $(n+1)$-valued counter, when decrement is performed in state 0 or $n+1$, the new state is $n+1$. Reset changes the state of a counter to 0. The only exception is that an $(n+1)$-valued counter in state $n+1$ does not change state. Both decrement and reset always return *ack*.

Theorem 6. *An n-process asynchronous shared-memory system with threshold-n or $(n+1)$-valued counters that also support a decrement or reset operation can simulate the n-process synchronous broadcast system.*

Proof. In this simulation, there is an additional counter, *ResetCounter*, and each round has a third phase. *ReadCounter* and *ResetCounter* are initialized to n. All other counters are initialized to 0.

In phase 1 of a round, a process that wants to broadcast the i-th possible message increments $M[i]$, then decrements or resets *ReadCounter*, and, finally increments $WriteCounter$. A process that does not want to broadcast in this round does not increment $M[i]$ for any i. In either case, each process then repeatedly reads $WriteCounter$ until it has value n. Note that when $WriteCounter$ first has value n, *ReadCounter* will have value 0.

In phase 2, each process first decrements or resets *ResetCounter*. Next, it reads the values from $M[1], \ldots, M[d]$ to obtain the number of occurrences of each possible message that were supposed to be sent during that round. Then it can simulate any necessary local computation. Finally, the process increments *ReadCounter* and then repeatedly reads it until it has value n. When *ReadCounter* first has value n, *ResetCounter* has value 0.

In phase 3, each process first decrements or resets $WriteCounter$. If it incremented $M[i]$ during phase 1, then it now decrements or resets it. Finally, the process increments *ResetCounter* and then repeatedly reads it until it has value n. When *ResetCounter* first has value n, $WriteCounter$ has value 0.

The space complexity of this algorithm can be changed from $\Theta(d)$ to $\Theta(n)$ as described in the proof of Theorem 5. ☐

5 When Counters Can Simulate Other Counters

We now consider the relationship between different types of bounded counters. A consequence of these results is that the asynchronous broadcast model is *strictly* stronger than some shared-memory models.

Definition 7. *Let m be a positive integer. An object is called* m-idempotent *if it is idemdicent and, for any initial state and for any operation op, the object state that results from applying op $m+1$ times is identical to the state that results from applying op once.*

If an object is m-idempotent, then, by induction, it is km-idempotent for any positive integer k. Any idempotent object (e.g. a read-write register) is 1-idempotent. A mod-m counter is m-idempotent. An m-idempotent object has the property that the actions of $m+1$ clones (i.e. processes behaving identically) are indistinguishable from the actions of one process.

Lemma 8 (Cloning Lemma). *Consider an algorithm for $n > m$ processes that uses only m-idempotent objects. Let γ be an execution of the algorithm in which processes $P_1, \ldots, P_m$ take no steps. Let $P \notin \{P_1, \ldots, P_m\}$. Let γ' be the execution that is constructed from γ by inserting, after each step by P, a sequence* *of steps in which each of processes $P_1, \ldots, P_m$ applies the same operation as P to the same object and gets the same result. If processes $P_1, \ldots, P_m$ have the same input as P, then γ' is a legal execution and no process outside $\{P_1, \ldots, P_m\}$ can distinguish γ from γ'.*

Proof. (Sketch) The state of the object that results from performing one operation is the same as the one that results from performing that operation $m+1$ times. Also, the response to each of those $m+1$ repetitions of the operation will be the same, since the object is idemdicent. Notice that P_i has the same input as P, and P_i is performing the same sequence of steps in γ' as P and receiving the same sequence of responses. Since the system is anonymous, this is a correct execution of P_i's code. □

The n-ary threshold-2 function is a binary function of n inputs whose value is 1 if and only if at least two of its inputs are 1.

Proposition 9. *Let m be a positive integer and let $m \leq n-2$. The n-ary threshold-2 function cannot be computed in an n-process synchronous shared-memory system if all shared objects are m-idempotent.*

Proof. Suppose there is an algorithm that computes the n-ary threshold-2 function in the shared-memory system. Let $P_1, \ldots, P_n$ be the processes of the system. Suppose the input to process P_n is 1, and the inputs to all other processes are 0. Let γ be an execution of the algorithm where processes $P_1, \ldots, P_m$ take no steps and all other processes run normally. (This is not a legal execution in a failure-free synchronous model, but we can still imagine running the algorithm in this way.)

Consider the execution α obtained by inserting into γ steps of $P_1, \ldots, P_m$ right after each step of P_{m+1}, as described in Lemma 8. This results in a legal execution of the algorithm where all processes must output 0.

Consider another execution, β, where the inputs of $P_1, \ldots, P_m$ are 1 instead, obtained from γ by inserting steps by processes $P_1, \ldots, P_m$ after each step of P_n, as described in Lemma 8. This results in a legal execution as well, but all processes must output 1.

Processes outside the set $\{P_1, \ldots, P_m\}$ cannot distinguish between α and γ, or between β and γ, so those processes must output the same result in both α and β, a contradiction. □

Since the n-ary threshold-2 function is symmetric, it is computable in the asynchronous broadcast model, by Proposition 4. However, by Proposition 9, for $n-2 \geq m \geq 1$, it cannot be computed in a synchronous (or asynchronous) shared-memory system, all of whose shared objects are m-idempotent. This implies that the n-process asynchronous broadcast model is strictly stronger than these shared-memory systems. In particular, it is stronger than the anonymous ARBITRARY PRAM and shared-memory systems with only read-write registers and mod-m counters for $m \leq n-2$.

Corollary 10. *For $1 \leq m \leq n-2$ and $t \geq 2$, a threshold-t counter cannot be simulated using registers and mod-m counters in an n-process synchronous shared-memory system.*

Proof. The n-ary threshold-2 function can be computed using a threshold-t object in an n-process synchronous system. The threshold-t object is initialized to 0. In the first round, each process with input 1 increments the threshold-t object. In the second round, all processes read the threshold-t object. They output 0 if their result is 0 or 1 and output 1 otherwise.

Since registers and mod-m counters are m-idempotent, it follows from Proposition 9 that the n-ary threshold-2 function cannot be computed in an n-process shared-memory system. □

A similar result is true for simulating m-valued counters.

Proposition 11. *If $1 \leq n \leq m-1$, than an m-valued counter cannot be simulated in an shared-memory system of n or more processes using only mod-$(n-1)$ counters and read-write registers.*

Proof. Suppose there was such a simulation. Let E be an execution of this simulation where the m'-valued counter is initialized to the value 0, and process P performs an increment on the m-valued counter and then performs a read operation. The simulated read operation must return the value 1.

Since $(n-1)$-counters and read-write registers are $(n-1)$-idempotent, Lemma 8, the Cloning Lemma, implies that there is a legal execution E' using n-processes which cannot be distinguished from E by P. Thus, P's read must return the value 1 in E'. However, there are n increments that have completed before P's read begins, so P's read should output n in E'. Notice that no non-deterministic behaviour can occur in the m-valued counter since $n \leq m-1$. □

Since an n-process broadcast system can simulate an m-valued counter, for any m, it follows that an n-process shared-memory system with mod-$(n-1)$ counters and read-write registers cannot simulate an n-process broadcast system.

The requirement in Proposition 11 that $n \leq m-1$ is necessary: In an n-process shared-memory system, it is possible to simulate an n-valued counter using only mod-$(n-1)$ counters and read-write registers.

Proposition 12. *It is possible to simulate an $(m+1)$-valued counter in an n-process shared-memory system with one mod-m counter and one read-write register.*

Proof. Consider the following implementation of an $(m+1)$-valued counter from a mod-m counter C and a register R. Assume C and R are both initialized to 0. The variables x and y are local variables.

```
INCREMENT
   increment C
   write 1 to R
end INCREMENT
```

```
READ
   y ← R
   x ← C
   if y = 0 then return 0
   elsif x = 0 then return m
   else return x
end READ
```

Linearize all INCREMENT operations whose accesses to C occur before the first write to R in the execution at the first write to R (in an arbitrary order). Linearize all remaining INCREMENT operations when they access C.

Linearize any READ that reads 0 in R at the moment it reads R. Since no INCREMENTS are linearized before this, the READ is correct to return 0. Linearize each other READ when it reads counter C. If at most m INCREMENTS are linearized before the READ, the result returned is clearly correct. If more than m INCREMENTS are linearized before the READ, the READ is allowed to return any result whatsoever. $\square$

The following result shows that the read-write register is essential for the simulation in Proposition 12.

Proposition 13. *It is impossible to simulate an n-valued counter in an n-process shared-memory system using only mod-$(n-1)$ counters.*

Proof. Suppose there was such a simulation. Consider an execution where $n-1$ clones each perform an INCREMENT operation, using a round-robin schedule. After each complete round of $n-1$ steps, all shared mod-$(n-1)$ counters will be in the same state as they were initially. So if the one remaining process P performs a READ after all the INCREMENTS are complete, it will not be able to distinguish this execution from the one where P runs by itself from the initial state. In the constructed execution, P must return $n-1$, but in the solo execution, it must return 0. This is a contradiction. $\square$

6 Counter Examples Demonstrating Non-robustness

This section proves that the reliable anonymous shared-memory model is not robust. Specifically, we show how to implement a 6-valued counter from mod-2 counters and mod-3 counters. Then we apply Proposition 11, which says that a 6-valued counter cannot be implemented from either only mod-2 counters and read-write registers or only mod-3 counters and read-write registers.

Let $m = \mathrm{lcm}(m_1, \ldots, m_r)$. We give a construction of an m-valued counter from the set of object types $\{$mod-m_1 counter$, \ldots,$ mod-m_r counter$\}$. We shall make use of the following theorem, which is proved in introductory number theory textbooks (see, for example, Theorem 5.4.2 in [16]).

Theorem 14 (Generalized Chinese Remainder Theorem). *The system of equations $x \equiv b_j \pmod{m_j}$ for $1 \leq j \leq r$ has a solution for x if and only if $b_j \equiv b_k \pmod{\gcd(m_j, m_k)}$ for all $j \neq k$. If a solution exists, it is unique modulo $\mathrm{lcm}(m_1, m_2, \ldots, m_r)$.*

Proposition 15. *Let $m_1, \ldots, m_r$ be positive integers. Let $m = \mathrm{lcm}(m_1, \ldots, m_r)$. For any number of processes, there is an implementation of an m-valued counter from $\{$mod-m_1 counter$, \ldots,$ mod-m_r counter$\}$.*

Proof. Let $q = 2m/m_1 + 1$. The implementation uses a shared array $A[1..q, 1..r]$ of base objects. The base object $A[i, j]$ is a mod-m_j counter, initialized to 0. The array $B[1..q, 1..r]$ is a private variable used to store the results of reading A. (We assume the m-valued counter is initialized to 0. To implement a counter initialized to the value v, one could simply initialize $A[i, j]$ to v mod m_j, and the proof of correctness would be identical.)

The implementation is given in pseudocode below. A process INCREMENTS the m-valued counter by incrementing each counter in A. A process READS the m-valued counter by repeatedly reading the entire array A (in the opposite order) until the array appears consistent (i.e. the array looks as it would if no INCREMENTS were in progress). We shall linearize each operation when it accesses an element in the middle row of A, and show that each READ operation reliably computes (and outputs) the number of times that element has been incremented. Note that the second part of the loop's exit condition guarantees that the result to be returned by the last line of the READ operation exists and is unique, by Theorem 14.

INCREMENT
  for $i \leftarrow q$ downto 1
    for $j \leftarrow r$ downto 1
      increment $A[i, j]$
    end for
  end for
end INCREMENT

READ
  loop
    for $i \leftarrow 1..q$
      for $j \leftarrow 1..r$
        $B[i, j] \leftarrow A[i, j]$
      end for
    end for
    exit when $B[i, j] \equiv B[1, j] \pmod{m_j}\ \forall i, j$ and
      $B[1, j] \equiv B[1, k] \pmod{\gcd(m_j, m_k)}\ \forall j \neq k$
  end loop
  return the value $x \in \{0, \ldots, m-1\}$ that satisfies
    $x \equiv B[1, j] \pmod{m_j}$ for all j
end READ

Consider any execution of this implementation where there are at most $m-1$ INCREMENTS. After sufficiently many steps, all INCREMENTS on the m-valued counter will be complete (due to the fairness of the scheduler). Let v_{final} be the number of increment operations on the m-valued counter that have occurred at that time. The collection of reads performed by any iteration of the main loop of the READ algorithm that begins afterwards will get the response v_{final} mod m_j from each mod-m_j counter, and this will be a consistent collection of reads. Thus, every operation must eventually terminate. If more than $m - 1$ INCREMENTS occur in the execution, READS need not terminate.

Let $s = m/m_1 + 1$. Ordinarily, we linearize each operation when it last accesses $A[s, 1]$. However, if there is a time T when $A[q, r]$ is incremented for the mth time, then all INCREMENTS in progress at T that have not been linearized before T are linearized at T (with the INCREMENTS preceding the READS). Each operation that starts after T can be linearized at any moment during its execution. Note that m INCREMENTS are linearized at or before T, so any READ that is linearized at or after T is allowed to return an arbitrary response.

Consider any READ operation R that is linearized before T. Let x be the value R returns. We shall show that this return value is consistent with the linearization. Let $a_{i,j} \in \{0, \ldots, m-1\}$ be the number of times $A[i,j]$ was incremented before R read it for the last time. Then $a_{i,j} \equiv x(mod m_j)$ for all i and j. Because INCREMENTS and READS access the base objects in the opposite order, $a_{i,j} \leq a_{i,j+1}$ and $a_{i,r} \leq a_{i+1,1}$. From the exit condition of the main loop in the READ algorithm, we also know that $a_{i,j} \equiv a_{1,j} \pmod{m_j}$. We shall show that $x = a_{s,1}$, thereby proving that the result of R is consistent with the linearization.

We first prove by cases that, for $i \geq 1$, $a_{i+1,1} \geq \min(x, a_{i,1} + m_1)$.

Case I ($a_{i+1,1} = a_{i,1}$): Since $a_{i,1} \leq a_{i,2} \leq \cdots \leq a_{i,r} \leq a_{i+1,1} = a_{i,1}$, we have $a_{i,1} = a_{i,2} = \cdots = a_{i,r} = a_{i+1,1}$. Thus, for all j, $a_{i+1,1} = a_{i,j} \equiv a_{1,j} \pmod{m_j}$. By the uniqueness claim of Theorem 14, $a_{i+1,1} = x \geq \min(x, a_{i,1} + m_1)$.

Case II ($a_{i+1,1} > a_{i,1}$): Since $a_{i+1,1} \equiv a_{i,1} \pmod{m_1}$, it must be the case that $a_{i+1,1} \geq a_{i,1} + m_1 \geq \min(x, a_{i,1} + m_1)$.

It follows by induction that $a_{i,1} \geq \min(x, a_{1,1} + (i-1)m_1)$. Thus, $a_{s,1} \geq x$, since s was chosen so that $(s-1)m_1 = m > x$.

We now give a symmetric proof to establish that $a_{s,1} \leq x$. We can prove by cases that, for $i < q$, $a_{i,1} \leq \max(x, a_{i+1,1} - m_1)$.

Case I ($a_{i,1} = a_{i+1,1}$): Since $a_{i,1} \leq a_{i,2} \leq \cdots \leq a_{i,r} \leq a_{i+1,1} = a_{i,1}$, we have $a_{i,1} = a_{i,2} = \cdots = a_{i,r}$. Thus, for all j, $a_{i,1} = a_{i,j} \equiv a_{1,j} \pmod{m_j}$. By the uniqueness claim of Theorem 14, $a_{i,1} = x \leq \max(x, a_{i+1,1} - m_1)$.

Case II ($a_{i,1} < a_{i+1,1}$): Since $a_{i,1} \equiv a_{i+1,1} \pmod{m_1}$, it must be the case that $a_{i,1} \leq a_{i+1,1} - m_1 \leq \max(x, a_{i+1,1} - m_1)$.

It follows by induction that $a_{i,1} \leq \max(x, a_{q,1} - (q-i)m_1)$. Thus we have $a_{s,1} \leq x$, since s was chosen so that $(q-s)m_1 = m$ and $a_{q,1} - (q-s)m_1 = a_{q,1} - m < 0 \leq x$. So we have shown that $x = a_{s,1}$, and this completes the proof of correctness for the implementation of the m-valued counter. □

Theorem 16. *The reliable, anonymous model of shared memory is* non-robust. *That is, there exist three object types A, B, and C such that an object of type A cannot be implemented from only read-write registers and objects of type B and an object of type A cannot be implemented from only read-write registers and objects of type C, but an object of type A can be implemented from objects of types B and C.*

Proof. Let A be a 6-valued counter, B be a mod-3 counter and C be a mod-2 counter. In a 4-process shared-memory system, an object of type A cannot be implemented from read-write registers and objects of type B, by Proposition 11. Similarly an object of type A cannot be implemented from read-write registers and objects of type C. However, by Proposition 15, an object of type A can be implemented using objects of type B and C. □

Acknowledgements

James Aspnes was supported in part by NSF grants CCR-0098078 and CNS-0305258. Faith Fich was supported by Sun Microsystems. Faith Fich and Eric Ruppert were supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.
2. James Aspnes, Gauri Shah, and Jatin Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 524–533, 2002.
3. Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2), pages 162–183, March 2002.
4. Paolo Boldi and Sebastiano Vigna. Computing anonymously with arbitrary knowledge. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 173–179, 1999.
5. Paolo Boldi and Sebastiano Vigna. An effective characterization of computability in anonymous networks. In *Distributed Computing, 15th International Conference*, volume 2180 of *LNCS*, pages 33–47, 2001.
6. Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitanyi. On the importance of having an identity or, is consensus really universal? In *Distributed Computing, 14th International Conference*, volume 1914 of *LNCS*, pages 134–148, 2000.
7. Catalin Drulă. The totally anonymous shared memory model in which the number of processes is known. Personal communication.
8. Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3), pages 121–163, September 2003.
9. Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9), August 1960.
10. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), pages 463–492, July 1990.
11. Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4), pages 592–614, July 1997.
12. Prasad Jayanti. Solvability of consensus: Composition breaks down for nondeterministic types. *SIAM Journal on Computing*, 28(3), pages 782–797, September 1998.
13. Prasad Jayanti and Sam Toueg. Wakeup under read/write atomicity. In *Distributed Algorithms, 4th International Workshop*, volume 486 of *LNCS*, pages 277–288, 1990.
14. John H. Reif, ed. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.
15. Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 173–179, 1999.
16. Harold N. Shapiro. *Introduction to the Theory of Numbers*. John Wiley and Sons, 1983.