Distributed Computing manuscript No.
(will be inserted by the editor)

James Aspnes · Faith Ellen Fich · Eric Ruppert

Relationships Between Broadcast and Shared Memory in Reliable Anonymous Distributed Systems

the date of receipt and acceptance should be inserted later

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 read-write registers plus 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 read-write 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.

Keywords Anonymous · Broadcast · Shared memory · Robustness · Simulations

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 1.2), researchers have 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), consensus objects 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 asyn-

James Aspnes
Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT, U.S.A., 06520-8285.

Faith Ellen Fich
Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada, M5S 3G4.

Eric Ruppert
Department of Computer Science and Engineering, York University, 4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3.

chronous system we can just wait until every process has taken a step before moving on to the next simulated synchronous round, but it also depends on the model's primitives providing enough power to detect when this has occurred.

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 5 that an n-process system with only mod-n counters is sufficient to simulate an n-process anonymous broadcast model, 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 impossibility result is generalized to a class of object types in Section 4.

In Section 6, we consider the power of mod-m counters when m is less than the number of processes, n. We show that, if $m \leq n-1$, systems containing only mod-m counters are inherently limited and, if $m \leq n-2$, this is also true even when 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* [16,17]: we show in Section 7 that mod-m counters with values of m which do not divide one another 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 [12] for a survey). The result given here is the first to use natural objects.

1.1 Pictorial summary of the main results

Some of the main results of the paper are summarized in Figure 1. The figure is divided into four quadrants for synchronous and asynchronous broadcast and shared-memory models. The parabola divides idemdicent shared-memory models from non-idemdicent ones. Solid arrows from one model to another indicate that the first can simulate the second. Dashed arrows indicate that such a simulation is impossible. The labels on the arrows refer the theorem numbers in this paper that prove the result. The vertical arrows are trivial simulations. In the diagram, n refers to the number of processes in the system. The types X and Y are arbitrary idemdicent and non-idemdicent objects, respectively. Many additional arrows can be drawn as consequences of the ones shown. For example, a shared-memory system equipped only with type X cannot simulate object type Y (regardless of synchrony) since the synchronous broadcast system can simulate type X but cannot simulate Y. The triangle in the bottom-right corner of the diagram describes the non-robustness result of Section 7.

1.2 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. Many of these results are surveyed in [12]. Typically, they assume that the underlying communication network is symmetric, often a ring or regular 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. For examples, see [6,7,23].

Randomization can be used to break the symmetry of an anonymous system. If processes can choose names at random from a large range, then, with high probability, each process will get a distinct name, turning an anonymous system into a non-anonymous system. Shared read-write registers can be used to detect whether all of the chosen names are unique if no failures occur [20,25]. This can also be done when failures occur using single-writer registers, which provide some ability to distinguish between different processes' actions [21]. However, this cannot be done using only multi-writer registers in an asynchronous system with halting failures [8,11,19]. Surprisingly, Buhrman *et al.* [8] show that randomized wait-free consensus can nonetheless be solved in this model, giving an algorithm based on Chandra's [9]. The upper bound for consensus has since been extended to an anonymous model with infinitely many processes by Aspnes, Shah, and Shah [3].

Attiya, Gorbach and Moran [4] give a systematic study of the power of asynchronous, failure-free anonymous shared-memory systems that are equipped with read-write registers which are initialized to known values. They characterize the agreement tasks that can be solved in this model if the number of processes is fixed but unknown. Drulă has shown the characterization is the same if the number of processes is known [10]. In their model, the only functions $f : D^n \to R$ that are computable with $n \geq 2$ processes are constant functions. Attiya *et al.* also show that consensus requires $\Omega(\log n)$ rounds and $\Omega(\log n)$ read-write registers and is solvable using $O(n)$ rounds and $O(n)$ read-write registers. However, consensus becomes unsolvable if the shared read-write registers are not initialized to known values [18].

Guerraoui and Ruppert [14] have considered the question of what can be implemented deterministically in an asynchronous anonymous system using read-write registers if failures can occur. They gave a wait-free implementation of snapshot objects and a bounded-space algorithm for obstruction-free consensus. They also characterized the types of objects that have obstruction-free implementations in this model.

The robustness question has been extensively studied in non-anonymous systems. It was first addressed by

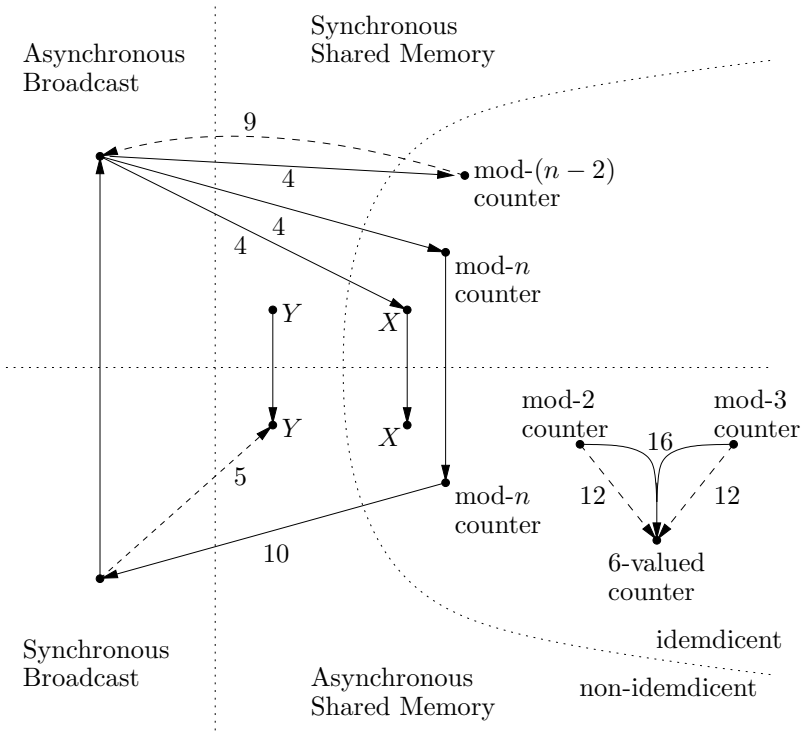

Fig. 1 Summary of the main results. Solid arrows represent simulations. Dashed lines indicate simulations are impossible. Arrow labels are theorem numbers. See Section 1.1.

Jayanti [16,17]. See [12] 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. All algorithms presented in this paper 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.

The *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 [15], 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 (including any operands), the process that performs this

operation, the object on which it is performed, and the result returned by the operation. An object type is *deterministic* if there is only one possible outcome of an operation: there is a unique response and state transition when any operation is applied to an object in any state. If this is not the case, the object is called *nondeterministic*.

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 value 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 can trivially implement 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.

When an asynchronous system is used to simulate a synchronous one, it is impossible to ensure that simulated events at different processes happen simultaneously. Instead, a correct simulation will guarantee that there exists some execution σ of the synchronous system such that the sequence of local events occurring at each process in the simulation is identical to the sequence of events that occur at that process in σ. This notion of simulation is defined formally by Attiya and Welch [5], who call it a *local simulation*. They remark that "local simulations preserve correctness for *internal* problems, those whose specifications do not depend on the real time at which events occur" (p. 248).

Since all executions of a synchronous system can be viewed as executions of an asynchronous system, it is stronger to design simulations to work in asynchronous systems, because they will also work in synchronous systems. Similarly, it is better to prove that simulations by synchronous systems are impossible because they imply the same impossibility results for asynchronous systems.

The anonymous ARBITRARY PRAM is a concurrent-read concurrent-write (CRCW) PRAM model where all processes run the same code and the adversary chooses which one of any set of simultaneous writes to the same read-write register succeeds. It is equivalent to a synchronous shared-memory model in which all writes in a given round are linearized before all reads. Consequently, processes with the same inputs behave as clones of one another: in each round they either all read the same value from the same read-write register or all write the same value to the same read-write register. When different values are written to the same read-write register during the same round, only the value written by the write that the scheduler linearized last can be read from that read-write register. In effect, one arbitrary value chosen by the adversary appears in the read-write register from among those values the processes are attempting to write there. PRAM models have been studied extensively for non-anonymous systems [22].

3 When Broadcast Can Simulate Shared Memory

In this section, we characterize the types of shared-memory systems that can be simulated by broadcast models in the setting of failure-free, anonymous systems.

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

It follows by induction that any number of repetitions of the same idemdicent operation on an object can all return the same result.

Definition 2 An operation defined on a shared object type is called *idempotent* if, for every starting state, for every operation, and every choice of operands for that operation, it is possible that two consecutive invocations of the operation with these operands return the same response and the second invocation does not alter the state of the object.

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

In other words, idempotent operations are idemdicent operations that can leave the object in the same state whether they are applied once or many times consecutively. Both of these definitions are applicable to nondeterministic objects. In such cases, it must be the case that *some* execution of two consecutive operations (starting from any state) have the desired property.

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 read-write registers, sticky bits, snapshots and resettable consensus objects. Counters are idemdicent objects that are not idempotent.

Definition 3 Let m be a positive integer. An object is called *m-idempotent* if it is idemdicent and, for every starting state, for every operation, and every choice of operands for that operation, it is possible that $m+1$ consecutive invocations of this operation with these operands return the same response and the state of the object after the first and last of these invocations is the same.

If an object is m-idempotent, then, by induction, it is km-idempotent for any positive integer k. Any idempotent object 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.

Theorem 4 *An 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 want to perform a shared-memory operation during the round, it broadcasts the message (r, nil, nil).) Then, P waits until it has received n messages of the form $(r, *, *)$, including the message it broadcast. (We include round numbers in the messages because broadcasts may be delivered in different orders at different recipients, and we do not want messages sent in round $r+1$ to be misinterpreted as round r messages.)

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 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. □

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

Proof Consider an asynchronous shared-memory system that has a non-idemdicent object X. Then there is a state q from which two consecutive invocations of some operation with the same operands always returns different values.

Suppose X is initialized in state q and each process accesses X once with this operation and these operands and outputs the values it receives as response. Then, in every execution, at least two outputs are different.

However, 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. Hence the simple asynchronous shared-memory algorithm described in the previous paragraph cannot be simulated in a synchronous broadcast system. □

Theorems 4 and 5 together show that the broadcast model can simulate a shared-memory model if and only if the objects in the shared-memory model are idemdicent, regardless of whether either model is synchronous or asynchronous.

4 When Shared Memory Cannot Simulate Broadcast

Next, we turn attention to the simulation of broadcast models by shared-memory models. We begin, in this section, by showing that there is some function which can be computed in the asynchronous broadcast model (and, hence, the synchronous broadcast model), but cannot be computed in certain shared-memory models.

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 6 *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. □

In contrast, we show that there is a symmetric function that cannot be computed in synchronous or asynchronous n-process shared-memory systems, if all of its shared objects are m-idempotent, for some $m \leq n-2$.

The following technical lemma is the key to this and later impossibility results. It describes how anonymous processes can behave as clones if the objects they are accessing are m-idempotent.

Lemma 7 *Consider an algorithm for $n \geq m+2$ processes that uses only m-idempotent objects. Let $P_1, \ldots, P_m, P, Q$ be distinct processes. Let α be a synchronous execution of the algorithm in which processes $P_1, \ldots, P_m$ all have the same input as P and, in each round, they are scheduled immediately following P, all get the same result as P, and the object they access has the same state after P has taken its step and after $P_1, \ldots, P_m$ have all taken their steps. Let β be the execution which is the same as α except that processes $P_1, \ldots, P_m$ all have the same input as Q and, in each round, they are scheduled immediately following Q, all get the same result as Q, and the object they access has the same state after Q has taken its step and after $P_1, \ldots, P_m$ have all taken their steps. Then no process outside $\{P_1, \ldots, P_m\}$ can distinguish α from β.*

Proof The proof is by induction on the number of rounds in α. To be formal, we prove the following four claims:
(1) each shared object is in the same state at the end of round r of α and β,
(2) each process except $P_1, P_2, \ldots, P_m$ is in the same state at the end of round r of α and β,
(3) processes $P_1, P_2, \ldots, P_m$ and P are all in the same state at the end of round r of α, and
(4) processes $P_1, P_2, \ldots, P_m$ and Q are all in the same state at the end of round r of β.

The base case is at the end of round 0, which is at the beginning of the execution. The claims follow because each process except $P_1, P_2, \ldots, P_m$ has the same input in α and β, $P_1, P_2, \ldots, P_m$ all have the same input as P in α, and $P_1, P_2, \ldots, P_m$ all have the same input as Q in β.

Assume the claims hold at the end of round r. Since $P_1, P_2, \ldots, P_m$ and P are all in the same state at the end of round r of α, they will perform the same operation to the same object in round $r+1$ of α. They all receive the same response and, hence, they will be in the same state at the end of round $r+1$ of α. Moreover, the m-idempotent object will be in the same state after P performs its round $r+1$ operation and after $P, P_1, \ldots, P_m$ perform their round $r+1$ operations. Similarly, $P_1, P_2, \ldots, P_m$ and Q will all be in the same state at the end of round $r+1$ of β and the m-idempotent object they access will be in the same state after Q performs its round $r+1$ operation and after $Q, P_1, \ldots, P_m$ perform their round $r+1$ operations.

Each process R except $P_1, P_2, \ldots, P_m$ has the same state at the end of round r in α and β, so it will perform the same operation to the same object in round $r+1$ of α and β. In both executions, this object has the same state just before R performs this operation, so it will have the same state immediately afterwards and R will have the same state at the end of round $r+1$. It follows that each shared object is in the same state at the end of round $r+1$ of α and β. Thus the claim is true at the end of round $r+1$ and, hence, by induction, at the end of the executions. □

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 8 *For $1 \leq m \leq n-2$, 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. Consider an execution α of the algorithm in which processes $P_1, \ldots, P_{n-1}$ all have input 0, P_n has input 1, and, in each round, processes $P_1, \ldots, P_m$ are scheduled immediately following P_{n-1}, they all get the same result as P_{n-1}, and the object they access has the same state after P_{n-1} has taken its step and after $P_1, \ldots, P_m$ have all taken their steps. This is possible since the object they access is m-idempotent.

Let β be the execution which is the same as α except that processes $P_1, \ldots, P_m$ all have input 1 and, in each round, they are scheduled immediately following P_n, they all get the same result as P_n, and the object they access has the same state after P_n has taken its step and after $P_1, \ldots, P_m$ have all taken their steps.

By Lemma 7, processes outside the set $\{P_1, \ldots, P_m\}$ cannot distinguish between α and β, so they must output the same result in both α and β. This contradicts the assumption that the algorithm correctly computes the n-ary threshold-2 function. □

Theorem 4 implies that the asynchronous broadcast model can simulate the anonymous ARBITRARY PRAM model. Because read-write registers are 1-idempotent, the following result says that the anonymous ARBITRARY PRAM model is strictly weaker than the asynchronous broadcast model for more than two processes.

Corollary 9 *A synchronous (or asynchronous) shared-memory system all of whose shared objects are m-idempotent cannot simulate the asynchronous (or synchronous) broadcast model for n processes if $n - 2 \geq m \geq 1$.*

Proof Since the n-ary threshold-2 function is symmetric, it is computable in the asynchronous (and, hence, synchronous) broadcast model, by Proposition 6. However, by Proposition 8, 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. □

5 When Counters Can Simulate Broadcast

In this section, we consider conditions under which unbounded counters and various types of bounded counters can be used to simulate broadcast. Specifically, we prove 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 broadcast systems are equivalent in power. This equivalence holds regardless of whether either model is synchronous or asynchronous.

Theorem 10 *An n-process asynchronous shared-memory system with mod-n counters or unbounded counters can simulate the n-process synchronous broadcast system.*

Proof An unbounded counter can be used to directly simulate a mod-n counter by taking the result of every read modulo n, so it suffices to construct a simulation from mod-n counters.

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.

First, we consider the case where the set of possible different messages that can be sent is finite and known to all processes. Let d be this number. We use an array of d shared counters $M[1..d]$. The 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.

The number of counters used by this algorithm is $\Theta(d)$. If d is very large, the space complexity of this algorithm is poor. The number of counters can be improved to $\Theta(n)$ by simulating the construction of a binary trie data structure [13] 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 *ReadCounter*) 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 $s_1 s_2 \ldots s_\ell$, 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 $s_1 s_2 \ldots s_k$. 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 each message continues until an iteration occurs where none of the M counters are incremented, indicating that the end of every message has been reached. 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 $x \in \{1, \ldots, n\}$ 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 $n + 1$. (Recall that this state indicates that the $(n + 1)$-valued counter has been accessed improperly, and the counter can behave arbitrarily once it is reached.) 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 11 *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 The simulation is similar to the one given in the proof of Theorem 10, but 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. Here we present the algorithm for the case where there are a finite number, d, of possible messages that can be sent. The space complexity of the following construction can be improved from $\Theta(d)$ to $\Theta(n)$, and we can deal with the possibility of unbounded messages in exactly the same way as in the proof of Theorem 10.

In phase 1 of a round, a process that wants to broadcast the i-th possible message increments $M[i]$. 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 decrements or resets *ReadCounter*, increments *WriteCounter*, and repeatedly reads *WriteCounter* until it has value n. 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* and each of the $M[i]$ counters has value 0. □

6 When Counters Can and Cannot Simulate Other Counters

We now consider when mod-m counters can simulate asynchronous systems containing different types of bounded counters for $m < n$.

Proposition 12 *If $m + 2 \leq v$, then a v-valued counter cannot be simulated in a shared-memory system of $n \geq m + 2$ processes using only mod-m counters and read-write registers.*

Proof Suppose there is such a simulation. We consider two executions of the simulation where the simulated v-valued counter has initial value 0. Let α be the synchronous execution where process P_n performs an increment of the v-valued counter followed by a read. All other processes perform no operations on the v-valued counter, and processes $P_1, \ldots, P_m$ are scheduled immediately following P_{n-1} in each round. Let β be the synchronous execution where processes $P_1, \ldots, P_m$ and P_n each increment the counter and then read it. All other processes perform no operations on the v-valued counter, and processes $P_1, \ldots, P_m$ are scheduled immediately following P_n in each round.

The simulated read operation by P_n must return the value 1 in α. Since $v \geq m + 2$, no non-deterministic behaviour can occur in the v-valued counter in β and P_n must return the value $m + 1$.

Read-write registers and mod-m counters are m-idempotent, so by Lemma 7, process P_n cannot distinguish between α and β. Thus it must return the same

```
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
```

Fig. 2 Implementation of an $(m+1)$-valued counter from a mod-m counter and a register.

result for its read in both α and β. This is a contradiction. □

The requirement in Proposition 12 that $m + 2 \leq v$ is necessary: A mod-m counter is an implementation of a v-valued counter for $v \leq m$. Furthermore, in an n-process shared-memory system, it *is* possible to simulate an $(m + 1)$-valued counter using only mod-m counters and read-write registers.

Proposition 13 *It is possible to simulate an asynchronous system containing an $(m + 1)$-valued counter using only one mod-m counter and one read-write register.*

Proof Consider the implementation of an $(m+1)$-valued counter from a mod-m counter C and a read-write register R in an asynchronous system given in Figure 2. Assume C and R are both initialized to 0. The variables x and y are local variables.

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 the 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. □

The following result shows that the read-write register is essential for the simulation in Proposition 13.

Proposition 14 *It is impossible to simulate an n-process asynchronous system containing an $(m + 1)$-valued counter using only mod-m counters, if $n > m$.*

Proof Suppose there is such a simulation. We consider two synchronous executions of the simulation where the simulated counter is initially 0, and processes $P_1, \ldots, P_n$ take steps in this order in each round. Execution α begins with processes $P_1, \ldots, P_m$ each executing an increment in lock step, while the other processes are not performing any operation on the simulated counter. Suppose it takes k rounds of the synchronous execution for the increments to be completed. Process P_n then performs a read operation while all other processes perform no operations on the simulated counter.

In execution β, the only process to perform an operation on the simulated counter is P_n, which executes a read, starting in round $k + 1$.

Note that, in each execution, the processes $P_1, \ldots, P_m$ will execute the same sequence of steps, since they are executing the same algorithm, run in lockstep, and access only idemdicent objects.

We prove by induction that, at the end of each round, $P_{m+1}, \ldots, P_n$ will each be in the same state in α and β and each shared mod-m counter will have the same state in α and β. Suppose the claim is true at the end of some round r. In both α and β, after the first m steps of round $r + 1$, each shared mod-m counter will be in the same state as it was at the end of round r, since processes $P_1, \ldots, P_m$ perform the same operation in round $r + 1$, and any mod-m counter they all access will end in the same state that it started in.

For any process P_i with $i > m$, P_i performs the same operation in round $r + 1$ of α and β, and it will get the same response in both executions. So any shared mod-$(n - 1)$ counters accessed by processes $P_{m+1}, \ldots, P_n$ in round $r + 1$ will have the same value at the end of the round.

P_n's read operation must return m in α and 0 in β. But α and β are indistinguishable to P_n. This is a contradiction. □

7 Counter Examples Demonstrating Non-Robustness

This section proves that the reliable, asynchronous, 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 12, which says that a 6-valued counter cannot be implemented using only mod-2 counters and read-write registers or using only mod-3 counters and read-write registers.

Let $m = \text{lcm}(m_1, \ldots, m_r)$. We give a construction of an m-valued counter from the set of object types $\{\text{mod-}m_1 \text{ counter}, \ldots, \text{mod-}m_r \text{ counter}\}$. We shall make use of the following theorem, which is proved in introductory number theory textbooks. (See, for example, Theorem 5.4.2 in [24].)

Theorem 15 (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 $\text{lcm}(m_1, m_2, \ldots, m_r)$.*

Proposition 16 *Let $m_1, \ldots, m_r$ be positive integers. Let $m = \text{lcm}(m_1, \ldots, m_r)$. In the*

```
INCREMENT
  for i ← q downto 1
    for j ← r downto 1
      increment A[i, j]
    end for
  end for
end INCREMENT

READ
  loop
    for i ← 1..q
      for j ← 1..r
        B[i, j] ← A[i, j]
      end for
    end for
    exit when B[i, j] ≡ B[1, j] (mod m_j) ∀i, j and
      B[1, j] ≡ B[1, k] (mod gcd(m_j, m_k)) ∀j ≠ k
  end loop
  return the value x ∈ {0, ..., m − 1} that satisfies
    x ≡ B[1, j] (mod m_j) for all j
end READ
```

Fig. 3 Implementation of m-valued counter

asynchronous model, there is an implementation of an m-valued counter from the set $\{$mod-m_1 counter, ..., mod-m_r counter$\}$ for any number of processes.

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 Figure 3. 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 15.

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, $a_{i,j} \le a_{i,j+1}$ and $a_{i,r} \le a_{i+1,1}$. 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 $i \ge 1$, $a_{i+1,1} \ge \min(x, a_{i,1} + m_1)$.

Case I ($a_{i+1,1} = a_{i,1}$): Since $a_{i,1} \le a_{i,2} \le \cdots \le a_{i,r} \le a_{i+1,1} = a_{i,1}$, 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 15, $a_{i+1,1} = x \ge \min(x, a_{i,1} + m_1)$.

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 $a_{i+1,1} \ge a_{i,1} + m_1 \ge \min(x, a_{i,1} + m_1)$.

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

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

Case I ($a_{i,1} = a_{i+1,1}$): Since $a_{i,1} \le a_{i,2} \le \cdots \le a_{i,r} \le a_{i+1,1} = a_{i,1}$, 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 15, $a_{i,1} = x \le \max(x, a_{i+1,1} - m_1)$.

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 $a_{i,1} \le a_{i+1,1} - m_1 \le \max(x, a_{i+1,1} - m_1)$.

It follows by induction that $a_{i,1} \le \max(x, a_{q,1} - (q-i)m_1)$. Thus we have $a_{s,1} \le x$, since s was chosen so that $(q-s)m_1 = m$ and $a_{q,1} - (q-s)m_1 = a_{q,1} - m < 0 \le x$. 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 17 *The reliable, asynchronous, 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 12. Similarly an object of type A cannot be implemented from read-write registers and objects of type C. However, by Proposition 16, an object of type A can be implemented using objects of type B and C. □

8 Conclusions and Open Problems

We have given an initial characterization of the relative powers of various communications mechanisms in reliable anonymous distributed systems, showing in particular that when the number of processes is known, an anonymous broadcast system has equivalent power to a shared-memory system with counters. We have further explored the relative power provided by different types and sizes of bounded counters, and shown the non-robustness of the asynchronous model by demonstrating that mod-p and mod-q counters can together implement a pq-valued counter, even though neither can implement a pq-valued counter alone (for primes p and q).

The question of what happens when the number of processes is unbounded or unknown remains open. We suspect that in a system with unbounded counters, it may be possible to have processes register themselves as they first enter the system, thereby reducing these cases to the case where n is bounded and known. But guaranteeing that all processes have a consistent and accurate view of the count appears difficult, and may require mechanisms tailored for specific applications.

Acknowledgements A preliminary version of this paper appeared as [2]. We thank Bernadette Charron-Bost for helpful discussions. James Aspnes was supported in part by NSF grants CCR-0098078, CNS-0305258, and CNS-0435201. Faith Ellen Fich and Eric Ruppert were supported by the Natural Sciences and Engineering Research Council of Canada. Faith Ellen Fich was also supported by Sun Microsystems.

References

1. Angluin, D.: Local and global properties in networks of processors. In: Proceedings of the 12th ACM Symposium on Theory of Computing, pp. 82–93 (1980)
2. Aspnes, J., Fich, F., Ruppert, E.: Relationships between broadcast and shared memory in reliable anonymous distributed systems. In: Proc. 18th International Symposium on Distributed Computing, *LNCS*, vol. 3274, pp. 260–274 (2004)
3. Aspnes, J., Shah, G., Shah, J.: Wait-free consensus with infinite arrivals. In: Proceedings of the 34th ACM Symposium on Theory of Computing, pp. 524–533 (2002)
4. Attiya, H., Gorbach, A., Moran, S.: Computing in totally anonymous asynchronous shared memory systems. Information and Computation **173**(2), 162–183 (2002)
5. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, second edn. Wiley-Interscience (2004)
6. Boldi, P., Vigna, S.: Computing anonymously with arbitrary knowledge. In: Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, pp. 173–179 (1999)
7. Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Distributed Computing, 15th International Conference, *LNCS*, vol. 2180, pp. 33–47 (2001)
8. Buhrman, H., Panconesi, A., Silvestri, R., Vitanyi, P.: On the importance of having an identity or, is consensus really universal? In: Distributed Computing, 14th International Conference, *LNCS*, vol. 1914, pp. 134–148 (2000)
9. Chandra, T.D.: Polylog randomized wait-free consensus. In: Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, pp. 166–175 (1996)
10. Drulă, C.: The totally anonymous shared memory model in which the number of processes is known. Personal communication.
11. Eğecioğlu, O., Singh, A.K.: Naming symmetric processes using shared variables. Distributed Computing **8**(1), 19–38 (1994)
12. Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing **16**(2-3), 121–163 (2003)
13. Fredkin, E.: Trie memory. Commun. ACM **3**(9) (1960)
14. Guerraoui, R., Ruppert, E.: What can be implemented anonymously? Tech. Rep. 200496, School of Computer and Communication Sciences, EPFL (2004)
15. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Progr. Lang. Syst. **12**(3), 463–492 (1990)
16. Jayanti, P.: Robust wait-free hierarchies. J. ACM **44**(4), 592–614 (1997)
17. Jayanti, P.: Solvability of consensus: Composition breaks down for nondeterministic types. SIAM J. Comput. **28**(3), 782–797 (1998)
18. Jayanti, P., Toueg, S.: Wakeup under read/write atomicity. In: Distributed Algorithms, 4th International Workshop, *LNCS*, vol. 486, pp. 277–288 (1990)
19. Kutten, S., Ostrovsky, R., Patt-Shamir, B.: The Las-Vegas processor identity problem (How and when to be unique). Journal of Algorithms **37**(2), 468–494 (2000)
20. Lipton, R.J., Park, A.: The processor identity problem. Inf. Process. Lett. **36**(2), 91–94 (1990)
21. Panconesi, A., Papatriantafilou, M., Tsigas, P., Vitányi, P.: Randomized naming using wait-free shared variables. Distributed Computing **11**(3), 113–124 (1998)
22. Reif, J.H. (ed.): Synthesis of Parallel Algorithms. Morgan Kaufmann (1993)
23. Sakamoto, N.: Comparison of initial conditions for distributed algorithms on anonymous networks. In: Proceedings of the 18th ACM Symposium on Principles of Distributed Computing, pp. 173–179 (1999)
24. Shapiro, H.N.: Introduction to the Theory of Numbers. John Wiley and Sons (1983)
25. Teng, S.H.: Space efficient processor identity protocol. Inf. Process. Lett. **34**(3), 147–154 (1990)

James Aspnes is an Associate Professor of Computer Science at Yale University. He received his Ph.D. from Carnegie-Mellon University in 1992.

Faith Ellen Fich is a Professor of Computer Science at the University of Toronto. She received her Ph.D. from the University of California, Berkeley in 1982.

Eric Ruppert was educated at the University of Toronto, where he completed his doctorate in 1999. He spent a year as a postdoctoral fellow at Brown University and is an Assistant Professor at York University.