ACM SIGACT News Distributed Computing Column 14

Sergio Rajsbaum*

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, Internet, and the Web.

This issue consists of the paper "A Pleasant Stroll through the Land of Infinitely Many Creatures" by Marcos Aguilera. Many thanks to him for contributing to this issue.

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, open problems, and authors willing to write a guest column or to review an event related to theory of distributed computing.

A Pleasant Stroll through the Land of Infinitely Many Creatures

Marcos K. Aguilera [1]

Abstract

Many distributed algorithms are designed for a system with a fixed set of n processes. However, some systems may dynamically change and expand over time, so that the number of processes may grow to infinity as time tends to infinity. This paper considers such systems, and gives algorithms that are new and simple (but not necessarily efficient) for common problems. The reason for simplicity is to better expose some of the algorithmic techniques for dealing with infinitely many processes. A brief summary of existing work in the subject is also provided.

1 Motivation

Many distributed algorithms are designed for a system with a fixed set of processes. These algorithms often make use of the number n of processes and their identity. However, some systems may dynamically change and grow over time, so that the number of processes does not remain bounded by n. For example, a network may allow new nodes to be added, or an operating system may allow new local processes to be created, where it may be desirable for the new local processes to participate in the distributed system. In those cases, distributed algorithms are needed that can

*Instituto de Matemáticas, UNAM. Ciudad Universitaria, Mexico City, D.F. 04510 `rajsbaum@math.unam.mx`.

[1] `firstname.lastname at hp dot com`. HP Laboratories, 1501 Page Mill Road, Palo Alto, California, 94304, USA.

work with a dynamic system that may have a growing set of processes—a growth that may tend to infinity as time tends to infinity.

This paper is about such systems with infinitely many processes [GK98, MT00]. These systems are not merely a theoretical diversion; rather, the requirement that an algorithm work in such systems forces the algorithm to be generic and have many nice characteristics. In fact, algorithms for infinitely many processes have the following advantages over algorithms for n processes:

- *They have no system size parameters to configure,* and as a result they are more robust and elegant. There is no need to predict bounds on the system size, or to reconfigure the bounds once the predictions become obsolete (which is often a non-trivial task in a widely deployed system). An interesting analogy here is the use of linked lists in place of vectors in programs, to allow a data structure to grow without preset bounds.

- *They automatically handle crash recovery of processes.* A process that crashes and recovers can join the algorithm simply by assuming a new identity. Therefore, the algorithm designer need not worry about designing ad-hoc recovery procedures.

- *They guarantee progress even if processes keep on arriving.* In fact, algorithms for infinitely many processes cannot be indefinitely delayed if new processes keep arriving as the algorithm tries to make progress. This is particularly important in extremely loosely-coupled systems, like peer-to-peer systems, where there is a large number of nodes that come and go all the time.

In addition to these motivations, from a conceptual viewpoint, it is important to investigate if knowledge of the system size, or of eventual absence of system growth, is inherently needed for algorithmic design; or if such a knowledge can improve the performance of algorithms; or if such a knowledge is merely an unjustified convenience for the algorithm designer. This type of investigation requires the development of algorithms and lower bounds for systems with infinitely many processes.

In this paper, my goal is to cover some basic algorithmic techniques to handle infinitely many processes in shared memory systems. To do so, I selected a few important problems and present some new solutions, which are considerably simpler than existing ones. My criterion for simplicity is that an algorithm is simple if either it is relatively short, or it consists of a small number of modifications to an algorithm that is widely understood. The hope is that simplicity can better expose the underlying core techniques. The solutions in this paper are not the most efficient ones; in fact, efficiency requires an additional set of techniques—not covered here—that sometimes get in the way of understanding the basics.

What is a system with infinitely many processes?

There are three ways in which infinitely many processes can occur in a model: across runs, along a run as time passes, or instantaneously in a run. More precisely, we have the following models:

- (Model M_1^n) *The system has a finite number n of processes.* This is the traditional model, where n may be used by algorithms. This model is called the *n-arrival model.*

- (Model M_2) *The system has infinitely many processes, but each run has only finitely many.* Therefore, there is no bound on the number of processes for all runs: for every N, there are runs with more than N processes. There is a bound on the number of processes *in each*

Model	Name	Brief description
M_1^n	Traditional model or n arrival model	System has $n < \infty$ processes.
M_2	Finite arrival model	System has ∞ processes, but each run only has finitely many.
M_3	Infinite arrival model or unbounded concurrency model	System has ∞ processes, processes may keep arriving forever.
M_4	Infinite concurrency model	Runs with ∞ processes taking steps concurrently (theoretical).
M_3^b	Infinite arrival model with b-bounded concurrency	M_3, but concurrency is at most b.
M_3^{finite}	Infinite arrival model with bounded concurrency	M_3, but concurrency does not grow indefinitely.

Figure 1: Models for finitely and infinitely many processes.

run, but an algorithm does not know what is that bound because it varies from run to run. The only guarantee is that for each run there is a time after which no new processes start executing. This is the *finite arrival* model, and algorithms designed for it are called *uniform algorithms* [GK98].

- (Model M_3) *The system has infinitely many processes, runs can have infinitely many processes, but in each finite time interval only finitely many processes take steps.* The total number of processes in a *single* run may grow to infinity as time passes. However, in any bounded time interval, only a finite number of processes take steps. Intuitively, the source of infinitely is the passage of time. This is the *infinite arrival model* or the *unbounded concurrency model*. Unlike with uniform algorithms, algorithms designed for model M_3 have the nice property that progress may not be repeatedly postponed as new processes start executing.

- (Model M_4) *The system has infinitely many processes, runs can have infinitely many processes, and a finite time interval can have infinitely many processes.* Thus, an infinite number of processes can take steps concurrently. This is the *infinite concurrency model*. This model is of theoretical interest only, and it does not seem to capture existing physical systems.

The literature also has variants of model M_3 that considers *departures* of processes. Roughly speaking, a process that departs is no longer active and it does not take part in the algorithm. The *concurrency* at a given time is the number of active processes at that time, that is, the number of process that have joined (arrivals) minus the number of departures. Based on concurrency, there are two variants of M_3:

- (Model M_3^b) Consists of model M_3, with the additional restriction that every run has a maximum concurrency bounded by constant b, which is known to algorithms. Therefore, in model M_3^b, it is possible to have infinitely many processes only if processes depart at the same rate that new processes join. This is the *infinite arrival model with b-bounded concurrency*.

- (Model M_3^{finite}) Consists of model M_3, with the additional restriction that each run has a maximum concurrency that is finite. This is the *infinite arrival model with bounded concurrency*.

Figure 1 shows all the models and their usual names, and Figure 2 shows their relationship.

In this paper I focus on algorithms for model M_3, which is very general: algorithms for M_3 also apply to all other models except M_4. In model M_3, I show how processes can acquire unique

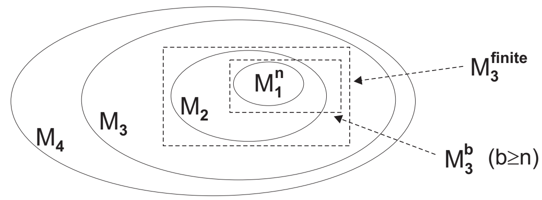

Figure 2: Inclusion relationship between models.

$\mathbf{N} = \{1, 2, \ldots\}$; $\mathbf{R}$ is the set of real numbers; $i..j$ is the set of integers from i to j, inclusive; $[x, y]$ is the closed interval of real numbers between x and y; a sequence is a function with domain $\mathbf{N}$; an array A is a set of pairs ($index$, $value$), and $A[index]$ indicates the value associated with $index$; if there are no such values, $A[index] = \perp$.

Figure 3: Notation used in this paper.

names, how they can count together, how they can take atomic snapshots of memory, how they can find the other active processes, and finally, how they can solve mutual exclusion. At the end of the paper, I show how the model M_3 itself can be implemented, and give a brief summary of other work, including work on the other models of infinitely many processes.

Bibliographic notes

Model M_2 was proposed by Gafni and Koutsoupias [GK98], who were the first to consider a system with infinitely many processes. Models M_3, M_4, M_3^b, and M_3^{finite} were proposed by Merritt and Taubenfeld [MT00], who were the first to consider a system where runs can have infinitely many processes. Many techniques presented in this paper were first introduced by Gafni, Merritt and Taubenfeld [GMT01].

2 Informal modeling

Some standard notation is given in Figure 3. We consider an asynchronous distributed system with a set $\Pi = \mathbf{N}$ of processes that can communicate through a shared memory. The shared memory has a set of registers or memory locations that support atomic read and write operations, with the usual semantics. The registers are *multi-writer multi-reader*: they can be accessed by any process in the system. Sometimes we consider a memory that supports *test-and-set*, which sets a location to 1 and returns its previous value.

Each process is an automaton, possibly with infinitely many states, that computes by taking steps. The system is asynchronous, that is, the next step of a process may occur after an arbitrarily long time. In each step, a process performs the following, in order: (1) either issue an output event, or read from shared memory, or write to shared memory, or none of the above, (2) possibly receive an input event, and (3) change state. The model has time that ranges over $\mathbf{R}$. This time is not accessible to processes; it is merely a technical convenience. A step includes a timestamp to indicate when it happens, and a process to indicate who takes the step. A *run* R is a countably infinite set of steps with distinct timestamps. A *process in* R is a process that has a step in R. We define models M_1^n, M_2, and M_3 (we do not try to define M_4). In M_1^n, all processes of every run are in $1..n$. In M_2, every run has only finitely many processes. In M_3, a run may have infinitely many processes. In M_1^n, M_2, M_3, a run has only finitely many steps with timestamp smaller than x for every $x \in \mathbf{R}$. Thus, if we order steps by increasing timestamp, we get a sequence.

A process may receive an input event with an operation *invocation*, that is, a request to execute an operation O. Later, the process may issue an output event with the *response* for O. Input events can happen at any time, except that a process will not receive two invocations without first issuing a response in between: intuitively, a process does not receive concurrent invocations. We say that operation[2] O is *active* at a given time if a process has received an invocation for O but has not yet output a response. We say that a process is *active* if it has an active operation; else the process is *inactive*. An inactive process continues to take steps, but it cannot issue output events, or read or write memory; intuitively, this prohibits background activities. The *concurrency at time* t is the cardinality of the set of active processes at time t. The *point contention* of an operation O is the largest concurrency from the time of O's invocation until its response.

A process that has taken steps may fail by crashing, in which case it stops taking steps. A *correct* process is one that takes infinitely many steps. An algorithm is *wait-free* [Her91] if it guarantees that correct processes always complete their operations (in a finite number of steps).

Linearizability [HW90] is a standard way of defining the concurrent behavior of an object from its sequential specification; roughly speaking, it requires that an operation appear to take place at some instantaneous time between its start and end. Linearizable objects are sometimes called atomic objects.

Finally, runs must conform with the automaton defined for each process (no byzantine failures) and with the semantics of read and write of shared memory.

3 Our first algorithm: naming the anonymous

To gain some understanding of the models for infinitely many processes, consider the problem of *naming*, where processes are all initially anonymous and execute the same code, and they must eventually output an identifier such that no two processes output the same identifier. Even in the traditional model M_1^n, this problem has no solution with a memory that only supports read and write operations[3]. Thus, to solve naming, let us assume that the memory has a test-and-set operation. Figure 4 shows a very simple algorithm, which uses an array of bits to indicate if a name is taken; a process uses test-and-set to iterate through the array to grab a name that is not yet taken.

This algorithm works well in model M_1^n, and even in model M_2, where each run has only finitely many processes, so that every correct process eventually finds a name and terminates. Unfortunately, in model M_3, a correct process p may execute forever because, just before p grabs some name i, a new process may start and execute very quickly so that it grabs name i before p. This can keep on happening forever because, in model M_3, new processes may keep on arriving.

This type of liveness problem is essentially what distinguishes model M_3 from M_2: in fact, any algorithm A (for any problem) that works in M_2 can only fail in M_3 by violating liveness, and only if new, distinct processes keep appearing. This is because, if A were to violate safety in M_3 at some time t, then we can prevent all new process arrivals after time t, and get a run in M_2 that also violates safety, contradicting the fact that A works in M_2.

In some sense, a similar problem occurs in the design of wait-free algorithms for M_1^n, when incoming operations may keep delaying a starving operation. In M_3 the problem is exacerbated, because new operations may originate from distinct processes, which affects the helping mechanisms,

[2]In this paper, we are sloppy about the distinction between operations and their execution.

[3]Intuitively, this is because processes may always execute in tandem, in a way that their computation never diverges from each other [HS80, JS85].

```
Shared variables: used[1..∞], initially all 0

procedure get-name()
    j ← 1
    while test-and-set(used[j]) = 1 do
        j ← j + 1
    name ← j
```

Figure 4: Wait-free algorithm for naming for model M_2 with test-and-set.

```
Shared variables:
    used[1..∞][1..∞], initially all 0
    currentcopy, initially 1

procedure get-name()
1   i ← currentcopy
2   currentcopy ← i + 1
3   j ← 1
4   while test-and-set(used[i][j]) = 1 do
5       j ← j + 1
6   name ← (i, j)
```

Figure 5: Wait-free algorithm for naming for model M_3 with test-and-set.

as we discuss in Section 4. Here, helping mechanisms do not apply because there are no process identifiers, so we need a different approach.

Our solution is based on the observation that if we limit the number of processes that are racing with p then p will not starve. One way to do so, shown in Figure 5, is to use (infinitely) many copies of the algorithm in Figure 4, and ensure that each copy is executed by only finitely many processes. When a process p executes, it picks one copy to "enter"; when p finishes executing that copy, it chooses as its name the value $\langle i, j \rangle$ where i is the copy number, and j is the name within the copy. The copy chosen is the value of shared register *currentcopy*, which then p increments. The chosen names are pairs $\langle i, j \rangle$, but it is easy to convert them into natural numbers with an injection f from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$, such as the one obtained by walking through each diagonal of $\mathbf{N} \times \mathbf{N}$.

Informal correctness argument

In a run, the value of *currentcopy* may bounce up and down a few times, but we show that it is impossible for an infinite set processes to read the same value from *currentcopy*. More precisely, to show correctness, we say a *process enters copy k if it executes line 2 with $i = k$*. We show by induction on k that only finitely many processes enter copy $\leq k$. For $k = 1$, after some process executes line 2, say at time t, *currentcopy* forever after has a value ≥ 2. Therefore, only processes that execute line 2 before time t—a finite number in model M_3—will enter copy 1, which shows the induction hypothesis. For the induction step, assume that only finitely many processes enter a copy $\leq k$, and let t_1 be the latest time when this happens. If no processes enter copy $k+1$ after time t_1, then the induction step holds. Else, at some time $t_2 > t_1$, some process enters copy $k+1$. From time t_2 onward, we have *currentcopy* $\geq k+2$, because *currentcopy* can only be set to a value $\leq k+1$ if a process enters a copy $\leq k$. Thus, only processes that execute line 2 before time t_2—a finite number—will enter a copy $\leq k+1$, which shows the induction step.

Since only finitely many processes enter each copy, termination is always guaranteed, even in model M_3.

Further reading and bibliographic notes

The idea in Figure 5 of using multiple copies of a data structure to bound concurrency was inspired by the sieve construction by Attiya and Fouren [AF03]. Although naming has no solutions with only registers, Aspnes et al [ASS02] show that probabilistic naming—where processes only need to obtain distinct names with high probability—is possible in model M_3^b with only registers and coin tosses. For M_3, it is shown that even this weaker naming is impossible with a *strong adversary*, i.e., an adversary that can schedule processes according to their state. Merritt and Taubenfeld [MT00] show that there is no wait-free algorithm for naming using only test-and-set bit objects (no read and write operations) in model M_3.

4 A helping mechanism for a simple problem

Henceforth we assume that each process has a unique identifier in $\mathbf{N}$, and that memory supports only read and write operations. When designing wait-free algorithms, a common problem is to prevent a slow process from executing forever when other faster processes keep executing operations—a problem that we already saw in Section 3. The so-called *helping* technique (e.g., [Her91]) is frequently used to solve this problem, by having a process help other slower processes to finish their operation. More precisely, whenever a process executes its own operation, it also checks if other processes are also concurrently executing some operation and, if so, the process helps them complete those operations. This technique has been extensively used in wait-free algorithms for model M_1^n and, as we show in this section, it also applies to model M_3 after some difficulties are solved.

The first difficulty is to determine whom to help: in model M_1^n, a process can check if any of the other $n-1$ processes need help, but in M_3, this checking is not possible because there is an infinite set of candidates. If we only had a mechanism in M_3 to tell who potentially has outstanding operations at a given time, then our problem would be solved: a process can simply obtain such a list—which will have a finite number of processes—and then help the processes in it. We will ultimately show how to build such a list in Section 6, but until we do so, we have a chicken-egg problem. This is because assembling the list needs to be done in a wait-free manner, and so we would like to use a helping mechanism for that.

A second difficulty is that, even if processes know exactly whom to help, in M_3 the helping mechanism needs to be careful with the order in which it does things. For instance, it might be tempting for processes to first (1) execute their own operation, and then (2) help other processes. This can work in M_1^n, but generally it will not in M_3, because an infinite number of processes can perform step (1) and then crash without performing (2): if that happens, a slow process will never get any help. Therefore, in M_3 a process should first help its peers before making data structure changes that can delay them [GMT01].

Whom to help?

Going back to the first difficulty, how can a process p determine whom it will check for need of help? First, note that p need not try to help everyone who has an outstanding operation: it can leave out processes, as long as they are guaranteed to be helped by someone else. More precisely, we would like to guarantee the following "check property":

- For every process p, if there are infinitely many operations executed that change the data structures, then infinitely many will check if p wants help.

One way to ensure the check property is to have the i-th operation execution of a process q to check (for need of help) all processes whose identifiers are between 1 and $i+q$. To see why this works, note that if there are infinitely many operations, then either (1) there is a process that executes infinitely many operations, or (2) there are infinitely many processes that execute operations. In both cases, any process p will be checked infinitely often.

Shared variables:
  bit[1..∞], initially all 0
  working[1..∞], initially all *false*
  help[1..∞], initially all 0

Code for process p:

```
procedure increment()
    n ← get-count()
    bit[n + 1] ← 1
```

```
procedure get-count()
    i ← 1
    working[p] ← true
    assist ← {}
    while bit[i] = 1 and help[p] = 0 do
        if working[i] then assist ← assist ∪ {i}
        i ← i + 1
    working[p] ← false
    if help[p] ≠ 0 then return help[p]
    else
        for each q ∈ assist in increasing order do help[q] ← i − 1
        return i − 1
```

Figure 6: One-shot wait-free algorithm for weak counters for model M_3.

There are also ad-hoc ways to ensure the check property, which are specific to the problem being solved, as we show below.

Weak counters

Let us illustrate more concretely the high-level ideas explained so far, by building a relatively simple data structure, which we call a *weak counter*. A weak counter supports two operations, *get-count* and *increment*, which returns the current value of the counter and increments the counter, respectively. The counter is weak because if multiple processes *increment* concurrently, the counter may only increment by 1. In fact, the only thing we require is that the counter never go back and that *increment* cause the counter to get a higher value. More precisely, a weak counter guarantees that *get-count* returns a value in $\mathbf{N}$ such that

- *Monotonicity*: if $\textit{get-count}_1$ precedes $\textit{get-count}_2$ then $\textit{get-count}_2$ does not return a smaller value than $\textit{get-count}_1$, and
- *Weak increment*: if $\textit{get-count}_1$ precedes *increment*, which in turns precedes $\textit{get-count}_2$, then $\textit{get-count}_2$ returns a larger value than $\textit{get-count}_1$,

where we say that an operation precedes another if it ends before the other starts [Lam86a].

Figure 6 gives a wait-free algorithm for weak counters where each process may execute at most one operation—a so-called *one-shot* implementation. We later remove this restriction. The algorithm's basic idea is to represent the value of the counter in unary in a bit array. To increment the counter, a process reads its current value, and then sets one more bit to one. To read the counter's value, a process p scans the array to count the number of bits set to one; to ensure that p terminates if the counter keeps increasing, we use a helping mechanism as follows. Before scanning, p announces that it wants help by setting *working*$[p]$ to true. A process q that scans beyond a value i will remember who among processes $1..i$ wants help. Once the scan is over, process q helps those found to be in need of help. A process that gets helped will not itself help other processes, because its help may be stale. When executing *increment*, this help is performed before a new bit is set to one. This is the place where the process helps before making a data change that may delay others.

To extend the algorithm for multiple uses per process, we can reserve a priori infinitely many identities per process, and let the process use a new identity for each operation [GMT01]. More precisely, if f is a injection from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$, then we reserve identities $f(p, 1), f(p, 2), \ldots$ to process p; upon executing its i-th operation, process p assumes identity $f(p, i)$.

We can also extend the implementation to multiple operations per process in a more traditional way. The problem with multiple operations is that a process p, upon executing a *get-count*, may get bogus help from some slow process trying to help one of p's previous operations, which can cause violation of monotonicity. To solve this problem, we replace the *help* vector with a matrix, where $help[p][i]$ is the location to help the i-th operation of p. A vector $current[p]$ keeps the count of operations for each process p. When a process q adds p to *assist*, it remembers the value in $current[p]$, to know which entry in $help[p][*]$ to use. In this way, if q is slow so that p completes the i-th operation, q's subsequent stale help will not harm p's operation.

Proof of correctness (one-shot algorithm)

Definition 1 *For any time t, let $f(t)$ be the number of ones in vector bit at time t.*

Lemma 2 *Function $f(t)$ is monotonically nondecreasing.*

PROOF: Trivial since entries of *bit* can only be changed from 0 to 1. ■

Lemma 3 *If get-count returns n at time t then, at time t, $bit[j] = 1$ for every $j = 1..n$.*

PROOF: We first claim that if some process q executes *get-count*, then while $help[q] = 0$ we have $bit[j] = 1$ for $j = 1..i_q - 1$, where i_q is the value of the i variable of q. This claim clearly holds because of the guard of the while loop.

Now suppose that upon executing *get-count*, some process p returns n at time t. There are two possibilities. (1) *p returns in line 13.* In this case, by the claim we have $bit[j] = 1$ for $j = 1..i_p - 1 = n$, which shows the lemma. (2) *p returns in line 10.* In this case, p returns $n = help[p] \neq 0$. The process q that previously set $help[p]$ does so at some time $t' < t$ in line 12. By the claim, at time t', $bit[j] = 1$ for $j = 1..i_q - 1$ where $i_q - 1 = help[p] = n$. After being set to 1, a bit is never set to 0, so at time t we also have $bit[j] = 1$ for $j = 1..n$. ■

Corollary 4 *At time t, if $bit[i] = 1$ then for any j in $1..i$ we have $bit[j] = 1$.*

PROOF: When a new $bit[j]$ is set to 1 in line 2, by Lemma 3 all previous entries are also set to 1. ■

Corollary 5 *At any time t, the $f(t)$ entries of bit that are set to 1 are $bit[1], \ldots, bit[f(t)]$.*

Lemma 6 *Suppose a process p executes line 12 of get-count, and let t be the time when p previously exits the while loop in line 6. Then the value of $i - 1$ during the execution of line 12 is $f(t)$.*

PROOF: If p executes line 12, then p exits the while loop because it finds $bit[i] = 0$ at time t. By Corollary 5 and the fact that $bit[i-1] = 1$, at time t, $i - 1 = f(t)$. Afterwards, p does not change the value of $i - 1$, so the lemma follows. ■

Lemma 7 *If get-count is called at time t and get-count returns, then it returns a value that is at least $f(t)$.*

PROOF: Suppose some process p calls *get-count* at time t and returns a value n at a time t_1. Let t_2 be the time when p leaves the while loop. Then $t \leq t_2 \leq t_1$. Now p either returns in line 10 or 13. If p returns in line 13, then by Lemma 6, p returns $n = f(t_2)$, and since $t \leq t_2$, by Lemma 2, $f(t) \leq f(t_2) = n$, which shows the lemma. If p returns in line 10, then p returns a value $n = \mathit{help}[p] \neq 0$. Consider the process q that sets *help*[p] to n. Process q does so in line 12 at some time that is on or before the time t'_2 when q leaves the while loop. By Lemma 6 applied to q, at time t'_2, the value assigned to *help*[p] is $n = f(t'_2)$. Process q adds p to the set *assist* at a time $t_3 \leq t'_2$, and t_3 is not before the time t_4 when p sets *working*[p] to *true*, and t_4 is not before the time t when p calls *get-count*. Thus, we have $t \leq t_4 \leq t_3 \leq t'_2$, and so $t \leq t'_2$. By Lemma 2, $f(t) \leq f(t'_2)$, and since $f(t'_2) = n$, we have $f(t) \leq n$. ■

Lemma 8 *If increment is called at time t_1 and returns at time t_2 then $f(t_1) + 1 \leq f(t_2)$.*

PROOF: Note that *increment* calls *get-count*. Let t_3 be the time when this happens, and t_4 be the time when *get-count* returns. Let n be the value returned by *get-count*. By Lemma 7, $n \geq f(t_3)$. Since $t_3 \geq t_1$, by Lemma 2, $f(t_3) \geq f(t_1)$. Due to line 2, and by Lemma 5, we have $f(t_2) \geq n + 1$. Therefore $f(t_2) \geq f(t_3) + 1 \geq f(t_1) + 1$. ■

Lemma 9 *If get-count$_1$ precedes get-count$_2$ then get-count$_2$ does not return a smaller value than get-count$_1$.*

PROOF: Let t_1 be the time when *get-count*$_1$ returns, and t_2 be the time when *get-count*$_2$ starts. Then $t_1 \leq t_2$. By Lemma 3, *get-count*$_1$ returns a value $v_1 \leq f(t_1)$. By Lemma 2, $f(t_1) \leq f(t_2)$. By Lemma 7, *get-count*$_2$ returns a value v_2 such that $f(t_2) \leq v_2$. Therefore $v_1 \leq v_2$. ■

Lemma 10 *If get-count$_1$ precedes increment, which in turns precedes get-count$_2$, then get-count$_2$ returns a larger value than get-count$_1$.*

PROOF: t_1 be the time when *get-count*$_1$ returns, t_2 be the time when *increment* starts, t_3 be the time when *increment* returns, and t_4 be the time when *get-count*$_2$ starts. Then $t_1 < t_2 < t_3 < t_4$. Let v_1 be the value returned by *get-count*$_1$ and v_2 be the value returned by *get-count*$_2$. Then we have $1 + v_1 \leq 1 + f(t_1) \leq 1 + f(t_2) \leq f(t_3) \leq f(t_4) \leq v_2$ where the inequalities hold, respectively, by Lemmas 3, 2, 8, 2, and 7. Thus $1 + v_1 \leq v_2$. ■

Lemma 11 *The algorithm in Figure 6 implements a weak counter where processes can execute at most one operation.*

PROOF: By Lemmas 9 and 10. ■

Lemma 12 *The algorithm in Figure 6 is wait-free.*

PROOF: To show wait-freedom, suppose for a contradiction that some process p starts *get-count* but never terminates. Then p loops forever in the while loop of lines 6–8. Therefore, for every i, when p reads *bit*[i], p gets 1. Thus, infinitely many calls to *increment* terminate. Thus, infinitely many calls to *get-count* terminate. There are two non-exclusive cases: (a) infinitely many *get-count*s return at line 13, and (b) infinitely many *get-count*s return at line 10. In case (a), there exists some process q such that (1) q starts *get-count* after p does, (2) q adds p to q's *assist* set, and (3) q executes lines 12–13 with $i \geq 2$. Therefore, q sets *help*[p] to a nonzero value. Soon after, p exits the while loop—a contradiction. In case (b), there are infinitely many processes q with identifier $q > p$ that get their *help*$[q]$ set to non-zero. For each such q, let p_q be a process that sets *help*$[q]$ to a non-zero value in line 12. There are infinitely many distinct p_q processes, and so there exists some $p_{q'}$ that starts *get-count* after p sets *working*$[p]$ to *true*. Then $p_{q'}$ adds p to *assist* and, before setting *help*$[q']$ to a non-zero value, $p_{q'}$ sets *help*$[p]$ to a non-zero value (since $q' > p$ and the for loop in line 12 is executed by increasing order of process identifiers). Soon after, p exits the while loop—a contradiction. ■

A trivial improvement to the algorithm

To avoid having *get-count* always scan all bits, we could try to keep the position of the last bit set in a shared variable *shortcut*. We then create a *fast-get-count* that starts the scan at position *shortcut*; it also does not perform the help in line 12. We modify line 13 so that the process sets *shortcut* to $i-1$ before returning. Within *increment*, we still call the original version of *get-count* (without the shortcut), so that helping works. But the fast version can be used by a process that just wants to read the counter. In this way, if *increment* operations stop, then eventually *fast-get-count* takes only constant time.

Bibliographic notes

The technique of allocating infinitely many identifiers to each process in M_3 to transform one-shot algorithms into long-lived ones was first proposed and applied to the name snapshot problem by Gafni et al [GMT01]. They were also the first to observe that, in model M_3, it is important for processes to "help first".

5 Taking snapshots and collecting values

Atomic snapshot [AAD$^+$93] is a useful primitive to build distributed algorithms by allowing processes to read data from multiple processes in an *atomic manner*, much like taking a snapshot of memory. More precisely, each process has one reserved location to store data using an *update* operation, and processes can atomically read the values of all reserved locations using a *scan* operation. To avoid returning infinitely large arrays in models M_2 and M_3, where there are infinitely many reserved locations, we define that all locations are initially empty, and require *scan* to return only the non-empty ones.

Figure 7 shows the classical atomic snapshot algorithm from [AAD$^+$93], for the standard model M_1^n (in the code, recall that we represent arrays as set of pairs $(index, value)$). The basic idea[4] is that, to execute a *scan*, a process p repeatedly collects the values of all processes, hoping that two consecutive collects will be equal, in which case a consistent snapshot is found (lines 7-8). If they are not equal, then some process must have finished executing an update. If p keeps finding such processes, then eventually one of them, say j, will have finished two updates. Therefore, the later update must have began after p started the scan. Every update performs itself a scan, and stores the resulting snapshot in a shared variable. Since the latter update began after p started the scan, p can return the update's snapshot as p's own snapshot (line 11).

We can modify this algorithm to work in M_3, by using the weak counters from the Section 4 to keep track of the largest process L to execute *update*, as follows: when a process p executes update, it increments the counter until it is at least p. Then, the idea is that a collect only needs

[4]This brief explanation is only a summary. For details, see the original paper [AAD$^+$93].

```
Shared variable: r[1..n], initially all ⊥

Code for process p:

   Private variable: seq, initially 0

   procedure scan()
1     moved ← {}
2
3     a ← {(j, r[j]) : 1 ≤ j ≤ n}
4     while true do
5
6        b ← {(j, r[j]) : 1 ≤ j ≤ n}
7        if for every 1 ≤ j ≤ n, a[j] = b[j] then
8           return {(i, a[i].data) : 1 ≤ i ≤ n and a[i] ≠ ⊥}
9        else for j ← 1 to n do
10          if a[j] ≠ b[j] then
11             if j ∈ moved then return b[j].snap
12             else moved ← moved ∪ {j}
13       a ← b

   procedure update(data)
14
15    snap ← scan()
16    seq ← seq + 1
17    r[p] ← (data, seq, snap)
```

Figure 7: Classical wait-free algorithm for atomic snapshot for model M_1^n.

```
Shared variable: r[1..∞], initially all ⊥

Code for process p:

   Private variable: seq, initially 0

   procedure scan()
1     moved ← {}
2*    n ← get-count();   ninit ← n
3     a ← {(j, r[j]) : 1 ≤ j ≤ n }
4     while true do
5*       n ← get-count()
6        b ← {(j, r[j]) : 1 ≤ j ≤ n}
7        if for every 1 ≤ j ≤ n, a[j] = b[j] then
8           return {(i, a[i].data) : 1 ≤ i ≤ n and a[i] ≠ ⊥}
9        else for j ← 1 to n do
10          if a[j] ≠ b[j] then
11*            if j ∈ moved or j > ninit then return b[j].snap
12             else moved ← moved ∪ {j}
13       a ← b

   procedure update(data)
14*   while get-count() < p do increment()
15    snap ← scan()
16    seq ← seq + 1
17    r[p] ← (data, seq, snap)
```

Figure 8: Modified wait-free algorithm for atomic snapshot for model M_3 (asterisks indicate a difference from the previous algorithm).

to look at processes 1 to L, rather than an infinite number. The only subtlety is that, if L keeps on increasing while a scan is executing, the scan might never find a process j that completes more than one update: it is possible that every new update comes from a new process. To solve this problem, note that if L increases during the scan due to a new process p, then p must have started the update after the scan began. Therefore, the scan can simply return the snapshot associated with p's update, even though p has not executed two updates. The full algorithm is in Figure 8 (in the code, if an index j is part of an array a, then $a[j] = \perp$).

Informal correctness argument

We now show correctness of the algorithm in Figure 8. Consider a run with calls to *scan* and *update*. We show how to choose a serialization point for each scan and update operation in the run. For an update, its serialization point occurs when the process executes line 17. For a scan operation S, we define its serialization point $P(S)$ by induction. If S returns in line 8, $P(S)$ occurs at the beginning of the last loop iteration in line 4 before the process returns in line 8. If S returns in line 11, $P(S)$ occurs at the serialization point $P(S_2)$ of the scan S_2 whose value is returned in line 11. We need to show that $P(S_2)$ occurs between the time t_1 of S's start and the time t_2 of S's end. Clearly $P(S_2)$ is before time t_2 because p reads S_2's value after S_2's value is written to $r[j]$ in line 17, which occurs after S_2's execution (line 15). To see that $P(S_2)$ is after time t_1, there are two cases to consider. (1) If the return in line 11 occurs because j is in *moved*, then process j changed $r[j]$ twice between times t_1 and t_2: once causing p to add j to *moved*, and then once again because p later finds j in *moved* in line 11. Now $r[j]$ is only changed in line 17. Thus, S_2's execution, which occurs in line 15 after the first change in line 17, must be after time t_1. Hence $P(S_2)$ is also after time t_1. (2) If the return in line 11 occurs because $j > \mathit{ninit}$ then (a) during execution of S there is a time $t_1' > t_1$ when j has not yet executed any *updates* (because j executes line 14 during an *update*). Therefore, $P(S_2)$ is after time t_1.

We now show that operations return values consistent with their serialization points. This is clear for *update*s, which return no value. For a scan S, if it returns in line 8, it clearly returns a value consistent with its serialization point $P(S)$ because there were no updates whose serialization point is between the two preceding reads of array r. If S returns in line 11, a trivial induction show that it also returns a value consistent with $P(S)$, because S returns the same value as a previous scan S_2 with serialization point $P(S_2) = P(S)$.

To show wait-freedom, there is only one loop in all code, which is in *scan*. In this loop, the value of *ninit* does not change. Thus, line 11 can only execute *ninit* times without returning, since every time it does not return, some $j \le \mathit{ninit}$ is added to *moved*. Therefore, if *scan* does not return in line 11, after some finite number of steps, $a[j] = b[j]$ for every $j = 1..n$, which ensures that *scan* returns in line 8.

Further reading and bibliographic notes

The collect operation. Lines 2 and 3, and lines 5 and 6 of the algorithm in Figure 8 perform a *collect* operation, used frequently in algorithms for model M_1^n. In model M_1^n, a collect operation consists of reading one shared variable per process, one at a time. Thus, intuitively a collect operation is a "non-consistent" version of snapshot, and we can take this to be the definition for models M_2 and M_3. In these models, one can perform a collect operation by using weak counters to keep track of the largest process identifier n to hold any data; in this way, processes can simply read the shared values of processes $1..n$. There is a more efficient algorithms that is *adaptive (to point contention)*, meaning that its running time depends not on the participants' largest identifier, but rather on the point contention, which is the largest concurrency during the execution of the operation. There is a vast literature on adaptive algorithms for the collect operation in model M_1^n. Gafni et al [GMT01] show how to modify one of these algorithms, by Afek, Touitou, and Stupp [AST99], to work in M_3, while retaining adaptiveness. By using this collect algorithm into an adaptive algorithm by Afek et al [AST99], one obtains an adaptive snapshot algorithm for M_3.

Renaming processes. In algorithms whose running time depends on the value of process identifiers, it is desirable for identifiers to be relatively small. This can be achieved by renaming processes; more precisely, a $f(k)$-renaming algorithm assigns unique names to processes, such that if k processes participate then all names are in the interval $1..f(k)$. There are two variants of renaming: one-shot and long-lived. The former only allows processes to get new names, while the latter also allows processes to release and recycle names. There is a vast literature on adaptive algorithms for renaming in model M_1^n. For model M_3, there is an adaptive algorithm for long-lived renaming, obtained by modifying the algorithm by Attiya and Fouren [AF00] as shown by Gafni et al [GMT01].

Name snapshot. Gafni et al [GMT01] propose an algorithm for *name snapshot*, a variant of one-shot snapshot for process names. Roughly speaking, name snapshot has just one operation *name-snapshot*() that outputs a list of processes that have called this operation; different processes may not exactly agree on this list, but different lists are always related by inclusion. More precisely, at any time a process i may start a name snapshot, and when it terminates it outputs a set of processes S_i, called the *snapshot of* i, such that the following properties hold:

- *(Validity)* The snapshot of i contains itself, that is, $i \in S_i$

- *(Total Ordering)* Snapshots form an inclusion chain, that is, for any i and j either $S_i \subseteq S_j$ or $S_j \subseteq S_i$
- *(Integrity)* If j does not start by the time i terminates then j is not in the snapshot of i.

Note that Validity, Total Ordering and Integrity imply that if a process i terminates before j starts then the snapshot of j contains i. Name snapshot can be used to get the standard atomic snapshot by (1) having processes store their data in a shared variable, and (2) using the technique to reserve infinitely many names per process. More precisely, to perform a $update(d)$, a process p chooses a new unique name p', stores d in a variable $data[p']$, and calls *name-snapshot*(). To perform a *scan*(), process p chooses a new name p', stores $\perp$ in variable $data[p']$, calls *name-snapshot*() and, for each process q' in the gotten snapshot, p reads $data[p']$. Then, for each real name q that has a reserved name q' in the name snapshot, p looks for the most recent reserved name q'' used by q with a non-$\perp$ value for $data[q'']$. This is the value returned for q in *scan*().

6 Group membership: determining who is around

It might be useful to have a primitive that tells exactly who is *present* in the system in a *consistent* manner, where "present" could be an application-specific meaning, such as actively executing an operation that changes state, and "consistent" means intuitively that processes get atomic answers, as we explain below. This functionality is provided by a *group membership* service (e.g., [Cri87, Bir93, vRBC$^+$93, ADKM92, KT91, EMS95, BDGB94]).

Group membership was initially defined and used in message-passing systems, but it can be useful as well in a shared-memory system with infinitely many processes. Roughly speaking, group membership provides three operations, *join*, *leave*, and *get-view*, all with no parameters. The first two are used by a process to inform that it is present or not present, respectively; the *get-view* operation returns a pair $\langle set, version \rangle$, where *set* contains the set of processes that are present and *version* is an indication of how many times the membership changed so far.

More precisely, the correctness condition for concurrent access is linearizability with the following sequential specification: a *get-view* returns (1) the set of processes that previously called *join* without having subsequently called *leave*, and (2) a counter that never decreases, and increases by at least one every time that *get-view* returns a set that differs from the last one returned. Note that condition (2) guarantees that if two calls to *get-view* return the same version, then they return the same set.

Group membership has a trivial implementation using atomic snapshot (Figure 9): processes use snapshot with a boolean variable to indicate whether they have joined or left. To maintain the version number, processes count the number of *join* and *leave* operations that they performed, and sum these values after a snapshot is taken.

Further reading and bibliographic notes

Group membership has been extensively studied in the context of message-passing systems (a survey is provided by Chockler, Keidar and Vitenberg [CKV01]). In shared memory, the closest related subject is the *active set* object from Afek, Stupp, and Touitou [AST99]. Roughly speaking, an active set object supports *join*, *leave* and *get-set* operations that are similar to the operations in group membership, but with weaker guarantees: *get-set* may not return a consistent view of the set of processes (analogously to the *collect* operation when compared to atomic snapshot), and it does not return version numbers.

```
Code for process p:

  Private variable: ops, initially 0

  procedure join()
      ops ← ops + 1
      update(⟨ops, true⟩)      (* ⟨ops, ismember⟩ *)

  procedure leave()
      ops ← ops + 1
      update(⟨ops, false⟩)     (* ⟨ops, ismember⟩ *)

  procedure get-view()
      snap ← scan()
      set ← {j : snap[j].ismember}
      version ← Σ_{(j,data)∈snap} data.ops
      return (set, version)
```

Figure 9: Wait-free algorithm for group membership using atomic snapshot in model M_3.

7 Using information of who is around to achieve mutual exclusion

Mutual exclusion ensures that at most one of a competing set of processes gets access to a critical region at a time [Dij65]. Crashes are not allowed. In this section, I show simple ways to solve this problem in model M_3, by using group membership. In particular, I show how to convert to model M_3 two popular mutual exclusion algorithms, namely, Dijkstra's algorithm and the Bakery algorithm.

7.1 Dijkstra's algorithm

Figures 10 shows Dijkstra's mutual exclusion algorithm [Dij65] for model M_1^n. It is quite easy to modify the algorithm to work in M_3 by using group membership, as follows. (1) Before competing for the critical region, processes call *join* if they have not yet done so previously. And (2) we replace the loop over n processes in line 7 with a loop over the processes returned by the group membership. The resulting algorithm is shown in Figure 11.

The proof of correctness of the modified algorithm is very similar to the original algorithm's. To show that the critical region is accessed by at most one process, assume by way of contradiction that two processes p and q could be simultaneously in the critical region at some time t, and consider their last execution of lines 5–8 before entering the critical region. Then one of them, say p, executed line 5 before the other, and so when q executes the loop in lines 7–8, it finds that $\mathit{flag}[p] = 2$ and goes to line 2—a contradiction. To show progress, consider any time t when $\mathit{flag}[\mathit{turn}] = 0$ (such as initially) and there are processes in the trying region. We show that some process eventually enters the critical region after time t, which proves that the algorithm guarantees progress, because after leaving the critical region, a process sets $\mathit{flag}[\mathit{turn}] = 0$. Suppose by way of contradiction that no processes ever enter the critical region. We claim that some process eventually executes the assignment in line 4. Indeed, if not then *turn* never changes, and so let T be its value. There are two cases: either (1) process T is always in the remainder region after time t (it is not trying to enter the critical section), or (2) process T tries to enter the critical section after time t. In case (1), $\mathit{flag}[T] = 0$ forever because a trivial invariant is that processes in the remainder region have their *flag*s set to 0. Therefore, a process in the trying region will execute line 4 and modify *turn*—a contradiction that shows the claim. In case (2), process T will get past the while loop in line 3, and other processes in the trying region will get stuck in the while loop. Therefore T does not find any *flag*s set to 2 in line 8, and enters the critical region—a contradiction that shows the claim. So, some process eventually executes the assignment in line 4 after time t; consider the first time $t' > t$ when that happens. An invariant is that from time t' onward, $\mathit{flag}[\mathit{turn}] \neq 0$: this is because nobody changes its *flag* to 0, and a process p only changes *turn* to p if $\mathit{flag}[p] \neq 0$. Therefore after

```
Shared variables:
  flag[1..n], initially all 0
  turn, initially 1

Code for process p:

  (trying region)

1
2    flag[p] ← 1
3    while turn ≠ p do
4       if flag[turn] = 0 then turn ← p
5    flag[p] ← 2
6
7    for j ← 1 to n do
8       if j ≠ p and flag[j] = 2 then goto 2

  (critical region)

  (exit)

9    flag[p] ← 0

  (remainder region)
```

Figure 10: Classical Dijkstra's mutual exclusion algorithm for model M_1^n.

```
Shared variables:
  flag[1..∞], initially all 0
  turn, initially 1

Code for process p:

  (trying region)

1*   if did not join yet then join()
2    flag[p] ← 1
3    while turn ≠ p do
4       if flag[turn] = 0 then turn ← p
5    flag[p] ← 2
6*   view ← get-view()
7*   for each j ∈ view.set do
8       if j ≠ p and flag[j] = 2 then goto 2

  (critical region)

  (exit)

9    flag[p] ← 0

  (remainder region)
```

Figure 11: Modified Dijkstra's mutual exclusion algorithm for model M_3, using group membership (asterisks indicate a difference from the previous algorithm).

t', the assignment in line 4 is executed only finitely many times, and the last time happens at some time $t'' > t'$ by some process p. Then forever after t'', $turn = p$. Any process q that reaches line 5 eventually also reaches line 8 and then goes back to line 2, because by assumption nobody enters the critical section. After doing so, it sets $flag[q] = 1$ and, if $q \neq p$, q loops forever in lines 3–4 with $flag[q] = 1$. Therefore, forever after some time $t''' > t''$, all processes but p are looping in lines 3–4 with $flag[q] = 1$. After time t''', p cannot be stuck in the while loop of line 3, and p will not find $flag[j] = 2$ in line 8, so p eventually enters the critical section—a contradiction.

7.2 The bakery algorithm

Another popular mutual exclusion algorithm is the bakery algorithm [Lam74], shown in Figure 12, whose nice feature is that it avoids process starvation, unlike Dijkstra's algorithm. Each process p first gets a numbered ticket greater than the one of other processes (line 4). Then p waits until the processes with smaller tickets are finished (lines 6–8), and then p enters the critical region. It is possible that two processes get the same number; in that case, they use their process identifiers to break ties.

This algorithm can be easily modified to work in model M_3 by using group membership, as shown in Figure 13. A similar flavor of modification to the Bakery algorithm was proposed by Afek et al [AST99] to produce an adaptive algorithm for M_1^n. At the beginning of the algorithm, processes call *join*() (if they have not done so already), and then *get-view* to get a list of potential competitors for the critical region, which is used as the set of processes to check in the loop of line 6. In line 8, the check takes into consideration not just the ticket number, but also the view's version number (we use lexicographic ordering for comparing tuples): intuitively, earlier views have priority over later views. The reason for that is that two processes p and q may get the same ticket

```
Shared variables:
   number[1..n], initially all 0
   choosing[1..n], initially all false

Code for process p:

   (trying region)

1
2
3     choosing[p] ← true
4     number[p] ← 1 + max_{q∈{1..n}} number[q]
5     choosing[p] ← false
6     for each q ≠ p ∈ {1..n} do
7        wait until choosing[q] = false
8        wait until number[q] = 0 or
               (number[p], p) < (number[q], q)

   (critical region)

   (exit)

9     number[p] ← 0

   (remainder region)
```

Figure 12: Classical Bakery algorithm for model M_1^n.

```
Shared variables:
   number[1..∞], initially all 0
   choosing[1..∞], initially all false
   View[1..∞], initially all (−1, {})

Code for process p:

   (trying region)

1*    if p has not previously executed join then join()
2*    View[p] ← get-view()
3     choosing[p] ← true
4     number[p] ← 1 + max_{q∈View[p].set} number[q]
5     choosing[p] ← false
6*    for each q ≠ p ∈ View[p].set do
7        wait until choosing[q] = false
8*       wait until number[q] = 0 or
            (View[p].version, number[p], p) < (View[q].version, number[q], q)

   (critical region)

   (exit)

9     number[p] ← 0

   (remainder region)
```

Figure 13: Modified Bakery algorithm for model M_3 using group membership (asterisks indicate a difference from the previous algorithm).

number, but p may not know about the existence of q if q joins later. In this case, p will have a smaller view version than q, so q lets p go first.

To show that the critical section is accessed by at most one process, suppose by way of contradiction that two different processes p and q could be in the critical region at the same time t. For $r \in \{p, q\}$, let v_r and n_r be the values of $View[r].version$ and $number[r]$ at time t, respectively. We claim that $(v_p, n_p, p) < (v_q, n_q, q)$, which establishes a contradiction because we can just switch the names of p and q (which are completely symmetric in the claim) and get the opposite inequality. To show the claim, there are two cases: (a) $q \notin View[p].set$ or (b) $q \in View[p].set$. In case (a), q executes *join* after p executes *get-view*, and so $v_p < v_q$, which shows the claim. In case (b), consider the time $t' < t$ when p finished line 8 for q. Then at time t' either (1) q had already assigned a value in line 2, or (2) q had not. In case (1), we have $v_p \le v_q$, else p gets stuck in line 8. In case (2), let v'_q be the value of $View[q].version$ at time t'. We have $v_p \le v'_q$ and $v'_q \le v_q$ since $View[q].version$ is monotonically nondecreasing. Thus, in both cases (1) and (2) we have $v_p \le v_q$. If $v_p < v_q$ then the claim follows. Now assume $v_p = v_q$, and consider the time $t'' < t$ when p finished line 7 for q. Then at time t'' either (1) q had already finished line 5, or (2) q had not executed line 3. In case (1), the guard in line 8 ensures that $(v_p, n_p, p) < (v_q, n_q, q)$, which shows the claim. In case (2), when q executes line 4, it picks $n_q > n_p$ because (a) $v_p = v_q$, and so *get-view* returns the same value for p and q, and so the set returned includes both p and q, and (b) q sees n_p when it takes the maximum in line 4. Since $v_p = v_q$ and $n_q > n_p$, it follows that $(v_p, n_p, p) < (v_q, n_q, q)$, and the claim follows.

To see that the algorithm guarantees progress, suppose by way of contradiction that the algorithm blocks forever while there are some processes in the trying region. Note that the values of $View[p].version$, $number[p]$ and p are all integers bounded below by -1. Therefore, in the whole run, some process p that gets blocked forever in the trying region has a smallest triple

$(View[p].version, number[p], p)$. Then, any process q with a smaller triple does not get blocked in the trying region, so q eventually has $number[q] = 0$. Therefore, p eventually finishes the for loop in lines 6–8 and enters the critical region—a contradiction to the definition of p.

To see that the algorithm avoids starvation, suppose by way of contradiction that some process p in the trying region starves. Let t be the time when p executes past line 5. Any process r that starts line 1 after time t will pick a pair $(View[q].version, number[r])$ bigger than p's. Therefore, by the guard in line 8, since p gets stuck, all such processes r get stuck too. The number of processes that have started line 1 before time t is finite, and so eventually nobody is in the critical region, but the processes in the trying section are stuck—a contradicting the fact that the algorithm guarantees progress.

Further reading and bibliographic notes

The first algorithm for mutual exclusion for model M_3 was proposed by Merritt and Taubenfeld [MT00]. Our modification of the Bakery algorithm to use a dynamic set of processes is similar to the one proposed by Afek et al [AST99] to get an adaptive algorithm for M_1^n.

8 Building the shared memory

Is it possible to build a memory shared consistently among infinitely many processes? If not then perhaps models M_2 and M_3 make little sense after all. We now consider this issue.

Traditional approaches that do not work well with infinitely many processes.

1. A series of papers has shown how to start with registers with very weak consistency and sharing, namely (single-writer single-reader) safe registers [Lam86b], and progressively construct better registers in a wait-free manner, all the way to atomic multi-writer multi-reader registers. Therefore, to get atomic registers in M_1^n, all we need is to implement the simpler safe registers and apply these constructions. This works for model M_1^n, but unfortunately not for models M_2 or M_3. In fact, in these models it is easy to show that there are no wait-free implementations of (multi-writer multi-reader) atomic registers from either (a) registers that are readable by only finitely many processes or (b) registers that are writable by only finitely many processes. So this approach does not work.

2. A traditional way to get shared memory is to link together processes and physical memory through an interconnect, and treat the memory as a component that responds to read and write requests, like in a multiprocessor system. Although a multiprocessor system is unlikely to scale to a large number of processors, this idea is also applicable to loosely-coupled systems. In fact, the machine rooms of large corporations often have so-called *network attached disks* that accept read and write requests; here, disks serve as a shared memory. These systems could scale to a very large number of hosts, but there is a problem: because a disk can fail, it does not exactly implement the standard atomic register, which never fails. Can we implement registers that do not fail from fail-prone ones? Yes, in model M_1^n [AGMT95, JCT98], and also in models M_2 or M_3 [AEG03a]. However, in models M_2 or M_3, any such implementation requires infinitely many fail-prone registers to implement as little as one fail-free register, even if at most one of the fail-prone registers may fail [AEG03a]. So this approach is inherently impracticable.

An approach that works.

Another way to implement shared memory is to simulate it over a message passing system [ABND95, LS97, ES00, LS02]. This approach can successfully implement models M_2 and M_3. To get a flavor for how it works, consider an asynchronous message-passing system with $2f + 1$

Code for client process p:

```
   Global variable: seq, initially 0

   To WRITE(v) do
1     seq ← seq + 1
2     send (GET-TS, seq) to all server processes
3     wait until received (TS, seq, ∗) from f + 1 processes
4     t ← largest ts in received (TS, seq, ts) messages
5     t.seq ← t.seq + 1; t.process ← p
6     send (WR, seq, v, t) to all server processes
7     wait until received (WR-OK, seq) from f + 1 servers

   To READ do
8     seq ← seq + 1
9     send (RD, seq) to all server processes
10    wait until received (VAL, seq, ∗, ∗) from f + 1 servers
11    t ← largest ts in received (VAL, seq, ts, ∗) messages
12    v ← value such that received message (VAL, seq, t, v)
13    while did not receive f + 1 messages of form
14          (VAL, seq, x, v) with x ≥ t do
15       send (RD, seq) to all server processes
16    return v
```

Code for server process q:

```
   Global variables:
      ts, initially ⟨0, 0⟩
      v, initially 0

   upon receive (GET-TS, seq) from p do
17    send (TS, seq, ts) to p

   upon receive (WR, seq, w, t) from p do
18    if t > ts then
19       send (PROPAGATE, w, t) to all server processes
20       wait until received (PROPAGATE, w, t) from f + 1 servers
21    send (WR-OK, seq) to p

   upon receive (RD, seq) from p do
22    send (VAL, seq, ts, v)

   upon receive (PROPAGATE, w, t) do
23    if have not yet sent (PROPAGATE, w, t)
24    then send (PROPAGATE, w, t) to all server processes
25    if t > ts and received (PROPAGATE, w, t) from f + 1 processes
26    then ts ← t; v ← w
```

Figure 14: Implementation of (multi-writer multi-reader) atomic registers for model M_3 over a message-passing system with $2f + 1$ server processes, up to f of which may fail

server processes and possibly infinitely many client processes. The idea is that the server processes implement the shared memory, and (client) processes communicate with the servers to read and write. Up to f of the servers may crash, and any number of clients may crash. Figure 14 shows a register implementation in this system.

Each server process keeps its latest known value for the register and an associated timestamp. To write v to a register, a (client) process p asks all server processes to send their timestamp (line 2), waits for a majority of responses (line 3), picks the largest timestamp (line 4), increments it (line 5), and sends v and the new timestamp to all servers (line 6). When a server process q receives v with the timestamp, it propagates this information to all servers using uniform reliable broadcast [HT94]. More precisely, q sends v and the timestamp to all server processes in a PROPAGATE message (line 19), and waits for $f + 1$ acknowledgements (line 20). Then q accepts v and the timestamp as its new value and timestamp (line 26) (if process q receives a PROPAGATE message while waiting in line 20, q first executes the code in lines 23–26 before continuing to line 21). Then q sends back an acknowledgement to p in an WR-OK message (line 21). When p receives a majority of acknowledgements, it returns (line 7). When a server process q' receives a value and timestamp propagated from q, q' itself propagates them to the other servers if q' has not done so already (line 24). When q' sees that $f + 1$ servers have also propagated the information, q accepts v and the timestamp as its new value and timestamps (lines 25–26).

To read, a process asks all servers to send their values with timestamps (line 9), waits for a majority of responses (line 10), picks the value with largest timestamp (lines 11–12), and then waits until a majority of servers have this timestamp (lines 13–15). An exercise for the reader is to show that this last waiting step is required for correctness.

Proof of correctness

In this proof, we sometimes subscript a variable local to a process p by p.

Lemma 13 *At every server process q, ts_q is monotonically nondecreasing.*

PROOF: Clear, since a server only updates ts_q with a larger value (lines 25–26). ■

Lemma 14 *If a correct process p starts* WRITE(v) *then p completes it.*

PROOF: Clearly, p does not get stuck waiting for TS messages. As for WR-OK messages, note that when a correct server q receives p's WR message, it either sends back a WR-OK message, or it sends a PROPAGATE message to all correct servers. Those will echo it back to q, so q will not get stuck waiting for PROPAGATE message, and so q will send a WR-OK message. Therefore, all correct servers send back a WR-OK message to p. Since there are at least $f+1$ such processes, p does not get stuck waiting for WR-OK messages. ■

Lemma 15 *If a correct process p starts* READ *then p completes it.*

PROOF: Clearly p does not get stuck waiting for VAL messages. Now consider the timestamp T chosen in line 11. Then p received a $(\text{VAL}, seq, T, *)$ message. If $T = \langle 0, 0 \rangle$ then clearly p does not get stuck in the while loop, because variable ts at any server is at least as large at T. So assume that $T \neq \langle 0, 0 \rangle$. Therefore, some server q has changed its t_q variable to T. Server q only does so if q receives PROPAGATE messages from $f+1$ servers. Therefore, q received at least one such a message from a correct server. This server causes all correct servers to receive and send the PROPAGATE message (if they have not done so already). Therefore, all correct servers receive at least $f+1$ PROPAGATE messages, and so they all eventually have their ts variable set to a value no smaller than T. Therefore, p does not get stuck in the while loop. ■

Definition 16 *We say that a read operation is* good *if its caller completes it without crashing. We say that a write operation is* good *if either (1) its caller p completes the operation without crashing, or (2) some correct server receives a* PROPAGATE *message with the same timestamp that p chooses during the execution of the write.*

Definition 17 *For a good read or write operations, we say that* ts *is its timestamp if* ts *is the timestamp chosen in line 4 or 11 during the execution of the operation.*

Lemma 18 *If a good write o_1 completes at some time t then at most f servers have their ts variable less than o_1's timestamp at time t.*

PROOF: Let p_1 be the process that executes o_1. Before a server q sends a WR-OK message to p_1, it must have $ts_q \geq t_{p_1}$ where t_{p_1} is the timestamp chosen by p_1. Since p_1 waits for WR-OK messages from $f+1$ servers and the ts variables at the servers are monotonically increasing, the lemma follows. ■

Lemma 19 *If a good read o_1 completes at some time t then at most f servers have their ts variable less than o_1's timestamp at time t.*

PROOF: This is clear from the while loop in the code for READ. ■

We now define a total ordering O of all good read and write operations according to their timestamp. If two such operations have different timestamps, then we order first the one with smaller timestamp. If a write and a read have the same timestamp, then we order the write before the read. If two reads have the same timestamp, then we order first the read that starts first. It is not possible for two writes to have the same timestamp, because the timestamp includes a process identifier to break ties (see line 5).

Lemma 20 *O respects the "precedes" relation on operations.*

PROOF: Indeed, consider two good operations o_1 and o_2 such that o_1 precedes o_2, that is, o_1 completes before o_2 starts. For $i \in \{1, 2\}$, let p_i be the process executing o_i and let t_i be the timestamp of o_i. By Lemma 18 or 19 (depending on whether o_1 is a write or read operation), and by Lemma 13, we have that at most f servers have their ts variable less t_1 when o_2 starts. Operation o_2 gets timestamps from $f + 1$ servers, so it gets at least one timestamp that is at least t_1. Thus, if o_2 is a write operation, $t_2 > t_1$, and so o_2 is ordered after o_1 under O. If o_2 is a read operation, then $t_2 \geq t_1$. If $t_2 > t_1$ then o_2 it is ordered after o_1 under O. If $t_2 = t_1$ then we have two cases: (1) if o_1 is a read then o_2 is ordered after o_1 under O, by the rule for ordering reads with the same timestamp. (2) if o_1 is a write then o_2 is also ordered after o_1 under O, by the rule for ordering a read and a write with the same timestamp. ■

Lemma 21 *O satisfies the sequential specification of registers.*

PROOF: (Sketch) We need to show that a read returns the value of the closest preceding write under $\mathcal{O}$ (if any) or the initial value 0 (if there are no preceding writes). This is clear because the value chosen by a read operation o in line 12 is the value associated with the timestamp of o, and that timestamp is identical to the timestamp of the closest preceding write under $\mathcal{O}$ (or, if there are no preceding writes, then the timestamp is 0 and the read value is 0). ■

We thus get the following

Theorem 22 *Consider a message-passing system with $2f + 1$ server processes where up to f may crash, and an arbitrary number of client processes, any of which may crash. Figure 14 implements a MWMR register shared among the client processes.*

Further reading and bibliographic notes

Our algorithm is inspired by the work by Attiya, Bar-Noy, and Dolev [ABND95], who gave the first simulation of an atomic register over a message passing system. However, Attiya et al only show how to implement single-writer multi-reader registers; this type of register is not very useful in models M_2 or M_3 because it cannot be used to implement multi-writer multi-reader registers. Subsequent papers have shown how to simulate multi-writer multi-reader atomic registers in a system where the set of server processes may dynamically change, by using quorums that vary over time [LS97, ES00, LS02]. Our algorithm is simpler than the ones in those papers because we do not allow the set of servers to change: dealing with a changing set of servers requires more complexity.

Our algorithm embeds an implementation of uniform reliable broadcast due to Hadzilacos and Toueg [HT94]. Roughly speaking, uniform reliable broadcast guarantees that (1) a broadcast message is either delivered by all targets or by none, (2) if the broadcaster does not fail, then the message is delivered by all targets, and (3) a message that is not broadcast is not delivered by anyone.

9 Other work on computing with infinitely many processes

Adaptive algorithms have been extensively studied in model M_1^n (e.g., [MT93, CS94, ADT95, MA95], etc). Roughly speaking, an algorithm is adaptive if its time complexity—measured in number of steps taken by a process—is proportional to the number of participating processes, rather than the total number n of processes that could participate. Most adaptive algorithms designed for M_1^n do not actually make use of n or use n only superficially (e.g., they may declare an array with indices $1..n$). These algorithms work in M_2 with no modifications or only trivial ones (e.g., finite arrays indexed by processes need to be changed to infinite arrays or linked lists). As a result, M_2 inherits many reasonably efficient algorithms for various problems like collect, atomic snapshot, immediate snapshot, renaming, lattice agreement, and mutual exclusion. These algorithms may fail in M_3 by losing liveness when new processes keep arriving, similarly to our algorithm in Figure 4 of Section 3.

Merritt and Taubenfeld [MT00] give various algorithms for election and mutual exclusion for models M_3 and M_3^b, as well as election algorithms for a variant of model M_3^b where some number of processes are known to participate.

An important problem is to determine what *tasks* have a fault-tolerant solution in a given model. Roughly speaking, a task is a multifunction from inputs of processes to allowable outputs of processes. Many important problems, like consensus, atomic commit, and k-set agreement, can be cast as a task. For example, consensus can be considered as the task where all processes must output the same value, which must be one of the input values. Some tasks do not have deterministic fault-tolerant solutions, so it is important to know which ones do. This question was originally answered for model M_1^n by Herlihy and Shavit [HS93, HS99], which gives a full characterization of tasks with t-resilient and wait-free solutions. Gafni and Koutsoupias [GK98, Gaf02] have extended the characterization result for model M_2. What if processes only have finitely many registers? Aguilera, Englert, and Gafni [AEG03b] have shown that some very simple tasks that cannot be solved in M_2. But the general characterization of solvable tasks with finitely many registers in M_2 is an open problem.

Can consensus be solved in a system with infinitely many processes? Even in M_1^n, consensus has no solution that can tolerate just one process crash [FLP85, LAA87], so this impossibility also applies to M_2 and M_3. However, there are *randomized* solutions for M_1^n (the first ones are in [Ben83, CIL94]), so perhaps there are also randomized solutions for M_2 and M_3? Aspnes, Shah, and Shah [ASS02] show that the answer is positive, by giving a randomized wait-free algorithm for M_3. The algorithm is based on a *weak shared coin*, which is a mechanism for processes to get a random bit that has a reasonable probability of being 0 at all processes, and a reasonable probability of being 1 at all processes, where reasonable means bounded below by a non-zero constant. Aspnes et al claim that such a coin can be used in a modification of the consensus algorithm of Chandra [Cha96], to obtain a consensus algorithm for model M_3. Finally, they give a universal construction [Her91] for model M_3, by plugging their consensus algorithm and a collect algorithm into the traditional universal construction for M_1^n.

Chockler and Malkhi [CM02] present a variant of the Disk Paxos protocol [GL00] for model M_3, which solves a weak version of consensus that sometimes aborts. To do so, they use a *ranked register* [BDFG03], which is a type of read-modify-write register with the nice property that a fail-free ranked register can be implemented from a set of fail-prone ones.

Merritt and Taubenfeld [MT03] show that (1) in M_3, for every $t \geq 1$, t-resilient consensus is possible using t-resilient consensus objects accessible by $t+1$ processes, and (2) in M_3^{finite}, wait-free consensus is possible using wait-free consensus objects accessible only by finitely many processes.

It is an open question whether (2) also holds in M_3.

What is the problem solvability situation for M_3, M_3^b, M_3^{finite}? Gafni et al [GMT01] have shown that (1) there is a problem solvable in M_3^b but not in M_3^{b+1}, (2) there is a problem solvable in M_3^b but not in M_3^{finite}, and (3) there is a problem solvable in M_3^{finite} but not in M_3. Therefore, in terms of problem solvability, M_3 is strictly weaker than M_3^{finite}, which in turn is strictly weaker than M_3^b for any b; and M_3^{b+1} is strictly weaker then M_3^b. However, general deterministic solvability characterizations for models M_3, M_3^b, M_3^{finite} are still open problems.

References

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merrit, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 292–312, November 1992.

[ADT95] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, May 1995.

[AEG03a] Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. On using network attached disks as shared memory. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 315–324, July 2003.

[AEG03b] Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. Uniform solvability with a finite number of mwmr registers. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 16–29, October 2003.

[AF00] Hagit Attiya and Arie Fouren. Polynomial and adaptive long-lived $(2k-1)$-renaming. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 149–163, October 2000.

[AF03] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, July 2003.

[AGMT95] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, November 1995.

[ASS02] 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, May 2002.

[AST99] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, pages 262–272, October 1999.

[BDFG03] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Deconstructing paxos. *ACM SIGACT News*, 34(1):47–67, March 2003.

[BDGB94] Özalp Babaoğlu, Renzo Davoli, Luigi-Alberto Giachini, and Mary Gray Baker. *RELACS: a communications infrastructure for constructing reliable applications in large-scale distributed systems*. BROADCAST Project deliverable report, 1994. Department of Computing Science, University of Newcastle upon Tyne, UK.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communication of the ACM*, 9(12):36–53, December 1993.

[Cha96] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 166–175, May 1996.

[CIL94] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, August 1994.

[CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[CM02] Gregory Chockler and Dahlia Malkhi. Active disk paxos. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 78–87, July 2002.

[Cri87] Flaviu Cristian. Issues in the design of highly available computing services. In *Annual Symposium of the Canadian Information Processing Soc iety*, pages 9–16, July 1987. Also IBM Research Report RJ5856, July 1987.

[CS94] Manhoi Choy and Ambuj K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, August 1994.

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

[EMS95] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, June 1995.

[ES00] Burkhard Englert and Alexander A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 454–463, April 2000.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gaf02] Eli Gafni. A simple algorithmic characterization of uniform solvability. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 228–237, November 2002.

[GK98] Eli Gafni and Elias Koutsoupias, 1998. On uniform protocols. http://www.cs.ucla.edu/˜ eli/eli.html.

[GL00] Eli Gafni and Leslie Lamport. Disk paxos. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 330–344, October 2000.

[GMT01] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20st ACM Symposium on Principles of Distributed Computing*, pages 161–169, August 2001.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[HS80] Maurice Herlihy and Nir Shavit. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 82–93, April 1980.

[HS93] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 111–120, May 1993.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

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

[JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, May 1998.

[JS85] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pages 13–22, August 1985.

[KT91] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222–230, Arlington, TX, May 1991.

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

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam86a] Leslie Lamport. On interprocess communication; part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[Lam86b] Leslie Lamport. On interprocess communication; part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[LS97] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 272–281, June 1997.

[LS02] Nancy Lynch and Alexander A. Shvartsman. RAMBO: a reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing*, pages 173–190, October 2002.

[MA95] Mark Moir and James Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[MT93] Michael Merritt and Gadi Taubenfeld. Speeding lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45(3):137–142, March 1993.

[MT00] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 164–178, October 2000.

[MT03] Michael Merritt and Gadi Taubenfeld. Resilient consensus for infinitely many processes. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 1–15, October 2003.

[vRBC+93] Robbert van Renesse, Kenneth P. Birman, Robert Cooper, Bradford Glade, and Patrick Stephenson. The Horus system. In Kenneth P. Birman and Robbert van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, Los Alamitos, CA, 1993.