At-Most-Once Semantics in Asynchronous Shared Memory*

Sotirios Kentros, Aggelos Kiayias,
Nicolas Nicolaou, and Alexander A. Shvartsman

Computer Science and Engineering, University of Connecticut, Storrs, USA
{skentros,aggelos,nicolas,aas}@engr.uconn.edu

Abstract. At-most-once semantics is one of the standard models for object access in decentralized systems. Accessing an object, such as altering the state of the object by means of direct access, method invocation, or remote procedure call, with at-most-once semantics guarantees that the access is not repeated more-than-once, enabling one to reason about the safety properties of the object. This paper investigates implementations of at-most-once access semantics in a model where a set of such actions is to be performed by a set of failure-prone, asynchronous shared-memory processes. We introduce a definition of the *At-Most-Once* problem for performing a set of n jobs using m processors and we introduce a notion of efficiency for such protocols, called *effectiveness*, used to classify algorithms. Effectiveness measures the number of jobs safely completed by an implementation, as a function of the overall number of jobs n, the number of participating processes m, and the number of process crashes f in the presence of an adversary. We prove a lower bound of $n-f$ on the effectiveness of any algorithm. We then prove that this lower bound can be matched in the two process setting by presenting two algorithms that offer a tradeoff between time and space complexity. Finally, we generalize our two-process solution in the multi-process setting with a hierarchical algorithm that achieves effectiveness of $n-\log m \cdot o(n)$, coming reasonably close, asymptotically, to the corresponding lower bound.

1 Introduction

The *At-Most-Once* semantic for object invocation ensures that an operation accessing and altering the state of an object is performed no more than once. This semantic is among the standard semantics for remote procedure calls (RPC) and method invocations and it provides important means for reasoning about the safety of critical applications. Uniprocessor systems may trivially provide solutions for at-most-once semantics by implementing a central schedule for operations. The problem becomes very challenging for autonomous processes in a shared-memory system with concurrent invocations on multiple objects. Although At-Most-Once semantics have been thoroughly studied in the context of

* This work is supported in part by the NSF Awards 0702670 and 0831306. The work of the first author is supported in part by the State Scholarships Foundation - Greece.

At-Most-Once message delivery [4, 13, 16, 23] and At-Most-Once process invocation for RPC [2, 14, 15, 16, 21], finding effective solutions for asynchronous shared-memory multiprocessors, in terms of how many at-most-once invocations can be performed by the cooperating processes, is largely an open problem. This paper brings the attention to the At-Most-Once problem in multiprocessor settings. We believe that solving this problem using only atomic memory, and without specialized hardware support, such as conditional writing, will provide a useful tool in reasoning about the safety properties of applications developed for a variety of multiprocessor systems, including those not supporting bus-interlocking instructions and multi-core systems.

We explore At-Most-Once implementations for asynchronous shared-memory processors that are prone to crashes. We model accesses to objects as tasks, where the correctness demands that each task is performed at most once. Any processor is able to perform any task and we aim to maximize the total number of performed tasks while preserving the at-most-once semantics. We define the notion of *effectiveness* used to assess the efficiency of solutions for the problem. Effectiveness measures the number of tasks performed using at-most-once semantics as a function of the number of tasks, the number of processors, and the number of crashes. We provide tight lower bounds for effectiveness, and we introduce three algorithms that solve the problem. The first two are formulated for two processors. The third algorithm is stated for an arbitrary number of processors and it uses a two-processor solution as a building block. We present rigorous analyses of the algorithms' work, space complexity, and effectiveness.

Related Work: A wide range of works study At-Most-Once semantics in a variety of settings. At-Most-Once message delivery [23, 16, 13, 4] and At-Most-Once semantics for RPC [2, 14, 15, 21, 16], are two areas that have attracted a lot of attention. Here the problem studied is different from the one we consider. Both in At-Most-Once message delivery and RPCs, we have two entities (sender/client and receiver/server) that communicate by message passing. Any entity may fail and recover and messages may be delayed or lost. In the first case one wants to guarantee that duplicate messages will not be accepted by the receiver, while in the case of RPCs, one wants to guarantee that the procedure called in the remote server will be invoked at-most-once [22].

Di Crescenzo and Kiayias in [5] demonstrate the use of the semantic in message passing systems for the sake of security. Driven by the fundamental security requirements of *one-time pad* encryption, the authors partition a common random pad among multiple communicating parties. Perfect security could be achieved only if every piece of the pad is used at most once. The authors show how the parties maintain security while maximizing efficiency by applying At-Most-Once semantics on pad expenditure.

One can also relate the At-Most-Once problem to the consensus problem [18, 9, 6, 12]; here one can view consensus as an at-most-once distributed decision.

Another related problem is the Write-All problem for the shared memory model [10, 1, 8, 19, 11].First presented by Kanellakis and Shvartsman [10], the *Write-All* problem is concerned with performing each task at-least-once. We note

that solutions to *Write-All* may be adapted to solve At-Most-Once, provided safeguards are in place to prevent more-than-once invocations.

Finally we note that the At-Most-Once problem becomes much simpler when shared-memory is supplemented by some type of read-modify-write operations. For example, one can associate a *test-and-set* bit with each task, ensuring that the task is assigned to the only processor that successfully sets the shared bit An efficient implementation can then be easily obtained from a Write-All solution, such as [1, 8, 11, 20]. Thus, in this paper we deal only with the more challenging setting where algorithms use atomic read/write registers.

Contributions: Our goal is to explore the feasibility and efficiency of solutions that satisfy the At-Most-Once semantic in the shared-memory model with asynchronous processors prone to crash failures. The At-Most-Once Problem is formulated for m processors and n jobs, where any processor can perform any job, provided that no job is performed more-than-once. Note that in such a setting it is impossible to distinguish between a slow and a crashed processor, consequently it is impossible to determine whether a processor delays while performing a job or if it crashes before performing the job. This means that generally some jobs may never be performed. Our contributions are as follows.

(1) We define the *At-Most-Once* problem and the correctness properties to be satisfied by any solution. We introduce a complexity measure we call *effectiveness*. This measure describes the number of jobs completed (at-most-once) by an implementation, as a function of the overall number of jobs n, the number of processors m, and the number of processor crashes f. (Section 2.)

(2) We present a lower bound for the effectiveness of the at-most-once implementations. In particular, we prove that no At-Most-Once solution may achieve effectiveness better than $n - f$. (Section 3.)

(3) We provide two algorithms that solve the At-Most-Once problem for 2 processors. The algorithms use a collision-avoidance approach. The importance of these algorithms is twofold: *a*) they can be used as building blocks to construct general implementations for larger number of processors, and *b*) they achieve optimal effectiveness. The algorithms differ substantially in their space requirements and work complexity, demonstrating the trade-offs between efficiency and space. We analyse work, space, and effectiveness. (Section 4.)

(4) Finally we present a multi-processor algorithm, that employs one of our two-processor algorithms as a building block. We prove the correctness of the algorithm, and we perform rigorous analysis of its effectiveness of $n - \log m \cdot o(n)$, and its work and space complexity. (Section 5.) The algorithms in this work are motivated by the *Write-All* algorithms from [3, 8], although the problem itself and the correctness criteria are quite different. Our work can be viewed as complementary to [5] that considers a similar problem in message-passing models. Here we use a shared-memory model in a deterministic setting.

Because of lack of space we omit some of the proofs in this manuscript. We encourage the reader to contact the authors for the detailed proofs.

2 Model, Definitions, and Efficiency

We define our model, the At-Most-One problem, and measures of efficiency.

2.1 Model and Adversary

We model a multi-processor as m asynchronous, crash-prone processes with unique identifiers from some set $\mathcal{P}$. Shared memory is modeled as a collection of atomic memory cells, where the number of bits in each cell is explicitly defined. We use the *Input/Output Automata* formalism [18, 17] to specify and reason about algorithms; specifically, we use the *asynchronous shared memory automaton* formalization [18, 7]. Each process p is defined in terms of its states $states_p$ and its actions $acts_p$, where each action is of the type *input*, *output*, or *internal*. A subset $start_p \subseteq states_p$ contains all the start states of p. Each shared variable x takes values from a set V_x, among which there is $init_x$, the initial value of x.

We model an algorithm A as a composition of the automata for each process p. Automaton A consists of a set of states $states(A)$, where each state s contains a state $s_p \in states_p$ for each p, and a value $v \in V_x$ for each shared variable x. Start states $start(A)$ is a subset of $states(A)$, where each state contains a $start_p$ for each p and an $init_x$ for each x. The actions of A, $acts(A)$ consists of actions $\pi \in acts_p$ for each process p. A transition is the modification of the state as a result of an action and is represented by a triple (s, π, s'), where $s, s' \in states(A)$ and $\pi \in acts(A)$. The set of all transitions is denoted by $trans(A)$. Each action in $acts(A)$ is performed by a process, thus for any transition (s, π, s'), s and s' may differ only with respect to the state s_p of process p that invoked π and potentially the value of the shared variable that p interacts with during π. We also use triples $(\{vars_s\}, \pi, \{vars_{s'}\})$, where $vars_s$ and $vars_{s'}$ are subsets of variables in s and s' respectively, as a shorthand to describe transitions without having to specify s and s' completely; here $vars_s$ and $vars_{s'}$ contain only the variables whose value changes as the result of π, plus possibly some other variables of interest.

We say that states s and t in $states(A)$ are *indistinguishable* to process p if: 1) $s_p = t_p$, and 2) the values of all shared variables are the same in s and t. Now, if states s and t are indistinguishable to p and $(s, \pi, s') \in trans(A)$ for $\pi \in acts_p$, then $(t, \pi, t') \in trans(A)$, and s' and t' are also indistinguishable to p.

An *execution* fragment of A is either a finite sequence, s_0,π_1,s_1, ...,π_r,s_r, or an infinite sequence, s_0,π_1,s_1, ...,π_r,s_r,..., of alternating states and actions, where $(s_k, \pi_{k+1}, s_{k+1}) \in trans(A)$ for any $k \geq 0$. If $s_0 \in start(A)$, then the sequence is called an *execution*. The set of executions of A is *execs(A)*. We say that execution α is *fair*, if α is finite and its last state is a state of A where no locally controlled action is enabled, or α is infinite and every locally controlled action $\pi \in acts(A)$ is performed infinitely many times or there are infinitely many states in α where π is disabled. The set of fair executions is $fairexecs(A)$. An execution fragment α' *extends* a finite execution fragment α of A, if α' begins with the last state of α. We let $\alpha \cdot \alpha'$ stand for the execution fragment resulting from concatenating α and α' and removing the (duplicated) first state of α'.

We model process crashes by action stop_p in $acts(A)$ for each process p. If stop_p appears in an execution α then no actions $\pi \in acts_p$ appear in α thereafter.

We then say that process p *crashed*. Actions stop_p arrive from some unspecified external environment, called *adversary*. In this work we consider an *omniscient*, *on-line adversary* [10] that has complete knowledge of the algorithm. The adversary controls asynchrony and crashes. We allow up to $f < m$ crashes. We denote by $fairexecs_f(A)$ all fair executions of A with at most f crashes.

2.2 At-Most-Once Problem, Effectiveness and Complexity

We consider algorithms that perform a set of tasks, called *jobs*. Let A be an algorithm specified for m processes with ids from set $\mathcal{P} = [0 \dots m-1]$, and with jobs with unique ids from set $\mathcal{J} = [0 \dots n-1]$. We assume that there are at least as many jobs as there are processes, i.e., $n \geq m$. We model the performance of job j by process p by means of action $\mathsf{do}_{p,j}$. For a sequence β, we let $len(\beta)$ denote its length, and we let $\beta|_\pi$ denote the sequence of elements π occurring in β. Then for an execution α, $len\left(\alpha|_{\mathsf{do}_{p,j}}\right)$ is the number of times process p performs job j. Now we define the number of jobs performed in an execution.

Definition 1. *For execution α we denote by $J_\alpha = \{j \in \mathcal{J} \mid \mathsf{do}_{p,j}$ occurs in α for some $p \in \mathcal{P}\}$. The total number of jobs performed in α is defined to be $Do(\alpha) = |J_\alpha|$.*

We next define the *at-most-once* problem.

Definition 2. *Algorithm A solves the At-Most-Once problem if for each execution α of A we have $\forall j \in \mathcal{J} : \sum_{p \in \mathcal{P}} len\left(\alpha|_{\mathsf{do}_{p,j}}\right) \leq 1$. We call any such execution α an AO-execution (at-most-once execution).*

Measures of Efficiency. We analyze our algorithms in terms of three complexity measures: *effectiveness*, *work*, and *space*. Effectiveness counts the number of jobs performed by an algorithm in the worst case.

Definition 3. *The **effectiveness** of algorithm A is: $E_A(n,m,f) = \min_{\alpha \in fairexecs_f(A)}(Do(\alpha))$, where m is the number of processes, n is the number of jobs, and f is the number of crashes.*

A trivial algorithm can solve the At-Most-Once problem by splitting the n jobs in groups of size $\frac{n}{m}$ and assigning one group to each process. Such a solution has effectiveness $E(n,m,f) = (m-f) \cdot \frac{n}{m}$ (consider an execution where f processes fail at the beginning of the execution). Thus our goal is to construct algorithms that achieve higher effectiveness.

Work complexity measures the efficiency of an algorithm in terms of the total number of memory accesses.

Definition 4. *The **work** of algorithm A, denoted by W_A, is the worst case total number of bits accessed in all memory reads and writes in any execution of A.*

Space complexity measures the memory space used by the algorithm.

Definition 5. *The **space** of algorithm A is the total number of bits in shared and internal variables used by the processes of A.*

3 Lower Bound

We show that any algorithm that solves the at-most-once problem in the presence of up to f crashes has effectiveness $E \leq n - f$. While the proof is subtle, the result itself is intuitive, based on the observation that one cannot distinguish a crashed process from a slow one. If an algorithm assigns job j to process p, and the process crashes, the algorithm is unable to revoke the job and assign it to another process, since process p may simply be slow and it may ultimately perform job j, violating at-most-once semantics.

Recall that in our setting we have at least as many jobs as processes ($n \geq m > f$). (The case where $n \leq m$ is less interesting and for this reason is not presented in this paper.) For our proofs we consider only the algorithms that satisfy Condition 1 below requiring that the algorithm is able to perform at least one job. Also let us denote by $F_\alpha = \{p \mid \mathsf{stop}_p \text{ occurs in } \alpha\}$ the set of crashed processes in execution α.

Condition 1. *For all infinite executions α of A, $Do(\alpha) \geq 1$ and for all finite executions α of A, there exists an execution fragment α', s.t. $\alpha \cdot \alpha' \in execs(A)$ and $Do(\alpha \cdot \alpha') \geq 1$.*

We proceed with a lemma, which shows that one may construct two executions that contain f failures and their states are indistinguishable to all correct processes, for algorithms that solve the at-most-once problem. Moreover we show that exactly f jobs are performed in the first execution, while no jobs are performed in the second one. Then we use these executions to prove the main theorem of this section, which shows that the second execution we construct from the lemma, cannot be extended to perform more than $n - f$ tasks. This implies that the effectiveness of any algorithm that solves the at-most-once problem is at most $n - f$.

Lemma 1. *If algorithm A solves the at-most-once problem in the presence of $f < m$ crashes and Condition 1 holds, then there exist finite executions $\alpha_1, \alpha_2 \in execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = f$, $Do(\alpha_1) = f$, $Do(\alpha_2) = 0$, and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$.*

Proof. We prove the lemma by induction on the number of crashes f.
Base case: First we find execution α s.t. $Do(\alpha) \geq 1$ and $F_\alpha = \emptyset$. Such an execution exists by Condition 1 and the fact that crashes are determined by the adversary. Let us consider the first do event in α. Let $\mathsf{do}_{p,j}$ be that event, and let s_1 and s_2 be the states in α before and after $\mathsf{do}_{p,j}$. Since $\mathsf{do}_{p,j}$ does not change shared memory, s_1 and s_2 differ only in the state of process p and thus are indistinguishable for all processes in $\mathcal{P} - \{p\}$. Let $\alpha' = \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2)$ be the prefix of α up to event $\mathsf{do}_{p,j}$. Clearly $\alpha' \in execs(A)$. We construct the executions $\alpha_1 = \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2, \mathsf{stop}_p, s_2')$ and $\alpha_2 = \alpha_0 \cdot (s_1, \mathsf{stop}_p, s_1')$. These executions are finite, and since the crashes are controlled by the adversary $\alpha_1, \alpha_2 \in execs(A)$. Moreover $F_{\alpha_1} = F_{\alpha_2} = \{p\}$ and $Do(\alpha_1) = 1$, $Do(\alpha_2) = 0$. Since stop_p affects only the state of p, s_1, s_1', s_2, s_2' are indistinguishable for all processes in $\mathcal{P} - \{p\}$.
Inductive step: For $k < f$ assume that there exist finite executions $\alpha_1, \alpha_2 \in$

$execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = k$, $Do(\alpha_1) = k$, $Do(\alpha_2) = 0$ and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$. We next construct the needed executions for $k+1$ failures.

We first take α_2. From Condition 1 there exists execution fragment α that has no crashes s.t. $\alpha_2 \cdot \alpha \in execs(A)$ and $Do(\alpha_2 \cdot \alpha) \geq 1$. Since $Do(\alpha_2) = 0$ only α has do events. Moreover since $\alpha_2 \cdot \alpha \in execs(A)$, α has only actions from processes in $\mathcal{P} - F_{\alpha_2}$. Let $\mathsf{do}_{p,j}$ be the first do event in α, where $p \in \mathcal{P} - F_{\alpha_2}$ and $j \in \mathcal{J}$, and let s_1, s_2 be the states in α before and after $\mathsf{do}_{p,j}$. Clearly s_1 and s_2 are indistinguishable for all processes in $\mathcal{P} - \{p\}$. Let us consider the prefix of $\alpha_2 \cdot \alpha$ up to event $\mathsf{do}_{p,j}$ and let us denote this as $\alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2)$. We have that $\alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{do}_{p,j}, s_2) \in execs(A)$.

Note that since the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_2}$, and α_0 contains only actions from process in $\mathcal{P} - F_{\alpha_2}$, the actions of the execution fragment α_0 can extend execution α_1 leading to a state s_3 that is indistinguishable for all processes in $\mathcal{P} - F_{\alpha_2}$ from state s_1. This means that there exists execution fragment α_0' that has the same sequence of actions with α_0, s.t. $\alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4) \in execs(A)$ and s_1, s_2, s_3, s_4 are indistinguishable for all processes in $\mathcal{P} - (F_{\alpha_1} \cup \{p\})$. Since $\alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4) \in execs(A)$, it must hold that $j \notin J_{\alpha_1}$.

We construct the executions $\alpha_2' = \alpha_2 \cdot \alpha_0 \cdot (s_1, \mathsf{stop}_p, s_1')$ and $\alpha_1' = \alpha_1 \cdot \alpha_0' \cdot (s_3, \mathsf{do}_{p,j}, s_4, \mathsf{stop}_p, s_4')$. We have that $\alpha_1', \alpha_2' \in execs(A)$, $F_{\alpha_1'} = F_{\alpha_2'} = F_{\alpha_1} \cup \{p\}$, $|F_{\alpha_1'}| = k+1$, $Do(\alpha_1') = k+1$, $Do(\alpha_2') = 0$, states s_1', s_4' are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1'}$.

Theorem 1. *If algorithm A solves the at-most-once problem in the presence of $f < m$ crashes, then there exists an execution $\alpha \in execs(A)$, s.t. either α is infinite and $Do(\alpha) \leq n - f$, or α is finite, and there exists no execution fragment α', s.t. $\alpha \cdot \alpha' \in execs(A)$ and $Do(\alpha \cdot \alpha') > n - f$.*

Proof. By contradiction. Assume the theorem to be false, with Condition 1 holding. Thus from Lemma 1 we can construct finite executions $\alpha_1, \alpha_2 \in execs(A)$, s.t. $F_{\alpha_1} = F_{\alpha_2}$, $|F_{\alpha_1}| = |F_{\alpha_2}| = f$, $Do(\alpha_1) = f$, $Do(\alpha_2) = 0$ and the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$. Also from the assumption, there exists execution fragment α' s.t. $\alpha_2 \cdot \alpha' \in execs(A)$ and $Do(\alpha_2 \cdot \alpha') > n - f$. Since $Do(\alpha_2) = 0$, it must be that $Do(\alpha') > n - f$. Clearly α' has only actions for processes in $\mathcal{P} - F_{\alpha_2} = \mathcal{P} - F_{\alpha_1}$. Because the final states of α_1 and α_2 are indistinguishable for all processes in $\mathcal{P} - F_{\alpha_1}$ the sequence of actions in α' can extend α_1 as well. This means that there exists execution fragment α'' that has exactly the same actions as α' s.t. $Do(\alpha'') > n - f$ and $\alpha_1 \cdot \alpha'' \in execs(A)$. But $Do(\alpha_1) = f$ and $J_{\alpha_1}, J_{\alpha''} \subseteq \mathcal{J}$. Since $n = |\mathcal{J}|$ it follows by the pigeonhole principle that $J_{\alpha_1} \cap J_{\alpha''} \neq \emptyset$ and thus $\alpha_1 \cdot \alpha''$ is not an AO-execution, a contradiction.

The main result follows as a corollary to the theorem.

Corollary 1. *For all algorithms A that solve the at-most-once problem with m processes and $n \geq m$ jobs in the presence of $f < m$ crashes it holds that $E_A(n, m, f) \leq n - f$.*

4 Two Process Algorithms for At-Most-Once Problem

We present algorithms for the at-most-once problem that use a collision-avoidance approach. First we give 2-process algorithms: $\text{AO}_{2,n}$ that uses n 1-bit shared memory variables, and $\text{AO}'_{2,n}$ that uses two shared memory variables of $\log n$ bits, thus achieving lower space complexity. Both algorithms achieve optimal effectiveness. The two-process algorithms can be used as building blocks to construct algorithms for larger numbers of processes. Here we use algorithm $\text{AO}_{2,n}$ to construct an m-process algorithm for the at-most-once problem.

4.1 Algorithm $\text{AO}_{2,n}$

The algorithm, given in Fig. 1, solves the at-most-once problem for n jobs, using two processes, numbered 0 and 1, and n 1-bit shared variables. The main idea is to have the processes move towards each other, with process 0 performing jobs in the ascending order, and process 1 in the descending order. The processes avoid a *collision*, i.e., doing a job twice, by adopting a *"look ahead decide for the current"* (LA-DC) approach.

The algorithm uses n shared bit variables $x_0, \ldots, x_{n-1}$ as a bookkeeping mechanism to record progress. Initially all shared variables are set to 0. If process p performs job j using action $\mathsf{do}_{p,j}$, then $status_p$ variable is changed to *set*. This enables action set_p that in turn sets the value of x_j to 1. The process decides whether a job can be performed in action check_p. Using the LA-DC approach, before a process performs job j, it decides that it is safe to do so, by checking the shared variable associated with the next job in its path, that is x_{j+1} for process 0 and x_{j-1} for process 1. If x_{j+1} (resp. x_{j-1}) is 0 then process 0 (resp. 1) proceeds to perform j; otherwise the status of the process is assigned the value *end*, and we say that the process *terminates*. The key idea is that since x_{j+1} (resp. x_{j-1}) is 0 then the competing process 1 (resp. process 0), did not yet perform the task $j+1$ (resp. $j-1$). Collision is avoided since it cannot be performing j.

To show correctness we first prove that if $cur_0 = k$ for some $k > 0$, then all shared variables "before" x_k are set to 1, and respectively that if $cur_1 = k$, then all shared variables "after" x_k are set to 1.

Lemma 2. *For any execution α of $\text{AO}_{2,n}$ and for any state s in α such that $s.cur_0 = k$ and $s.cur_1 = k'$ for $1 \leq k \leq k' \leq n-2$, then for $i \in \{0, ..., k-1\} \cup \{k'+1, ..., n-1\}$, $s.x_i = 1$, and actions $\mathsf{do}_{*,i}$ precede s in α.*

Using Lemma 2 we prove that $\text{AO}_{2,n}$ solves the at-most-once problem.

Theorem 2. *Algorithm $\text{AO}_{2,n}$ solves the at-most-once problem.*

4.2 Algorithm $\text{AO}'_{2,n}$

This algorithm, also uses the LA-DC idea. The difference is that we use two integer shared variables, x_{left} and x_{right}, each of $\log n$ bits, that serve as pointers

Shared Variables: $\mathcal{X} = \{x_0, \ldots, x_{n-1}\}$, boolean, initially all 0

Signature:

Input:	Output:	Internal:
stop_p, $p \in \{0,1\}$	$\mathsf{do}_{p,j}$, $p \in \{0,1\}$, $j \in \mathcal{J}$	next_p, $p \in \{0,1\}$
		Read: check_p, $p \in \{0,1\}$
		Write: set_p, $p \in \{0,1\}$

State:

$status_p \in \{check, set, do, done, end, stopped\}$, initially $check$
$cur_p \in \{0, \ldots, n-1\}$, initially $cur_0 = 0$ and $cur_1 = n-1$
$step_p \in \{-1, 1\}$, initially $step_0 = 1$ and $step_1 = -1$

Transitions of process p:

Internal Read check_p
Precondition:
  $status_p = check$
Effect:
  if $(cur_p + step_p) \geq 0$ AND $(cur_p + step_p) \leq n-1$
  then
    if $x_{cur_p + step_p} = 0$
    then $status_p \leftarrow do$
    else $status_p \leftarrow end$
  else
    $status_p \leftarrow end$

Internal next_p
Precondition:
  $status_p = done$
Effect:
  $cur_p \leftarrow cur_p + step_p$
  $status_p \leftarrow check$

Internal Write set_p
Precondition:
  $status_p = set$
Effect:
  $x_{cur_p} \leftarrow 1$
  $status_p \leftarrow done$

Output $\mathsf{do}_{p,j}$
Precondition:
  $status_p = do$
  $cur_p = j$
Effect:
  $status_p \leftarrow set$

Input stop_p
Effect:
  $status_p \leftarrow stopped$

Fig. 1. Algorithm $\text{AO}_{2,n}$: Shared Variables, Signature, States and Transitions

to the progress of each process. Initially x_{left} and x_{right} are set to 0 and $n-1$ respectively, and thereafter each time process 0 or 1 performs a job with action $\mathsf{do}_{*,*}$, x_{left} is incremented or x_{right} is decremented respectively at event set. The decision (made in action check) on whether it is safe to perform a job is based on the differences $x_{right} - cur_0$ and $cur_1 - x_{left}$ for processes 0 and 1 respectively. If the difference is greater than 1, then it is safe to perform the job. With similar arguments as in Theorem 2 the result follows.

Theorem 3. *Algorithm* $\text{AO}'_{2,n}$ *solves the at-most-once problem.*

4.3 Effectiveness, Work and Space Complexity

We now present the efficiency results for both algorithms.

Effectiveness: We show that algorithms $\text{AO}_{2,n}$ and $\text{AO}'_{2,n}$ perform $n-1$ jobs in the presence of at most one stopping failure (optimal given Corollary 1).

Theorem 4. *The effectiveness of* $\text{AO}_{2,n}$ *with* $f < 2$ *is* $E_{\text{AO}_{2,n}}(n, 2, f) = n - 1$.

Theorem 5. *The effectiveness of* $\text{AO}'_{2,n}$ *with* $f < 2$ *is* $E_{\text{AO}'_{2,n}}(n, 2, f) = n - 1$.

Work and Space: Next we asses the work and space complexity of algorithms $\text{AO}_{2,n}$ and $\text{AO}'_{2,n}$. Recall that algorithm $\text{AO}_{2,n}$ uses single bit shared variables and $\text{AO}'_{2,n}$ uses shared variables of $\log n$ bits.

Theorem 6. *Algorithm* $\text{AO}_{2,n}$ *has work* $2(n+1)$ *and space* $n + 2\log n + 8$ *bits.*

Theorem 7. *Algorithm* $\text{AO}'_{2,n}$ *has work* $2(n+1)\log n$ *and space* $4\log n + 10$ *bits.*

5 Multiprocess Solution for the At-Most-Once problem

We now present m-process algorithm $\text{AO}_{m,n}$, given in Fig. 2, where $m = 2^h$, and the number of jobs is $n = k^h$ (non-powers are handled using standard padding techniques). The algorithm is a hierarchical generalization of algorithm $\text{AO}_{2,n}$. It uses an abstract full k-ary tree of h levels to keep track of progress and guarantee at-most-once semantics. All processes start at the root of the tree at level 0. At each node λ at level μ processes are split in two groups according to their process identifiers and look for subtrees with jobs that are safe to perform in the children of node λ. Thus at each node λ we can see the processes as two groups, group 0 and group 1, solving a sub-problem with k groups of jobs (the subtrees rooted at the children of node λ) using the approach of algorithm $\text{AO}_{2,n}$. Group 0 starts from the leftmost child of node λ and moves to the right, while group 1 starts from the rightmost child and moves to the left. Both groups use the LA-DC approach to define whether it is safe to perform a group of jobs (sub-tree rooted at a child of node λ).

We store the tree on a shared memory array by associating each node with a shared variable. Variable x_0 is associated with the root at level 0, $x_1, \ldots, x_k$ with the nodes at level 1, $x_{k+1}, \ldots, x_{k^2}$ with the nodes at level 2, and so on. In general the nodes at level $\mu \in [1 \ldots h]$ are associated with the shared variables $x_{u_\mu}, \ldots, x_{u_\mu + k^\mu - 1}$, where $u_\mu = 1 + k + k^2 + k^3 + \ldots + k^{\mu-1}$. The tree has a total of $v = u_{h+1}$ nodes. We denote by node λ the node associated with the shared variable x_λ, that has children associated with $x_{\lambda \cdot k+1}, \ldots, x_{\lambda \cdot k+k}$ and a parent associated with $x_{\lfloor \frac{\lambda-1}{k} \rfloor}$. Node $\lambda \in [0 \ldots v-1]$ is at level $\mu = \lfloor \log_k (\lambda \cdot (k-1) + 1) \rfloor$. Finally, job j is associated with leaf $x_{u_h + j}$. Next we present $\text{AO}_{m,n}$ in more detail.

Internal Variables of process p

$status_p \in \{check, set, up, down, do, done, end, stopped\}$ records the status of process p and defines its next action as follows: *down–p* can move to the children of its current node, *up–p* finished the current level and can move one level higher, *set–p* can set the shared variable associated with its current node to 1, *check–p* has to check whether it is safe to work at the current node, *do–p* is at a leaf and can perform the associated job, *done–p* finished working at the current node and can move to the next, *end–p* terminated (it is not safe for p to work on the tree), *stopped–p* crashed. All processes start at node 0, with $status_p = down$.

$pid_p[0 \ldots h]$ is a binary expansion of p into $h+1$ bits. Note that $p \in [0, 2^h - 1]$ and thus $\forall p \in \mathcal{P},\ pid_p[0] = 0$.

$cur_p \in \{0, \ldots, v-1\}$ marks the node at which process p is positioned.

$left_p, right_p \in \{0, \ldots, v-1\}$ keeps the leftmost and rightmost siblings of the current node.

$lvl_p \in \{0, \ldots, h\}$ stores the level μ of the current node.

$step_p \in \{-1, 1\}$ tracks of whether process p is moving from right to left or left to right at the current level.

Actions of process p

down_p: Process p moves one level down. If a leaf is reached, it sets $status_p = do$ in order for the job associated with the leaf to be performed. If p is at an internal node, it checks whether $pid_p[lvl_p]$ is 0 or 1. If it is 0, then p moves to the

leftmost child of node cur_p, otherwise it moves to the rightmost child. Process p sets lvl_p, cur_p, $left_p$, $right_p$ and $step_p$ accordingly. The status of p remains *down*.

check_p: If p works left-to-right and cur_p is the rightmost child of its parent, it sets $status_p = up$. Similarly if p works right-to-left and cur_p is the leftmost child of its parent, it sets $status_p = up$. Otherwise, p performs a look-ahead read in shared memory to determine if it is safe to work on the subtree rooted at node cur_p. If the shared variable associated with the next node ($cur_p + step_p$) is 0, it is safe to work on the subtree of node cur_p and thus sets $status_p = down$. Otherwise it sets $status_p = up$.

up_p: Process p moves one level up. If it is at level 1 (only root is above), it sets $status_p = end$ and terminates. If by moving up an internal node is reached, p updates its internal variables accordingly by checking the proper bit of its pid_p variable, and sets $status_p = set$.

set_p: Process p writes 1 to the shared variable associated with the node cur_p and sets $status_p = done$.

next_p: Process p moves to the next node (left or right, per value of $step_p$), and sets $status_p = check$.

$\mathsf{do}_{p,j}$: Process p preforms job j. Then p sets $status_p = set$.

stop_p: Process p crashes by setting $status_p = stopped$.

Correctness. We show that algorithm $\text{AO}_{m,n}$ solves the at-most-once problem. First we prove that at any internal node λ at level μ, either only processes with $pid_p[\mu] = 0$, or only processes with $pid_p[\mu] = 1$ enter the subtree rooted at λ.

Lemma 3. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' in α and processes $p, q \in \mathcal{P}$ s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = \left\lfloor \frac{s'.cur_q - 1}{k} \right\rfloor = \lambda$, for some node λ at level μ, then $pid_p[\mu] = pid_q[\mu]$.*

Proof. For node λ at level μ, if it is the leftmost child of its parent, then from the first **if** clause of action check_p, only processes with $pid_p[\mu] = 0$ may enter the subtree rooted at λ. Similarly if node λ is the rightmost child, only processes with $pid_p[\mu] = 1$ may enter the subtree rooted at λ. If node λ is between the leftmost and rightmost children of its parent $\left(\lambda \in \left[\left\lfloor \frac{\lambda-1}{k} \right\rfloor \cdot k + 2 \ldots \left\lfloor \frac{\lambda-1}{k} \right\rfloor \cdot k + k - 1\right]\right)$, then processes with $pid_p[\mu] = 0$ will approach it from the left, while processes with $pid_p[\mu] = 1$ will approach it from the right. In order to get a contradiction let us assume that there exists execution α that has states s, s' and processes p, q with $pid_p[\mu] = 0$ and $pid_q[\mu] = 1$, s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$. This means that both processes have entered the subtree rooted at node λ. For this to happen, there exist in α transitions $(\{cur_p = \lambda, status_p = check\}, \mathsf{check}_p, \{cur_p = \lambda, status_p = down\})$ and $(\{cur_q = \lambda, status_q = check\}, \mathsf{check}_q, \{cur_q = \lambda, status_q = down\})$, that precede s and s' respectively. Recall that p moves left-to-right and q right-to-left, and before moving to a new node at a level, they set the shared variable associated with the previous node to 1. Hence it follows that either $x_{\lambda+1} = 1$ when action check_p took place or $x_{\lambda-1} = 1$ when action $check_q$ took place. If the

Shared Variables: $\mathcal{X} = \{x_0, \ldots, x_{v-1}\}$, x_i boolean initially 0

Signature:

Input:	Output:	Internal:	
stop_p, $p \in \mathcal{P}$	$\mathsf{do}_{p,j}$, $p \in \mathcal{P}$, $j \in \mathcal{J}$	next_p, $p \in \mathcal{P}$	*Read:* check_p, $p \in \mathcal{P}$
		up_p, $p \in \mathcal{P}$	*Write:* set_p, $p \in \mathcal{P}$
		down_p, $p \in \mathcal{P}$	

State:

$status_p \in \{check, set, up, down, do, done, end, stopped\}$, initially $down$

$pid_p[0 \ldots h]$, where $pid_p[i] = \left\lfloor \frac{p}{2^{h-i}} \right\rfloor \mod 2$ (the binary expansion of p to $h+1$ bits)

$cur_p \in \{0, \ldots, v-1\}$, initially 0

$left_p \in \{0, \ldots, v-1\}$, initially 0

$right_p \in \{0, \ldots, v-1\}$, initially 0

$lvl_p \in \{0, \ldots, h\}$, initially 0

$step_p \in \{-1, 1\}$, initially undefined

Transitions of process p:

Input stop_p
Effect:
  $status_p \leftarrow stopped$

Internal Read check_p
Precondition:
  $status_p = check$
Effect:
  if $(cur_p + step_p) \geq left_p$
  AND $(cur_p + step_p) \leq right_p$
  then
    if $x_{cur_p + step_p} = 0$
    then
      $status_p \leftarrow down$
    else $status_p \leftarrow up$
  else
    $status_p \leftarrow up$

Internal next_p
Precondition:
  $status_p = done$
Effect:
  $cur_p \leftarrow cur_p + step_p$
  $status_p \leftarrow check$

Internal up_p
Precondition:
  $status_p = up$
Effect:
  if $lvl_p = 1$ **then**
    $status_p \leftarrow end$
  else
    $lvl_p \leftarrow lvl_p - 1$
    $cur_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor$
    $left_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor \cdot k + 1$
    $right_p \leftarrow \left\lfloor \frac{cur_p - 1}{k} \right\rfloor \cdot k + k$
    if $pid_p[lvl_p] = 0$ **then**
      $step_p \leftarrow 1$
    else
      $step_p \leftarrow -1$
    $status_p \leftarrow set$

Internal Write set_p
Precondition:
  $status_p = set$
Effect:
  $x_{cur_p} \leftarrow 1$
  $status_p \leftarrow done$

Internal down_p
Precondition:
  $status_p = down$
Effect:
  if $lvl_p = h$ **then**
    $status_p \leftarrow do$
  else
    $lvl_p \leftarrow lvl_p + 1$
    $left_p \leftarrow cur_p \cdot k + 1$
    $right_p \leftarrow cur_p \cdot k + k$
    if $pid_{lvl_p} = 0$ **then**
      $cur_p \leftarrow left_p$
      $step_p \leftarrow 1$
    else
      $cur_p \leftarrow right_p$
      $step_p \leftarrow -1$

Output $\mathsf{do}_{p,j}$
Precondition:
  $status_p = do$
  $cur_p = u_h + j$
Effect:
  $status_p \leftarrow set$

Fig. 2. $\text{AO}_{m,n}$: Shared Variables and Signature, States and Transitions of processes

first case is true, then the state of p becomes $\{cur_p = \lambda, status_p = up\}$ preventing p from entering the subtree rooted at λ. Otherwise the state of q becomes $\{cur_q = \lambda, status_q = up\}$ and q never enters the subtree rooted at λ. So it cannot be the case that both process p and q entered the subtree rooted at node λ in α and that completes the proof.

Lemma 4. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' in α and processes $p, q \in \mathcal{P}$ s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = \left\lfloor \frac{s'.cur_q - 1}{k} \right\rfloor = \lambda$, for some node λ at level μ, then $pid_p[0 \ldots \mu] = pid_q[0 \ldots \mu]$.*

Proof. We prove this by induction on the level μ of node λ.

Base Case: Here we consider level $\mu = 0$, meaning that all processes that reach the children of the root (node 0) have the same $pid_*[0]$ bit. This holds since $\forall p \in \mathcal{P}, pid_p[0] = 0$. Thus for any execution α of $\text{AO}_{m,n}$, if there exists state s in α s.t. $\left\lfloor \frac{s.cur_p - 1}{k} \right\rfloor = 0$ for some process $p \in \mathcal{P}$, $pid_p[0] = 0$.

Induction Hypothesis: Assume that for any execution α if there exist states s, s' and processes p, q s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$, for all nodes $\lambda \in [x_{u_\mu} \dots x_{u_\mu + k^\mu - 1}]$ at level μ, then $pid_p[0...\mu] = pid_q[0...\mu]$.
Induction Step: By Lemma 3 we show that $\forall \lambda \in [x_{u_{\mu+1}} \dots x_{u_{\mu+1} + k^{\mu+1} - 1}]$ at level $\mu + 1$, for any execution α, if there exist states s, s' and processes p, q s.t. $\lfloor \frac{s.cur_p - 1}{k} \rfloor = \lfloor \frac{s'.cur_q - 1}{k} \rfloor = \lambda$, then $pid_p[0 \dots \mu + 1] = pid_q[0 \dots \mu + 1]$.

From Lemma 4 we get Corollary 2 that says, that in any execution α of $\text{AO}_{m,n}$, only one process p, if any, may reach the decision to perform job j associated with leaf $u_\mu + j$. This decision is reflected in α by a state s, where $s.cur_p = u_\mu + j, s.status_p = do$.

Corollary 2. *For any execution α of algorithm $\text{AO}_{m,n}$ if there exist states s, s' and processes p, q s.t. $s.cur_p = \lambda, s.status_p = do$ and $s'.cur_q = \lambda, s'.status_p = do$, for some leaf $\lambda \in [u_h \dots u_\mu + k^h - 1]$, then $p = q$.*

Theorem 8. *Algorithm $\text{AO}_{m,n}$ solves the at-most-once problem.*

Work and Space: Next we assess work and space of algorithm $\text{AO}_{m,n}$. According to the algorithm specification, only the actions check_p and set_p perform memory accesses, and every time they do so, they access exactly one bit.

Theorem 9. *The work complexity of algorithm $\text{AO}_{m,n}$ is $O(n + m \log m)$.*

Proof. We observe that for each subtree rooted at an internal node λ at level μ we have a *sub-instance* of the problem for $k^{h-\mu}$ jobs and $2^{h-\mu}$ processes. All processes of such sub-instance have the same prefix at the first μ bits of their *pid* from Lemma 4. Let W_μ be an upper bound on work of the sub-instance. Now we consider the first level of the subtree. Processes are split in groups 0 and 1 (with $2^{h-(\mu+1)}$ processes each), according to the value of their $pid_*[\mu+1]$. Group 0 starts at the leftmost child, group 1 at the rightmost child, and they move towards each other. From Lemma 4 we have that only one of the groups, if any, will continue to the sub-instance of the next level, thus we have at most k sub-instances derived at level $\mu + 1$. From algorithm $\text{AO}_{m,n}$, we have that before a process enters a node, it does a look ahead memory read, and when it leaves a node, it sets the shared variable associated with the node to 1. This means that we have a total of $k + 2$ reads and k writes from the two groups. Since each group has $2^{h-(\mu+1)}$ processes, we get $(k + 2) \cdot 2^{h-(\mu+1)}$ reads and $k \cdot 2^{h-(\mu+1)}$ writes. From the above discussion we have the following recurrence relation: $W_\mu = k \cdot W_{\mu+1} + (2k + 2) \cdot 2^{h-(\mu+1)}$.
Also for level h (k jobs and 2 processes), we have $k + 2$ reads and k writes by Theorem 6, thus: $W_h = 2k + 2$. Combining the above we get:
$W_0 = k \cdot W_1 + (2k + 2) \cdot 2^{h-1} = (2k + 2) \cdot 2^{h-1} \cdot \sum_{i=0}^{h-1} \left(\frac{k}{2}\right)^i$.
Case $k = 2$: $(2k + 2) \cdot 2^{h-1} \cdot \sum_{i=0}^{h-1} \left(\frac{k}{2}\right)^i = 6 \cdot 2^{h-1} \cdot h = 5m \log m$
Case $k > 2$: $(2k + 2) \cdot 2^{h-1} \cdot \sum_{i=0}^{h-1} \left(\frac{k}{2}\right)^i = (2k + 2) \cdot 2^{h-1} \cdot \frac{\left(\frac{k}{2}\right)^h - 1}{\frac{k}{2} - 1} = \frac{2k+2}{k-2} \cdot (n - m) \leq 8(n-m)$, where the penultimate relation follows form $m = 2^h, n = k^h$.
We conclude that $W_0 = \Theta(n + m \log m)$.

Theorem 10. *The space complexity of algorithm* $\text{AO}_{m,n}$ *is* $\Theta(n + m \log n)$.

Effectiveness: We now assess the effectiveness of algorithm $\text{AO}_{m,n}$.

Theorem 11. *Algorithm* $\text{AO}_{m,n}$ *has effectiveness* $E_{\text{AO}_{m,n}}(n, m, m-1) = (n^{\frac{1}{\log m}} - 1)^{\log m} = n - \log m \cdot o(n)$.

Proof. We observe that for each subtree rooted at an internal node λ at level μ we have a sub-instance of the problem for $k^{h-\mu}$ jobs and $2^{h-\mu}$ processes. Moreover if we consider only the first level of such a sub-instance, we have to solve a problem of k groups of jobs (with $k^{h-(\mu+1)}$ jobs each) and 2 groups of processes (with $2^{h-(\mu+1)}$ processes each). Furthermore, as we pointed out before, algorithm $\text{AO}_{m,n}$ follows the same principles for solving this instance as algorithm $\text{AO}_{2,n}$. Thus at each level we match the effectiveness of $\text{AO}_{2,n}$ that by Theorem 4 performs $E_{\text{AO}_{2,n}}(k, 2, 1) = k - 1$ jobs. If we go all the way down to level $h = \log_k n$, we have an exact instance of the 2-process problem (Section 4.1) and hence by Theorem 4 it follows that $E_{\text{AO}_{m,n}}(k, 2, 1) = E_{\text{AO}_{2,n}}(k, 2, 1) = k - 1$. From the above we get the following recurrence:

$$E_{\text{AO}_{m,n}}(n, m, m-1) = (k-1) \cdot E_{\text{AO}_{m,n}}\left(\tfrac{n}{k}, \tfrac{m}{2}, \tfrac{m}{2} - 1\right) = \cdots =$$
$$= (k-1)^{h-1} \cdot E_{\text{AO}_{m,n}}\left(\tfrac{n}{k^{h-1}}, \tfrac{m}{2^{h-1}}, \tfrac{m}{2^{h-1}} - 1\right) = (k-1)^{h-1} \cdot E_{\text{AO}_{m,n}}(k, 2, 1)$$

Thus $E_{\text{AO}_{m,n}}(n, m, m-1) = (k-1)^h$.

Finally, we note that since $E_{\text{AO}_{m,n}}(n, m, m-1) = n - \log m \cdot o(n)$, the effectiveness of the algorithm comes reasonably close, asymptotically in n, to the corresponding lower bound of $n - f$.

6 Conclusions

We examined the implementation of at-most-once semantics in an asynchronous multiprocessor shared memory model. We first defined the problem, proposed a new efficiency measures, we called *effectiveness* and counts the number of jobs performed by a given implementation, and we showed that at-most-once algorithms that tolerate f failures cannot perform more than $n - f$ jobs. Then we devised and analyzed two effectiveness-optimal algorithms for 2 processors using the collision avoidance paradigm, and finally we used those algorithms as building blocks to construct an algorithm for n processors. Our results reveal an effectiveness gap as the number of processes in the system increases. Thus we challenge the discovery of more complex collision detection techniques that would achieve higher effectiveness. Finally we question the existence and efficiency of algorithms that try to implement at-most-once semantics in systems with different means of communication, such as message-passing systems.

References

[1] R. J. Anderson and H. Woll. Algorithms for the certified write-all problem. *SIAM J. Computing*, 26(5):1277–1283, 1997.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.
[3] J. F. Buss, P. C. Kanellakis, P. Ragde, and A. A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.
[4] S. Chaudhuri, B. A. Coan, and J. L. Welch. Using adaptive timeouts to achieve at-most-once message delivery. *Distrib. Comput.*, 9(3):109–117, 1995.
[5] G. Di Crescenzo and A. Kiayias. Asynchronous perfectly secure communication over one-time pads. In *Proc. of 32nd International Colloquium on Automata, Languages and Programming(ICALP '05)*, pages 216–227. Springer, 2005.
[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
[7] K. J. Goldman and N. A. Lynch. Modelling shared state in a shared action model. In *Logic in Computer Science*, pages 450–463, 1990.
[8] J. Groote, W. Hesselink, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14(2), 2001.
[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.
[10] P. C. Kanellakis and A. A. Shvartsman. *Fault-Tolerant Parallel Computaion.* Kluwer Academic Publishers, 1997.
[11] D. R. Kowalski and A. A. Shvartsman. Writing-all deterministically and optimally using a non-trivial number of asynchronous processors. In *Proc. of the 16th annual ACM Symp. on Par. in Alg. and Arch.(SPAA '04)*, pages 311–320. ACM, 2004.
[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
[13] B. W. Lampson, N. A. Lynch, and J. F. S-Andersen. Correctness of at-most-once message delivery protocols. In *Proc. of the IFIP TC6/WG6.1 6th International Conference on Formal Description Techniques(FORTE '93)*, pages 385–400, 1994.
[14] K.-J. Lin and J. D. Gannon. Atomic remote procedure call. *IEEE Trans. Softw. Eng.*, 11(10):1126–1135, 1985.
[15] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, 1988.
[16] B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. Comput. Syst.*, 9(2):125–142, 1991.
[17] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.
[18] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
[19] G. Malewicz. A work-optimal deterministic algorithm for the asynchronous certified write-all problem. In *Proc. of the 22nd annual Symp. on Principles of Distributed Computing(PODC '03)*, pages 255–264. ACM, 2003.
[20] G. Malewicz. A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors. *SIAM J. Comput.*, 34(4):993–1024, 2005.
[21] F. Panzieri and S. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14(1):30–37, 1988.
[22] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, 1982.
[23] R. W. Watson. The delta-t transport protocol: Features and experience. In *Proc. of the 14th Conf. on Local Computer Networks*, pages 399–407, 1989.