Open access • Proceedings Article • DOI:10.1145/1583991.1584003

At-most-once semantics in asynchronous shared memory — Source link

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

Institutions: University of Connecticut

Published on: 11 Aug 2009 - ACM Symposium on Parallel Algorithms and Architectures

Topics: Asynchronous communication, Shared memory and Upper and lower bounds

At-Most-Once Semantics in Asynchronous Shared Memory
(Brief Announcement)

Sotirios Kentros * Aggelos Kiayias Nicolas Nicolaou
Alexander A. Shvartsman

Computer Science and Engineering
University of Connecticut
Storrs, CT 06268
{skentros,aggelos,nicolas,aas}@engr.uconn.edu

February 23, 2009

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 specific instance of 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 semantic for the model with failure-prone, asynchronous shared-memory multiprocessors. The focus here is on the setting, where any processor is able to perform any task on any object, where the total number of tasks is performed is to be maximized while preserving the at-most-once semantics. The paper introduces formal definitions of the *At-Most-Once* and *Do-Exactly-Once* problems for performing tasks (including accessing memory) in the assumed model, and defines the notion of efficiency, called *effectiveness*, that allows for precise characterizations of algorithms solving these problems. Effectiveness for an at-most-once implementation is the number of tasks completed (at-most-once) by the implementation, as a function of the overall number of tasks, the number of participating processors, and the number of processor failures. We show a lower bound on the effectiveness in our model for at-most-once and do-exactly-once implementations that states that at least f tasks cannot be completed, where f is the maximum number of crashes. Following this finding we present two effectiveness-optimal at-most-once algorithms for two-processes (the second improving the space performance of the first) and then we propose an algorithm for the model with n processors. The last algorithm being a hierarchical generalization of a two-processor solution. The algorithms are presented using Input/Output Automata formalism. We prove correctness of the algorithms and analyze their performance in terms of effectiveness.

Keywords: Asynchronous Shared Memory, At-Most-Once Semantics

*Part of this work is supported by the State Scholarships Foundation of Greece

1 Introduction

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 and method invocations in decentralized systems, 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 the operations to be performed. The problem becomes very challenging for autonomous processors in a shared-memory system, in which processes perform concurrent object invocations.

At-most-once message delivery protocols have been a subject of much research, e.g., [4, 1, 2]. The authors in [4] motivate their work in terms of remote procedure calls and method invocations, obtaining at-most-delivery with reliance on the estimation of message lifetimes. The work [1] achieves stronger results by eliminating timing dependencies. The work [2] considers the at-most-once semantic motivated by the security requirements of *one-time pad* encryption. The authors partition a *shared* random pad among multiple communicating parties. Perfect security can be achieved only if every piece of the pad is used at most once by any process in the system. Unlike the works [4, 1], the authors of [2] connect the at-most-once problem to the problem of multiple processor coordination, where each process has to decide whether or not to use certain parts of the single shared pad.

Contributions. This paper explores the possibility of At-Most-Once implementations for asynchronous shared-memory multiprocessor environments where processors are prone to crashes. We are concerned with a setting where a collection of objects exists in shared memory, such that each object has a specific task that needs to be performed for the object using at-most-once semantics. Here any processor is able to perform the specific single task for any object, and we are interested in maximizing the total number of performed tasks while preserving the at-most-once semantics. We formally define our problem and the notion of *effectiveness* used to assess the efficiency of solutions for the problem. Effectiveness measures the number of operations (tasks) performed using at-most-once semantics as a function of the number of objects to be accessed, the number of processors, and the maximum number of processor 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 and they achieve optimal effectiveness. The third algorithm is stated for an arbitrary number of processors comprising the system. The algorithms in this work are motivated by the algorithm for the *Write-All* problem from [3], although the correctness criteria for our algorithms is quite different. Our work can be viewed as complementary to [2] that considers a similar problem in message-passing models.

2 Model, Problems, Definitions

We consider a system of n asynchronous processors, where the processors are susceptible to crashes. The model includes shared memory, where the memory locations are atomic. We use the *Input/Output Automata* formalism [5]. In particular an *asynchronous shared memory automaton*, consists of a finite set of *processes* that interact by means of a finite collection of *shared variables* [5].

The adversary controls the asynchrony and the crashes of the processes. We allow up to $f \leq n-1$ crashes in our system, where n is the number of processes in A.

In our setting we consider algorithms whose purpose is to perform a set of tasks or activities that we call *jobs*. Let A be such an algorithm that is comprised of a set of processes $\mathcal{P}$, where $|\mathcal{P}| = n$, and where the jobs come from the set $\mathcal{J}$ with $|\mathcal{J}| = m$. We assume that there are at least as many jobs as there are processors, i.e., $m \geq n$. A job is *performed* in an execution α of A by process $p \in \mathcal{P}$, if α includes 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 2.1 *Let the set of performed jobs in execution α be denoted by $J_\alpha = \{j \in \mathcal{J} \mid \exists\, \mathsf{do}_{\mathsf{p,j}}$ event in α, where $p \in \mathcal{P}\}$. The total number of jobs performed in execution α of A is defined as: $Do(\alpha) = |J_\alpha|$*

We next define the *at-most-once* and *do-exactly-once* properties.

Definition 2.2 *Property AO (at-most-once correctness): An execution α of A satisfies AO if:* $\forall j \in \mathcal{J} : \sum_{p \in \mathcal{P}} len\left(\alpha|_{\mathsf{do}_{p,j}}\right) \leq 1$. *In this case α is called an AO-execution.*

We say that algortihm A satisfies the AO property if any of its executions is an AO-execution. The *at-most-once problem* consists of devising such algorithms.

Definition 2.3 *Property EO (do-exactly-once correctness): An execution α of A satisfies EO if:* $\forall j \in \mathcal{J} : \sum_{p \in \mathcal{P}} len\left(\alpha|_{\mathsf{do}_{p,j}}\right) = 1$. *In this case α is called an EO-execution.*

Algorithm A satisfies the EO property if any fair execution of A is an EO-execution. Devising such algorithms is the *do-exactly-once problem*. We note that AO is a safety property, thus it must hold in any execution. On the other hand we intend for EO to be a liveness property, thus it must hold in fair executions. It is easy to see that any prefix of an EO-execution or an AO-execution is also an AO-execution. Next we define the *effectiveness* measure that counts the number of jobs performed by an automaton in the worst case.

Definition 2.4 *The **effectiveness** of algorithm A is: $e_A(m, n, f) = \min\{Do(\alpha)\}$ where α is any fair execution of A with n processes, m jobs, and at most f crashes.*

Here we consider only the case where $m \geq n > f$, since the most interesting case is when there is at least one job for each processor. (In the full version of the paper we also consider the case $m \leq n$).

3 Lower Bound and Impossibility

We show that if the at-most-once property is to be preserved, it is impossible for any algorithm to sustain f crashes and perform more than $m - f$ jobs.

Theorem 3.1 *For any algorithm A that satisfies the AO property with n processes, $m \geq n$ jobs, and in the presence of $f < n$ crashes it holds that $e_A(m, n, f) \leq m - f$.*

Moreover, we show the do-exactly-once problem cannot be solved even in the presence of a single crash:

Theorem 3.2 *For any algorithm A, there exists a fair execution of A that does not satisfy the EO property in the presence of $f \geq 1$ crashes.*

4 Algorithms for the At-Most-Once Problem

We developed three solutions for the at-most-once problem. Two of them (P2A and P2B) assume a two process system, while the third one (P2^l) is more general and assumes n-process participation. The last solution is motivated by the algorithm of Groote et al. [3]. We now provide a brief description of the three algorithms and state their effectiveness.

Algorithm P2A. Algorithm P2A solves the at-most-once problem for m jobs, using two processes p and q, and m 1-bit shared variables. The main idea is to have the processes move in opposite directions, trying to avoid a collision (i.e., doing a job twice) by adopting a *"look ahead decide for the current"* (LA-DC) approach.

In particular we use m shared variables $\{x_1, \ldots, x_m\}$ as a bookkeeping mechanism on the progress of the processes. Initially all shared variables are set to 0 and let process p start from job 1 and process q start from job m. When some process, say p, performs job j, it sets $x_j = 1$. Deploying the LA-DC approach, a process decides that it is safe to perform some job by checking the shared variable associated with the next task in its path. More precisely p (resp. q) performs a job j only if $x_{j+1} = 0$ (resp. $x_{j-1} = 0$). The key idea here lies on the fact that since the shared variable x_{j+1} (resp. x_{j-1}) is 0 then the competing process q (resp. process p), did not yet perform the task $j + 1$ (resp. $j - 1$). Hence it cannot be performing j, which the acting process attempts

to perform, and thus collision is avoided. It can be shown that the effectiveness of P2A in the presence of at most one stopping failure is $e_{P2A}(m, 2, 1) = m - 1$, where m is the number of tasks (this is optimal by Theorem 3.1).
Algorithm P2B. Algorithm P2B is also based on LA-DC idea. The main difference here is that we only use two integer shared variables, $x_{left} = 1$ and $x_{right} = m$, each of size $\log m$ bits. Each variable represents a pointer to the progress of each processor. Initially $x_{left} = 1$ and $x_{right} = m$, and thereafter each time process p or q performs a job, x_{left} is incremented and x_{right} is decremented respectively. The decision on whether it is safe to perform a task j, is based on the difference of the shared variables, $x_{right} - j$ and $j - x_{left}$, for processes p and q respectively. If the difference is greater than 1, then there is a safety gap between the progress of the two processes and thus it is safe to perform the task, hence collision is avoided. It can be shown that P2B has the same (optimal) effectiveness as P2A, $e_{P2B}(m, 2, 1) = m - 1$.
Algorithm P2^l. This is an n-processor algorithm for the at-most-once problem, where n is 2^l (non-powers of two are easily handled using standard padding techniques). Let $m = k^l$ be the number of jobs in $\mathcal{J}$, for some integer $k \geq 2$. Algorithm P2^l is a hierarchical generalization of algorithm P2A. The algorithm uses a full k-ary tree of l levels to keep track of progress and satisfy property AO. The tree, is reflected in the shared memory as follows: no variable represents the root, variables $x_1, \ldots, x_k$ represent level 1, variables $x_{k+1}, \ldots, x_{k^2}$ form level 2 and so on. The leaves are shared memory variables $x_{c+1}, \ldots, x_{c+k^l}$, where $c = k + k^2 + k^3 + \ldots + k^{l-1}$. An internal node x_λ, has children $\{x_{(\lambda \cdot k)+1}, \ldots, x_{(\lambda+1)\cdot k}\}$. The parent of node x_λ is node $x_{\lfloor \frac{\lambda-1}{k} \rfloor}$. Each leaf is associated with one of the jobs in $\mathcal{J}$: job j is associated with leaf x_{c+j}.

The algorithm proceeds as follows. All processes start at level 1. At each level processes are split in half according to their process identifiers and compete to find subtrees with jobs that are safe to perform. Essentially, in each level we can see the processes as two multiprocessors, solving a subproblem with k jobs (the subtrees) using the approach of algorithm P2A. The effectiveness of algorithm P2^l, can be expressed by:

$$e_{\mathrm{P}2^l}(m, n, n-1) = (m^{\frac{1}{\log n}} - 1)^{\log n} = m - \log n \cdot o(m)$$

where the number of processors is $n = 2^l$ and the number of jobs is $m = k^l$.

5 Discussion

We defined the At-Most-Once (AO) and Do-Exactly-Once (EO) problems in the asynchronous multiprocessor shared memory model and we propose a new efficiency measure we call *effectiveness* that counts the number of jobs performed by a given implementation. We showed that there exists no implementation that tolerates even a single crash while satisfying the EO-property. Moreover we showed that algorithms that tolerate f failures cannot perform more than $m - f$ jobs and still satisfy the AO-property. We presented three algorithms for the At-Most-Once problem. The first two are two-processor algorithms that achieve optimal effectiveness. The third is an n-processor algorithm that achieves good effectiveness.

References

[1] 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.

[2] G. D. Crescenzo and A. Kiayias. Asynchronous perfectly secure communication over one-time pads. 3580:216–217, 2005.

[3] 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.

[4] 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.

[5] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.