Yale University
Department of Computer Science

Adaptive Algorithms for PASO Systems

Jeffery Westbrook Lenore Zuck

Department of Computer Science
Yale University
New Haven, CT 06520

YALEU/DCS/TR-1013
August 1994

Abstract: We describe a fault-tolerant distributed storage system for local area networks. Our system implements Persistent, Associative, Shared Object (PASO) memory. A PASO memory stores a set of data objects that can be accessed by associative search queries from all nodes in an ensemble of machines. This approach to distributed memory has been used in a number of systems, and provides a convenient and useful model for parallel and distributed applications. PASO memory is amenable to adaptive implementations that relocate data objects in response to changing network configurations and access patterns, making it a good candidate for an efficient, fault-tolerant storage system. The paper defines the semantics of PASO memory, gives a basic design strategy, discusses memory primitives and their costs, and discusses adaptive techniques for improving efficiency.

1 Introduction

This paper presents PASO, a Persistent, Associative, Shared Object memory, and studies algorithms that implement fault-tolerant and efficient PASO memories using adaptive algorithms for data replication.

PASO memories store data objects that can be accessed by associative search queries, based on "search criteria", from all nodes in an ensemble of machines. An object in a PASO memory is a tuple of values drawn from ground sets of basic data types. The memory contains a collection of objects, each of which has an arbitrary number of fields. Programs manipulate the PASO memory through three atomic operations: `insert`, `read`, and `read&del`. A PASO memory is *associative* in the sense that objects are accessed by pattern-matching. For example, a `read` takes an object template (search criterion) specifying acceptable values for each field, and returns any one object matching that template. There is no `modify` operation; modifying a field is logically equivalent to destroying the old object and creating a new one. There is no loss of generality, since a mutable distributed data structure can be built out of collections of immutable atomic objects. The memory is "shared" in the sense that any object can be read or deleted by any participating process. It is "persistent" in the sense that once an object is inserted into the memory, it remains there until it is deleted, irrespective of whether its creating process is still alive.

A PASO memory is built on a network of n machines, each of which has local memory and supports a set of *compute processes* and one *memory server*. A compute process executes a user program that generates requests for access to the PASO memory by means of the basic PASO operations. A memory server manages some collection of PASO objects stored in the local memory of the machine and is responsible for serving the PASO requests generated by compute processes. The primary type of fault we consider are crash errors, in which a machine crashes and all processes on that machine are halt.

Shared memories that qualify informally as PASOs have been used as coordination languages in a variety of parallel programming systems, e.g., in the context of C [10], Scheme [18], Prolog [9], distributed object-oriented systems [24], Modula-2 [8], program visualization systems [25], math libraries [11], and as part of other coordination mechanisms [1, 22]. They have proven to be an effective basis for concurrent programming in many multi-processing environments [10].

Despite these various implementations, there is no widely accepted formal semantics for PASO memories (often referred to as "tuple spaces" in the literature). One of the

contributions of this paper is a simple semantics for PASO systems that covers most tuple-space implementations. The fact that informal PASO memories are a pragmatic success makes them good candidates for formal, algorithmic, and theoretical research that aims at improving them. Kambhatla and Walpole make a case that tuple spaces are "a particularly suitable model for fault-tolerant applications" [20], because of the "highly asynchronous" or "uncoupled" nature of the tuple space model. (Because processes are mutually anonymous and never deal with each other directly, it becomes relatively simple to substitute a new process for a failed one.)

A PASO memory that is able to tolerate many rapidly occurring failures is especially useful in designing parallel algorithms that adapt to changing availability of computational resources—*adaptive parallelism* [15]. Today's ubiquitous work-station networks are huge reservoirs of power and wasted potential, reservoirs that can be tapped by adaptive parallel programs designed to gain or lose processing units during the computation. Adaptive fault-tolerant techniques can allow a distributed application to retire gracefully from work-stations that are being reclaimed for personal use, and expand onto nodes that become available. It has been argued that adaptive-parallel programs executing on networked multiprocessors will be one of the most important arenas for high-performance computing over the next decade[16] .

In fault-tolerant systems data must be replicated to survive machine failure. The main drawback of fault-tolerant systems has been that the benefits of preserving data in the face of failures may be outweighed by the loss of efficiency when errors are infrequent. Recent research in on-line algorithms using *competitive analysis* [23, 21] has presented schemes for adaptive file replication and migration that reduce overall message complexity in distributed file management (*e.g.* [3]). This paper suggests using the same adaptive on-line techniques to provide both efficiency and fault-tolerance. We consider adaptive object replication in PASO memory management, with the goals of reducing response time, message cost, and total workload.

We assume a basic communication and coordination mechanism based on the ISIS [7, 17] model of virtual synchrony. The mechanism is discussed in further detail in Section 3. ISIS offers elegant tools to deal with faults, and basing our PASO system on ISIS allows us to focus on the issues that are of main interest to us, namely, efficient replication of objects. The ISIS fault model covers a large range of frequently occurring faults, and the ISIS system is widely used. A prototype PASO system built on top of ISIS is currently running at Yale.

In previous work, Xu and Liskov [28] discuss the use of the virtual partition algorithm to maintain the consistency of tuple replicas in the tuple space. Bakken and Schlichting [4, 5] assume a reliable tuple space, and propose a new atomic tuple-swap operator that can be used to build reliable applications of a certain type ("bag of task" applications). Anderson and Shasha's [2] work on Persistent Linda includes support for transactions, but doesn't focus on the problem of reliable distributed tuple spaces. Kambhatla and Walpole [20, 27] discuss reliable tuple spaces, and the use of a reliable tuple space to build a reliable application. The degree to which this previous work is generalizable is not clear, however, since current implementations of tuple spaces restrict searches in various ways that are not true of our more general PASO model. This becomes quite significant when making a system fault tolerant, since permitting general search criteria implies a need for more synchronization among the processors, which is harder to achieve in the presence of faults.

In addition, these papers do not directly address efficiency considerations.

The paper is organized as follows. Section 2 presents the semantics of the PASO. Section 3 describes the physical model, fault model, and communication model. Section 4 presents a family of algorithms that implement PASO primitives and their cost. The algorithms in this family differ in their assignment of the "write groups" which determine on which machines objects are replicated. Section 5 discusses adaptive strategies for assigning write groups whose goal is to reduce the overall workload. The strategies are analyzed using competitive analysis.

2 Semantics of PASO

The PASO model is similar to the Linda model. Unfortunately, there is no formal semantics for Linda that is accepted and used [14]. Recently, Jagannathan proposed a semantics for multiple tuple spaces [19], which does not extend to the multi-process case. The semantics outlined here was derived by studying operational Linda and Network Linda implementations; hence, it captures the semantics of the currently running Linda-like systems. A full description of the semantics is in [29].

The set of *objects* is denoted by $\mathcal{O}$. Each object has a "life". It is initially *prenatal*. If `inserted`, the object becomes *live*. If `read&deled`, the object becomes *dead*. *Search criteria*, used as arguments in `read` and `read&del` commands, are predicates over $\mathcal{O}$. We assume a set P of *processes*, each executing some program. The programs are "standard" programs (e.g., C) augmented with the special PASO primitives: `insert`, `read`, and `read&del`.

A *global state* of a PASO system consists of the *local states* of each of the processes and a mapping from the objects to their state. An *initial state* is a global state where all objects are prenatal, all processes are at their initial local states, and the value of the local variables of each process is as indicated by the program code.

All non-PASO commands are assumed to be atomic. Each PASO command is associated with two atomic commands, its issuing, and its return. A *joint transition* is defined by a set of (possibly null) atomic commands for each of the system's processes. Each joint transition defines a (global) state-to-(global) state successor function. For the non-PASO commands in the joint transition, this successor function is the obvious one.

A run of a PASO system is a sequence of alternating global states and joint transitions, starting with a state, and, if finite, ending with a state, such that the first state is initial and every state is the successor of the previous one under the successor function of the joint transition in between them.

Properties (A1)–(A3) describe the life-cycle of objects in every run of a PASO system. Property A1 describes the life cycle of an object in the run. Property A2 describes what in the run determines an object's life. Property A3 describes the processes in the run.

A1 (a) A prenatal object may remain so forever or become alive.
(b) A live object may live forever or die.
(c) A dead object remains dead.

A2 An object may become alive only after it is inserted. It may later die if returned from an `read&del`. Moreover, there is at most one `insert`(o) command and at most one `read&del` command that returns o.

A3 The individual run of each process as determined by the run is indeed a plausible run of the process.

It remains to describe the rules of each of the PASO commands. From A2 it follows that an object becomes alive only after an `insert` command. The rules of the `insert` commands require objects to become alive after being `inserted`. A `read` command returns either fail or an object that satisfies the search criterion and is alive at some time in between the issue and the return of the `read`. It may return fail only when there is no object that satisfies the search criterion and is consistently alive from the time the `read` is issued until the `read` returns. The rules of the `read&del` command are similar to the rules of the `read` command, but, in addition, require that an object that is returned from a `read&del` dies sometimes after the issue of the `read&del`. Note that the `read` and `read&del` operations defined here are "non-blocking". The operations defined in Linda are blocking. The semantics of the blocking version of `read` and `read&del` is rather straightforward. In Section 4, we discuss how to implement them.

3 The Physical Model

This section defines the fault model and the basic communication primitives and their cost. The communication model described here is motivated by ISIS [17], which offers correct and efficient implementation of these communication primitives. In fact, the PASO system now being implemented at Yale is running on top of ISIS.

3.1 Fault Model

A PASO system consists of a set Mach of machine. Let n denote the size of Mach. Each $M \in$ Mach supports a a set of processes which are either *compute processes* or *memory servers*. A memory server manages some collection of PASO objects stored in the local memory of the machine. For simplicity, we assume that every $M \in$ Mach supports a single memory server. Machines may crash and leave the system, and then be fixed and re-join the system.

We expect our system to tolerate up to λ simultaneous *crash* crashes of machines where $\lambda < n$ is some fixed constant. When a machine crashes, all its local memory is erased, and, consequently, the memory server hosted on it fails. When a faulty machine re-joins the system, the memory server hosted on it performs an initialization phase during which it obtains copies of the objects that it supports.

The time the initialization phase lasts depends on the the set $\mathcal{O}$ of objects the PASO memory can handle, the topology of the network, and the set SC of search criteria used in `read` and `read&del` operations. We assume it is bounded above and below. Both upper and lower bounds on the initialization phase are expected to be several minutes. The roughly correspond to the maximal and minimal time it takes to boot a machine.

A machine is considered faulty while in its initialization phase. Hence, we assume that at any time, there are at least $n - \lambda$ servers in the system that are past their initialization phase and are hosted on operational machines.

3.2 Synchronization

The main tool for achieving communication and synchronization in the system is the notion of "groups", which are essentially equivalent to the ISIS groups ([17]).

Let Names be a set of *group names*. Let group: Mach $\rightarrow 2^{\text{Names}}$ be a partial function that maps every operational machines to set of named *groups*. The group function is dynamic—at any point in a run, it may assign different groups to different machines. Group membership is determined by both the servers, who can leave and join groups at will, and by crashes that force servers on crashed machines to leave the groups they belong to.

To leave a group *g-name*, a server residing on an operational machine M (where *g-name* $\in$ group(M)) can execute a g-leave operation. To join a group *g-name* $\notin$ group(M), the server can execute a g-join operation. Both g-leave and g-join operation have two parameters: the name of the groups to be left or joined, and a name of a procedure that is executed to terminate of initiate the group change.

All communication occurs by means gcast operations: The operation gcast(*g-name*, *msg*, *resp*) broadcasts a message *msg* to each server currently subscribing to the group *g-name*. It is assumed that each of the servers that receives the message processes it and produces a response. *One* of these responses is sent to the issuing process and stored in (the local variable) *resp*. While this definition may seem unintuitive, in our model all responses are equal, hence, sending only one of them is used for efficiency.

The gcast primitive is assumed to be reliable. Namely, it eventually delivers the designated message to all group members. Moreover, the messages are delivered to all group members in the same order. Finally, all gcasts from the same process to the same group reach the group's members in the order they are sent. We assume that the groups are always in a stable state when receiving a gcast—memory servers cannot join or leave a group during a gcast to the group. In addition, all g-leave and g-join events to the same group are notified to all group members, in the same order they occur. In fact, every group member receives all message addressed to the group, and g-join and g-leave operations to the group in the same order. (The ISIS primitives provide for all of these properties.) Thus, there is some synchronization (called "virtual synchrony" in ISIS) in the system guaranteed by the group structure.

3.3 Communication Cost

We assume that the cost of transmitting a message *msg* is *msg-cost*(*msg*) $= \alpha + \beta|msg|$ where α and β are some constants. Hence, *msg-cost*(*msg*) consists of an initial startup cost α and a cost which depends on the *msg*'s length.

We assume no multicasts are available. (A multicast allows a single message to be sent to multiple recipients using only one bus transmission; this assumption is pessimistic, but is the actual situation in standard Unix work-station Ethernet systems.) Executing gcast(*g-name*, *msg*, *resp*) then involves sending *msg* to all members of *g-name* and gathering responses from them. Since only one of the responses is sent to the issuing process, and this response is stored in *resp*, the cost of transmitting this single response is $\alpha + \beta|resp|$. However, this response is sent only after *all* members of *g-name* have finished processing their copy of *msg*. We therefore assume that each of *g-name*'s members sends an empty message to some designated server (*g-name*'s "leader") indicating that it has finished processing

msg. Putting it altogether, we get

$$\begin{aligned}
&\textit{msg-cost}(\mathsf{gcast}(\textit{g-name}, \textit{msg}, \textit{resp})) \\
&= |\textit{g-name}|(\alpha + \beta|\textit{msg}|) + |\textit{gname}|\alpha + \alpha + \beta|\textit{resp}| \\
&= \alpha(2 \cdot |\textit{g-name}| + 1) + \beta(|\textit{msg}| \cdot |\textit{g-name}| + |\textit{resp}|) \\
&\approx |\textit{g-name}|(2 \cdot \alpha + \beta(|\textit{msg}| + |\textit{resp}|))
\end{aligned}$$

where $|\textit{g-name}| = |\{M \in \mathsf{Mach} : \textit{g-name} \in \mathtt{group}(M)|$.

4 A Basic Strategy

The section presents a family implementations of PASO systems. Without loss of generality, assume that every object can be inserted once. This is easily guaranteed, for example, by attaching to each object some unique identification signed by its creating process. Section 4.1 introduces the *write groups*. The write groups determine where objects are stored and how queries are answered. Section 4.2 describes the algorithms run by the memory servers, and Section 4.3 describes the macro expansions of the PASO primitives.

4.1 Write Groups

Objects are stored and searched for by partitioning them into object classes and associating a *write group* with every class. Each server of the write group stores every live object of the object class. All requests to insert a particular object are made to the write group of the (object class of the) object. All requests to read of remove an object based on a search criterion are made to write groups of classes that may include object satisfying the search criterion.

Let $\mathcal{C}$ be a set of *object classes*. Let $\mathsf{obj\text{-}cls}: \mathcal{O} \to \mathcal{C}$ be a function that partitions $\mathcal{O}$ into set of object classes, and for every global state g, let $\mathsf{wg}_g: \mathcal{C} \to \mathsf{Names}$ be a function that maps every object class to a group, whose members replicate the objects in the class. to be replicated.

From the definition it follows that the write groups of a class may change over time. In addition, as described in Section 3.2, group membership can change over time. To guarantee the correctness of the system, at any time, each living object should have at least one copy. We therefore require that in every run, in every global state g in the run, if there are $k \leq \lambda$ memory servers that have failed, then for all $C \in \mathcal{C}$, $|\{M \in \mathsf{Mach} : \mathsf{wg}_g(C) \in \mathtt{group}(M)\}| > \lambda - k$ for every class $C \in \mathcal{C}$. This condition is called the *fault tolerance condition.*

Let SC be the set of *search criteria.* Each $sc \in \mathsf{SC}$ is a subset of object. Let *sc-list*: $\mathsf{SC} \to \mathcal{C}^+$ be such that *sc-list*(sc) is an exhaustive list of classes that may contain objects in sc. Formally, we require that for every $sc \in \mathcal{S}$, if *sc-list*(sc) $= C_0, C_1, \ldots, C_n$, then $sc \cap \mathsf{obj\text{-}cls}^{-1}(C) \neq \emptyset$ for each $i = 0, \ldots, n$, and $sc \subseteq \bigcup_{i=0}^{n} \mathsf{obj\text{-}cls}^{-1}(C)$.

4.2 Memory Servers

For every machine $M \in \mathsf{Mach}$, the memory server that resides in M supports three atomic operations: $\mathtt{store}_M$ takes an object and stores it in the memory. Objects are stored according to their object class and in the order in which the `stores` occur. $\mathtt{mem\text{-}read}_M$

takes an object class C and a search criterion sc, and returns an C-object in sc if M has contains such an object, or fail otherwise. `remove`$_M$ has the same parameters as the `mem-read`; It returns the *oldest* C-object in M satisfying sc and removes it from M if such an object exists, and otherwise it returns fail.

Recall (Section 3) that servers join groups by performing some "initiation" procedure. The procedure is such that each server receives the objects that are in the classes supported by the group it joins. Therefore, when M performs g-join to *g-name*, one of *g-name*'s servers, say M', is chosen to supply M with the appropriate information. That is, M' sends M all the objects that it has in classes whose write group is *g-name*. Until this transfer is completed, no communication to *g-name* is to be processed by any of *g-name*'s members. Hence, when M becomes a member of *g-name*, its state is consistent with the state of all *g-name*'s object with respect to the information (i.e., classes) that *g-name* is in charge of.

When M leaves *g-name* (by performing g-leave), it should "erase" all the *g-name* information it has. The algorithms presented below would work even if servers do not erase the relevant information when they leave groups, however, for sake of space efficiency, servers should erase all information when leaving a group.

Finally, when a machine is restarted, the memory server residing on it should determine which groups it belongs to, and, one by one, g-join these groups.

4.3 Implementing PASO Primitives

We next describe the expansion of each of the PASO macros: `insert`, `read`, and `read&del`. We chose to describe the algorithms in details since their exact description is of crucial importance to both their correctness proof and their cost analysis. There are three cost measures associated with each of the PASO primitives: *msg-cost* measures the message cost, *time* measures the amount of time required to complete the macro, and *work* measure that amount of work the macros requires. Roughly speaking, *time* is the maximum amount of time any single server spends on completing the macro, and while *work* is the sum of the times the various severs spend to complete the macro. The complete macro expansions are described in Appendix A.

When a process wants to `insert` an object o in memory, it issues an `insert`(o) command. When an `insert`(o) is issued, the message "`store`(o)" is gcasted to wg(obj-cls(o)). No response is expected.

To read an object based on a search criterion, a process p computes a the search list of the search criterion. Let M be the memory server residing on the same machine p is on. The process p then goes through the classes in the list. For each class C that M is a member of its write group, M performs `mem-read`(sc, C) and returns the result to p. For each class C that M is not a member of, p gcasts wg(C) a request to `mem-read`(sc, C). The procedure terminates when p receives some non-fail response or when all classes return a fail response (and then `read` returns fail.)

To implement a *blocking* read (see Section 2), one can use our non-blocking read and busy-wait while cycling among the classes. This strategy may be inefficient when only a small number of the requests are expected to be satisfied. An alternative to busy-waiting is to leave read-message markers at nodes supporting each class. There are also hybrid approaches in which read-markers are left and then expired.

	msg-cost	*time*	*work*
`insert`(o)	$\|g\|(2 \cdot \alpha + \beta\|o\|) + \alpha$	$I(\|\mathsf{live}(\mathsf{obj\text{-}cls}(o))\|)$	$\|g\| \cdot time(\texttt{insert}(o))$
`read`(sc) ($M \in C$)	0	$Q(\|\mathsf{live}(C))$	$Q(\|\mathsf{live}(C))$
`read`(sc) ($M \notin C$)	$\|g\|(2 \cdot \alpha + \beta(\|sc\| + \|r\|))$	$Q(\|\mathsf{live}(C)\|)$	$\|g\| \cdot time(\texttt{read}(sc)$
`read&del`(sc)	$\|g\|(2 \cdot \alpha + \beta(\|sc\| + \|r\|))$	$D(\|\mathsf{live}(C)\|)$	$\|g\| \cdot time(\texttt{read\&del}(sc))$

Figure 1: Costs of PASO Operations

Since the size of the write groups is unbounded, and a `read` entails no changes to the memories, there is some inefficiency involved in gcasting the read requests to all members of the write groups. Suppose that for every class C there is a *read group*, $\mathsf{rg}(C) \subseteq \mathsf{wg}(C)$, such that at every point $\lambda - k < |\mathsf{rg}(C)| \leq \lambda + 1$ where k is the number of non-operational machines in the system. Note that $\mathsf{rg}(C)$ satisfies the fault tolerant condition. Then obviously it suffices to gcast read requests only to the members of the read group.

The `read&del` routine is similar to the `read` routine. However, since all the servers in the write groups have to remove an object once an object is found, there is no reason to deal with requests locally. Just like the `read`, a blocking `read&del` can use busy-waiting and suffers from the drawbacks discussed above. To use markers here is more complicated than in the previous case, and will be studied further in the future.

The cost of all the operations is summarized in Table 1. Since both `read` and `read&del` are implemented by a sequence of blocking operations, when measuring their cost we consider the cost of performing a single non-blocking operation. In Table 1, g denotes the set of processes to whom the gcast is directed, $\mathsf{live}(C)$ denotes the set of live objects in the object class C, $|r|$ is the size object returned (in `read` and `read&del`) if such an object is found and 0 when the operations fails, $I(.)$ is a function that determines the time complexity of the `store` operation, $Q(.)$ is a function that determines the time complexity of the `mem-read` operation, and $D(.)$ is a function that determines the time complexity of the `remove` operation. If a read group is used in `read` operations, then the $|g|$ in the third entry of Table 1 should be relaced by a number that is bounded from above by $\lambda + 1$.

In the full version of the paper we prove:

Theorem 1 *Assume that at any point, for every object class C, $\mathsf{wg}(C)$ satisfies the fault tolerance condition. Then, the* PASO *implementation above satisfies the semantics of Section2.*

5 Optimizing Performance

In this section we present and analyze some simple adaptive algorithms intended to improve the performance of the system. We study the total work incurred during a run of the system. On a bus-based local area network, the total message cost is a lower bound on the time to complete the run, since messages must be sent one-at-a-time. The total work divided by the number of processors is also a lower bound on the time to complete the run.

We do not directly examine message cost or response time in this paper. The on-line algorithm described below for minimizing work can also be used to minimize message cost,

since the relationships between message cost for reads and updates and the message cost for copies is essentially the same, and that is what determines the nature of the algorithm. Response time is a valid concern, and a load-balancing scheme designed to reduce response time is described in [13]. It remains an open problem to design a system with guaranteed good behavior in all three cost measures.

In this paper we concentrate on optimizations local to the management of a given object class. That is, we will attempt to minimize the cost of reading and updating an single object class, in the hope that these local optimizations will lead to global efficiency over all object classes. Since object classes form the basic structures of PASO memory this is a reasonable approach.

Fix an object class C. Assume that all the objects in C are roughly of the same size. Let ℓ denote the number of live objects in C, Depending on the type of queries to be supported, the data structure implementing the local storage for the class may be one of various kinds: a hash table for dictionary queries; a binary search tree for range queries; a linear list for text pattern matching. In fact, several such data structures may be used for a single class. However, *time*(g-join(C)) should almost always be $O(\ell)$ since all is required is to copy the memory containing the data structure as is.

To present the basic algorithm, we make some simplifying assumptions. We assume `read&del` and `insert` operations to C come in pairs, and hence ℓ remains fixed. We assume $I(.) = D(.) = Q(.) = O(1)$, as in a hash table, and normalize the costs with respect to the most expensive of these operations, so that `read&del` or `read` take 1 time unit and joining a group requires K time units, for some positive integer K.

5.1 The Basic Algorithm

The following is a simple scheme to maintain an individual write group. A fixed set $B(C)$ of $\lambda + 1$ machines is chosen as the *basic support* for C. If $m \in B(C)$ is operational, it is required to belong to wg(C). At any point, let rg(C) be the subset of operational machines in $B(C)$ and let $F(C) = B(C) - \text{rg}(C)$ denote the machines in $B(C)$ that are currently failed.

Machines other than those in $B(C)$ may join or leave wg(C) in order to reduce the cost of reads. In general, it is advantageous for machine M to join the write group whenever processes local to M are actively reading object class C, and advantageous for M to leave the write group when the time M spends updating its local copy outweighs the benefits from local reads. To determine when a machine $M \notin B(C)$ should join or leave wg(C), the memory server process $m \in M$ keeps a *cost counter*, $c(C)$. We abbreviate this by c when the object class C is understood. Server m behaves by the following rules.

- If $M \in \text{wg}(C)$, when m serves a `mem-read` generated by a process in M, it does the lookup in its local copy and sets c to $\max\{c+1, K\}$.
- If $M \notin \text{wg}(C)$, when M serves a `mem-read` generated by a process in M, it broadcasts the request to rg(C) and increments c by $\lambda + 1 - |F(C)|$. If $c \geq K$, M performs g-join(wg(C),.) and sets $c = K$. (A process gcasting to rg(C) can learn the value of $|F(C)|$ by having it piggyback on the response to the gcast.)

- When m serves an `insert` or `read&del` request for C (hence $M \in \mathsf{wg}(C)$), c is set to $\min\{c-1, 0\}$. If $c = 0$ and $M \notin B(C)$, M leaves $\mathsf{wg}(C)$.

The Basic algorithm is quite closely related to algorithms for snoopy caching [21] and file allocation on a uniform network [6]. The analysis and extensions below differ, however. We analyze the algorithm using the method of competitive analysis. Definitions and basic proof techniques of competitive analysis are given in Appendix B.

Theorem 2 *The Basic algorithm is* $(3 + \lambda/K)$*-competitive.*

Proof Let σ be a sequence of memory requests from processes in M to object class C. We compare our performance against that of an optimal algorithm for this sequence. Let m_o denote a server $m \in M$ controlled by the optimal algorithm, and m_b denote the server as controlled by our on-line Basic algorithm. If $M \in B(C)$ then the optimum off-line algorithm and the on-line algorithm must control M in exactly the same way, since M must always belong to $\mathsf{wg}(C)$. If $M \notin B(C)$, we analyze the competitive ratio using the following potential function:

$$\Phi_m = \begin{cases} 2c & m_o, m_b \notin \mathsf{wg}(C) \\ 3K - 2c & m_o, m_b \in \mathsf{wg}(C) \\ c & m_o \notin \mathsf{wg}(C),\ m_b \in \mathsf{wg}(C) \\ 3K + \lambda - c & \text{otherwise} \end{cases}$$

where $m_o \in \mathsf{wg}(C)$ (resp. $m_b \in \mathsf{wg}(C)$) is used to abbreviate "the optimal (resp. Basic) algorithm causes M to be in $\mathsf{wg}(C)$. The potential Φ is the sum of Φ_m over all servers m.

A case analysis shows that for each possible case and request, the amortized cost charged to the on-line algorithm is at most $3 + \lambda/K$ times the optimal cost for that request. By standard arguments of amortized analysis, this shows that the total cost to the on-line algorithm is no more than $3 + \lambda/K$ times the total optimum cost. This is essentially the best possible [21]. ∎

The Basic algorithm can be extended to other data structures for the internal memory. In typical data structures (e.g. trees and linked lists), $I(.)$ and $D(.)$ are the of the same order, while $Q(.)$ is more expensive. Normalize insertion and deletion to cost 1 time unit, and let the query cost q time units. (We are still assuming that the size of the object class is fixed at ℓ.) Modify the counter algorithm so that after a read, the counter is increased by $q(\lambda + 1 - |F|)$. The remainder of the algorithm is the same. Letting $\Phi_m = 3K + \lambda - c$ if $m_o \in \mathsf{wg}(C)$, $m_b \notin \mathsf{wg}(C)$, and the same as above in all other cases, we can show that the competitive ratio is $3 + \frac{q\lambda}{K}$.

The Basic algorithm also be extended to handle the general situation in which ℓ, the number of live object in C, changes over time. Each memory server then maintains, in addition to c, a counter k_m which has the current value of K, the current number of time units to g-join class C. Obviously, $c \leq k_m$. Roughly speaking, the algorithm resets itself every time the ratio between join cost and update cost changes by a factor of 2. In resetting, it either doubles or halves K, and then adjusts the ratio q as appropriate. If P belongs to E, then K is kept updated to the current value. If $M \notin \mathsf{wg}(C)$, then k_m is updated by piggybacking the current value of K whenever a read to C is performed.

Doubling K may drive up the value of the potential function for those processors where $m_b \in \mathsf{wg}(C)$ but $M_o \notin \mathsf{wg}(C)$. This can be amortized against the cost incurred by the optimum algorithm in having m_o service the read or delete requests that led to the doubling.

The algorithm remains the same in all other aspects. We can show

Theorem 3 *The doubling/halving algorithm is* $6 + \frac{2q\lambda}{K}$ *competitive.*

5.2 The Support Selection Problem

The result of the previous section gives a provably good on-line algorithm when we have a fixed basic supporting for each object class. It is natural to ask what happens if we relax this requirement. We define the general **Support Selection Problem:** choose on-line a set of machine for $\mathsf{wg}(C)$ so as to minimize total work (respectively, message cost) subject to the constraint $|\mathsf{wg}(C)| = \min\{\lambda+1, n-f\}$, where $f < \lambda$ is the number of failed machines. In other words, if $\mathsf{wg}(C) \leq \lambda + 1$ and a machine supporting C fails, it must be immediately replaced, unless there is no other operational machine. This implies that the initialization period for a machine may vary substantially depending on how many classes it is required to support.

Choosing the right machine can make a big difference to the total work. When a machine supporting class C fails, a cost of $g(\ell)$ is incurred to copy the memory state for C to a non-failed machine, where ℓ is the current number of live objects in class C and $g(\ell)$ is the size of the data structure storing those objects. If a frequently failing machine is often chosen, the system will spend a lot of time copying memory states. As in the basic support problem, allowing many processors to join the write group may reduce overall work by localizing reads.

We observe that support selection is at least as hard as a classical on-line problem: virtual memory management. The *virtual paging problem* is defined as follows. A machine has a virtual memory of n pages, but a physical cache can only hold $k < n$ pages at a time. A page of memory cannot be read or written unless it is contained in the physical cache. If a program generates an access to a page i not in cache, a *page fault* occurs. Some page j currently in cache must be ejected, and replaced by page i. The goal of a virtual memory management algorithm is to choose the pages to eject so that the total number of page faults is minimized.

Theorem 4 *No deterministic algorithm for support selection can be better than* $n - \lambda - 1$ *competitive. No randomized algorithm can be better than* $\log(n - \lambda - 1)$ *competitive.*

Proof (Sketch.) Assume that $n \geq 2\lambda + 1$. Hence C can always be supported by $\lambda + 1$ machines that are either in $\mathsf{wg}(C)$ already or else have servers that are about to g-join $\mathsf{wg}(C)$. Also, assume ℓ is fixed. An instance of the virtual paging problem is easily converted to an instance of support selection. Map each page i of memory to a machine M_i. Let $\lambda+1 = n-k$. Page i will be in the cache if and only if processor $M_i \notin \mathsf{wg}(C)$. Each reference to page i is mapped to a failure of M_i. If M_i is in $\mathsf{wg}(C)$, then the support selection algorithm must replace it by some other unfailed machine $M_j \notin \mathsf{wg}(C)$. This corresponds to an ejection of page j from the cache. Then M_i is brought back to life, outside the write group. This corresponds to a loading of page i into cache. The total cost incurred by the support

selection algorithm in copying state, divided by the copy cost $g(\ell)$, gives the total number of faults in the corresponding instance of paging. The proof is completed by applying lower bounds for paging [26, 12]. No deterministic paging algorithm is better than k competitive and no randomized algorithm is better than $\log k$ competitive. ∎

Although Theorem 4 gives quite high lower bounds for the competitiveness of support selection algorithms, we can take some comfort from the fact that there are many deterministic and randomized heuristics for paging whose observed behavior on real-life instances is much better than k competitive. One of the best known rules for paging is "LRU": on a page fault, eject the least recently used page from memory. In the support selection problem, this rule translates to "LRF": if a machine in the write group fails, replace it by the least recently failed machine. This heuristic uses the relatively plausible assumption that the longer a machine stays up, the more reliable it is. LRF does not completely solve support selection, however, as it does not permit expanding the write group. At this time finding a competitive support selection algorithm whose implementation does not impose excessive communication and coordination cost remains an interesting open problem. On a more general level, extension of any of our results to wide-area networks is an very interesting open problem.

Acknowledgement: We would like to thank David Gelernter for introducing us to the problem, Nick Carriero for his patience and guidance through the various Linda compilers as well as many helpful comments and criticism, and Eric Freeman for his contribution to an earlier model and his current work in implementing the new model.

References

[1] G. Agha and C. Callsen. Actorspaces: An open distributed programming paradigm. In *Proc. 4th ACM SIPLAN Symp. on Principles and Practice of Parallel Programming, San Diego*, May 1993.

[2] B. Anderson and D. Shasha. Persistent Linda. In J. B. Banatre and D. L. Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*. Springer-Verlag, Berlin, 1992.

[3] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 164–173, 1993.

[4] D. E. Bakken and R. D. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *Proc. 11th IEEE Int. Symp. Fault Tolerant Computing*, pages 248–255, 1991.

[5] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in linda. Technical Report TR93-18, Univ. Arizona Dept. Computer Science, 1993.

[6] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. ACM Symp. on Theory of Computing*, pages 39–50, 1992.

[7] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. Computer Systems*, 5(1):47–76, Feb. 1987.

[8] L. Borrman, M. Herdieckerhoff, and A. Klein. Tuple space integrated into Modula-2, implementation of the Linda concept on a hierarchical multiprocessor. In Jesshope and Reinartz, editors, *Proc. CONPAR '88*. Cambridge Univ. Press, 1988.

[9] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Trans. on Programming Languages and Systems*, 13(1):99–123, 1991.

[10] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, April 1989.

[11] C. C. Douglas. A tupleware approach to domain decomposition methods. *Applied Numerical Mathematics*, 8:353–373, 1991.

[12] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. Young. On competitive algorithms for paging problems. *Journal of Algorithms*, 12:685–699, 1991.

[13] E. Freeman, D. Gelernter, J. Westbrook, and L. Zuck. A fault tolerant paso for lans (extended abstract). Technical Report YALEU/DCS/TR-1012, Yale University, Feb. 1994.

[14] D. Gelernter. personal communication, 9 1993.

[15] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with piranha. In *Proc. 1992 ACM Int. Conf. Supercomputing*, July 1992.

[16] D. Gelernter, J. Westbrook, and L. Zuck. Towards an efficient fault tolerant PASO memory system. Technical Report YALEU/DCS/TR-1000, Yale University, Dec. 1993.

[17] I. Isis Distributed Systems. The Isis distributed toolkit, version 3.0. Systems Reference Manual, 1992.

[18] S. Jagannathan. TS/Scheme: Distributed data structures in Lisp. In *Proc. 2nd Workshop on Parallel Lisp: Languages, Applications and Systems*. Springer-Verlag LNCS, Oct 1992. Also published as: NEC Research Institute Tech Report: 93-042-3-0050-1.

[19] S. Jagannathan. personal communication, Apr 1994.

[20] S. Kambhatla and J. Walpole. Recovery with limited replay: fault-tolerant processes in linda. Technical Report CS/E 90-019, Oregon Grad. Inst., Dept. C.S. and Eng., September 1990.

[21] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[22] B. Liskov. Position paper. The panel discussion at OLDA2, Vancouver, October 18 1992.

[23] M. Manasse, L. A. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. In *Proc. 20th ACM Symposium on Theory of Computing*, pages 322–333, 1988.

[24] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proc. OOPSLA '88*, pages 276–284, Nov 1988.

[25] G. Roman, K. Cox, C. D. Wilcox, and J. Plun. Pavane: a system for declarative visualization of concurrent computations. *J. Visual Languages and Computing*, 3:161–193, 1992.

[26] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[27] J. Walpole and S. Kambhatla. Replication issues for long-lived parallel computations in a loosely-coupled distributed environment. In *Proc. Workshop on Management of Replicated Data*, 1990.

[28] A. S. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of linda. In *Proc. 9th IEEE Int. Symp. Fault Tolerant Computing*, pages 199–206, 1989.

[29] L. Zuck. The semantics of PASO systems. Unpulished manuscript, final version in preparation, Sept. 1993.

A Macro Expansions

```
% Macro expansion for read(sc)
begin
  C' := sc-list(sc).
  for each C ∈ C'
      begin
        if local server M is in C
          then begin
                  r := mem-read_M(sc, C)
                  if r ≠ fail then return(r)
          end
          else begin
                  gcast(wg(C),"mem-read(sc, C),", r)
                  if r ≠ fail then return(r)
          end
      end
      return(fail)
end

% Macro expansion for read&del(sc)
begin
  C' := sc-list(sc).
  foreach C ∈ C'
    begin
      gcast(wg(C), "remove(sc, C)", r)
      if r ≠ fail then return(r)
    end
  end
  return (fail)
end
 % Macro expansion for insert(o)
 begin
     gcast(wg(obj-cls(o)), "store(o)", )
 end
```

B Competitive Analysis

A standard tool for studying on-line algorithms is competitive analysis [23, 21]. For a given sequence of operations σ, the cost incurred by the on-line algorithm is compared against the minimum possible cost had the algorithm made all the right decisions at the right time. The cost incurred by algorithm A on σ is denoted $A(\sigma)$ and the minimum possible cost is denoted $\mathrm{OPT}(\sigma)$. An on-line algorithm A is said to be *c-competitive* for a constant c if for all sequences σ,

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + B$$

The value B is a constant independent of σ that accounts for initialization cost.

Let A denote the algorithm for PASO that consists of the individual algorithms for the `insert`, `read`, `read&del`, g-join, and g-leave operations. Competitive analysis proceeds by comparing simultaneous runs of A and the optimum algorithm for σ, OPT, on the request sequence σ. The actions of A and OPT are partitioned into events, which together account for all costs to both algorithms. The first kind of event is the servicing of an `insert`, `read`, or `read&del` request by both A and OPT. The second kind of event involves A or OPT making processors join or leave groups. The second kind of event is OPT making a processor join or leave a group. The third kind of event is a processor failing or recovering. We define a *potential function* Φ that is a non-negative real-valued functions of the states of the systems as maintained simultaneously by A and OPT.

Each event has an actual cost to OPT, and an *amortized cost* to A, defined to be the actual cost of the event to A plus the change in the potential function at the t^{th} step, $\Delta\Phi = \Phi_t - \Phi_{t-1}$. The potential Φ is always non-negative, and $\Phi_0 = 0$. This properties imply that if we sum the amortized costs for each event, the total amortized cost is an upper bound on the total actual cost incurred by A. To prove a competitive bound, therefore, it suffices to show that for any single event the amortized cost to A is bounded by c times the actual cost to OPT.