TR-1012

A Fault Tolerant PASO for LANs (Extended Abstract)

Eric Freeman David Gelernter Jeffery Westbrook Lenore Zuck

Department of Computer Science
Yale University
New Haven, CT 06520

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 appeared in a number of systems, and provides a convenient and useful mechanism for parallel and distributed applications. PASO memory is very amenable to adaptive implementations that relocate data objects in response to changing network configurations and access patterns, and so makes a good candidate as an efficient, fault-tolerant storage system. We define the semantics of PASO memory; give a basic design strategy; present one implementation and prove its correctness; and discuss potential techniques for improving efficiency.

Contact author: Lenore Zuck, Department of Computer Science, Yale Station 2158, New Haven, CT 06520-2158. E-address: zuck@cs.yale.edu, telephone: 1-203-432-1236, fax: 1-203-432-0593.

1 Introduction

This paper discusses the design of fault-tolerant storage based on 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. The design is targeted at local area networks connected by a bus such as an Ethernet.

An object in a PASO 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. A `read` takes an object template (search criterion) specifying acceptable values for each field, and returns any one object matching that template. Both `read` and `read&del` are blocking, *i.e.*, they cannot return until they succeed in finding a matching object. 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.

The local area network consists of n machines, each of which has local memory and supports a set of processes. A process may be either a *compute process* or a *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. It is responsible for serving the PASO requests generated by compute processes. The primary type of fault we consider is *fail-stop* errors, in which a machine crashes and all processes on that machine are killed. We assume a communication system that handles communication faults such as message loss and corruption; such communication systems have been studied extensively in previous research. The restriction to local area networks connected by a bus eliminates the possibility of network partitions caused by communication link failures.

The PASO model is a hybrid of the message passing and the shared address space approaches to inter-process communication. Like a shared address space, a PASO memory hides the physical location of data. A programmer simply manipulates an abstract data space. A PASO memory also preserves some of the efficiency of message passing, allowing the programmer to distinguish local computations from potentially expensive communication/coordination actions. 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 [7], Scheme [16], Prolog [6], distributed object-oriented systems [19], Modula-2 [17], program visualization systems [12], math libraries [10], and as part of other coordination mechanisms [1, 18]. They have proven to be an effective basis for parallel computations, distributed databases, groupware and related software systems[7, 8]. 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.

As observed in [3], one can separate the problem of fault-tolerant computation into two issues. The first is the design of parallel programs that are fault-tolerant given the assumption of a stable storage. This area is well studied and there are many approaches based on checkpointing, message logging, and rollback recovery (*e.g.*, see [15]). The second issue is the design of the

stable storage. It is on this second issue that we focus. We take some predefined constant $\lambda < n$ and assume that at any given time at most λ machines can simultaneously fail. The PASO memory is reliable if throughout any series of faults the abstract object space remains unchanged and all active processes have a consistent view of the object memory.

Current PASO-like systems either provide no fault-tolerance or provide basic fault tolerance at the cost of substantial overhead (see, *e.g.*, [3, 7, 22]). One argument against fault-tolerance is that the obvious benefits of preserving data in the face of failures are outweighed by the loss of efficiency when errors are infrequent. Our thesis is that both goals of fault-tolerance and efficiency can be achieved. The requirement of fault tolerance implies that data will need to be adaptively replicated in response to machine failures. But since we are forced to relocate data, we may as well commit to inherently adaptive data management schemes and take advantage of the potential optimizations that adaptive schemes offer.

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* [13]. Today's ubiquitous workstation 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. Our fault-tolerant techniques will allow a distributed memory to retire gracefully from workstations that are being reclaimed for personal use, and expand onto nodes that become available. We believe that adaptive-parallel programs executing on networked multiprocessors will be one of the most important arenas for high-performance computing over the next decade. For more details on the Yale PASO project see [14].

Section 2 provides an overview of the semantics of PASO memories. Section 3 describes the physical model, communication model, and fault model. Section 4 describes the basic strategy for memory management. Section 5 describes a particular correct implementation of a PASO system. Section 6 discusses strategies for improving efficiency. We conclude in Section 7 with summary and description of present and future work.

2 The PASO Semantics

We give an overview of the semantics of PASO memories. A detailed discussion is in [23].

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 also 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 the state of the object space. We assume some set Φ of propositions and an evaluation function that determines whether each proposition is true or false in each of the global states. Of special importance to us are the propositions $\mathsf{pre}(o)$, $\mathsf{live}(o)$, and $\mathsf{dead}(o)$, for every $o \in \mathcal{O}$, denoting whether o is prenatal, alive, or dead. With each global state we associate a partition of $\mathcal{O}$ into three sets, PRE, LIVE, and DEAD according to the state of the objects in the global state. An *initial state* is a global state where all objects are prenatal (i.e., PRE = $\mathcal{O}$) and all processes are at their initial local states. 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, denoted by ι, and its return, denoted by ρ. For `read` and `read&del` commands, we sometimes abuse notation and use two arguments for ρ, the first denoting the terminating command, and the second denoting the result. For example, $\rho(\texttt{read\&del}(sc), o)$ is the return of a `read&del` with search criterion sc, whose result is o.

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 $r = s_0, \tau_0, s_1, \ldots$ of alternating global states and joint transitions, starting with a state, and, if finite, ending with a state, such that s_0 is an initial state and every state s_{i+1} is the successor of s_i under the successor function of τ_i.

Properties (A1)–(A3) below are some of the properties that should be satisfied by every run r of a PASO system. Property A1 describes the life cycle of an object in r. Property A2 describes what in r determines an object's life. Property A3 describes the processes in r.

A1 A prenatal object may remain so forever or become alive. A live object may live forever or die. A dead object remains dead. An object may become alive at most once, and may die at most once.

A2 An object o may become alive only after a transition includes $\iota(\texttt{insert}(o))$. It may later die after, and only after, a transition which includes $\rho(\texttt{read\&del}(sc), o)$.

A3 The individual run of each process as determined by r is indeed plausible run of the process. In particular, for every process p, if r_p denotes p's run as determined by r, then every ρ in r_p is the immediate successor of the corresponding ι in r_p. Also, every $\iota(\texttt{insert})$ in r_p is immediately followed by a corresponding ρ. Obviously, a PASO command of p blocks when its ι is the last element in r_p.

It remains to describe the rules of each of the PASO commands. We require that an object becomes alive at some time after its `insert` is issued. The rules of `read` commands are somewhat more complicated since they describe both the correctness of objects returned by `read` and the conditions under which `read` commands may and may not block. We require that a `read` command returns an object that satisfies the search criterion and is alive at some time in between the issue and the return of the `read`. A `read` should not block if there is an object that satisfies the search criterion and is alive from some time onward. It may block in all other cases. The rules of the `read&del` command are similar to the rules of the `read` command. We do require, however, that an object that is returned from a `read&del` eventually dies.

3 The Physical Model

As described in Section 1, a local area network consists of n machines, each of which has local memory and each supports a set of processes. A process is either a compute process or a memory server. A memory server manages some collection of PASO objects stored in the local memory of the machine. For simplicity we assume that each machine hosts exactly one memory server. Machines may crash and leave the system, and then be fixed and re-join the system. When a machine crashed, the memory server hosted by it also crashes. We assume that a crashing memory server leave the system and never return. When the hosting machine re-joins the

system, the hosted memory server is treated as if it is a new memory server. The set of memory servers is therefore dynamic. Assume some universal set of memory servers. At any point, we assume that $\mathcal{M}$ denotes the subset of memory servers that are active (i.e., hosted by operational machine) at that point.

3.1 Communication Mechanism

All communication occurs by means of a simple primitive, **gcast**. The **gcast** primitive is derived from the ISIS system for robust distributed communication [5], which we are using as a basis for experimental implementations of PASO memory.

A **gcast** broadcasts a message to all members of a specified *group*, a construct roughly analogous to a mailing list. At any time, a given process may join or leave a group. The operation **gcast**(*name*, *msg*, *resp-type*, *resp*) broadcasts a message *msg* to each process currently subscribing to the group identified by *name*. The flag *resp-type* is either **a** or **s**. If it is **a** then control returns to the issuing process only after all the group subscribers respond to the broadcast. If it is **s** then control returns to the issuing process as soon as one group member responds. In this latter case, while the process receives only one response, the response is sent only after all group members are ready to send. Hence, response type **s** is used when the process needs only one copy of the response; we use this form to to minimize contention as not all responses need to be sent. Responses are stored in the local variable *resp*.

Let **Names** be the (finite) set of group names. For any point in every run, let **group: Names** $\rightarrow 2^{\mathcal{M}}$ be a mapping such that **group**(*name*) is the set of memory servers belonging to group *name*. The communication subsystem that implements **gcast** is responsible for maintaining this mapping. The use of group names thus provides a simplifying level of indirection for the compute and memory servers.

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 **gcast**s 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. (The broadcast primitives of ISIS provide for all of these properties.)

In the formal model, we refine the notion of runs (see Section 2) to account for the groups membership and **gcast** primitive in the obvious way. Details are deferred to the full version of the paper.

3.2 Fault Model

We expect our system to tolerate up to λ simultaneous *fail-stop* 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 that is associated with it fails. Failed memory servers are assumed to leave the system and never return. Similarly, memory servers that join the system are assumed to be new.

Once a faulty machine re-joins the system, the memory server performs an initialization phase. During the initialization phase, the server obtains copies of the objects that it should store (see Section 4). Hence, this phase is expected to be rather lengthy. We consider a machine in its initialization phase faulty, since it cannot answer all queries correctly. We assume that at any time, there are at least $n - \lambda$ non-faulty machines in the system.

4 Memory Management

In order to determine where objects are stored and how they are searched for, we partition the object space and the search criteria space into classes. Each class of objects is associated with a *write group*—a set of servers each of which stores every live object that belongs to the class. All requests to insert and remove particular objects are therefore made to the write group that is responsible for the class that contains the object. Similarly, every class of search criteria is associated with a *read group*—a set of servers that contains at least one server from every class that may hold an object satisfying the search criteria in the class. Hence, search requests are directed to the read classes, and update requests are directed to the write classes.

The set $\mathcal{O}$ is partitioned into a set of object classes $\mathcal{C}$ by a function **obj-cls**: $\mathcal{O} \rightarrow \mathcal{C}$. We place no restrictions on the number of object classes, nor on the function **obj-cls**. The function may or may not be known to memory servers. It can be predetermined during compilation or generated at run-time. It can be a dynamic function, changing over time. At each point in a run, the live objects in every class $C \in \mathcal{C}$ are replicated across some group of memory servers that is said to *support* C and is called the *write group of* C. The write group of C is denoted **w-grp**(C). The write group of a class is dynamic.

The set of search criteria, SC, is also partitioned into a set of search classes $\mathcal{S}$, by a function **srch-cls**: SC $\rightarrow \mathcal{S}$. Again this function may or may not be known in advance to memory servers, and it may be modified at runtime. Like object classes, each search class is also supported by a group of servers, called the called the *read group* for search class S, and denoted **r-grp**(S).

A given memory server may support multiple read and write groups. In addition, the membership of read and write groups can change over time. Memory servers may fail and recover, joining different write groups. In addition, it may be useful to reassign servers among write groups in order to optimize communication. For example, if compute processes on a machine are frequently accessing a given class C, it may be advantageous for the memory server on that machine to begin supporting C. Then read requests can be handled locally, without using communication. Although read and write groups can change, at all times they must satisfy the *intersection condition*:

> For every search class $S \in \mathcal{S}$ and object class $C \in \mathcal{C}$, if $sc \cap C \neq \emptyset$ for some search criterion $sc \in S$, then **r-grp**(S) $\cap$ **w-grp**(C) $\neq \emptyset$.

That is, if some $o \in C$ satisfies some $sc \in S$, then there is at least one memory server that is in both the write group of C and the read group of S. In addition, the write groups must satisfy the *fault tolerance condition*:

> Let λ be the fault-tolerance parameter. In every run, in every point in the run, if there are $k < \lambda$ memory servers that have failed, then for all $C \in \mathcal{C}$, $|$**w-grp**(C)$| \geq \lambda - k$.

Lemma 1 *In any implementation that correctly tolerates up to λ simultaneous memory server failures, the write groups must satisfy the fault tolerance condition*

Proof The proof follows immediately from the observation that when k memory servers fail, at most $\lambda - k$ additional memory servers can fail at the next point. Hence, when k servers fail, there is at least one correct (non-failing) server in each write group. ∎

5 Algorithms

We now describe a correct implementation of a PASO system that can tolerate up to λ simultaneous faults. The algorithms described here consist of a correct fault-tolerant (up to λ faults) implementation of PASO memory for any read and write groups that satisfy the intersection condition and the fault tolerance condition. The only assumption taken here is that the rate of faults is such that each process can eventually execute part of its code while the system does not experience new faults. Since recovery is assumed to take some time (see Section 3), this assumption is reasonable in our model.

The implementation below makes no assumption about the ratio of inserting and removing objects into the system, the size of the read and write groups, or the nature of the search criteria. Any prior knowledge about one or more of these parameters may imply a different, possibly more efficient, implementation. For instance, many existing PASO-like systems assume that for a given class C, the w-grp(C) = r-grp(C). In this case our `read` and `read&del` algorithms can be substantially more efficient.

Our implementation uses the gcast primitive to the write groups whenever an object is inserted or removed. In the full version of the paper we show that our assumption about the system and the properties of gcast guarantee that for every object class C, it is always the case that all the servers that support C have the same view of C, and other servers do not have any view of C.

5.1 Memory Servers

Each memory server M is defined by an abstract data type. A formal definition of the memory servers will be given in the full version of this paper.

Every $M \in \mathcal{M}$ supports four atomic operations: $\texttt{store}_M$ takes an object and stores it in the memory. Roughly speaking, objects are stored in the order in which the `stores` occur. $\texttt{search}_M$ takes a search criterion sc and returns an object class identifier C if there is an C-object in M that satisfies sc and fail otherwise. If there is more than one C-object then $\texttt{search}_M$ returns the oldest such object. $\texttt{mem-read}_M$ is similar to $\texttt{search}_M$, only it returns an object satisfying sc instead of a class identifier. $\texttt{remove}_M$ takes a search criterion sc and an object class C. It returns the oldest C-object in M satisfying sc and removes it from M if such an object exists, and fail otherwise. For the purpose of this paper we assume that each memory server can store infinitely many objects.

5.2 Insert

When a process wants to `insert` an object o in memory, it issues an `insert`(o) command. When an `insert`(o) is issued, the process uses obj-cls to determine the class C that o belongs to, and then w-grp to determine the current write group of C. It then issues a gcast to the relevant memory servers with a request to write o onto them.

Figure 1 describes the code that is generated when a process p issues a `insert`(o) command.

In the full version of the paper we prove:

Lemma 2 *The* `insert` *routine in Figure 1 is correct. That is, it satisfies the semantics of the* PASO `insert` *operation.*

```
% Macro expansion for insert(o)
begin
   gcast(w-grp(obj-cls(o)), "store(o)", s)
end
```

```
% Macro expansion for lookup(sc)
begin
   found := false
   while ¬found
   begin
      gcast(r-grp(sc), "search(sc)", a, resp)
      r := {set of non-fail responses}
      if r ≠ ∅ then found := true
   end
{r is all classes containing some o ∈ scr}
return(r)
end
```

Figure 1: The `insert` and `lookup` macro expansions

5.3 Read

To read an object based on a search criterion, a process may broadcast a read request to the appropriate read group. If there are many servers that contain a matching object (*i.e.*, if the criterion covers objects in many object classes), objects are rather large, and communication cost grows fast with the message length, then the cost of the answers may be prohibitively expensive.

We therefore decided to request servers in the read group to give only some minimal information that would help in finding objects that satisfy the search criterion. Namely, servers are only requested to supply the process with a list of classes the support and that have a live object satisfying the search criterion. This is accomplished by means of the routine `lookup`, described in Figure 1.

Given the set of class identifiers returned, the process then successively broadcasts to the servers in each write-group(*name*) a request to read a C-object satisfying *sc*. If such a C-object exists in w-grp(C), one such object is returned. Otherwise, the next class is tried. If all classes returned by `lookup` fail to have an object that satisfies the search criteria then the process goes back to `lookup`. The `read` expansion is presented in Figure 2.

Obviously, if the `read` returns, it returns an object the satisfies the search criterion and is alive at some point during the execution of the `read`. From the code of `read` and the assumption that `search` returns the class with the oldest object satisfying *sc*, it can be shown that the `read` doesn't block unless no object remain in the system long enough.

In the full version of the paper we formally prove:

Lemma 3 *The* `read` *routine in Figure 2 is correct. That is, it satisfies the semantics of the* PASO `read` *operation.*

Both `lookup` and `read` routines use busy-wait to block. This is potentially inefficient if many processes are blocking for long periods of time, and may end up flooding the network with messages. On the other hand, if most requests are expected to be satisfied, and blocking is rare, then the overhead is small. An alternative to busy-waiting is to leave read-message markers at nodes supporting each class. Then, whenever a new object is created, the supporting nodes

```
% Macro expansion for read(sc)
begin
  while true
  begin
    C := lookup(sc)
    foreach C ∈ C
      begin
        gcast(w-grp(C),
              "mem-read(sc, C)", ss, r)
        if r ≠ fail then return(r)
        {r is some existing o ∈ sc}
      end
  end
end
```

```
% Macro expansion for read&del(sc)
begin
  while true
  begin
    C := lookup(sc)
    foreach C ∈ C
      begin
        gcast(w-grp(C),
              "remove(sc, C)", s, r)
        if r ≠ fail then return(r)
        {r is some existing o ∈ sc}
      end
  end
end
```

Figure 2: The `read` and `read&del` macro expansions

check to see if it matches any pending search request. In this case, however, additional effort is needed to guarantee that the request markers are not lost in node failures. An appealing hybrid of these schemes is to have each message marker expire after some time period Δ, and have waiting reads send a fresh set of read requests every Δ time units.

5.4 Read&Delete

The `read&del` routine is similar to the `read` routine. The only difference is that in the `read` routine, after `lookup` returns a set of classes that contain an object satisfying the search criterion, a process requests each write group to `mem-read` one such object, while in the `read&del` routine, the process requests the write group to `remove` such an object.

The `read&del` expansion is presented in Figure 2.

Note that, similar to the `read` algorithm, `read&del` also uses busy-waiting and suffers from the drawbacks discussed above.

In the full version of the paper we prove:

Lemma 4 *The* `read&del` *routine in Figure 2 is correct. That is, it satisfies the semantics of the* PASO `read&del` *operation.*

From Lemmata 2, 3, and 4, we formally prove in the full paper:

Theorem 1 *For any read and write groups that satisfy the intersection and the fault tolerant conditions, the implementation described here satisfies the* PASO *semantics and can tolerate up to λ failures.*

6 Managing the Read and Write Groups

The basic framework of read and write groups is very flexible both in the number of groups and the assignment of memory servers within them. The organization of the groups can be tailored

for specific kinds of application, so that the types of searches occurring in the application can be performed efficiently.

Any implementation must provide

1. A description of the read and write classes.

2. An algorithm for assigning processors to read and write groups such the intersection condition is satisfied and the fault tolerance condition is satisfied.

3. An algorithm to compute the mapping from objects to write groups and the mapping from search criteria to read groups.

We now describe an implementation called "Load Balancing." The primary goal of load balancing is to evenly distribute the set of tuples across write groups. The scheme is intended for situations where the search criteria are arbitrary predicates that may be satisfied by objects from almost any object class. Recall that n is the number of machines. Let $m = \delta n$, where $0 < \delta \leq 1$ and $1/\delta \geq \lambda + 1$. The parameter δ can be tuned as desired for efficiency. We assume m is an integer. Note that $\lambda < n/m = 1/\delta$.

Let h_w be a hash function mapping $\mathcal{O}$ to $[1..m]$, chosen uniformly at random from a *universal* family of hash functions [9]. Similarly, let h_r be a hash function mapping SC to $[1..1/\delta]$, also drawn at random from a universal family. Use h_w to partition $\mathcal{O}$ into object classes $C_1, C_2, \ldots, c_m$ so that obj-cls$(o) = C_{h_r(o)}$. Similarly, use h_r to partition SC into search classes $S_1, S_2, \ldots, S_{1/\delta}$ so that srch-cls$(sc) = S_{h_r(sc)}$. (This assumes that search criteria are encoded in a standard way.)

At time t, let live(t) be the set of all alive objects. Then by applying standard results on universal hash functions, we have that the expected size of class C_i is live$(t)/m$. Furthermore, the expected size of the largest class is $|\text{live}(t)|\frac{\log n}{m \log\log n}$, under the assumption that the hash function achieves uniform random hashing [9]. Similarly, if S_t is the set of active searches at time t, the expected number occurring in read group r_j is $|S_t|\delta$.

Next we consider the assignment of servers to read and write groups. In the simplest approach, processor p_i is always assigned to support object class i mod m. Should p_i fail and then recover, it will be reassigned to the same write group. The support of a class can decrease to 1, but no less, hence the fault-tolerance condition is met. At any time, the set of servers assigned to write group i are also assigned to all $1/\delta$ read groups, so that the read groups are evenly distributed over the servers. Specifically, if there are k servers in the write group i, then each server belongs to $O(1/\delta k)$ read groups. If p_j fails, then the remaining processors in write group j mod m join the read groups to which p_j formerly belonged. Similarly, if p_j returns, then it joins some read groups currently supported by other processors in class j mod m, and those other processors resign from those read groups. There are various on-line load-balancing algorithms that can be used to determine which processors join what groups while guaranteeing that the maximum load is $O(1/\delta k)$ (See, *e.g.*, [20]).

An `insert` of object o is done by broadcasting an insertion instruction to all members of the write group $C_{h_w(o)}$. A `read` on search criterion sc is done by a broadcast to the read group $S_{h_r(sc)}$, waiting for responses from the processors in the group. A `read&del` is done by a broadcast to a read group to find a matching object, followed by a broadcast to a write group to delete the object. This are easily done with **gcast**'s by associating each write group with a mailing list name.

Let us assume that the bulk of the running time can be ascribed to computations within local memory, *i.e.*, that communication overhead can be ignored. In the case of very general

search queries, this assumption is not unwarranted. Consider the time spent managing the local memories. We make a simplifying assumption: if a local memory contains ℓ objects, then the time to insert or delete an object o is $O(f(\ell))$ for some f independent of the size of o, and that the time to search the local memory using search criterion sc is $O(g(\ell))$, for some function g independent of sc. The functions f and g depend on the nature of the searches being performed and the data structure used to store the objects. We assume that f and g are at worst linear in ℓ. Since a search criterion and an object can be compared in constant time, a simple linked list implementation will run in time $O(\ell)$ per operation.

Suppose W `insert` operations are performed. The write groups can perform insertions in parallel. Given uniform hashing, the expected number of objects assigned to class $1 \leq i \leq m$ is W/m, expected time per local insertion or deletion is $O(f(W/m))$, and the expected time per local search is $O(g(W/m))$. The total parallel time required to insert all objects is determined by the maximum size of a class. As discussed above, the expected value of the maximum size is $O(\frac{W \log n}{m \log \log n})$, implying that the total parallel time is $O\left(\frac{W}{m}\left(\frac{\log n}{\log \log n}\right)^2 f(\frac{W}{m})\right)$.

Suppose R `read` operations occur. In the absence of faults, the expected number of `read` requests assigned to class $1 \leq j \leq 1/\delta$ is δR. Using the bound on the expected maximum number of requests to any one read group, we conclude that the expected total parallel time for all reads is $O\left(\delta R g(\frac{W}{m})\left(\frac{\log n}{\log \log n}\right)^2\right)$.

If faults occur, then in the worst case, all but one of the machines in a given write group fails permanently. Then any read requests satisfied by objects in that class must be satisfied by that one processor. This increases the total read time by a factor of $1/\delta$. If faults occur uniformly at random, however, with each processor being equally likely to fail, then at any time the expected number of processors per write group is $(1/\delta) - (\lambda/m)$. For $\lambda = O(m^\varepsilon)$, $0 \leq \varepsilon < 1$, the probability of one failure in a given write group is $O(1/m^{1-\varepsilon})$, and the expected running time per read remains of the same order as in the no failure case. A random distribution of failures can be guaranteed by initially randomly permuting the labels on processors.

Note that if the function g is linear in the size of the local memory, then the value of δ does not affect the expected time for serving requests, since $m = \delta n$. In this situation, the total write time is minimized by maximizing δ, subject to the fault tolerance condition. If g is sublinear, however, than minimizing δ reduces read time, and there is a tradeoff in the choice of δ between read and write times.

7 Conclusion

PASO memory offers a simple and flexible model of distributed data storage. It offers many avenues for improvement in efficiency. The implementation described in this paper is intended to handle very general search criteria. In the case of data that can be aggregated into classes subject to simple search based on predetermined key fields (*i.e.*, a distributed data file), adaptive algorithms for "file allocation" [2, 4, 11, 21] may be used dynamically change the support groups for a given class. This would allow processors that are frequently accessing information in a given file to support that file locally, thereby reducing communication overhead. We are currently implementing a PASO system on a testbed network of 150 nodes.

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

[3] 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.

[4] 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.

[5] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpanguo, K.Kane, F. Schmuck, and M. Wood. The ISIS system manual, version 2.1. Systems User Manual, Sept. 1990.

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

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

[8] N. Carriero and D. Gelernter. *How to write parallel programs: A first course.* MIT Press, Cambridge, 1990.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1990.

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

[11] B. Gavish and O. R. L. Sheng. Dynamic file migration in distributed computer systems. *Commun. ACM*, 33(2):177–189, 1990.

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

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

[14] 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.

[15] J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzulo, and I. Traiger. The recovery manager of the system R database manager. *ACM Computing Surveys*, 2(13):223–242, 1981.

[16] 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.

[17] M. H. L. Borrman 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.

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

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

[20] J. Westbrook. On the power of preemption. Technical Report YALEU/DCS/TR-999, Yale University, 1993.

[21] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, 1991.

[22] 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.

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