Yale University
Department of Computer Science

Toward an Efficient Reliable Persistent Associative Shared-Object Memory (Position Paper)

David Gelernter Jeffery Westbrook Lenore Zuck

YALEU/DCS/TR-1000
December 1993

This work was supported in part by ONR under grant N0014-93-1-0573

Toward an Efficient Reliable Persistent Associative Shared-Object Memory
(Position Paper)

David Gelernter Jeffery Westbrook Lenore Zuck
Department of Computer Science
Yale University

Abstract

We describe a fault-tolerant distributed storage system. 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.

This work was supported in part by ONR under grant N0014-93-1-0573

1 Introduction

The goal of the Yale PASO project is to develop fault-tolerant, efficient distributed storage based on a model called *Persistent, Associative, Shared Object* (PASO) memory. A PASO memory stores a set of objects that can be accessed from all nodes in some ensemble of machines by associative search queries rather than by means of a shared address space. We assume that the objects are physically stored among the local memories private to each machine, and that memory operations are implemented by exchanging information over the communication network connecting the ensemble. The faults the memory is intended to tolerate are fail-stop processor crashes in which any objects stored at the failing machine are permanently lost. The thesis of the project is that this fault tolerance can be obtained efficiently through the use of state-of-the-art *competitive on-line* algorithms to manage dynamic replication of data both in response to loss of processors and to changing patterns of data access.

There are several compelling reasons to develop an efficient, fault-tolerant PASO memory. Systems based on the PASO model have had pragmatic success. PASO memory is a good basis for a reliable and efficient parallel programming system. It provides a versatile communication system on which a variety of distributed applications, groupware, distributed database and related software systems can be built.

Our goal is to produce a theoretical design and analysis of efficient, reliable PASO memory, an implementation of the theoretical design, and an empirical performance evaluation of the implementation.

1.1 Definition of PASO Memory

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 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.

1.2 The Advantages of PASO Memory

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 [15], Scheme [26], Prolog [13], distributed object-oriented systems [30], Modula-2 [12], program visualization systems [32], math libraries [19], and as part of other coordination mechanisms [2, 29]. They have proven to be an effective basis for parallel computations, distributed databases, groupware and related software systems[15, 16]. 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 [5], 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 [24]). 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.*, [5, 15, 41]). 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* [23]. 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.

1.3 The Proposed Solution

Our proposed solution derives from a synergy between the use of data replication for fault tolerance and data replication for efficiency.

As stated above, our basic model assumes that the objects in the shared memory are physically stored in the local memories of the machines. nodes in a processor The primary fault we are

concerned with is fail-stop processor errors in which the data stored at a node is irrecoverably lost. For the object memory to survive, some measure of data replication is required.

Data redundancy can also improve efficiency. If many machines are reading the same object, then by replicating copies of that object among several machines, the overhead per machine is reduced as is the communication cost. There is a considerable literature on the use of static replication schemes to improve efficiency. Recently the algorithms community has begun to study adaptive, run-time algorithms for replication. Here the goal is to respond effectively to changing access patterns. Changing configurations and changing access patterns are closely related. Our thesis is that these theoretical techniques for adaptive replication can be used to efficiently manage the redundant data needed for fault tolerance. Furthermore, as the required degree of fault-tolerance is reduced, an implementation of PASO memory using data replication can actually become more efficient than any that now exist.

2 Technical Definitions and Issues

2.1 Network Architecture

The target architecture for PASO sytems is a loosely-coupled network of independent machines, typified by a local-area workstation cluster. The only storage available is the local storage controlled by each machine. This private storage can be stable, as in a hard disk, but for efficiency the memory objects will typically reside in volatile RAM memory. The network is modeled by a *communication graph* where nodes represent machines and edges represent communication links. An edge or node may be marked *inactive*, indicating that the corresponding communication link or processor has failed. Two processes can communicate only when in the communication graph there is a path that consists entirely of active edges and nodes between the nodes representing the machines where the two processes reside.

We limit ourselves to network architectures whose communication graphs are complete, so we exclude errors that partition the network. This model includes all bus-based networks, an important and common family of architectures.

We will use two testbeds for our research. The first is the Yale Computer Science research subnet, consisting of two servers and a number of Sun SparcStations connected by ethernet. The second testbed is provided by the Yale Center for Parallel Supercomputing (YCPS), a joint venture between Yale and IBM. The primary computing power of YCPS resides in a network of six IBM RISC System/6000 servers and ten IBM RISC System/6000 workstations. All 16 computers are connected by ethernet and a high speed (100 MB) IBM switch. In addition, the six servers are also connected by a FDDI ring.

2.2 Pattern Matching and Searching

An object is read or deleted from the PASO memory by means of a `read` or a `read&del` operation whose operand is the template to be matched. The range of templates allowed determines the degree of associativity of the memory. One extreme example is allowing no wildcarding, that is, allowing only templates that fully specify tuples. The degree of associativity obtained then is minimal. Less extreme is insisting that a template consist of a set of fully specified key fields and fully wildcarded data fields. This corresponds to many database applications. More general templates might be of

the form `(1-10, 5, a*)` which would match any object whose first field is an integer between 1 and 10, second field is 5, and third field is any string beginning with 'a'.

Let us assume for the moment that there is only one copy of each object, thus ignoring the issue of fault tolerance. The search problem of satisfying a **`read`** or **`read&del`** request can be broken into two subproblems. The collection of objects is spread across many physical location. The first problem is sending enough messages so that of all nodes containing an object that satisfies the request, at least one such node receives a request message. The second problem is then pattern matching within a node's local storage for a desired object.

The problems of search strategy and optimal data replication are intimately related. For example, one may have a class of objects that will always be accessed using by a template in which the first field is a fully specified key. It may be useful to bundle this class of objects together in a file, or several subfiles, and replicate the entire file over some set of nodes. On the other hand, one may have a class of tuples that will be subjected to very general search. It may be best to scatter these tuples all over the network. Previous work on PASO-like systems shows that the objects generated by parallel programs can often be preassigned to various such classes, based on information generated at compile and link time. For each class, a different technique for managing replicated data may be used.

2.3 Correctness and Efficiency Measures

2.3.1 Correctness

In Section 4.2.1 we gave an informal definition of reliable PASO memory: throughout some series of faults belonging to the fault model, the abstract object space remains unchanged and all active processes have a consistent view of the object memory. The fault model limits us to λ simultaneous errors.

The inherent concurrency of the PASO memory makes a formal definition of correctness somewhat intricate. We shall define a *universal* PASO *implementation*, which generates the set of legal PASO traces—sets of sequences of operations performed on the memory and the results they return. The universal PASO operates in the absence of faults. An implementation of a fault-tolerant PASO memory is correct if each of its traces can be produced by the universal PASO.

2.3.2 Efficiency of a PASO implementation

There are several measures for the performance of a PASO implementation.

Total communication: the total cost of all messages processed by individual processors; Total communication time is a measure of how much overhead the PASO system is placing on the network.

Network contention: the maximum number of simultaneous messages sent through one path of the communication network.

Processor contention: the maximum number of simultaneous messages one processor may receive.

Response time: the amount of time before a (non-blocking) create, read, or delete operation completes; Response time depends on contention.

3 Previous and Related Research

3.1 PASO-like Systems

Network Linda [14] is a PASO-like system that allows limited pattern matching in `read` and `read&del` operations. Each object is stored at a single memory location and efficiency issues are addressed by "partial evaluation" at compile and link time. Assuming a linked image of the entire program is available, one has at linktime a complete list of every access in the program, including the number and type of fields in the object. Using this information a good deal of "proto-matching" can be done at linktime. Because an n-field search pattern with type signature t can only match an n-field, type-t object, the search at runtime can be restricted to exactly those data structures inhabited by appropriately sized and typed objects. Consistently-used constant fields can be pre-matched. When runtime matching is needed and, the compiler can determine whether there is a search key. Special cases include singleton objects with constant fields that function as distributed semaphores, and matrices that are are implemented as collections of objects, whose first fields identify the matrix and indices: e.g. "(MatrixA, 4, 5, $\langle data \rangle$)". The latter kind of objects can be stored in a distributed hash table based on the search key.

In previous work on reliable Linda systems, Xu and Liskov [41] discuss the use of the virtual partition algorithm to maintain the consistency of tuple replicas in the tuple space. This work builds on capabilities that are native to the Argus distributed programming system, upon which we cannot rely. Bakken and Schlichting [5, 6] 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, although the their swap operator would seem to be applicable to a broader range of master-worker programs). Anderson and Shasha's [4] work on Persistent Linda includes support for transactions, but doesn't focus on the problem of reliable distributed tuple spaces. Closest to the work proposed here is Kambhatla and Walpole's [27, 36]. They discuss reliable tuple spaces, and the use of a reliable tuple space to build a reliable application (by means of a "log space," similar to a tuple space, in which information about the status of an ongoing computation is stored). Kambhatla and Walpole make an interesting case for Linda as "a particularly suitable model for fault-tolerant applications" [27], 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.) We expect that many of Kambhatla and Walpole's observations will be useful in the course of the (somewhat more general) work we plan to do.

3.2 Fault-tolerance in Distributed Database Systems

The failures in a distributed system can be roughly divided into communication errors and processing errors. Communication errors which involve a loss, duplication, or reordering of messages transmitted across a link can be overcome if they are appropriately restricted. See [1, 3, 25, 37] for various solutions and impossibility results. Communication errors that involve corruption of data in messages be solved by means of error-correcting codes, again under certain restrictions [35].

As to process failure, most of the research in the area is about the "Byzantine Generals" problem. For a survey of research until 1985 see [21]. Following the terminology of Byzantine generals, we use "fail-stop" or "crash" errors to refer to cases where processes simply stop. A Byzantine failure, in contrast, allows the process to remain active but to send spurious messages of the worst possible

kind.

Many known fault-tolerant protocols for communication errors are incorporated into existing software for distributed communication, such as the ISIS system [9]. Our initial system, described below, actually uses ISIS to handle communication reliably.

Our primary concern is in dealing with fail-stop errors and consequent loss of data. Data replication must be used to avoid loss, and this imposes various problems of maintaining consistency throughout transactions. A number of standard protocols are available for maintaining consistency, assuming reliable communication. Chapter 12 of [8] gives a broad discussion of the handling of failures in distributed databases.

3.3 On-line Optimization and Competitive Analysis

The problem of optimizing the replication and distribution of shared data has been studied in several contexts. The *file allocation* problem is to assign one or more copies of a database file among a collection of nodes. By distributing multiple copies, the time for a node to read data can be reduced, since the node may look at its local copy of the file and avoid an expensive network communication. On the other hand, distributing copies increases the time to perform a write, since all copies must be updated. In static file allocation, one computes a static allocation of data files based on some model of read and write frequencies. Dowdy and Foster[20] give a survey of many different models and algorithms.

There has been recently been considerable theoretical research into *adaptive* algorithms for problems of this nature, as examples of *on-line optimization*. On-line optimization problems involve making on-line decisions in response to changing patterns of memory accesses with the intention of minimizing the ratio of the on-line cost to the best possible cost had the entire future been known. This measure was codified as the *competitive ratio* [28, 34]. A c-competitive algorithm guarantees worst-case behavior that is never more than a factor of c away from optimal. Experience has shown that algorithms with good worst-case behavior typically have good behavior in real-life applications. Examples include algorithms for paging and searching linked lists [34].

Adaptive on-line algorithms respond to changing patterns of access by moving or replicating data to where it can be used most effectively. In the *page migration* problem, pages of shared memory can be migrated around a network of multiprocessors in response to changing locality of reference. Variations of this problem, dependent on the nature of the communication network, have been studied in [10, 11, 18, 33, 28, 31, 39]. In adaptive file allocation [40] both the number and location of copies of the file are allowed to vary. For page migration [11, 38, 17] there are relatively simple adaptive algorithms that guarantee competitive ratios of 3 or less on the total communication incurred in servicing any sequence of read and writes. Adaptive algorithms for file allocation have been presented in [7, 40]. Here the ratio is between 2 and 3 for simple networks such as buses (ethernet) and trees, but can be as bad as $O(\log n)$ for complex point-to-point networks of n nodes. This work does not explicitly address issues of contention nor response time. A bound on total communication does imply a bound on response time, however. Furthermore, in a distributed system allowing the movement of data files also adds the overhead of maintaining a system map that allows nodes to find the nearest copy of the desired data file.

Some of this theoretical work is directly applicable to the restricted searching problems, described in Subsection 3.1, where a collection of homogeneous tuples is to be searched via a fixed key field. We plan to utilize these algorithms in our fault-tolerant PASO memory.

General associative queries, however, require a different model of data access than used in the current theoretical literature. Each memory operation generates a branching collection of requests, not just a single targeted request. There are many interesting open problems, and we hope that our work will suggest useful models and problems for the theoreticians studying on-line algorithms.

4 Current Work

We defined semantics for PASO, a necessary step for specifying a "correct" fault-tolerant PASO system. We also desinged some generic memory management strategies and outlined some algorithms based for special cases of PASO systems. We are currently working on a prototype of a PASO system built on top of ISIS.

4.1 Semantics of PASO

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.

For a detailed discussion of the semantics see [42].

4.2 A Prototype PASO system

4.2.1 Basic Assumption

In the system outlined here, all communication occurs by means of a simple primitive, **gcast**. 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.)

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. 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.

4.2.2 Outline of Stragety

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 $\textsf{obj-cls}: \mathcal{O} \rightarrow \mathcal{C}$. 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 $\textsf{w-grp}(C)$. 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 $\textsf{srch-cls}: \mathsf{SC} \rightarrow \mathcal{S}$. 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 $\textsf{r-grp}(S)$.

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 $\textsf{r-grp}(S) \cap \textsf{w-grp}(C) \neq \emptyset$.

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 $C \in \mathcal{C}$, $|\textsf{w-grp}(C)| \geq \lambda - k$.

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. ∎

4.2.3 Possible Implementation of PASO operations

We describe macros that implement the PASO operations on a system of memory servers that can tolerate up to λ simultaneous faults. The algorithms described here are correct for any allocation of the read and write groups. Their efficiency depends on the ratio of inserting and removing object to the memory. Every memory server M is assumed to be able to perform four basic operations: $\texttt{store}_M$ takes an object and stores it in the memory. $\texttt{search}_M$ 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.

Figures 1 and Figure 2 describe the macro expansion for the basic PASO operations. Both `read` and `read&del` macros use an auxliary macro, `lookup`, described in Figure 1.

```
% 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

We can show:

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.*

```
% 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

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.

For more details on our prototype system and the issues involved, see [22].

5 Future Work

Future work can be separated into short-term and long-term projects.

5.1 Short-Term Work

- Complete the preliminary system of Section 4 and subject it to careful empirical evaluation. Of primary interest is the overhead imposed on the system by fault tolerance, and how much slower it is than a non-fault tolerant system.
- Study various startegies for determining the read and write groups. We expect different assumptions about the ratio of insert/removal of objects to/from the systems, as well as restrcited search patterns, to lead to different assignments of the read/write groups. Some of these strategies may require on-line optimization.

5.2 Long-Term Work

- Generalize the notion of read and write groups. A read group for processor p is the set of processors that p queries when searching for an object. A write group is the set of processors to which that p sends copies of a new object it creates. For general correctness of searching, p requires that its read group intersect all write groups. If two write groups are having little activity, then it may be beneficial to merge them, thereby reducing the size of everyone's read group. This increases fault tolerance. On the other hand, if two groups are experiencing many creations and deletions of objects, then it may be beneficial to split the write group, thereby reducing total communication and network contention. This reduces fault tolerance. Re-allocation of read and write groups is done at run-time as patterns of access change.

- Examine schemes for specialized classes of search objects. As discussed in Section 3.1, it is possible that certain objects may always be searched for using a fixed set of search fields. In that case, objects can be classified by search key into groups, using a hashing scheme, for example. Each groups can be treated as a data file, and we may then apply some of the algorithms for file allocation discussed in Section 3. Fault tolerance can be based upon replication of groups. This allows finer control.

- Evaluate the role of ISIS in the implementation. The capabilities of ISIS provide great convenience, but if its overhead is substantial it may be desirable to investigate designs that do not depend on ISIS.

- Extension of theoretical models of distributed data to capture our problems. Current theoretical models such as that in [7] are overly simple and abstract. Conversely, practical models as discussed in [20] are quite complex and system-specific. Our goal is to develop an easy-to-use model that still captures the essential nature of the problem.

- Design an adaptive PASO implementation for loosely-coupled networks of shared-memory multiprocessors. While previous-generation parallel supercomputers were often made of custom components, current machines tend to use commodity processors. A simple extrapolation of current trends suggests that future high performance computing might well center on an environment in which tightly-coupled multiprocessor nodes, consisting of standard commodity processors and shared local memory, are tied together via a slower local area network. With the growing availability of multiprocessor file servers and workstations, such networks of multiprocessors will be commonplace in the near future. In such heterogeneous communication environment, identical processors may deal with each other via fast shared memory or slower LANs and WANs. Defining orthogonal read and write sets in such an environment poses a complex series of choices.

6 Summary: Efficient, Fault-tolerant PASO Memory

We plan to design an efficient and fault-tolerant PASO memory for distributed local area networks. The system will use dynamic run-time data replication both to guarantee fault tolerance and to give improved efficiency during associative search. The primary faults we are concerned with are processor crashes, either hard or soft, which destroy that portion of the shared memory stored at the processor. A soft crash for our purposes is the deliberate withdrawal of a processor, as in an

adaptively parallel system. The system will take as a parameter a certain degree of fault tolerance, expressed in terms of the number of processors that can be simultaneously unavailable while still guaranteeing that the memory remains correct.

Our thesis is that through the mechanism of data replication we can not only provide fault tolerance but efficiency improvements relative to current systems with neither fault tolerance nor data replication. The PASO model lends itself well to embedding fault tolerance into the basic structure of the memory system. We need not treat fault-tolerance as an issue independent of how the database software functions. A synthesis of fault-tolerant systems and efficient competitive replication algorithms has not been previously attempted, nor has a tunable parameterized degree of fault tolerance.

We hope our work will also contribute a formalization of what it means to be fault tolerant in PASO memory, new algorithms for managing replicated data, and new methods for measuring workload and analyze the effectiveness of such algorithms.

Our project deals with the foundations of parallel and distributed computing. In addition, we plan to achieve an implementation that can serve both as a test bed for new ideas and as a valuable software system in its own right.

Acknowledgements: We would like to thank Nick Carriero and Eric Freeman for many discussions and comments.

References

[1] Y. Afek, H. Attyia, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D.-W. Wang, and L. Zuck. Reliable communication using unreliable channel. to appear in JACM.

[2] 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.

[3] A. V. Aho, J. D. Ullman, and M. Yanakakis. Modeling communication protocols by automata. In *Proc. 20th IEEE Symp. on Foundation of Computer Science*, pages 267–273, 1979.

[4] 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.

[5] 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.

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

[7] 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.

[8] A. Bernstein and P. Lewis. *Concurrency in Programming and Database Systems*. Jones and Bartlett Publishers International, 1993.

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

[10] D. Black, A. Gupta, and W. Weber. Competitive management of distributed shared memory. In *Proceedings, Spring Compcon 1989*, pages 184–190. IEEE Computer Society, San Francisco, CA., 1989.

[11] D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[12] 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.

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

[14] N. Carriero and D. Gelernter. The s/net's linda kernel. *ACM Trans. Comp. Sys.*, May 1986.

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

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

[17] M. Chrobak, L. L. Larmore, N. Reingold, and J. Westbrook. Page migration algorithms using work functions. Technical Report YALEU/DCS/TR-897, Yale University, November 1991. Submitted to JCSS.

[18] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node butterfly parallel processor. In *Proc. International Conf. on Parallel Processing*, pages 531–540. IEEE Computer Society, 1985.

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

[20] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982.

[21] M. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report RR-273, Yale Univ Dept CS, June 1983.

[22] E. Freeman, D. Gelernter, J. Westbrook, and L. Zuck. A fault tolerant PASO for LANs (extended abstract). Submitted to PODS 1994, Dec. 1993.

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

[24] 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.

[25] J. Halpern and L. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 7 1992.

[26] 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.

[27] 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.

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

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

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

[31] G. Pfister and et al. The IBM research parallel processor prototype: Introduction and architecture. In *Proc. International Conf. on Parallel Processing*, pages 764–771. IEEE Computer Society, 1985.

[32] 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.

[33] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.

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

[35] T. M. Thompson. *From Error-Correcting Codes through Sphere Packing to Simple Groups.* The Mathematical Association of America, 1983. The Carus Mathematical Monographs, number 21.

[36] 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.

[37] D. Wang and L. Zuck. Tight bounds for the sequence transmission problem. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 73–83, Aug. 1989.

[38] J. Westbrook. Randomized algorithms for multiprocessor page migration. In *Proceedings of the DIMACS Workshop on On-Line Algorithms, American Mathematical Society DIMACS Series, Vol. 7*, pages 135–150, 1992.

[39] A. Wilson. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. 14th International Symp. on Computer Architecture*, pages 244–252. ACM SIGARCH/IEEE Computer Society, 1987.

[40] O. Wolfson. A distributed algorithm for adaptive replication data. Technical Report CUCS-057-90, Department of Computer Science, Columbia University, 1990.

[41] 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.

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