Fault Tolerant Distributed Shared Memory Algorithms

Michael Stumm and Songnian Zhou
University of Toronto
Toronto, Canada M5S 1A4

Abstract

Distributed shared memory (DSM) has received increased attention as a mechanism for interprocess communication in loosely-coupled distributed systems because of its perceived advantages over direct use of message passing or remote procedure calls. One problem with most DSM algorithms proposed to date, however, is that they do not tolerate faults.

In this paper, we extend four basic DSM algorithms to tolerate single host failures and argue that this degree of fault tolerance is sufficient for most applications. We analyze the performance behavior of the fault tolerant DSM algorithms and show that for some algorithms the additional overhead for fault tolerance is quite small, but that for other algorithms the extra overhead can be substantial and even unpredictable.

1 Introduction

Distributed shared memory (DSM) is a mechanism for interprocess communication in (loosely-coupled) distributed systems, where processes running on separate hosts can access a shared, coherent[1] address space through memory access operations, such as **read** and **write**. To emulate this abstraction, distributed shared memory is implemented in a layer of software between the application and a message passing system.

The advantages of DSM over direct use of message passing and RPC made distributed shared memory to be the focus of recent study [2, 4, 7, 9, 10, 11, 12, 13, 14, 18]. Performancewise, it has been shown that DSM can be competitive to direct use of message passing and in some cases may even result in superior performance [4, 7, 12, 14, 18], despite the fact that DSM is implemented using underlying message passing primitives. For an overview of DSM and its advantages, see [14].

Nevertheless, the DSM algorithms proposed to date also have disadvantages. One of the disadvantages is that they cannot tolerate faults; the fault of a single participating host may cause a portion of the shared address space to be permanently lost.

Fault tolerance is important for many applications which are traditionally not considered to be fault-critical. Consider, for example, a distributed editor application that allows multiple users sitting at different workstations to edit the same document and see the updates of all users in real time. This application is typical of many cooperative shared workspace applications [8], for which distributed shared memory is well suited. Since the shared data is transparently managed by the DSM system, the user has no control over the location of the data so that data critical to all participating users may be lost if a host crashes.

As another example, distributed shared memory has been extensively used for implementing parallel, computationally intensive applications in order to exploit the vast processing power of workstation clusters. Such computations often run for long periods of time. If the shared memory cannot tolerate faults, then the computation must be restarted from the beginning if data is lost due to a host failure.

In this paper, we present and analyze basic algorithms that implement fault tolerant distributed shared memory (FTDSM). Our approach is to take existing, well-known *basic* DSM algorithms and extend them to make them more tolerant of faults. (Most of the DSM algorithms proposed and implemented to date are variations on these basic algorithms.) This approach allows us to directly compare the complexity and performance of the fault tolerant algorithms to that of their non-fault tolerant counterparts.

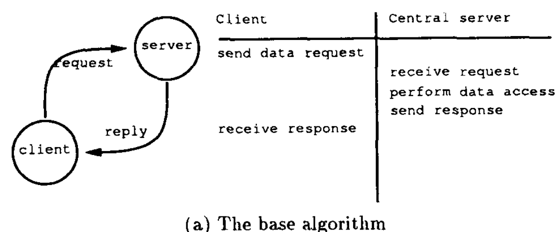
Our aim is to make the resulting algorithms as efficient as possible so that, from a performance point of view, FTDSM remains viable as an IPC mechanism and competitive to message passing. We assume a distributed system environment consisting of a cluster of hosts connected by a local area network, such as an Ethernet, where communication between processors is unreliable and slow relative to local memory access. Hence, the performance of distributed and parallel applications in this environment is strongly affected by communication costs. For our performance analyses and comparisons, we abstract communication costs in terms of the number messages sent and the number of *packet events*, where a packet event is the cost associated with either receiving or sending a small packet (about 1 ms on a Sun 3/50). We also assume that broadcast or multicast communication is available.

We treat fault tolerance somewhat differently than is commonly the case. Instead of trying to survive all types of faults, we attempt to survive only the most common and probable ones and attempt to do so at a reasonable cost. Since we observe that host failures in a distributed system are usually independent of each other[2] and relatively infrequent (say, at most several a day), we design our algorithms to tolerate only single faults. As soon as a host failure is detected, the algorithms enter a *recovery phase*, after which they are again capable of tolerating another fault. 1-resilient algorithms appear desirable because the cost for providing this degree of resiliency is relatively modest (compared to more resilient versions of the algorithm) and because the time it takes to recover from a fault can be made short, so that the probability of a second fault occurring during the recovery phase is small. The cost

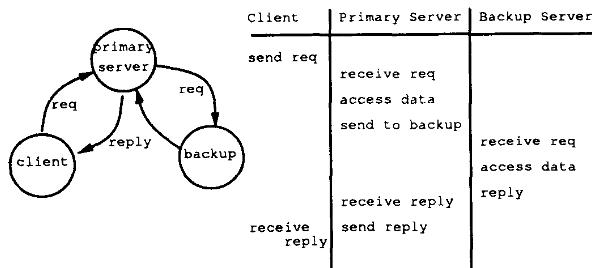
[1] Non-coherent, problem specific distributed shared memory has also been proposed [4], but in this paper we only consider coherent DSM.

[2] This is not true in the case of a power glitch or failure, which may well affect all hosts.

TH0328-5/90/0000/0719/$01.00 © 1990 IEEE



(a) The base algorithm



(b) The fault tolerant algorithm for writes

Figure 1: The Central Server Algorithm

of t-resilient algorithms with $t > 1$ are substantially higher and appears to be acceptable to only the most critical applications.

We achieve fault tolerance by replicating *critical state* onto physically separated hosts; critical state includes DSM data and state information that cannot be otherwise recovered if lost. Access to replicated data must be managed such that DSM remains consistent even during and after a recovery phase. We do not save data onto persistent storage, because of the overhead and delays it would cause. We only consider host failures (detected through timeouts, taking on the order of a few seconds in an Ethernet-based environment) and assume hosts act in a fail-stop way. We assume communication faults are detected (by checksums and sequence numbers) and corrected by underlying communication protocols. Finally, we do not consider network partitions.

2 Fault Tolerant DSM Algorithms

In this section, we extend four basic DSM algorithms to make them fault tolerate. We always first briefly describe the non-fault tolerant, base algorithm, before describing the fault tolerant version. For a more complete discussion of the base algorithms we refer the reader to [14].

2.1 The Central-Server Algorithm

The Central Server algorithm is the simplest strategy for implementing distributed shared memory. A single server is responsible for servicing *all accesses to shared data*.[3] *It maintains the only* copy of the shared data. Both read and write operations involve the sending of a request message to the data server by the process executing the operation, as depicted in Figure 1.a. The data server executes the request and responds either with the data item in the case of a read operation or with an acknowledgement in the case of a write operation.

[3] Data can be partitioned and managed by several servers but the conceptual central server model remains valid.

A simple request-response protocol can be used for communication in an implementation of this algorithm. For communication reliability, a request is retransmitted after each time-out period with no response from the server. A failure condition is raised after several timeout periods with no response. Duplicate requests (due to retransmissions) are detected at the server with sequence numbers, as is typical in communication protocols. Duplicate read requests can be serviced again, since read operations are idempotent, and duplicate write requests are simply reacknowledged. As can be seen in Figure 1.a, this algorithm requires[4] two messages and four packet events.

One way to make this algorithm tolerate single faults is to replicate the data maintained by the data server to a backup server, as depicted in Figure 1.b. With this setup, read requests could be serviced from the primary server as before (or from the backup server, since it has the same data). However, write requests received by the primary server are also sent to the backup server where it is also processed. The primary server does not reply to the client *until it receives* the acknowledgement from the backup.

We now informally argue the fault tolerance of this algorithm by considering the failure of individual processes at different points in time and by describing the steps necessary to recover from failures. If a client fails, then either the data server received the client's most recent write request in which case the system behaves as if the client failed immediately after the write, or the data server did not receive the most recent write request in which case the system behaves as if the client failed without having executed a write.

If the backup server fails, then the primary server will detect this (by repeatedly timing out when trying to send a message), start up a new one and pass it a copy of the shared data space. (If the backup fails while in the process of servicing a write request, then the new backup data copy will reflect that change.) If the primary server fails[5], then the backup server becomes the new primary server, creates and starts a new backup server and passes it a copy the shared data space. The fact that a new process is now serving as the primary data server causes a minor addressing problem: clients will address their next data access request to the failed primary server and therefore will timeout. Clients can resort to broadcasting to determine the identity of the new server.[6] Also, to reduce the need for such broadcasts, the new primary server may broadcast its address to the clients.

In terms of overhead, the fault tolerant extensions to the Central Server algorithm requires two additional messages and 4 additional packet events, as can be seen in Figure 1.b. The maximum time to recover, $t_{recover}$, is on the order of several seconds. If a watchdog process polls each server once a second, and we allow 5 to 6 timeouts at appr. 300 msecs each, then it will take at most 3 seconds to establish the existence of a failure. In addition, 1 to 2 seconds are needed to create and start a new server, as are several seconds to copy the shared data space (at appr. 3 seconds per Mbytes in an Ethernet environment).

2.2 The Full-Replication Algorithm

The Full-Replication Algorithm replicates all data on each host. Read requests are serviced locally, requiring no communication

[4] We assume for this analysis that no communication errors occur, although we do count the acknowledgement message needed to determine this.

[5] A practical way to detect failures is to have a separate watchdog process that periodically polls the server to see if it is still operational. (Of course, the watchdog process also needs to be watched by a process on a different host; in our case, this can be done by the backup server.)

[6] If group communication and process group addressing as, for example, in the V system[5] is available, then the primary server can always be addressed with the same (singlton group) address even if the server processes change.

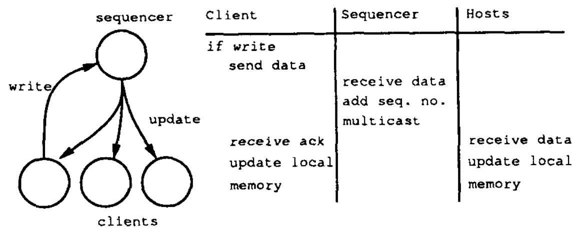


Figure 2: The base Full-Replication algorithm

overhead, but write requests are sent to all hosts so that all copies can be updated. One way to maintain consistent copies of data on all hosts is to globally sequence all write operations. This can be implemented with a central sequencer: write requests are sent to the sequencer, which appends it with the next sequence number before broadcasting it to all hosts. If a host detects a gap in the sequence numbers, then it missed a write request, in which case it requests a retransmission from the sequencer. The broadcast message also serves as an acknowledgement to the writing process. As can be seen from Figure 2, an implementation of this algorithm requires two messages and $S + 2$ packet events for each write request, assuming all messages arrive in their proper order and a total of S hosts.

This algorithm maintains no critical state, since all data is replicated at all hosts. Only a recovery procedure is needed to make it fault tolerant. A client can recover by querying all other participating hosts to determine which host has the most up-to-date copy of the data (by comparing the highest sequence number received at each host) and obtaining a copy of the data from that host. The sequencer can recover in the same way. After a sequencer failure, all hosts must make sure their replicas are up-to-date, possibly obtaining a new copy from other hosts.

The time needed to recover a sequencer failure (i.e., the time it takes to detect a fault, to poll all hosts and for those hosts that do not have an up-to-date copy of data to obtain a new copy) is again on the order of several seconds. In the worst case, the sequencer fails immediately after a broadcast which is only received by one client, requiring all but one host to obtain a copy of the shared data space from the one host that did receive this last update.

To reduce the recovery time, but at the cost of increasing the overhead of every write operation, it is possible to back up the sequencer. A write request sent to the sequencer is first forwarded to a backup sequencer which broadcasts the write operation to all hosts. The sequencer now needs to keep track of the most recently received request from each client and the sequence number assigned to it, so that retransmissions can be detected and forwarded (with the correct sequence number) to the backup processor. Requests for missed write requests by other clients can be serviced by the sequencer directly. The recovery procedure in this case is similar to that of the Central Server algorithm.

2.3 The Migration Algorithm

In the Migration Algorithm, data is always migrated to the host where it is accessed. This is a "single-reader/single writer" (SRSW) protocol, since only the threads executing on one host can read or write a given data item at any one time. Typically, data is migrated between hosts in fix-sized blocks in order to facilitate the management of the data. If an application exhibits high locality of reference, then the cost of block migration is amortized over many accesses, since accesses to a block held locally incur no communication overhead.

One advantage of the Migration algorithm is that it can be integrated with the virtual memory system of the host operating system if the size of the block is chosen to be equal to (or a multiple of) the size of a virtual memory page. If a shared memory page is held locally, it can be mapped into the application's virtual address space and accessed using the normal machine instructions for accessing memory. An access to a data item located in a data block not held locally triggers a page fault so that the fault handler can communicate with the remote hosts to obtain the data block before mapping it into the application's address space. The location of a remote data block can be found, for example, by broadcasting a query to all remote hosts. When a data block is migrated away, it is removed from any local address space it has been mapped into.

This algorithm can be made fault tolerant by ensuring that

(i) whenever a block is migrated from a host A to another host B, the copy at host A is maintained, but marked *invalid*; and

(ii) whenever a *dirty* block, p, is migrated from host A to another host B, all dirty blocks from A are also copied to B; all blocks copied to B except for p are marked *invalid* at B and marked *clean* at A. (A block is dirty if it is valid and has been modified since it was obtained or marked clean.)

Action (i) ensures that every block has a copy on at least two hosts. It is not necessary to make a copy of a block every time it is written to (or to log each write operation) because if a host crashes after several writes to a block but before it is copied to another host, then the system will behave as if the host crashed before these writes were executed. Action (ii) is needed for sequential consistency; otherwise, when a host crashes it is possible for a situation to occur where a modification m to the shared address space is visible to others while modifications made earlier than m are not. It should also be noted that all blocks must be transferred in an atomic fashion; that is, either all blocks are accepted at the destination host or none of them are. This is simple to implement, but is necessary for consistency for the case where the sender crashes while it is transmitting blocks.

The recovery procedure of this algorithm is straight-forward. All pages a host held at the time of its failure must be recovered from their backup copy and duplicated. This set of blocks can be determined on demand: whenever a host cannot locate a block it wishes to access, it locates the copy of the block. If that copy is local, then a new copy is migrated to a remote host and marked invalid there (in order to have a copy on two physically separated hosts); otherwise, a copy of it is migrated to this host where it can then be accessed. Since recovering the lost blocks on demand makes the recovery time indeterminate, it makes more sense to determine and recover the set of lost blocks as soon as a host failure is detected. The time needed to do that is again on the order of seconds: each host must be querried to determine which valid and which backup blocks they have and (possibly) several blocks need to be transferred.

Since copies are continuously being generated as a block migrates from host to host, a problem with respect to garbage collection arises. Instead of communicating with the previous owner of the block to inform it that the old, now obsolete copy of the block can be freed, it is possible to attach a sequence number with each block which is incremented each time the block is migrated to a new host. Old copies are freed when the primary copy of the block returns to the host. If this is done, then the copy of the block with the highest sequence number must be located during recovery.

Many implementations of the Migration Algorithm statically assign each block a managing server that always "knows" the location of the data block in order to eliminate the need for broadcasts. Migration requests are always sent to the manager that forwards

the request to the current owner. It is not difficult to make these block managers resilient; they do not contain critical state, since the current owners of the blocks can be determined by querying all hosts. However, to speed up the recovery procedure, a scheme similar to the backup of the central server could be used. In this case, the recovery proceedure is fast, because managers know which hosts have backup copies of lost blocks.

2.4 The Read-Replication Algorithm

The Read-Replication algorithm is very similar to the Migration algorithm except that it allows multiple read-only copies of blocks; that is, it supports "multiple readers/single writer" (MRSW) replication. For a read operation on a data item in a block that is currently not local, it is necessary to communicate with remote sites to first acquire a read-only copy of that block and to change to read-only the access rights to any writable copy if necessary before the read operation can complete. For a write operation to data in a block that is local, but for which the local host has no write permission, all copies of the block held at all hosts must be invalidated before the write can proceed. (The block must first be acquired from another host, if it is not local.)

This algorithm can be made fault tolerant in the same way as the Migration algorithm.

3 Performance Comparisons

The algorithms presented in the preceding section are intended to provide a satisfactory degree of fault tolerance against loss of data in shared memory, while minimizing the overhead during the normal execution of applications. In this section, we analyze the average costs of accesses to data in DSM under each of the four algorithms, for both the base and the fault tolerant versions[7]. To do this, we need to first establish a metric for DSM data access cost. Distributed shared memory is used to support distributed and parallel applications in which multiple threads of execution may be in progress on a number of hosts. We therefore choose the *average cost per data access* to the *entire* system as the performance measure. Hence, if a data access involves one or more remote hosts, the message processing costs on both the local and remote host(s) are included. An alternative metric would be the data access delay to the process making the access. However, possible costs incurred on other hosts would not be reflected in this measure, even though other processes executing there would be affected.

3.1 Model and Assumptions

The following parameters are used to characterize the basic costs of accessing shared data, and the application behaviors:

p: The cost of a packet event, i.e., the processing cost of sending or receiving a short packet, which includes possible context switching, data copying, and interrupt handling overhead. Typical values for real systems range from one to several milliseconds [3, 5].

P: The cost of sending or receiving a data block. This is similar to p, except that P is typically significantly higher. For an 8 KB block, typical values range from 15 to 40 ms ; often multiple packets are needed.

S: The number of participating hosts.

r: The read/write ratio. This parameter is also used to refer to the access pattern of entire blocks. Although the two ratios may be different, we assume they are equal in order to simplify our analyses.

f: The probability of an access fault on a *non-replicated* data block in the Migration algorithm, which is the inverse of the average number of consecutive accesses to a block by a single host, before another host accesses the same block, causing a fault. f characterizes the locality in data access for the Migration algorithm.

f': The access fault probability on *replicated* data blocks in the Read-Replication algorithm, which is the inverse of the average number of consecutive accesses to data items in blocks kept locally, before a data item in a block not kept locally is accessed. f' characterizes the locality in data access for the Read-Replication algorithm.

m: The probability of a page transfer request being made for a dirty block, i.e., one that has been modified since it was last transferred to another host.

d: The average number of dirty blocks a host has when servicing a page transfer request of a dirty page owned by this host.

The last four parameters have a large impact on the performance of the corresponding algorithms, but, unfortunately, are difficult to assess, since they vary widely from application to application. It should also be pointed out that the above parameters are not entirely independent of one another. For instance, the size of a data block and therefore the block transfer cost, P, influences both f and f', in conflicting directions. As the block size increases, more accesses to a block are possible before another block is accessed; however, access interferences between hosts become more likely. S also has direct impact on the fault rates. Nevertheless, the analyses below suffice to characterize the DSM algorithms.

To focus on the essential performance characteristics of the algorithms and to simplify our analyses, a number of assumptions are made:

1. The amount of message traffic will not cause network congestion; hence, we do not consider the network bandwidth occupied by messages.
2. Server congestion is not serious enough to cause significant queueing or processing delay in remote access. Effective methods to distribute the load of the servers are discussed in [14].
3. The cost of accessing a locally available data item is negligible compared to remote access cost.
4. Message passing is assumed to be sufficiently reliable, so retransmissions are rare, and not considered in our analyses.[8]

3.2 Access Costs and Comparisons

Using the basic parameters and the simplifying assumptions described above, the average access costs of the four algorithms, in both the base and the fault tolerant versions, may be expressed as in Table 1. Each of the cost formulas consists of two components. The first component, to the left of the '$*$', is the probability of an access to a data item being remote. The second component, to the right of the '$*$', is equal to the average cost of accessing a remote data item. Since the cost of local accesses is assumed to be negligible, the average cost of accessing a data item is therefore

[7] An analysis of the access costs of the four base algorithms were presented in an earlier paper [14].

[8] Note, however, that the cost for acknowledgment messages, required to determine whether a retransmission is necessary, is included in our models.

Algorithm	Base Version	Fault Tolerant Version
Central	$(1-\frac{1}{S})*4p$	$\frac{r}{r+1}*(1-\frac{2}{S})4p + \frac{1}{r+1}*(1-\frac{1}{S})8p$
Migration	$f*(2P+4p)$	$f*[2(md+(1-m))P+8p]$
Read-Repl.	$f'*[2P+4p+\frac{Sp}{r+1}]$	$f'*[2(md+(1-m))P+8p+\frac{Sp}{r+1}]$
Full-Repl.	$\frac{1}{r+1}*(S+2)p$	$\frac{1}{r+1}*(S+2)p$ without backup
		$\frac{1}{r+1}*(S+4)p$ with backup

Table 1: Cost of algorithm for an average memory access operation

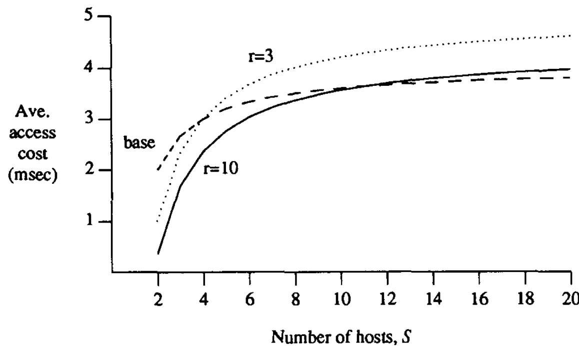


Figure 3: Performanance comparison between the base and fault tolerant versions of the Central Server algorithm

equal to the product of these two components. Below, we explain the derivation of these models and comment on the extra costs due to fault tolerance.

Central Server Algorithm

Under the assumption that data is uniformly distributed over all hosts, the probability of an accessed data item being remote is $1-\frac{1}{S}$ for the base algorithm, in which case four packet events are necessary for the access, as described in Section 2.1. The overall cost is mainly determined by the cost of a packet event, as long as the number of hosts is over four or five. For the fault tolerant version, a read access is processed in the same way as in the base algorithm. Since both the server for the block and its backup has an up-to-date copy of the shared data, however, the probability of an access being remote is $1-\frac{2}{S}$. A write access is more expensive in the fault tolerant algorithm, since both the server and its backup need to be contacted, incurring a total of eight packet events for an access from a third host. The cost is only four packet events if the access originates from the server or its backup, since only one other host needs to be contacted. Again, assuming uniformity of accesses, the average update cost is $(1-\frac{1}{S})8p$. Considering both read and write accesses, with a ratio of r, the overall average access cost for the fault tolerant Central algorithm is as shown in Table 1, third column.

Comparing the cost formulas for the base and fault tolerant versions of the algorithm, one can see that the cost of the fault tolerant version is close to that of the base algorithm for typical values for the read/write ratio (over three). The replication of the shared data at two hosts actually helps reduce the read access cost. Figure 3 illustrate our observation graphically.

Full-Replication Algorithm

For the base Full-Replication algorithm, the remote access probability is simply the write access probability; the associated cost is always a message from the local host to the sequencer (two packet events), followed by a multicast update message (S packet events). There is no additional cost for fault tolerance if no backup is used. Otherwise, the cost of remote access is increased by $2p$. Given the tradeoff between increased access cost during normal operation, and a more complicated recovery procedure, the algorithm with no backup is generally preferable. As in Central, the additional overhead due to fault tolerance is minimal.

Migration Algorithm

For the base Migration algorithm, f represents the probability of an access to a data item being remote. Assuming the variant of the algorithm with a manager process that is backed up, the cost of bringing the data block containing this data item to the local host includes a total of four packet events distributed across the local, manager, and server hosts[9], and one block transfer ($2P$). For the fault tolerant version, if the data block causing the fault is not dirty on the current owner host (the probability of which is $1-m$), then only this block needs to be transferred to the faulting host. Otherwise, a total of d dirty blocks need to be transferred to ensure sequential consistency. Two extra packet events are needed to keep the backup for the manager of this block up to date. The cost of garbage-collecting old backup copies of blocks is not considered, since it can be done in the background.

Assessment of the extra overhead due to fault tolerance is generally difficult, since the values of the crucial parameters, m and d, vary greatly from application to application. On the one extreme, if the probability of a fault on a dirty block is low (read-only or read-mostly accesses), or the average number of dirty blocks on the owner host is close to one, then the extra cost for tolerating faults may be little more than four more packet events, which is relatively small compared to the block transfer cost. On the other extreme, if a fault on a dirty block occurs after its owner host has recently updated a large number of other blocks, then these blocks all need to be transferred, even if they are not shared at all. Applications performance may be seriously degraded in this case.

Read-Replication Algorithm

For the base Read-Replication algorithm, the remote access cost is similar to that of the base Migration algorithm, except that, in the case of a write fault (with a probability of $\frac{1}{r+1}$), a multicast invalidation packet must be serviced by all S hosts. The block transfer cost is always included in our expression, although it may not be necessary if a write fault occurs and a local (read) copy of the block is available. The characterization of the overhead for fault tolerance is also similar to that for the Migration algorithm. However, the values of the crucial parameters, m and d, are generally different from those in the fault tolerant Migration algorithm, even for the same application.

Summary

The fault tolerant versions of the Central Server and Full-Replication algorithms do not introduce substantially more overhead, whereas the extra overhead introduced by the fault tolerant versions of the Migration and Read-Replication algorithms is very heavily dependent on the applications' data access behavior. In [14], it is pointed out that the Central Server algorithm is suitable when accesses to shared data are infrequent, and the Full-Replication algorithm performs well when accesses are mostly

[9] We assume that the local, manager, and server hosts are all distinct, and that the request is forwarded by the manager to the server. The sequence of packet events are send (on local host), receive (on manager host), forward (on manager host), and receive (on server host).

read. The Migration and Read-Replication algorithms support frequent shared data accesses better, provided that sufficient access locality exists. In particular, Read-Replication is suitable for many applications and is the most widely implemented DSM algorithm. Unfortunately, fault tolerance potentially makes the applications' performance with the Migration and Read-Replication algorithms significantly poorer and less predictable.

4 Related Work

Distributed shared memory is an active area of research [2, 4, 6, 7, 9, 10, 11, 12, 13, 14, 18], but little work has been done with respect to making the algorithms fault tolerant. Xu and Liskov [17] describe an fault tolerant implementation of Linda [1], which is a language based on the distributed shared memory model. In their design, data is replicated onto t hosts, where t must be larger than 2 since a majority is needed for memory operations to be successful. The semantics of some of the Linda operations (`in` and `out`) make communication with each server for each of these operations necessary, and in the case of `in`, each server must be contacted twice in a two phase protocol. Therefore, the communication overhead for this implementation is far higher than for the protocols presented in this paper.

Wu and Fuchs [16] extend the Read-Replication algorithm to make it fault tolerant. They effectively checkpoint the entire system on each page fault. In contrast to our algorithm that copies blocks onto other hosts, they copy blocks onto a single backing store (i.e. disks).

5 Concluding Remarks

In this paper we extended four basic DSM algorithms to tolerate single host failures. In doing so we had to strike a balance between performance and fault tolerance. Our algorithms are only 1-resilient, but we argue that this degree of fault tolerance is sufficient for most applications; a higher degree of fault tolerance would degrade performance substantially. Instead, we attempted to make FTDSM as efficient as possible to remain competitive as an inter-process communication mechanism. For two of the algorithms, namely the Central Server and the Full Replication algorithms, the extra overhead for fault tolerance is very small, but for the other two, Migration and Read Replication, the extra overhead can vary from being very small to being very large, depending on the data access patterns of the application.

This paper only considered making the distributed shared memory resilient. Additional mechanisms may be needed to make applications that use FTSDM fault tolerant. For example, a checkpointing facility, as described in Section 4, would allow an application to restart computation from the last checkpoint instead of from the beginning. But less expensive mechanisms would also suffice for many applications. For instance, some applications, like the distributed editor of Section 1, can use FTDSM as is and can simply continue to run with fewer processes after a host failure. For other applications, say those based on the Work-Crew model of parallel computation [5, 15], subtasks that never complete can be detected relatively easily and simply restarted. An atomic write operation (that gets executed at the end of the subtasks) could help prevent problems due to partially executing a subtask multiple times.

Further work is still necessary to assess the performance implications of the presented FTDSM algorithms in practice. Both fault tolerant versios of the Migration and the Read-Replication algorithm are susceptible to anomalous behavior and an excessive amount of data copying is possible. For example, in many applications, worker processes iteratively process and modify large amounts of data locally before interacting with other processes. The Read-Replication algorithms can be very well suited for this type of access behavior, but the fault tolerant version of the algorithm causes all blocks containing locally updated data to be copied to a remote host in each iteration, even though the data in these blocks is not shared, resulting in a far higher overhead than in their base counterparts.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Symp. on Priciples and Practice of Parallel Programming*, March 1990.

[3] L.F. Cabrera, E. Hunter, M. Karels, and D. Mosher. A user-process oriented performance study of ethernet networking under berkeley UNIX 4.2BSD. Technical Report UCB/CSD 84/217, University of California, Berkeley, EECS, 1984.

[4] D.R. Cheriton. Problem oriented shared memory: A decentralized approach to distributed system design. In *Proc. 6th ICDCS*, pages 190–197, May 1986.

[5] D.R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[6] B.D. Fleisch and G.T. Popek. Mirage: A coherent distributed shared memory design. In *Proc. 12th ACM Symp. Operating System Principles*, pages 211–222, December 1989.

[7] A. Forin, J. Barrera, M. Young, and R. Rashid. The shared memory server. In *Proc. 1988 Winter USENIX Conf.*, pages 229–243, January 1989.

[8] I. Greif. *Computer-Supported Cooperateive Work: A Book of Readings*. Morgan Kaufmann, 1988.

[9] R.E. Kessler and M. Livny. An analysis of distributed shared memory algorithms. In *Proc. 9th Intl. Conf. on Dist. Comp. Sys.*, pages 498–505, June 1989.

[10] O. Krieger and M. Stumm. An optimistic apoproach for consistent replicated data for mulitcomputers. In *Proc. 1990 HICSS*, volume 2, pages 367–375, 1990.

[11] K. Li. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, Yale University Dept. of Computer Science, 1986.

[12] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4), November 1989.

[13] R.G. Minnich and D.J. Farber. Reducing host load, network load and latency in a distributed shared memory. In *Proc. 10th ICDCS*, June 1990.

[14] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 22(5), May 1990.

[15] M.T. Vandevoorde and E.S. Roberts. WorkCrews: An abstraction for controlling parallelism. *Intl. Journal of Parallel Programming*, 17(4):347–366, August 1988.

[16] K.L. Wu and W.K. Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[17] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proc. 1989 Conf. on Reliability*, 1989.

[18] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *Proc. 10th ICDCS*, June 1990.