Exploiting Temporal and Spatial Constraints on Distributed Shared Objects *

Richard West
Karsten Schwan
Ivan Tacic
Mustaque Ahamad
College of Computing
Georgia Institute of Technology

October 4, 1996

Abstract

The advent of gigabit network technologies has made it possible to combine sets of uni- and multiprocessor workstations into a distributed, massively-parallel computer system. Middleware, such as distributed shared objects (DSO), attempts to improve programmability of such systems, by providing globally accessible 'object' abstractions. Early research on distributed shared object systems concerned protocols to maintain consistency across replicated 'memory' objects. Such systems are well suited to scientific applications but have limited support for multimedia or groupware applications. This paper addresses the state sharing needs of complex distributed applications with (1) high-frequency symmetric data accesses to shared objects, (2) unpredictable and limited locality of data access, (3) dynamically changing sharing behavior, and (4) potential data races. We show that a DSO system that exploits application-level temporal and spatial constraints on shared objects can outperform shared object protocols which do not exploit application-level constraints. We describe the features of our S(emantic) DSO and compare three application specific consistency protocols, developed to run on our system against entry consistency for a sample application having the four properties mentioned above.

*This work is supported in part by the Engineering and Physical Sciences Research Council grant 92600699 and DARPA contract DABT63-95-C-0125

1 Complex Parallel and Distributed Systems

The advent of gigabit network technologies has made it possible to combine sets of uni- and multiprocessor workstations into a distributed, massively-parallel computer system. However, such platforms are difficult to program, in part because implementors must explicitly code complex message passing protocols with which concurrent processes exchange the state they must share. Distributed shared objects (DSO) have the potential of simplifying the implementation of such shared state, by providing globally accessible 'object' abstractions. To the user of such an abstraction, all accesses to it appear to be local. The abstraction's implementation, however, may be internally distributed across multiple memory units on physically different machines. It is up to object implementors and the underlying run-time system to maintain consistency across distributed object components.

Early research on distributed shared objects concerned 'memory' objects accessible via 'read' and 'write' operations (DSM) [26] or 'fragmented' objects offering relatively simple operational interfaces[11, 33]. Since then, object-based research has moved toward more general representations of shared abstractions, including the support of arbitrary type hierarchies in object access[17, 34]. Furthermore, many researchers have investigated the efficient implementation of DSM, including the development of efficient consistency maintenance protocols[7, 4, 8, 21, 6, 14, 20, 28], experimentation with alternative representations of shared memory pages[22] and with alternative methods for dealing with specific implementation issues, like false sharing[9]). Ongoing DSM research is developing hardware[27, 25] and operating system[23] support for efficient DSM implementations. Two particular issues addressed by such work are (1) the levels of concurrency attainable by programs using DSM abstractions[18] and (2) the scalability of DSM implementations in terms of the amounts of memory shared and the number of processors involved in such sharing[14, 28]. Evaluations of (1) and (2) are typically performed with application programs similar to those running on shared memory supercomputers[10].

Complex high performance applications. This paper considers the state sharing needs of the complex distributed applications targeted by ongoing research in distributed, object-based systems, including distributed groupware applications[16], client-server systems[30], and distributed and interactive high performance codes[32]. Specifically, we focus on the required consistency of the state shared in such applications. Toward that end, this work utilizes the relatively 'simple' model of shared state presented by DSM abstractions, accessed with 'read' and 'write' operations. For such shared 'memory' objects of varying sizes, we develop and evaluate methods for consistency maintenance suitable for the efficient and scalable implementation of concurrent programs that exhibit dynamically changing and unpredictable patterns of resource access.

The sample applications considered in our research include (1) distributed multimedia games; (2) collaborative applications in which substantial amounts of state may be shared, such as distributed virtual environments[13], shared visualization of 3D datasets manipulated by multiple end users[32], or distributed design environments[24]; (3) distributed real-time applications in which many entities share common state, such as distributed command and control[29]; and (4) high performance applications exhibiting dynamic data access patterns[36]. The characteristics of such applications relevant to our work are:

- *Poor and unpredictable locality* – distributed processes may read and write shared data with high frequency, thereby making it difficult to cache shared state. This is exemplified by the dynamic data redistribution necessary for the irregular scientific applications investigated in [36].

- *Symmetric data access* – in contrast to distributed programs like World Wide Web browsers where servers occasionally write shared data while clients simply read the data, these applications' distributed processes may behave symmetrically in terms of their read and write operations on shared state. For example, in virtual environments used for distributed games, the movement of one 'player' may affect the actions of another player within range of each other.

- *Dynamic changes in sharing behavior* – the sharing behavior of these applications may change rapidly. For instance, in distributed design[24] or in scientific collaboration via shared 3D data[32], participants may jointly manipulate certain shared state with high frequency, then cease such sharing and operate on private data or instead, begin interacting with other collaborators on related tasks. As a result, such applications will potentially access not just one but many different shared data objects, and they may access many different locations within a single shared data object. Furthermore, whether or not a write to shared memory affects any other process's future writes depends on the causal relationships between such write operations. For example, in distributed command and control applications, the state an entity must share with other entities may depend on its current location in reference to them, not on its past or future ability to access such shared state.

- *Data races may occur* – data races occur when two or more processes attempt to access the same memory location, where at least one process is performing a 'write' to that location[1, 2, 4]. Most existing DSM systems assume that the programs using them are data race free, because this property is easily guaranteed if such programs' synchronization on shared state is implemented correctly. In other words, by using synchronization, programs simply enforce strictly sequential accesses to the memory locations in question. This approach is reasonable since most parallel supercomputer applications employ algorithms that attempt to minimize explicit synchronization. However, the applications considered in our research do not have this property. Instead, data races are highly probable and may occur for any shared object. For example, when manipulating shared documents, it is quite possible that two end users attempt to update the same portion of the document at the same time. Rather than prohibiting such simultaneous updates by use of synchronization, it may be more appropriate to employ application-specific methods for dealing with data races, like maintaining version histories.

Solution approach. We posit that high levels of concurrency and scalability for complex distributed applications may be attained if programmers can exploit the specific semantics of these applications when implementing and using shared state abstractions. For 'memory' abstractions used for representing shared state, this implies that they should facilitate the use of application-level semantics for deciding *when* and *who* should be informed of updates to them. Given this ability, we claim that complex distributed applications will perform comparably well to programs using explicit message passing, with improved programmability compared to such programs. In addition, a DSM system facilitating the exploitation of application semantics can outperform DSM systems that are not able to do so.

Contributions. The S(emantic)-DSO system presented in this paper permits applications to specify *when* and *which* processes should see updates to shared memory objects of varying sizes, using additional parameters associated with object accesses, called *attributes*. Therefore, rather than having the DSO system determine the resource-sharing patterns among processes at different times, users can exploit their knowledge of such patterns to improve program performance. S-DSO does not offer a single consistency protocol, nor does it implement some particular lock management scheme for use in synchronization (when desired by programmers). Instead, it builds on the generality of the Indigo[22] system to present to developers low level primitives with which they may construct exactly the shared object functionality and consistency semantics they desire. In addition, using attributes, consistency maintenance may exploit application-level semantics specified by end users. Last, and in contrast to Indigo's initial implementation with PVM, S-DSO is implemented on an efficient lower-level communication substrate that supports heterogeneous distributed computing platforms.

S-DSO is evaluated with a novel distributed multimedia application implementing a multi-player game and patterned after the complex command and control applications now being developed by companies like Honeywell[19] and TRW[29]. Novel characteristics of this application include its exploitation of user-specified attributes to improve the performance of consistency maintenance for its replicated DSO objects, its high levels of concurrency and asynchrony while performing what appear to be sequentially consistent actions on shared data, and its scalability in terms of the amounts of data shared and the number of parties accessing shared data. More specifically, the application relies on a *lookahead* consistency protocol developed using S-DSO's attribute infrastructure. We use the term 'lookahead' to describe any protocol that has the ability to predict the future times at which groups of processes must exchange information regarding modifications to shared objects that each process may later need. Our lookahead protocol's realization exploits the semantics of interactive distributed applications to enable their concurrent and asynchronous execution. This means that application processes need not synchronize until:

- two or more processes attempt to access the same shared object, with at least one access involving a write, or
- a process requires updated information to decide its next operation.

Furthermore, with lookahead consistency, processes communicate only with the *subset* of other processes currently accessing objects that they need to know about in the imminent future. Such processes do not synchronize (ie., they proceed asynchronously) at the access to a shared object unless they require knowledge about that object's current state. In other words, information about a shared object is only made known to a process if and when it is required.

Evaluation. Several specific lookahead protocols are evaluated in this paper, and they are also compared against one of the common consistency protocols used in DSM systems, called entry consistency (EC)[7]. The first lookahead protocol is called *BSYNC* (see Section 3.2); it assumes that all updates to shared objects must be made known to all other processes whenever any object is modified. BSYNC uses application-level knowledge to predict the worst-case time when two or more processes may attempt to access the same shared object. Such knowledge also permits us to avoid data races, without having to use explicit synchronization. The second pair of lookahead protocols are called *MSYNC* and *MSYNC2* (see Section 3.2); each attempts to communicate object updates only to those processes that may need them for their next operations. The following observations summarize insights derived from these protocols' experimental evaluations:

- A 'lookahead' protocol can be made to outperform an 'entry consistent' protocol if it makes full use of application-level program semantics.
- The 'entry consistent' protocol performs poorly when applications manipulate multiple shared objects with rapidly changing sharing patterns amongst these objects.

Overview. In the remainder of this paper, we first present a sample distributed program that is representative of the complex distributed applications addressed by our research. Using this program, we define in more detail the notions of 'temporal' and 'spatial' consistency that capture the application-level semantics exploited by this paper's novel consistency protocols. Section 3 describes the S-DSO system, its interfaces, and its implementation to the extent necessary for comprehending the BSYNC and MSYNC protocols and their experimental evaluation. An interesting aspect of the S-DSO system is the manner in which it permits end users to specify application-level knowledge of the spatial and temporal consistency constraints utilized by these protocols. Section 4 shows an evaluation of the S-DSO system as well as the sample application to demonstrate the utility of protocols like BSYNC and MSYNC. Section 5 shows related work, while conclusions and future research appear in Section 6.

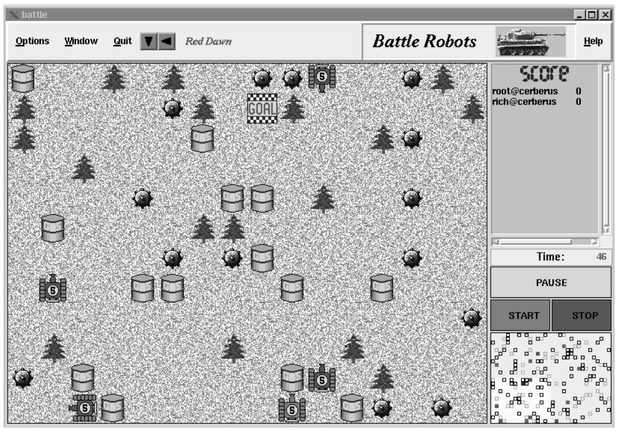


Figure 1: A sample distributed multimedia application: a video game

2 Memory Consistency for Interactive Distributed Applications

2.1 A Sample Application

A common feature of multimedia collaborative and groupware applications is a shared environment. If the application is interactive, like a distributed whiteboard, document, spreadsheet, or a video game, then at any one time, each user interface manipulates some part of the shared environment. The run-time realization of such a user interface is a process, while the shared environment may be realized as physical memory shared by these processes. This physical memory may totally reside in some single server process, or be distributed physically across participating processes. For reasons of scalability and performance, we assume the physical distribution of the shared environment across all interacting processes, while proposing to exploit the fact that at any one time, each process tends to operate only on some section of this environment, separated from other sections by some known distance (hence the term 'spatial' locality).

The sample distributed multimedia application developed in our research is an interactive game with a shared environment, as shown in Figure 1. The objective of this game is much like "Capture the Flag". A player must maneuver her team of tanks to some known goal as quickly as possible, while picking up bonus items and avoiding bombs and enemy tanks along the way. Clearly, at any one time, each player need not know about the moves of all other players, particularly if those players' teams are located in remote areas of the shared environment. However, whenever an enemy team's tanks appear close to your own tanks, then it is important to know exactly where those tanks are whenever they move, because such tanks may pose a threat to your own team.

The 'spatial' constraints on consistency for this application are well defined. Namely, if one team's tanks move to within distance d of another team's tanks, each such team must know the exact position of the other team. When tanks are separated by a distance larger than d, it is not necessary to update each team with the locations of enemy tanks. From this example, it is clear that such formulations of 'spatial' locality are possible only in reference to application-level program semantics.

The contributions of our work are (1) to construct a DSO system that is capable of utilizing such semantics and (2) to develop the aforementioned lookahead consistency protocols that exploit such knowledge. Specifically, the manner in which such semantic knowledge is provided to S-DSO by the application programmer is by specification of an *exchange* function. An exchange function capturing the spatial consistency constraints for the interactive game is one that permits the consistency protocol to construct a list of *(exchange-time, process)* pairs based on the minimum possible time that an enemy tank could appear within distance d of one of the local team's tanks. In the worst case, this function would assume that the closest tank in an enemy team will always move towards the closest tank in the local team, and vice versa. Furthermore, with this application, in the worst case, all teams are within distance d of each other, thereby obviating the potential benefits of an exchange function specifying spatial consistency constraints. However, as shown by the experimental evaluations in Section 4, such worst cases appear rare for most game scenarios we have experienced.

Clearly, the 2D shared environment used in the interactive game gives rise to fairly simple functions with which spatial consistency constraints may be computed, and similarly straightforward formulations may be found for 3D shared environments like those used in shared virtual environments. Even scientific applications exhibit such spatial consistency constraints, as is evident in n-body simulations, where the gravitational effects of bodies on each other are considered only when two bodies are within minimum distance d of each other. Likewise, molecular dynamics simulations[12] tend to consider only those interactions of molecules within some known cut-off radius. Furthermore, when two users 'walk' through a shared virtual world, there may be known and quantifiable semantics other than distance that determine whether they need to know about each other (e.g., consider obstacles like mountains or walls).

In summary, the sample interactive application presented in this section has unique requirements with respect to when each of its distributed processes must have access to the latest values of shared objects. A general consistency function, like entry consistency, is not designed to meet such requirements. To support such requirements, the S-DSO system may not only be used to implement 'standard' consistency functions, like 'entry consistency', but it also permits developers to devise application-specific consistency management schemes using *exchange* functions. The sample exchange functions evaluated in this paper exploit application-level knowledge about spatial and temporal properties of the distributed game. We posit that similar functions may be developed for most distributed applications being developed in commercial and research settings.

The distributed game exhibits the four features described in Section 1. Firstly, there is poor and unpredictable locality to shared objects, where each object is at the granularity of a team's tank. Secondly, each user can manipulate her own team of tanks independently of other users, which results in symmetric data accesses. Thirdly, at any point in time, one team is only concerned with a subset of all possible shared objects within range, but that set of objects may change with time. Consequently, the DSO system must deal with dynamic changes in sharing behavior. Finally, data races may occur, since two tanks may attempt to move to the same location in the shared environment.

2.2 Temporal and Spatial Consistency

The previous sections introduced the notions of spatial and temporal consistency. This section characterizes these notions more precisely.

All DSM systems deal with *temporal consistency*. Temporal consistency determines *when* (and in what order) changes to a shared object are made visible to all processes interested in that object. However, this says nothing about *which* processes should be informed of changes, and a system that does not utilize such knowledge can only assume that changes must be known by all processes that have a copy of the shared resource. Spatial consistency considers this latter problem.

Spatial consistency determines *which* processes should be updated with changes to shared objects based on the locations of those objects in the shared space. For example, a contiguous block of memory, such as an array, may be shared between two process P_i and P_j. A write to any element of this memory by P_i should be made visible to P_j, if the change is to an element in a range of elements that P_j needs to know about. Likewise, if two moving objects in a virtual environment are within a certain distance of each other, each object must know about the other. This 'spatial' consistency property is stated more precisely below in the context of the distributed game.

Consider a process P_i that writes to a location x at time t. At time $t + \tau$, P_j reads location x and, based on the value read, generates a write to one of two possible locations, y or z. Alternatively, P_j may decide to write to location y no matter what the value of x, but the actual value written to y does depend on the value P_j sees at x. Thus, we have a dependency between the write of P_i at time t and the write of P_j at time $t + \tau$, based on the read of x by P_j also at time $t + \tau$[1]. However, if P_i writes to a location x' at time t, out of the range of necessary locations that P_j must know about for its next operation, P_j can avoid being updated with the new value at location x'. Spatial constraints permit P_i and P_j to see inconsistent views of the value at x' at time $t + \tau$.

Defining the value of τ is critical for the performance of a system in which updates must be exchanged in this manner. Between t and $t+\tau$, P_i and P_j can be inconsistent, but at $t+\tau$

[1]This assumes the real-time taken to perform a read before a write is negligible.

they must both see the same values in all locations that affect their next write operations.

We can define τ as the interval of time in which two or more processes may concurrently perform a write operation. These write operations will be based on the states of each process' local copy of the shared environment at time t. Only at time $t + \tau$ will each process P_i be required to synchronize with those processes that have generated write operations that may affect $P_i's$ operations in the interval $[t + \tau, t + 2\tau]$.

In the worst case, each process must barrier synchronize with every other process after each interval τ, in which each process performed exactly one read of the shared environment, followed by one write. Synchronization at time t is required for two reasons:

1. To update the state of each process' copy of the shared environment, thereby ensuring any writes in the interval $[t, t + \tau]$ are based on correct previous states.

2. To identify new locations in the shared environment where access races may occur as a result of the dynamically changing environment.

The synchronization actions described above do not eliminate data races, because in any period τ, two or more processes may be performing concurrent accesses to the same location, where at least one is a write. In such circumstances, only one process may access the common shared object. All other processes must block or perform access operations on other locations in shared space. Some policy must be adopted for blocking a process from performing a potentially conflicting write, while another process proceeds. A simple scheme would be to allocate a token to the process with the lowest ID, or randomly grant a token to one process while all others blocked. Lock-based consistency protocols explicitly avoid data races by requiring each process acquire a lock to an object before modifying that object.

In shared-world applications, where processes behave in a symmetric manner (ie., all processes are equally likely to perform write operations), each process has to synchronize with those processes that have performed write operations that may affect its next writes. In the worst case, every process barrier synchronizes with every other process after each write operation. However, in most multimedia-collaborative applications, not all writes in the interval $[t, t + \tau]$ affect write operations in $[t + \tau, t + 2\tau]$. Namely, writes in the interval $[t, t + \tau]$ do not need to be conveyed to those processes that (a) don't depend on them for their next write operations, and (b) will not be involved in data races regardless of all writes being known or not. This reinforces the notion that application-level semantics should be used at run-time to decide *when* write operations must be exchanged and with *whom*. Combining *when* (temporal) and *which* (spatial) constraints on object-sharing in distributed applications enables greater levels of concurrency and asynchrony, and reduces the number of update messages transferred around the system.

2.3 Conventional Consistency Protocols

Previous consistency protocols for DSM systems have been designed for scientific rather than multimedia or groupware applications. We mention three such prominent protocols, namely causal, entry, and lazy release consistency (LRC), and describe the reasons why these protocols are inadequate for the distributed multimedia applications addressed by our work.

Causal consistency. Despite its proven high performance with scientific applications, causal memory[3] is not a good candidate for distributed multimedia applications, in part because the overheads of avoiding data races severely curtail the potential levels of asynchrony and concurrency. Specifically, with causal memory, each process must enforce strict sequential access to *all* shared objects, since it does not know which objects will be needed at what times by each process. Without strict sequential access to all shared objects, two users could inadvertently attempt to modify the same object at the same time and, although this is legitimate with causal consistency, it does not ensure the correct execution of collaborative applications. Data races may be prevented with some lock-based scheme, but with a *push-based* protocol like causal memory, all processes have to send their acknowledgments to a lock-manager on receipt of an object-update message prior to lock release. This is undesirable for reasons of performance and scalability.

Causal consistency is also inefficient when an object's modification can potentially affect any other process' future operations. Namely, causal memory cannot determine which subset of processes should be informed of such changes, so that all processes sharing a given object must barrier synchronize if they need updates to the object that could affect their future operations. Such synchronization can cause some processes to block when they are actually able to continue to run. Moreover, frequent barrier synchronization across thousands of processes would be undesirable. Therefore, large-scale distributed applications would perform poorly in such circumstances. Furthermore, even in small-scale collaborative applications, causal memory will still exchange more messages then necessary among process. This may consume substantial network bandwidth if the application is rich in visual detail, such as a 3D virtual world.

One way of reducing the number of processes that must synchronize is to explicitly specify which objects are shared in a program at which times, using share(*object*) and unshare(*object*) calls to the underlying DSM run-time system, as done in Indigo[22]. However, for applications where the sharing patterns change frequently, the code would be cluttered with such calls, and frequent calls would result in substantial overheads since for each such call, the system would have to build and rebuild associations of shared objects with different groups of processes. A better solution would be to hide the semantics of when and who should be informed of updates in a user-specifiable attribute list used by the system, and to declare all objects as sharable once-only at the the time of program initialization. In effect, this method is used by our BSYNC and MSYNC protocols.

Entry consistency. Due to the problems with causal consistency explained above, this paper compares our 'lookahead' schemes against entry consistency. This comparison is not performed because entry consistency is popular, but because: (1) 'entry consistency' explicitly deals with data races by associating distributed locks with objects, (2) 'entry consistency' only enforces consistency among those objects accessed by locks, thereby avoiding unnecessary updates to objects that are not needed, and (3) 'entry consistency' permits processes that lock separate objects to proceed concurrently. However, in comparison to the 'lookahead' schemes, entry consistency does not deal well with multiple shared objects that are spatially related and have dynamically changing relationships. For a process to have consistent copies of those objects it needs for its subsequent operations, the process must first acquire locks on all those objects. Acquiring a lock ensures that updates to the locked object are 'pulled' from the owner of the up-to-date copy. If the object is exclusively write-locked by another process, then the process attempting to acquire the lock must block. Furthermore, entry consistency can cause deadlocks in applications that lock multiple objects simultaneously. This implies additional run-time overheads due to the use of distributed deadlock detection schemes, or it requires the use of deadlock prevention schemes that must distinguish between the accessibility of different shared objects by assignment of totally ordered identifiers to them (using some general and most likely, non-intuitive algorithm for assigning such identifiers).

Lazy release consistency. Like EC, lazy release consistency uses locks to synchronize access to shared objects and, as a result, to enforce consistency. With LRC, updates to shared data are propagated when locks are transferred between processes. Unlike EC, LRC has no explicit associations between shared data and synchronization primitives. To ensure that a process acquiring a lock receives all changes to shared data that were known to the process that released the lock, data dependencies are recorded using vector timestamps, and a history-based mechanism determines what data modifications have to be transferred with the lock. The Treadmarks[20] system implements LRC, while EC is used in Midway[7]. We chose to restrict our protocol comparisons to entry consistency, since entry consistency has explicit associations between locks and objects, ensuring that only data associated with a lock is ever transferred with the lock itself. LRC, on the other hand, must include information about changes to all shared data objects.

Deadlocks. Entry, release and lazy-release consistency are lock-based paradigms. Therefore, if processes using these protocols acquire multiple resources simultaneously, as is the case with the interactive applications considered in our work, these protocols must be enhanced by application-level functionality with which possible deadlocks may be handled. In comparison, the lookahead protocols can push updates to appropriate processes as soon as updates are available and are therefore able to avoid object locking and consequently, obviate the need for deadlock handling. We believe that their additional run-time overheads due to predicting future exchange times between groups of processes are less than the potential overheads of deadlock handling and blocking in protocols like entry consistency.

We next describe a framework for building application-specific consistency protocols and show how the lookahead protocols, suitable for our sample application can be built using this framework.

3 S-DSO – A Framework for Application-Specific Consistency Protocols

3.1 System Description

The *S-DSO* infrastructure extends the functionality of Indigo[22] by allowing users to write application-specific functions that can be invoked by a consistency protocol to minimize message exchanges and to increase asynchrony, concurrency, and scalability in distributed applications that share objects. *S-DSO*'s current implementation is directly layered onto sockets, eliminating the overhead of the PVM library used in Indigo. Its future implementations will also exploit higher performance network interfaces, like 'fast messages', so that substantial virtual environments may be efficiently shared between many players.

To support configurable semantic-based consistency protocols, *S-DSO* allows users to write functions detailing *when* each process must see the most recent updates to *which* objects. The S-DSO system uses the information from user-defined *semantic functions* to calculate the future times at which each process must send to and receive from other processes updates to different objects. Semantic functions, hereafter referred to as *s-functions*, provide a specification of application-level attributes the underlying consistency protocol may use.

S-DSO offers the following library calls:

- **async_put**: initiates a send of a data object to a remote process and returns without waiting for acknowledgment that the remote process has received the data;
- **sync_put**: initiates a send of a data object to a remote process and blocks the calling process until it receives an acknowledgment from the remote process;
- **async_get**: requests a data object from a process and continues without blocking; and
- **sync_get**: requests a data object from a remote process and blocks until it arrives. This call is used in our implementation of entry consistency, to 'pull' the up-to-date copy of an object from the owner.

The above four calls are of the form:
`call(obj_t *shared_obj, int remote_proc, int local_process)`

S-DSO also uses a **share(obj_t *shared_obj)** call, but there is no corresponding unshare() call. The share() call is used to register shared objects with S-DSO. All objects are declared shared at the initialization phase of a program. Since S-DSO is really designed for configurable application-specific protocols, there is no need for explicit object sharing and unsharing calls strategically placed throughout a program. Instead, a carefully-written semantic function can be exploited by a lookahead consistency protocol to decide which copies of objects need to be updated when in the future. However, to allow conventional DSM/DSO protocols to run, we plan to add a full set of library calls similar to those described for the Indigo system[22].

The most interesting attribute of the S-DSO system is its provision of the *exchange* call:

```
void exchange (obj_t *shared_obj,
               boolean resync_flag,
               send_t how,
               void (*s_func) (),
               any_t arg);
typdef char obj_t;
typedef enum {
    multicast,
    broadcast
} send_t;
typedef void *any_t;
```

The *exchange* function takes, as its first argument, a pointer to the shared object. If 'resync_flag' is true, then exchange() will wait for all processes that receive updates at the current time to exchange any object updates they have made since the local process last exchanged with them. If 'resync_flag' is false, exchange simply *pushes* changes out to appropriate processes. Thus, 'resync_flag' switches exchange between two modes of operation: *push* and *push-pull*. Furthermore, if 'resync_flag' is true, S-DSO uses *s_func* to calculate when to exchange (and, hence, resynchronize) in the future with the processes with which it has just performed exchange operations. *S_func* is specified by the user and thereby, may convey to the consistency protocol the semantic attributes it should use for calculating future synchronization times with other processes. The *arg* argument to exchange() is the argument associated with *s_func*.

Every time an application process (running on a lookahead-consistent system) modifies a shared object, it calls exchange(), and a logical system clock is advanced one time-tick. The modified object is referenced by the first argument to exchange(). The S-DSO system can use the s-function specified in the exchange call to determine whether or not the updated object information should be sent to remote processes that have a local copy of that object.

S-DSO maintains a time-ordered list of (exchange-time, process) pairs for each process that must be updated with object modifications in the future (see Figure 2). The object modifications themselves must be buffered if they are not immediately sent to remote processes. Accordingly, S-DSO maintains a slotted buffer at each process for outstanding modifications to be exchanged with remote processes (see Figure 3). There is one slot in the buffer for each remote process. To reduce buffering needs, the buffered changes are *diffs* of the state of

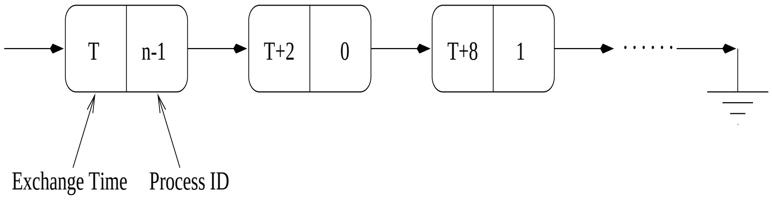


Figure 2: A sample exchange-list. Each list-member is an (exchange-time, process) pair. Only those processes requiring future exchanges appear in the list. The list is ordered 'earliest exchange-time first' and not by process IDs.

each object since their previous modification. In each slot is the list of modifications about which the corresponding process must be informed when it needs the latest information on those objects. S-DSO uses the s-function to calculate the times at which each list of buffered changes must be flushed (i.e., sent to the corresponding remote process). With this model, object changes are not sent to processes that will never need them. Furthermore, exchange() only exchanges with the subset of processes that need to know the changes immediately. Also note that each process may need knowledge of different objects at different times. This implies that the buffered changes maintained by S-DSO differ across processes at any one point in time. Furthermore, S-DSO can be tuned to merge multiple diffs to the same object into one diff since the last exchange with a given process. This kind of optimization is especially useful for real-time applications and games, since many such applications will not consider 'old' values when newer values of shared objects are available. Last, observe that an application process does not explicitly specify which processes receive changes to shared objects. Instead, the s-function is used to determine *who* receives the information.

To override the multicasting capabilities of exchange(), the *how* argument can be set to 'broadcast'. This forces the modifications to the object referenced by *shared_obj* as well as all buffered modifications to be immediately flushed to all remote processes. Under normal operation, the *how* argument is set to 'multicast'.

The exchange() function will update remote object copies for all objects that must be updated at the current time. If the *resync_flag* is true, exchange() will then block until all such remote processes have exchanged their buffered modifications with the local process. At this point, the local process is consistent with the remote processes involved in the exchange, for those objects just exchanged by this group of processes. The exchange() function will then use the s-function to recalculate (for each local process) the future time at which it must re-exchange information with this same set of processes. Note that every time an application process calls exchange(), it may do so with a different s-function. Figure 4 shows the pseudo-code for the operation of the exchange() call.

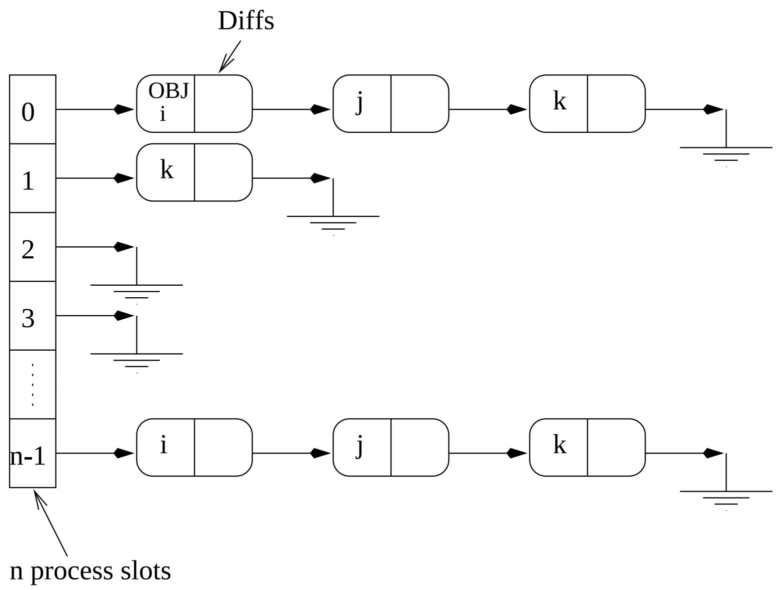


Figure 3: A sample slotted-buffer at process 2. Each slot may have a unique set of object diffs which will eventually be sent to the corresponding processes. In this example, process 3 is not in the exchange-list, which implies that we need not buffer and, hence, send updates to process 3. Likewise, updates for the local process need not be buffered, shown here as process 2.

3.2 Lookahead Consistency – Combining Temporal and Spatial Constraints

We have implemented several semantic-based consistency protocols using the S-DSO system. These protocols are tailored to the sample video game described in Section 2.1. The first protocol, called BSYNC, broadcasts all object updates to every other process after each object modification. The s-function for this protocol is only used to establish when data-races can occur and thus avoids them without recourse to a locking protocol. Each time the local process broadcasts a synchronous update, it blocks until all other processes have responded with their updates. In this way, each process exchanges with every other process after each object modification. When two processes are in contention for the same object[2], the process with the lowest ID is blocked, while the other process generates an event that potentially modifies the common object. During update exchanges, blocked processes simply exchange SYNC control messages and wait for all other processes to respond with their data updates and/or SYNC messages. Unblocked processes send a SYNC control message paired

[2]For our application, this occurs when two enemy tanks are within one block of each other. In this case, a block is a single object in a two-dimensional array representing the shared environment for the tanks.

```
exchange: /* Called after each object modification. */
  current_time++;
  /* Apply updates to local objects with data messages
     whose timestamp == current_time. */
  apply_buffered_data_msgs(current_time);

  if (how == multicast) {
    check exchange list to determine processes to exchange with
    at current time;
    send any appropriate buffered (data,SYNC) pairs;
    send (data,SYNC) pairs with current timestamps to remote
        processes;
  }
  else
    send (data,SYNC) pairs to all remote processes;

  for each process i not sent updates
    add object diffs to buffer-slot i;

  while (outstanding_replies to sent msgs) {
    for (each process i sent a (data,SYNC) pair) {
      wait for (data,SYNC) pair from process i with
        timestamp == current_time;
      if (data has timestamp > current_time) {
        buffer data;
        continue; /* Go back to waiting for correct message pair. */
      }
      else {
        apply updates locally;
        delete current exchange time from list for process i;
        call s-function to recalculate new exchange time for
          process i;
        outstanding_replies--; /* Decrement reply count. */
      }
  }
```

Figure 4: Pseudo-code for the exchange function.

with a data message.

In any one time quantum τ, the BSYNC protocol allows all processes to perform concurrent object writes, with each process performing at most one object modification before broadcasting its updates and waiting for responses from everyone else. Under this policy, a process can be executing in a time quantum at most one τ earlier or later than any other process. Thus, all processes' logical clocks are synchronized to within one time-tick, and their real-time clocks are synchronized to within τ seconds. This means that integer-valued logical timestamps must be sent with each update, to ensure that any early updates (by at most one logical time-tick) are not applied to object copies too soon. This also means that each process must buffer at most one early message from every other process. Unlike other consistency protocols, BSYNC does not require vector-timestamps and does not require unbounded buffer space for early update messages.

The BSYNC protocol is nothing more than a temporal consistency protocol. It knows when each process must be informed of updates. Furthermore, the BSYNC protocol uses application-level semantics (attributes) to avoid using locks or serialized access to all shared objects (which, in our application, form a totally shared environment) that can involve data-races.

Clearly, exchanging updates among all application processes is not a scalable solution to maintaining consistency across shared object copies. In response, we have developed two additional lookahead protocols, called MSYNC and MSYNC2. The MSYNC variants are similar in operation to BSYNC, but they perform synchronous exchanges with a multicast group of processes, rather than broadcasting exchanges to all other processes. This solution significantly reduces the number of message transfers and improves concurrency and asynchrony among processes.

Both MSYNC and MSYNC2 use of the exchange-list and slotted-buffer provided by S-DSO. For the video game application, the s-function for MSYNC computes the logical exchange times with each process (i.e., team of tanks) by halving the distance between the nearest tanks in any two teams. This approach is based on the assumption that, in the worst-case, one team's closest tank to an enemy will always move towards the other team's closest tank, and vice versa. Every logical time-tick, each team's tank moves from one block to another in the shared virtual environment. In the context of the video game, MSYNC assumes that any enemy tank in the same row or column of the shared environment as a local tank can potentially affect a local tank's next operation. MSYNC2 refines this assumption by only exchanging tank locations and their image information with those processes whose tanks could have moved into the same row or column as a local tank, and the distance to those enemy tanks is less than a d blocks.

The two MSYNC protocols capitalize on application-specific temporal and spatial constraints on shared objects. They are lookahead protocols that invoke a user-defined s-function to compute these temporal and spatial attributes. Like BSYNC, both MSYNC and MSYNC2 avoid using a lock-based scheme for resolving data-races. In the next section, these synchronous protocols are compared against entry consistency for the video game application.

4 S-DSO and Lookahead Consistency – An Experimental Evaluation

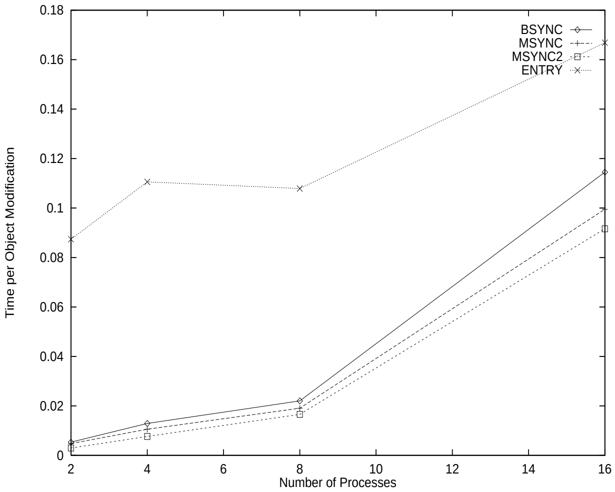
The purpose of this section is to demonstrate performance improvements when exploiting application-level spatial and temporal consistency knowledge for consistency maintenance. These improvements are attained for distributed applications with (1) high-frequency symmetric data accesses to shared objects, (2) unpredictable and limited locality of data access, (3) dynamically changing sharing behavior, and (4) potential data races. In addition, we show that consistency protocols that do exploit application-semantics can outperform shared object protocols not able to do so.

The three specific protocols exploiting temporal and spatial semantics evaluated in this section for the sample distributed video game are efficient, easily implemented, and they act to improve the degree of concurrency and asynchrony of the game's distributed processes. Protocol implementations are straightforward, since all manipulation of shared objects is left to the S-DSO system, with the exception of the user-supplied semantic function informing the system of when and with whom it must perform its consistency operations. For our application, the semantic function is less than 50 lines of C, and we envision that s-functions should be easy to implement for applications with more complex sharing behavior. Our system imposes relatively little overhead, due in part to the relatively 'thin' layer S-DSO places between the application and the underlying communication media.

The entry consistent protocol is implemented as efficiently as possible within the framework of S-DSO. Each object is associated with one lock, and a lock is acquired by sending a request to the associated lock manager. The lock managers are distributed evenly and statically amongst the processors in the system. Each lock manager maintains a list of pending writers and the identity of the owner of the most up-to-date object copy. Processes can acquire either exclusive write-locks or shared-read locks.

4.1 Results on a Workstation Cluster

All measurements presented in this section are conducted on a cluster of 16 SGI Indy workstations (each with a single MIPS R4400 processor and 64MBytes of memory) connected via switched 10Mbps Ethernet, and using TCP. In each of the experiments, the video game is configured to run non-interactively, and the 2D shared environment consists of 32x24 blocks (shared objects). There is one team per process and one process per physical processor, so that every process runs on its own machine in the workstation cluster. In all cases, team size is fixed to one tank. Each tank's objective is to reach the goal as quickly as possible, while trying to acquire as many bonus points as possible and avoid being destroyed by enemy tanks. For correct operation of the application, it is mandated that each tank base its decision to move, rotate, or fire at an enemy by looking a certain number of blocks in each of four directions: north, south, east and west. This implies that, at the very least, all blocks within range in each direction have to be consistent when the corresponding tank looks at the contents of those blocks. Each tank performs a simple iteration each logical clock-tick: (1) look at all the blocks within range in each direction, north, south, east and west; (2) generate a task to modify a block object; and (3) goto (1), unless the goal is reached or tank is destroyed. The consistency protocol ensures that the necessary blocks, in the range of a tank, are all always consistent.



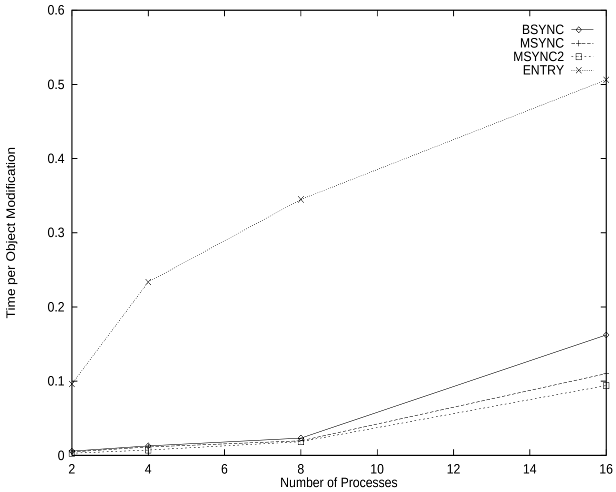


Figure 5: Average execution time per process normalized by average number of object modifications

Figure 5 depicts two separate graphs of the average execution times (in seconds) of processes. These times are normalized by the average number of object modifications performed by each process. Such normalization removes from these graphs random 'game' effects, such as favorable locations of tanks (e.g., when their initial locations are close to the goal). For all cases, we use the same random seed value to place the teams of tanks in the shared environment. In all left-hand figures, each tank can 'see' one object away in each direction[3]. This means that for entry consistency, each process must lock 5 objects before its tank can move; one lock is for the location of the tank itself, and four other locks are for all adjacent locations in which a tank might move. Such locking must be performed in order to prevent two tanks from attempting to move to the same block. In contrast, in all right-hand figures, each tank can 'see' 3 objects away in each direction. With entry consistency, this means that 13 objects have to be locked by each tank for every move, and 5 of those 13 are write-locked objects.

[3]For brevity, interpret 'range x' as corresponding to x visible objects in each direction.

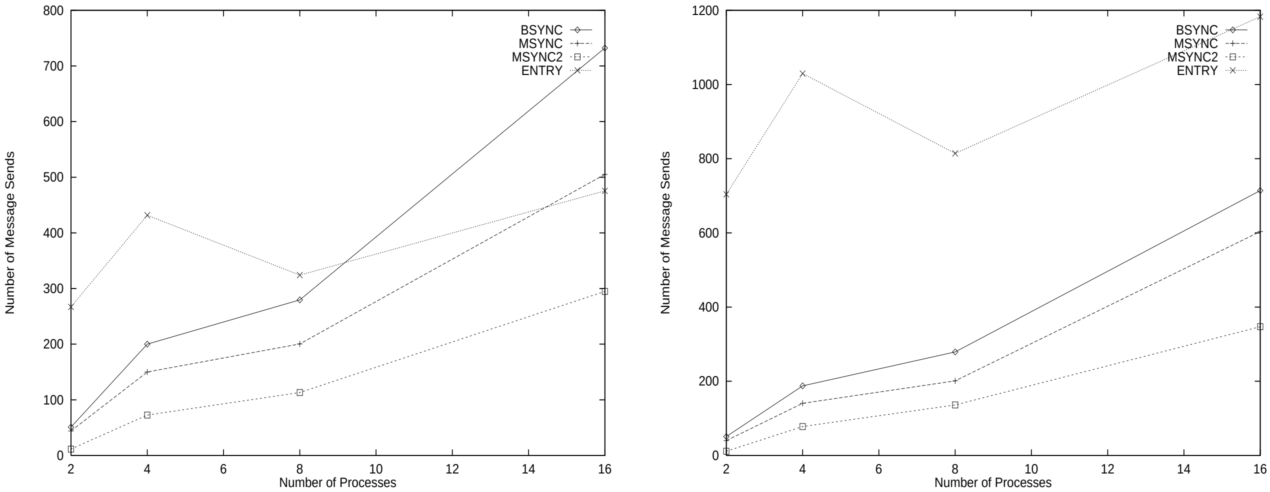


Figure 6: Total control and data message transfers by each protocol

From Figure 5, it is clear that entry consistency performs worse than all of the semantically richer synchronous 'lookahead' protocols, when the number of processes varies from 2 to 16. It remains unclear what will happen for significantly larger numbers of processes. The gradients of the left-graph, moving from 8 to 16 processes, suggest that eventually entry consistency will outperform all the synchronous protocols. However, when the number of dynamically shared objects is larger, as in the case of the right-hand graph, it appears that entry consistency will remain worse than the other three protocols, no matter how many processes there are. Note that MSYNC2 exhibits the highest performance, because its s-function captures application-level behavior more precisely than the functions used for MSYNC and BSYNC and thereby, avoids sending unnecessary updates to remote processes.

Figure 6 depicts the total number of message transfers for each protocol, as a function of the number of processes in each experiment. Once again, the left-hand graph shows results for a range of 1 object in each direction, while the right-hand graph shows results for a range of 3 objects in each direction. In these graphs, the number of messages is the total number of control and data messages used by each consistency protocol. In all cases, the average data size is the same as the average control message size; both are 2048 bytes. With a range of 1 and only two active processes, entry consistency performs significantly worse than the synchronous (lookahead) protocols. The major overhead to entry consistency is the number of lock-acquire messages it must send around the network to acquire all of the objects each process needs for each of its iterations. As the number of processes increases to 16, the left-hand graph shows entry consistency performing better. This is because BSYNC is sending exchange messages among all 16 processes on a broadcast basis, while in the same time interval, each entry consistent process only needs to acquire 5 locks and maybe obtain updated objects from corresponding owners. With our entry consistency implementation,

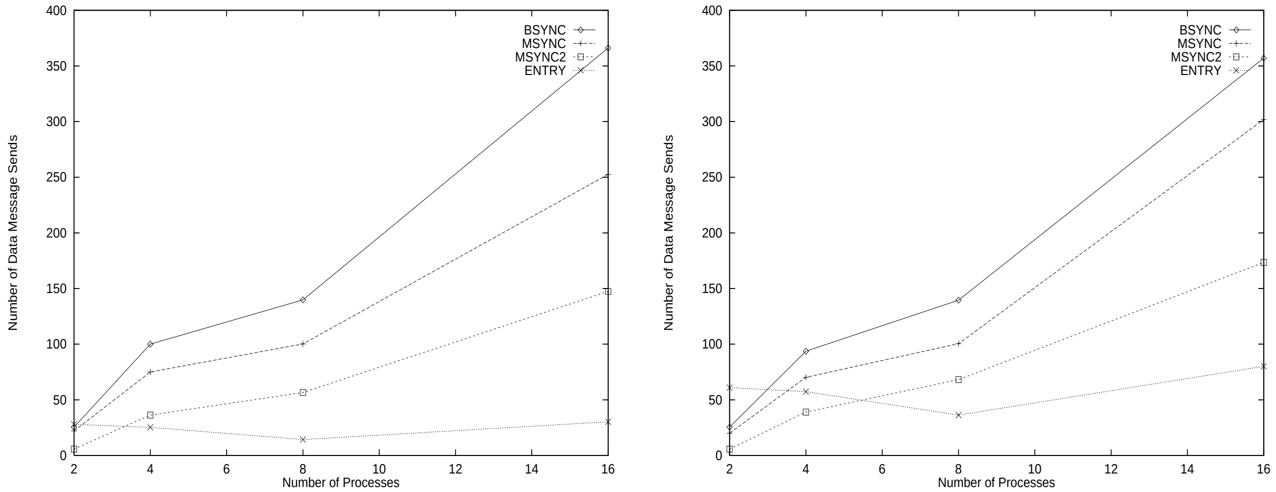


Figure 7: Total data message transfers by each protocol

the lock-managers for each lock-able object are evenly and statically spread across all host machines. With n processes there is a $1/n$ chance of the lock manager residing on the same machine as the process requesting the lock. Thus, as the number of dynamically shared objects increases (in our case, from 5 to 13), the majority of the lock-acquire messages are sent to remote machines. As a consequence, for 16 processes and when the number of shared objects is increased, entry consistency sends far more control messages than even BSYNC. Once again, MSYNC2 performs better than all of the other algorithms, at the cost of using a slightly more complicated s-function algorithm[4]. From Figure 5, it appears that entry consistency is spending a significant amount of time waiting for the acquire-lock messages to return the appropriate locks.

The problem with simply analyzing the total number of message transfers is that this doesn't differentiate between the number of control and data messages. Figure 7, shows only the number of data messages transferred by each protocol, as a function of the number of processes. It is interesting to note that entry consistency transfers the fewest number of data messages overall, in both graphs. The reason lies in the fact that entry consistency is a 'pull-based' protocol that only pulls updates to objects when it is certain it needs them. The three lookahead protocols are sending updates to objects unnecessarily, even in the case of MSYNC2. The problem with MSYNC2 is that it assumes worst-case prediction of the minimum time that one object modification will affect another. Clearly, with the tank application, it isn't certain that two tanks will ever come within range of each other, but

[4]The s-function complexity of MSYNC and MSYNC2 is $O(n^2)$, where n is the number of tanks in each team. The only difference between the two protocols is that MSYNC exchanges between processes that, in the worst-case, have tanks in the same row or column as each other, while MSYNC2 only exchanges with those processes with tanks in the same row or column *and* within range.

MSYNC2 assumes that this will eventually happen. Entry consistency, on the other hand, avoids updating objects that are never modified.

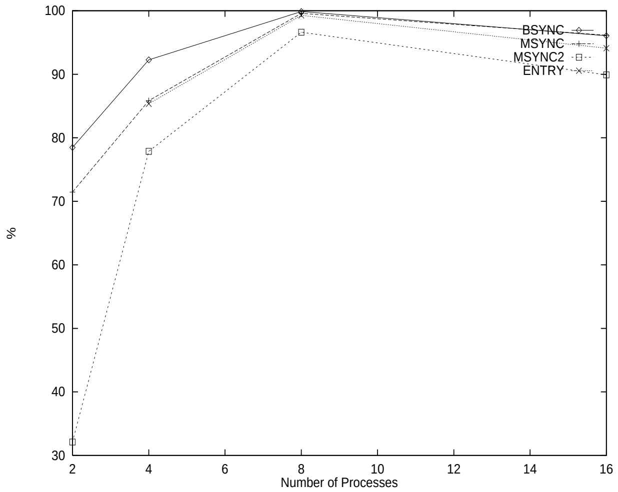


Figure 8: Protocol overheads as a percentage of the total execution time of each process

The final graph, shown in Figure 8 shows the overheads of each protocol during the execution of each application process. These overheads are measured for runs of the sample application with each tank having a range of 1 block in each direction. The major overheads for entry consistency are acquiring locks, and retrieving object updates upon acquiring those locks to modified objects. The locking overhead for entry consistency rises when the number of dynamically shared objects increases, since more locks have to be acquired each time. For the other three protocols, the cost of exchanging updates dominates the runtime cost. MSYNC2 has lower overheads compared to MSYNC and BSYNC because, on average, each process needs to rendezvous at an exchange time with a smaller number of other processes. Entry consistency has slightly less runtime costs than MSYNC and BSYNC, although it is more expensive than MSYNC2 for the experiments shown. In all cases, the protocol overheads dominate the execution time of each process, since the application processes have only a minimal amount of local processor processing to perform.

One problem, affecting the overhead of the lookahead protocols is that exchanges may be transitively dependent upon one another. Consider a process P_i at logical time t, synchronizing with process P_j. P_j is at logical time $t - \delta t$, waiting for P_k to exchange data. As a result, P_i is blocked, waiting for P_k even though P_i and P_k are not explicitly rendezvousing with each another.

The effectiveness of any lookahead protocol is dependent on how accurately we can predict the worst-case time of pairwise accesses by two or more processes to the same shared object, or at least spatially-related objects. If the data size is small but the number of dynamically shared objects is large, it would appear a lookahead protocol with a suitable s-function allows far greater concurrency and scalability that a pull-based protocol like entry consistency. For large numbers of dynamically shared objects, we believe that entry consistent processes are spending far greater amounts of time in blocked modes, while waiting for locks that are potentially held by other processes. Although a lookahead scheme might send more data messages under the same conditions, it is able to do so with far less blocking overhead and therefore, exhibit better performance than the entry consistent scheme.

By the time this paper appears, we will have explored two additional interesting issues: (1) an analysis of the blocking overhead of lock-based protocols such as entry consistency, versus the overheads of multicast synchronization in generic lookahead schemes, and (2) the effects of different data sizes. Specifically, concerning (2), it is interesting to understand the effect of changes in the resolution of shared objects, where either more or less data is transferred in each data message carrying object state. In realistic distributed command and control applications, data sizes may be large when sensor images of enemy tanks are employed, for instance. Furthermore, *diffs* may not be suitable when images are encrypted. Similarly, when multiple 'players' interact via shared images of scientific data, as done for immersive virtual environments (e.g., the shared CAVE systems explored in the recent I-Way demonstration), substantial amounts of data may have to be shared and transported from one 'move' to another. In the final version of this paper, we will present measurements evaluating the lookahead vs. the entry consistency protocols, again using the game application.

5 Related Work

Birman et al[8] have built the ISIS toolkit as a programming platform for group communication. Their system supports a wide range of multicast protocols but they focus on causality amongst process groups, while our work looks at application-specific shared object protocols. [5, 37] explores the notion of Δ-causality in unreliable networks, supporting multimedia real-time collaborative applications. Δ-causality is somewhat different from our work in that it deals with the delivery of messages that respect causal ordering only for messages received within their deadline-time, Δ. Messages arriving greater than Δ time units after their original transmission are never delivered in such a system. Yavatkar has built a Multi-Flow Conversation Protocol (MCP)[37] to support the temporal synchronization of multimedia collaborative applications, with the explicit ability to support Δ causally-ordered message transfers. There has been much work by people such as[35] to investigate temporal synchronization in distributed multimedia systems. However, this work is primarily focused on synchronizing media streams across networks, and detecting and bounding the levels of asynchrony between streams to meet the quality of service constraints of different media types.

[31] surveys the various synchronization mechanisms used in distributed shared memory systems. Our work is not restricted to shared memory abstractions but generalizes to shared

object systems. We hope to look at shared objects that involve data and methods for manipulating that data. Furthermore, the DSM systems described in [31] are traditional systems that have little or no support for exploiting application-level semantics to improve performance.

Finally, [15] describes the Unify system for exploring scalable approaches to designing distributed multicomputer systems. They mention spatial and temporal consistency, but their notion of spatial consistency differs from ours in that it determines the relative order of data contained in various replicated objects, such as log files, associative and sequentially-ordered memories. In comparison, this paper's formulation of spatial consistency determines when processes should be updated with changes to shared objects, based on the locations currently accessed by these processes in shared space.

6 Conclusions and Future Work

We have implemented a framework, called S-DSO, which permits applications to specify the semantics of when and which processes should see updates to shared memory objects of varying sizes, using additional parameters associated with object accesses, called 'attributes'. We have shown that for applications with dynamic sharing behavior, poor spatial locality, data-races, and symmetric object accesses, conventional DSM protocols like entry-consistency may be inadequate, particularly when there is a large number of shared objects.

The S-DSO system presented in this paper will be one of several configurable substrates of the CORBA-compliant, distributed object system for high performance applications now being developed at Georgia Tech, called COBS. Once integrated within the COBS framework, S-DSO applications may execute on heterogeneous distributed platforms, across a variety of network links. As a result of this integration, we expect to be able to investigate the effects of wide area as well as the effects of high performance communication media on consistency protocols, and on applications like the video game presented here and the large-scale distributed virtual environments now being developed at Georgia Tech and elsewhere[32].

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report 1051, University of Wisconsin, Madison, September 1991.

[3] Mustaque Ahamad, Ranjit John, Prince Kohli, and Gil Neiger. Causal memory meets the consistency and performance needs of distributed systems! In *SIGOPS*, 1994.

[4] Mustaque Ahamad, Gil Neiger, Prince Kohli, James E. Burns, and Phillip W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, Aug 1995.

[5] Roberto Baldoni, Achour Mostefaoui, and Michel Raynal. Causal delivery of messages with real-time data in unreliable networks. *Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 10(3):245–262, May 1996.

[6] John K. Bennett, John B. Carter, and Willy Zwaenpoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[7] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[8] Kenneth Birman, Andre Shiper, and Pat Stephenson. Lightweight causal and atomic group multicast. Technical Report 91-1192, Department of Computer Science, Cornell University, February 1991. To appear in *ACM Transactions on Computer Systems*.

[9] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *4th Symposium on Experimental Distributed and Multiprocessor Systems*, pages 57–71, September 1993. Also available as MSR-TR-93-1, Microsoft Res. Lab., Sep. 1993.

[10] Russell Carter. Nas kernels on the connection machine. Technical Report RND-90-005, NASA Ames Research Center, April 1990.

[11] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. In *Proc. of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 227–246. USENIX, September 1993. Also as TR# GIT-CC-93/25.

[12] Greg Eisenhauer and Karsten Schwan. Design and analysis of a parallel molecular dynamics application. *Journal of Parallel and Distributed Computing*, 35(1):76–90, May 25 1996.

[13] C. Cruz-Neira et al. The cave audio visual experience auto virtual environment. *Communications of the ACM*, pages 65–72, June 1992.

[14] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[15] James Griffoen, Raj Yavatkar, and Raphael Finkel. Extending the dimensions of consistency: Spatial consistency and sequential segments. Technical Report cs248-94, University of Kentucky, April 1994.

[16] S. Gronemberg and D. Marwood. Real-time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Cooperative Support for Cooperative Work, ACM press*, pages 207–217. ACM, 1994.

[17] X Consortium Working Group. Fresco specification draft version 0.7, April 1994.

[18] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distrbuted shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[19] Rakesh Jha, Mustafa Muhammad, Sudharkar Yalamanchili, Karsten Schwan, Daniela Ivan-Rosu, and Chris DeCastro. Adaptive resource allocation for embedded parallel applications. conference submission, August 1996.

[20] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Technical Report Rice COMP TR93-214, Department of Computer Science, Rice University, 1993.

[21] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium of Computer Architecture*, 1992.

[22] Prince Kohli, Mustaque Ahamad, and Karsten Schwan. Indigo: User-level support for building distributed shared abstractions. In *Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC-4)*, August 1995.

[23] Leonidas I. Kontothanassis and Michael L. Scott. Distributed shared memory for new generation networks. Technical Report TR 578, University of Rochester, March 1995.

[24] Robin Kravets, Ken Calvert, and Karsten Schwan. Dynamically configurable communication protocols and distributed applications: Motivation and experience. submitted to Multimedia Computing and Networking 1997 (MMCN97).

[25] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *IEEE*, 1990.

[26] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, pages II 94–101, Aug 1988.

[27] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4):321–359, November 1989.

[28] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[29] Jim McDonald and Karsten Schwan. Ada dynamic load control mechanisms for distributed embedded battle management systems. In *First Workshop on Real-time Applications, New York*, pages 156–160. IEEE, May 1993.

[30] J. G. Mitchell, J. G. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the spring system.

[31] Mahendra Ramachandran and Mukesh Singhal. On the synchronization in distributed shared memory systems. Technical Report OSU-CISRC-10/94-TR54, Ohio State University, 1994.

[32] Beth Schroeder, Gerg Eisenhauer, Jeremy Heiner, Vernard Martin, Karsten Schwan, and Jeffrey Vetter. From interactive apllications to distribted laboratories. Submitted to the Visual Supercomputing special issue of IEEE Computational Science and Engineering - June 1996, 1996.

[33] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 198–204. IEEE, May 1986.

[34] Jon Siegel. *CORBA - Fundamentals and Programming.* John Wiley and Sons, 605 Third Ave, New York, NY 10158. ISBN 0471-12148-7, 1996.

[35] Sang Son and Nipun Agarwal. Synchronization of temporal constructs in distributed multimedia systems with controlled accuracy. Technical Report CS-93-57, University of Virginia, October 1993.

[36] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandam. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.

[37] Raj Yavatkar. Mcp: A protocol for coordination and temporal synchronization in multimedia collaborative applications. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems*, pages 606–613. IEEE, 1992.