Robust State Sharing for Wide Area Distributed Applications*

Brad Topol
Mustaque Ahamad
John T. Stasko

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{topol,mustaq,stasko@cc.gatech.edu}

Abstract

In this article, we present the Mocha wide area computing infrastructure we are currently developing. Mocha provides support for robust shared objects on heterogeneous platforms, and utilizes advanced distributed shared memory techniques for maintaining consistency of shared objects that are replicated at multiple nodes to improve performance. In addition, our system handles failures that we feel will be common in wide area environments. For example, to ensure that the state of an object is not lost due to a node failure, updated state of the object can be disseminated to several other nodes. The overhead of such state dissemination can be controlled based on the level of availability needed for shared objects. We have used an approach that makes use of multiple communication protocols to improve the efficiency of shared object state transfers in Mocha. We also provide an empirical evaluation of our prototype implementation for local area, wide area, and home service networks and show that several classes of applications may be written using the mechanisms provided by the system.

*This work was supported in part by NSF grants CCR-9619371 and CDA-9501637, and the Broadband Telecommunications Center at Georgia Tech.

1 Introduction

The growth in popularity of the World Wide Web (WWW) has resulted in the development of a new generation of tools tailored to Internet computing activities. Prominent examples include the Java programming language and Java capable Web browsers. These Web spinoffs are having a profound impact on the field of distributed computing. Whereas distributed computing has traditionally focused on improving the functionality of local clusters of computers, technology is progressing such that wide area computing networks are now becoming a popular target environment for research in distributed computing.

With wide area distributed computing environments, geographically distributed resources such as workstations, personal computers, supercomputers, graphic rendering engines, and scientific instruments will be available for use in a seamless fashion by parallel applications[16, 12]. Many envision that it will be possible to transport application code to remote sites in the wide area virtual computer where it may be executed in the presence of needed resources. The area of research devoted to bringing this vision to reality in the context of scientific applications is referred to as *metacomputing*. A metacomputing system is the software infrastructure which orchestrates collections of hosts and networks into functioning as a large virtual computer for use by parallel and distributed applications.

In the near future, applications such as electronic commerce (e.g., service to the home) will be deployed in the wide area distributed computing environment. Such applications require more advanced capabilities than those provided by a standard web browser, and could benefit from advances made in wide area distributed computing. Electronic commerce applications, such as those that facilitate collaborative home shopping, are a prime example of this. In these types of applications, consumers in different geographical locations can shop together in "virtual stores". These virtual stores can be 2D or 3D. In the virtual stores, each consumer can quickly acquire his or her peers' opinions of the items the consumer is considering purchasing. Group members may locate and suggest items that others in the group have overlooked. Essentially, the consumers are able to maintain in the virtual environment shared access to information as well as the high level of interaction that exists naturally when shopping together in a real store. Shared information access and interactivity are facilities that are not well supported by the mechanisms available in wide area computing environments. Thus, collaborative applications such as these are examples of wide area applications that are poised to become reality once wide area computing environments supporting these mechanisms become available.

Wide area computing environments have several salient differences from traditional local area network computing environments. For example, there is less autonomy and control over resources (e.g., workstations) as many of the resources are remotely located and controlled by others. Furthermore, failures in a wide area computing environment are a relatively common occurrence. Failures, when detected by timeout mechanisms, result from network congestion and varying workloads at nodes. Also, the autonomy of nodes can result in a remote node reboot or the owner of such a node can choose to terminate foreign tasks.

There are many problems that need to be solved to enable wide area applications to become common. One important issue that we address here is a robust shared object model for wide area applications. Because these applications are themselves distributed applications, they are dependent on some mechanism for sharing state across workstations to allow processors to cooperatively work together. The shared memory and shared object models are attractive for state sharing because they are simpler to program than standard message passing. They have been explored in [19, 4, 14, 1]. Our focus is on providing efficient state sharing that is integrated with failure handling support.

The following are the contributions of our work in robust state sharing for a wide area computing environment:

- The Mocha system, which is a wide area computing infrastructure that provides basic facilities such as remote evaluation support, security, debugging support, and a highly scalable thread-safe network communication library[28].
- Support for shared objects on heterogeneous platforms. To improve performance and mitigate communication latencies, copies of objects can be created and accessed locally.
- Object sharing support that utilizes advanced distributed shared memory techniques for maintaining consistency of shared objects.
- Fault tolerance support that allows its overhead to be controlled based on the level of availability needed by an application for its objects.
- The design of a "multiple protocol" approach which combines the capabilities of Mocha's network library and the TCP protocol to support efficient transfer of object replicas in a scalable fashion.
- Applications have been written that demonstrate that the system's mechanisms are capable of supporting several classes of applications.
- An evaluation of the system in local area, wide area, and home service networks.

The following section provides an overview of Mocha and includes a description of its program interface as well as its shared object model. An overview of Mocha's support software, with an emphasis on basic algorithms and key implementation features, follows. Next, failure handling refinements are presented. We then present several applications that illustrate the flexibility of Mocha's mechanisms. We also provide an evaluation of both the overheads of Mocha's state sharing capabilities as well as its application level performance. Related work is then discussed and the concluding section reports on ongoing and future work.

2 Mocha Overview

To motivate the need for a framework such as Mocha, we begin with a description of a potential wide area distributed application. Consider an application in which a consumer at home wishes to add a new formal dinner table place setting composed of flatware, plates, and glassware. At the consumer's home, a graphical user interface is executing which allows various flatware, plates, and glassware to be viewed together so that the consumer may "mix and match" these items and end up with a pleasing coordinated table setting. Additionally, a sales associate located at the retail outlet may also have a copy of the graphical user interface which permits the associate to see what the customer is selecting and may suggest alternatives which are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. In this scenario, it is expected that the platforms on which the GUI executes in each of the homes would be vastly different from the platform at the retail outlet.

For a wide area computing infrastructure to support the above scenario, there are several features it must provide. First, it must provide support for shipping platform independent

GUI code to the remote sites and allow the code to execute in a secure environment. Second, the infrastructure must provide support that allows the GUI's to share objects among each of the sites. Finally, the infrastructure must enable applications to be written such that they are resilient to failures at some of the remote sites.

The Mocha system is capable of supporting an application scenario such as the one described above. It is written entirely in Java and provides the application programmer with a framework for developing wide area distributed applications in the Java language. In addition to providing constructs for distributed computing such as the remote spawning of threads and state sharing between these threads, the Mocha system provides features that enable it to serve as a wide-area computing environment. These include security, the ability to ship and dynamically link in application code (i.e., remote evaluation support[25]), and basic debugging and event logging facilities that provide insight into execution of code at remote locations.

In order to start a distributed application, Mocha assumes that at each site that wishes to participate in the execution of the distributed application (i.e., a workstation or Java capable PC), someone will startup one of its *Site Managers* which listens on a well known port. The *Site Manager* listens on this port for requests to utilize the workstation and is responsible for controlling the number of true processes on the workstation that are allocated for use by remote tasks. These processes are referred to as *Mocha Servers* because they are able to "serve" a thread executing on behalf of a remote task with anything it may need. For example, a *Mocha Server* provides a thread with the ability to receive and link in more application code, as well as other features such as communicating results and reporting error conditions.

Mocha's basic constructs for distributed computing are fashioned after constructs for popular local area distributed computing environments such as PVM[26]. However, the primitives have been modified to take advantage of Java's object oriented capabilities.

In the Mocha environment, the application is composed of threads which execute in the address spaces of Mocha Servers. These threads may be initiated (i.e., spawned) using a method from the provided `Mocha` class. A typical section of code executed by a thread to spawn another Mocha thread is shown in Figure 1. When a new instance of the Mocha object is created, a host file is read which provides a list of potential sites at which remote threads may be spawned. The Mocha system provides a tool to generate this host file. As shown in the code fragment, a `Parameter` object is utilized for organizing the parameters that will eventually be sent to a remotely spawned thread. The actual spawn is performed by calling the `spawn()` method of class `Mocha` with the name of the Mocha thread class that is to be spawned and the `Parameter` object that is to be sent to the remotely instantiated thread. Other spawn methods are available which allow the application to specify the exact host in the host file on which a remote thread should execute. In Figure 1, the class being spawned as a thread is the "Myhello" class. This and other classes that are intended to be spawned are provided by the application programmer.

In contrast to a spawn performed by a network computing environment such as PVM, Mocha's spawn provides remote evaluation support which allows it to transport and dynamically link in thread code at a remote Mocha server as necessary. Mocha's model for remote evaluation is that of an initial "push" of application code followed by "demand pulling" of new application code object classes as they are encountered during execution.

Mocha threads may be derived from any Java class that implements the `MochaTask` interface. Through this interface the Mocha runtime is able to provide the application thread with a `Mocha` object that is essentially a "travel bag". The `Mocha` object currently provides a `Parameter` object from which the remotely evaluated task may retrieve the initial execution parameters denoted in the `spawn()` method that started this task. Also provided is a

```
import mocha.*;
public class TestMocha  {

   public static void main(String args[]) {
      Mocha mocha;
      ResultHandle rh;
      Parameter p;

      mocha = new Mocha();
      p = new Parameter();
      p.add("param1", 5);            // create parameters to
                                     // send to remotely evaluated class

      rh = mocha.spawn("Myhello", p); // spawn class named Myhello
                                      // remote site requests other
                                      // classes as necessary

   }
}
```

Figure 1: Mocha application code that spawns a class for remote evaluation.

`Result` object in which the task may place results. Additionally, the `Mocha` object provides a variety of useful objects and methods that support remote printing, remote stack dumps, support for making replicated object copies and accessing the replicas, and enables a thread to recursively spawn other wide area computing threads. Figure 2 shows an example user class that utilizes the `Mocha` class to acquire initial startup parameters, perform remote printing and stack dumps, and then return its subresult. In this example, `Myhello` is a user class that has been written in a fashion that permits it to be shipped and executed at a remote site. After the class is shipped, the Mocha runtime creates a new thread that begins executing at the `mochastart()` method. The Mocha runtime provides this thread with a `Mocha` object from which the thread may invoke the travel bag related activities described above.

2.1 Mocha's Shared Object Model

Mocha's shared object model permits threads in separate Mocha Servers to share state. The primary goals of the model are (i) to provide an efficient scheme for the consistency maintenance of shared state between threads across Mocha servers running at different nodes and (ii) support the sharing of complex objects.

2.1.1 Maintaining Shared State Consistency

Mocha's model for maintaining shared state is motivated by weakly ordered shared memory models such as Entry Consistency[4], Release Consistency[14], and Lazy Release Consistency[19]. These models have been shown to provide highly efficient state sharing implementations. In these memory models, shared memory is guarded by synchronization constructs (i.e., lock acquire and release) and is only made consistent with the most recent updates when certain synchronization points in the code are reached. Thus, in these models, the shared memory may only be properly accessed between lock acquire and release synchronization points. This approach towards maintaining consistency has been shown to reach performance levels that are comparable to message passing models and thus is a very strong starting point for Mocha's shared object model.

```
import mocha.*;
import java.lang.*;

public class Myhello implements MochaTask {
  public Myhello() {
  }
  public void mochastart(Mocha mocha) {
    double start=0.0, sum = 0.0;
    try {
      start = mocha.parameter.getdouble("start");
      sum = start + 1;

      mocha.mochaPrintln ("Returning as a return value " + sum);
      mocha.result.add("returnvalue", sum);
      mocha.returnResults();

    } catch (MochaParameterException e) {
      mocha.mochaPrintStackTrace(e);
    } catch (Throwable t) {
      mocha.mochaPrintStackTrace(t);
    }
  }
}
```

Figure 2: Remotely evaluated user class that illustrates the use of the Mocha object.

It may appear that the object consistency model, a variant of entry consistency[4] that is chosen by us, is not appropriate in wide area computing environments. We believe there are several benefits of this model that make it appropriate for wide area computing. First, since the model requires an explicit association of objects to locks, should a failure occur, it can be quickly determined which objects need to be recovered. This permits fault tolerant refinements discussed in the next section to be implemented in a more efficient fashion. Second, the model does not require all replicas to be synchronously updated when a local update occurs. Thus, in some cases, when there are many local updates occurring in succession, it may not be necessary to incur the latency of updating remote replicas. This can dramatically improve the response time for interactive applications. Finally, while the consistency model provided may be too strong for some applications, the flexibility provided by lock and object associations can be utilized to provide weaker notions of consistency which may be appropriate for some applications. This approach is described more thoroughly in Section 5.3.

Mocha's shared object model consists of `Replica` and `ReplicaLock` objects. All objects that are desired to be shared in the Mocha system must be of type Replica or subclass from it. Replicas are not required to represent a fixed size of data; the amount of shared data contained in a Replica may grow and shrink as the needs of the Replica vary during application execution. Replicas may contain both homogeneous arrays of primitive data types as well as bona fide Java objects which are serializable[22]. A thread that wishes to create a shared object might use the following section of code:

```
int myarray[] = new int[10];
Replica replica1 = new Replica ("flatwareIndex", mocha, myarray, 5);
```

Here, the Replica constructor utilizes methods in the Mocha object to create a shared object that is an array of integers, will have 5 replicas, and is identified by the string *flatwareIndex*. Threads that wish to acquire replicas of this object would use a similar constructor:

```
Replica replica1 = new Replica ("flatwareIndex", mocha);
```

```
int intarray[] = new int[5];
String string = new String("Hello World");
Replica replica1 = new Replica ("flatwareIndex", mocha,  intarray, 5);
Replica replica2 = new Replica ("plateIndex", mocha, intarray, 5);
Replica replica3 = new Replica ("glasswareIndex", mocha, intarray, 5);

// The following is a generated subclass of Replica used for sharing
// a Java String,  a general purpose object.
StringReplica replica4 = new StringReplica ("text", mocha, string, 5);

// need to create a ReplicaLock
ReplicaLock rlock1 = new ReplicaLock( 1, mocha);

// associate Replicas with lock
rlock1.associate(replica1);
rlock1.associate(replica2);
rlock1.associate(replica3);

rlock1.lock();  // Lock must be acquired and Replicas are
                // made consistent before proceeding
// Replicas may now be accessed or updated in a consistent manner
replica1.intdata[0] = 1;
replica2.intdata[0] = 1;
replica3.intdata[0] = 1;
replica4.string  = "Good Choice";
rlock1.unlock();  // Lock is now released
```

Figure 3: Associating and locking shared object Replicas.

Here it is not necessary to denote what type of shared object the replica refers to as this information is already known to the Mocha runtime. For programming purposes, the Replica class provides signature methods that enable the application to determine the type and amount of data the Replica represents.

In their generic form, Replica objects enable the Mocha system to replicate or *publish* shared data at either all or a subset of cooperating Mocha threads. Since multiple nodes share Replica objects, in order for such objects to serve in a manner that resembles a shared object, each Replica must be *associated* with a ReplicaLock object. This association is exploited to efficiently maintain consistency between Replicas. This approach allows Mocha's runtime to exploit the synchronization present in an application to improve the performance of consistency maintenance, a technique well established by advanced DSM systems[4, 19]. Figure 3 shows a typical section of code in which Replicas are associated with a ReplicaLock to enable them to serve as a form of consistent distributed shared memory. These type of Replicas could be used in the service to the home table setting coordinator application described earlier in this section. In this example, several shared index Replicas are provided. These shared objects would be utilized to control which images of the retail items are presented at the same time. A String object is also provided to allow the users of the GUI's to send comments to each other. As shown in Figure 3, these Replicas are associated with a ReplicaLock. Once the ReplicaLock's `lock()` method returns, the lock has been acquired and the Replicas are consistent. At this point the Replicas may be accessed and modified as desired. The ReplicaLock may then be released via the `unlock()` method.

2.1.2 Supporting General Purpose Java Objects

By themselves, Mocha's Replica objects are able to store homogeneous arrays of Java's primitive data types such as `byte`, `int`, or `double`. However, it is expected that many applications may desire to share more complex objects that are either user defined or those provided by Java itself. Examples of the latter include `java.util.Date`, `java.util.Hashtable`, *etc.* These objects are more complicated to support as shared objects because they must be serialized into a byte array to enable them to be transported over the network. Java provides an object serialization package that greatly simplifies the process of serializing these objects. Typically, only a few lines of user code are required to serialize or unserialize an object. With object serialization available, the only difficulty is to guarantee that objects are serialized before they may be sent to another thread and then unserialized when an updated copy of a replica is received.

Mocha supports the above functionality by allowing users to subclass from the standard Replica object and add a user specific complex object to the derived class as the object to be shared. Two methods are provided in Replica that may be overloaded in the derived class with serialization and unserialization code appropriate for the complex object that is desired to be shared. The Mocha runtime will automatically call these methods when it needs to marshal or unmarshal these shared objects. While creating a derived class and inserting the appropriate serialization/unserialization code into the overloaded methods is quite straightforward, it would nonetheless require extra coding effort from the application developer. To prevent this, a tool, `MochaGen`, is provided which generates a custom subclass of Replica which contains the object the user desires to share as well as a new custom constructor and the appropriate serialization/unserialization methods. Thus, complex objects may be shared in a manner very similar to Mocha's standard Replica object.

It is worth noting that more experienced Java users are permitted to replace the code that the `MochaGen` tool generates for serialization/unserialization with more optimized code when a priori knowledge regarding the use of objects is available. For example, assume a large object is being shared in which only a few integer variables might change value. Because only a small amount of state changes, much more efficient versions of the serialization routines are possible. Instead of serializing the whole object, more advanced routines might simply serialize/unserialize only the few integer variables that have been modified.

3 Basic Object Consistency Algorithm

The implementation of Mocha's shared object replicas utilizes a daemon thread at each site manager, and a single synchronization thread at the home site. The home site in our system is simply the site at which the initial application thread executes. All of the threads mentioned above are implemented using the standard Java threads library. All objects that the application threads wish to share are registered with the local daemon thread allowing it to directly access the shared objects themselves. The daemon threads execute at maximum priority which guarantees they may interrupt lower priority application threads when necessary. This behavior permits the daemon threads to perform the transfer of shared objects to remote sites as well to accept shared object consistency updates from remote sites when necessary. The asynchrony present in this architecture allows Mocha the flexibility to dynamically configure itself during execution from a streamlined shared object system to one that can handle common failures.

The synchronization thread handles *lock* and *unlock* requests from application threads. In addition, the synchronization thread directs daemon threads to perform operations such as transferring replicas to remote sites and is also responsible for deducing when such activities are necessary.

As described in the previous section, the Mocha shared object model consists of replicas which must be associated with a ReplicaLock object thereby supporting a variant of entry consistency[4]. When an application thread desires exclusive access to shared replicas, it calls the ReplicaLock's `lock()` method. The method call sends a message to the synchronization thread that states its desire to acquire the lock, and it suspends the thread's execution until lock acquisition is granted and up-to-date copies of the replicas are sent to this site, or it is determined that up-to-date copies are already present at this site.

An application thread releases access to shared replicas by calling the ReplicaLock's `unlock()` method. This method call is responsible for sending the synchronization thread a message that indicates that the lock is being released. The message contains the identifier for the lock and also an updated version number associated with the replicas.

The more complex aspects of Mocha's shared object support may be found in the operation of the daemon threads and the synchronization thread. Each daemon thread is responsible for startup activities such as initialization, and responding to requests regarding the transfer and acceptance of replicas. The synchronization thread at the home node is responsible for granting locks, queuing requests, and deducing whether a new version of replicas must be sent to an application thread.

Several aspects of the basic algorithm are worth emphasizing. First, replica data is transmitted directly from one application thread address space to another application thread without having to be transmitted via the (central) synchronization thread. This allows the system to exploit locality which may exist in a wide area distributed computing environment. Second, application threads never assume that replicas will arrive but instead examine a flag to determine this aspect. This approach provides the flexibility needed to efficiently augment the basic algorithm with advanced features such as a "push-based" replication scheme that is described in the next section. Third, the user may notice that version numbers are being maintained however not yet used in any significant manner. Their purpose is also described in the next section. Finally, for simplicity, we described the basic algorithm assuming exclusive locks. It can easily be modified to support shared (i.e., read-only) locks. More extensive implementation details regarding the basic object consistency algorithm may be found in the Appendix.

4 Fault Tolerant Refinements

As previously discussed, we anticipate failures to be more common in wide area computing environments than traditional local area network computing environments. This has motivated us to modify Mocha's basic state sharing algorithm by incorporating fault handling refinements. The failure of a remote application thread can result in the following shortcomings in the basic algorithm. First, if an application thread which has released the lock to a shared object fails before another thread has pulled a copy of the shared object, this most recent version of the shared state will be lost. Thus, the next thread that desires a copy of this shared object will not be able to acquire this most recent version of the object. Second, if a thread which currently has acquired a lock fails, then no other thread will be permitted access to the shared object and this may result in a deadlock in the system.

The object sharing system can be enhanced to provide fault tolerance using a number of different techniques. These include transactional support, server replication and virtual synchrony, and checkpoint/restart mechanisms. Our focus is on developing basic support that can be used to make the object sharing system robust with minimal overhead. As a result, we focus on the following two refinements:

- We detect failures of remote Mocha servers using timeouts and take appropriate actions to handle such failures (e.g., release lock held by a failed server).
- To ensure that a shared object's state is not lost because of a node failure, an application can choose to disseminate the object's state to multiple sites. Such dissemination is done to achieve high availability even when it is not required by the consistency protocol.

Our refinements are based on the assumption that the home node, where the synchronization thread executes, is less prone to failures because it is controlled by the user initiating the application. Thus, the focus is on dealing with failures of remote nodes.

Mocha allows objects to be replicated for state sharing. We also exploit the replicas to increase the probability that there is an operational thread that has an up-to-date copy of the desired objects. Support for updating multiple replicas of objects has been added in Mocha by permitting ReplicaLock objects to reconfigure themselves to employ a "push-based" update scheme. In this approach, when a thread is ready to release access to the replicas associated with a ReplicaLock, it may disseminate a copy of the replicas to a subset of threads that have registered to utilize these replicas. This is supported by having ReplicaLocks maintain state information regarding other application threads that have registered that they also desire access to the replicas. Specifically, the ReplicaLock keeps track of the daemon threads associated with these application threads. Recall that these daemon threads have access to the shared replica objects and may apply the disseminated updates directly. Assume that R is the number of such daemon threads that share the copy of an object.

Additionally, ReplicaLock objects maintain state information regarding UR, the number of up-to-date copies of the shared object. Thus, UR represents a subset of the object replicas that store the most up-to-date values of the objects. If no fault tolerance is desired, $UR = 1$ and only the node that produced the current value stores it. The new value will be sent to other nodes only when they acquire the lock associated with the object. When $UR = k$, the value will be sent to k nodes prior to releasing the lock even when such dissemination is not required by the consistency protocol. The changing of R and UR (and hence the reconfiguration of the level of availability of shared objects) may be performed by either application threads or by the Mocha runtime which has access to this state information through its daemon threads.

With application threads now able to dictate the amount of update dissemination, it is possible that a thread has the most up-to-date object copy when it requests a lock. In this case, we do not transfer the replica with the lock. The synchronization thread can decide whether or not a new replica value must be sent to an application thread based on the information it maintains about which UR nodes received the updated value of the object. For example, if an application thread has configured a ReplicaLock to disseminate its version of replicas to a subset of threads and one of these threads now desires access to these replicas, it is no longer necessary for the synchronization thread to direct a daemon thread to transmit the new version because it is already there. To permit the synchronization thread to make this type of decision, the ReplicaLock's `unlock()` method has been modified to include in its lock release method a set of identifiers for the daemon threads to which it has disseminated copies of the replicas. When granting the lock to next application thread that desires it, the synchronization thread consults this set to determine if the application thread requires a new copy of the replicas.

With replication support in place, it is now possible for failure resiliency to be integrated into the Mocha system. The following subsections present the modifications necessary for failure detection and handling.

Failure of Non-Lock Owning Application Thread. A failure by an application thread that does not own a lock may be detected in several ways. First, if this thread had the most up-to-date version of the replicas then the synchronization thread will contact the thread's daemon thread to perform a transfer of replicas. Assuming a fault was due either to a system crash or a local user terminating the site manager process, the daemon thread will also no longer be executing. Thus, the synchronization thread will detect the failure when the transfer message it sends to the daemon thread times out. In this particular instance, the synchronization thread handles the failure by polling other daemon threads to obtain the most recent version of the replicas available. If the new object values are disseminated to multiple nodes then in all probability (depending upon the number of failures) the most recent version of the replicas will be available and the synchronization thread may forward the replicas to the requesting thread. If the values produced by the failed site were not disseminated then the synchronization thread will only receive older versions of the replicas. It can then examine version numbers to forward the most recently available old version of the replicas. This weakened consistency may be appropriate for certain classes of applications such as service to the home whereby the home user can recognize unwanted characteristics of the old version and reapply the appropriate updates to the replicas.

A second way the failure of an application thread may be detected is when another application thread utilizes the update facilities and attempts to disseminate its version to other daemon threads. Here, we again assume that the failure of an application thread implies its associated daemon thread will also fail. When attempting to disseminate the new replicas, the send message will time out. The failure has been detected and can be handled by choosing another daemon thread at another site to receive a copy of the new version of replicas.

In Mocha, we implemented fault detection and handling for this type of failure by having the synchronization thread detect failures when the transfer message it sends to a daemon thread times out. The synchronization thread then handles the failure by polling other daemon threads to obtain the most recent version of the replicas available.

Failure of Lock Owning Application Thread. Failure of an application thread that currently owns a lock is another situation that must detected and handled. This failure may be detected by having the synchronization thread timestamp lock acquisitions and having threads indicate approximately how long they need to hold a lock. The synchronization thread can periodically peruse its list of held locks to determine if any threads are holding locks for an extraordinary amount of time and therefore a candidate for being a failed thread. The synchronization thread can confirm this suspicion by sending a "heartbeat" message to the appropriate daemon thread. If this message times out the synchronization thread can assume the application thread has failed and thus the failure has been detected. Here, the synchronization thread can simply break the lock and give it to the next application thread that desires it. The most recent copy of the replicas will now be those available from the daemon thread of the previous owner of the lock or if necessary the synchronization thread may resort to polling daemons for the most recent version of the replicas. Furthermore, an application thread that fails in this manner is prevented from making future requests.

In Mocha, we implemented fault detection and handling for this type of failure by having the user threads indicate how long they need to hold a lock. The synchronization thread timestamps lock acquisitions and if a lock is held for longer than expected, the synchronization thread breaks the lock and gives it to the next application thread that desires it. The next application thread to acquire the lock receives the most recent version of the replica that is available.

Failure of Synchronization Thread. In general, we assume the synchronization thread executes at the initial home site and therefore will be less likely to fail compared to application threads executing at remote sites. Failures that take place when a machine reboots, a task is killed by another user, or failures due to network contention are not expected to happen very frequently at the home site. This is due to the fact that in the local environment there is more control over the computing resources. Because of these reasons and to keep the overhead of failure handling low, we chose not to implement the synchronization thread in a failure resilient manner. However, we do have some ideas on how to mitigate failures of the synchronization thread. Failure detection and handling of the synchronization thread could be handled by logging its state and employing a recovery protocol whereby a new synchronization thread is spawned which informs the daemon threads of its existence. Application threads which time out attempting to contact the failed synchronization thread can query the local daemon thread to obtain the location of the newly created surrogate synchronization thread.

5 Applications

We are exploring a range of applications that can run on Mocha. In this section we explore several classes of applications that Mocha is able to support. The first class is electronic commerce home service applications. These applications execute in heterogeneous environments that include archetypical home computing platforms such as a PC running Windows 95 operating system.

The next class of applications we investigate is scientific applications suitable for execution in a wide area computing environment. We illustrate how the traveling salesperson problem, a parallel branch and bound application that requires state sharing, may be developed in a failure resilient manner using the mechanisms provided by Mocha.

A third class of applications we explore is traditional collaborative applications such as a shared calendar. These applications in some circumstances are able to exploit a weaker notion of shared state consistency when available. We illustrate that in addition to the entry consistency Mocha provides by default, its model is flexible enough that it may also be utilized to provide weaker notions of consistency.

5.1 Electronic Commerce Applications

Figure 4 presents a sample interactive home service application written with the Mocha system that utilizes the system's state sharing mechanisms to support collaborative home shopping. The application is a formal dinner table setting coordinator application similar to the one described in Section 2. As discussed in Section 2, in this home shopping scenario, a consumer residing at home wishes to add a new formal dinner table place setting composed of flatware, plates, and glassware. At the consumer's home, a graphical user interface (GUI) is executing which allows various flatware, plates, and glassware to be viewed together so that the consumer may "mix and match" these items and end up with a pleasing coordinated table setting.

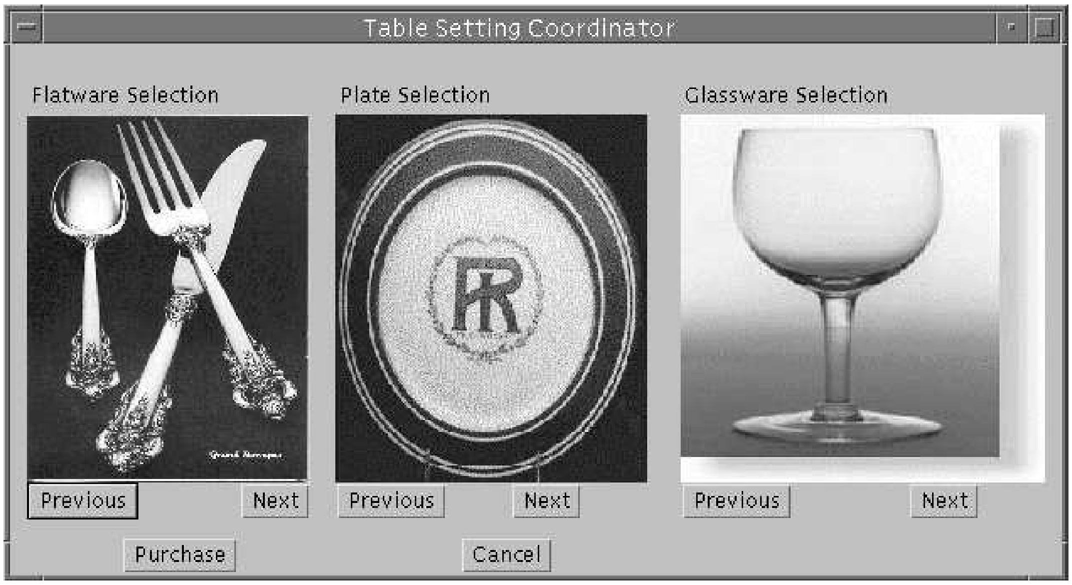

Figure 4: Mocha table setting coordinator home service application.

Additionally, a sales associate located at the retail outlet may also have a copy of the graphical user interface which permits the associate to see what the customer is selecting and may suggest alternatives that are then presented in the customer's GUI. Furthermore, the home consumer may have requested friends located at other homes to also participate in this decision making and therefore they too may be running a GUI and viewing the possibilities and also making suggestions. In this scenario, it is expected that the platforms on which the GUI executes in each of the homes would be vastly different from the platform at the retail outlet. Moreover, it is critical that the latency of keeping the GUI's consistent be kept small in order to support an acceptable response time for user actions.

5.1.1 Implementation

In this application, the GUI shown in Figure 4 is sent via Mocha's remote evaluation support to execute at several remote sites. Each site may modify the flatware, plates, or glassware currently being displayed by pushing the appropriate `previous` or `next` button. These buttons result in activating callbacks which modify shared index variables associated with each item. A thread which executes in each remote GUI periodically polls the shared index variables for new values and updates the local display as needed. The graphical images are also shared as replicas but are not associated with a `ReplicaLock`. Thus, they are cached at each host without any consistency maintenance being performed on them. The shared indexes do however rely on the system's consistency maintenance facilities.

5.2 Scientific Applications

Scientific parallel computing applications are another domain in which the functionality provided by Mocha may be beneficial. Obviously, when possible, it is best to execute a parallel computing application in a low latency network environment using a non-interpretive language.

In some cases, however, it may be necessary to execute a parallel computing application in a wide area environment. For example, an application may require scientific data that is only available at a remote site. Furthermore, sufficient computing resources may not be available locally and therefore remote sites must be utilized to obtain the desired amount of computing cycles. Finally, as techniques such as just in time compilation[11, 24] which improve the performance of interpretive languages become more mature, languages such as Java may become an acceptable platform for some types of parallel applications.

When developing parallel computing applications that are intended to be executed in wide area computing environments, a significant starting point is to begin with implementation strategies developed for cluster environments. These environments are in many ways similar to wide area computing environments. For example, both environments are composed of workstations that that have varying configurations in terms of CPU speed and memory. Furthermore, these workstations are part of an open network computing environment and both the workstations as well as the network may be subject to uncontrollable external loads. These factors often result in load imbalances and dynamic fluctuations of delivered resources which can be a major source of performance degradation for a parallel application[23].

One strategy for developing network based concurrent computing applications shown to be particularly effective is use of an *agenda parallel* model that utilizes dynamic load balancing to combat the effect of load imbalance[23]. In this model, the application is implemented using a "bag of tasks" algorithm where pending work is in the form of a queue of tasks maintained by a master process. The master process is responsible for assigning tasks to processing elements as they become idle and also collects results. With this scheme, more work is assigned to processing elements with lighter loads and this dynamic load balancing helps to mitigate the performance degradation that occurs when executing an application in a network based concurrent computing environment.

In this section, we investigate the development of a bag of tasks implementation of the traveling salesperson problem (TSP) suitable for execution in a network computing environment using the functionality provided by Mocha. This parallel application allows us to investigate the performance of Mocha's state sharing ability to serve as a distributed shared memory (DSM) facility for parallel applications.

We also use this application to show how it is possible to utilize Mocha's functionality to develop a failure resilient version of TSP using locks that have timeout values to permit failure detection. Furthermore, we utilize timeout values that are large enough to be suitable for wide area execution. The next section provides an overview of the implementation of TSP for network computing using Mocha as well as the extensions that are necessary to develop a failure resilient version of the application.

5.2.1 Implementing TSP

The TSP algorithm is similar to those used by Bal *et al*[2] as well as Kohli*et al*[21]. The application creates a statically defined job queue that is shared between tasks. The initial queue is composed of partial tours that are generated by unraveling possible paths through the cities to a fixed depth. Locks are utilized to dequeue jobs from the queue as well as to update the value of the best tour. Thus, there are two structures which must be shared between tasks; the job queue and the best tour value. The best tour value is checked frequently to insure that the tour currently being investigated by a task has the potential to become the new best tour. Tasks constantly read the best tour value and these accesses are permitted to occur without acquiring a lock and hence data races can occur. In contrast, however, all updates to the best tour value are protected by a lock. The ability to support push-based dissemination when updating the best tour value has the potential to improve the performance of the application.

This is because receiving an updated tour value may result in the task realizing that the tour it is investigating has no chance of being the best tour and completing the investigation would be a wasteful endeavor. Instead, the task may choose the next available partial tour job and investigate it.

Developing a Failure Resilient TSP. A failure resilient TSP application can be created from the algorithm described above by combining Mocha's mechanisms for failure recovery (i.e., replication, breakable locks) with some simple algorithmic modifications that permit the application to (i) become aware of failures and (ii) take appropriate action to reproduce the work that was lost during the failure.

We utilize Mocha's mechanisms for failure recovery by using the `setLockHoldTime()` method of class `ReplicaLock` in order to set a timeout value for the lock used by the application. That is, we enable the synchronization server to break a lock if it has been held for a duration of time that is much longer than the period of time it should take for a non-failed task to release the lock. Additionally, we modify the application to use Mocha's update dissemination capabilities to insure that should a failure occur, it will be possible to find an up-to-date copy of the shared job queue as well as the best tour value.

In order to be able detect a failure as well as determine which jobs were not completed because of the failure, we must disseminate to all tasks which job each task is currently investigating. Thus, we add a new shared object whose purpose is to store the identifier of the current job a task is working on. We refer to this as a *will* structure as it essentially denotes a request that the task desires some other task to complete should this task die (i.e., fail). When a task chooses a new job it marks it in its will structure and when it completes the job it clears its will structure. This clearing of the will structure is performed *after* the best tour value has been updated (if necessary). At the end of execution, the will structures for all tasks are examined to determine if there exists any jobs that have not been completed. These jobs are then performed a second time to their full completion. In the next section, we examine the performance for both the basic implementation as well as the failure resilient implementation of TSP.

5.3 CSCW Applications

Computer Supported Cooperative Work (CSCW) applications is another domain in which the functionality provided by Mocha may be beneficial. These types of applications require state sharing support to facilitate the collaborative work efforts these applications support. In this section, we investigate the development of two types of shared calendar applications which utilize Mocha's mechanisms to support both a tightly coupled shared calendar as well as a loosely coupled calendar that relies on timestamps to mitigate update conflicts that can occur when using the looser coupling.

Figure 5 shows the graphical user interface (GUI) utilized by both versions of the shared calendar. Essentially, the calendar presents a weekly schedule where each hour has a text field associated with it that denotes an activity that is planned to occur during that hour. The text string is updated by using a mouse to press the button that is labeled with the hourly interval that needs to be updated. This triggers a dialogue box to appear. The user may then type in the dialogue box a new message and submits the message by pressing the dialogue's button labeled *OK*. This activity results in the modification of a `StringReplica` that is associated with each text field. The `StringReplica` is a subclass of `Replica` that is generated using the `MochaGen` tool described in Section 2.1.2.

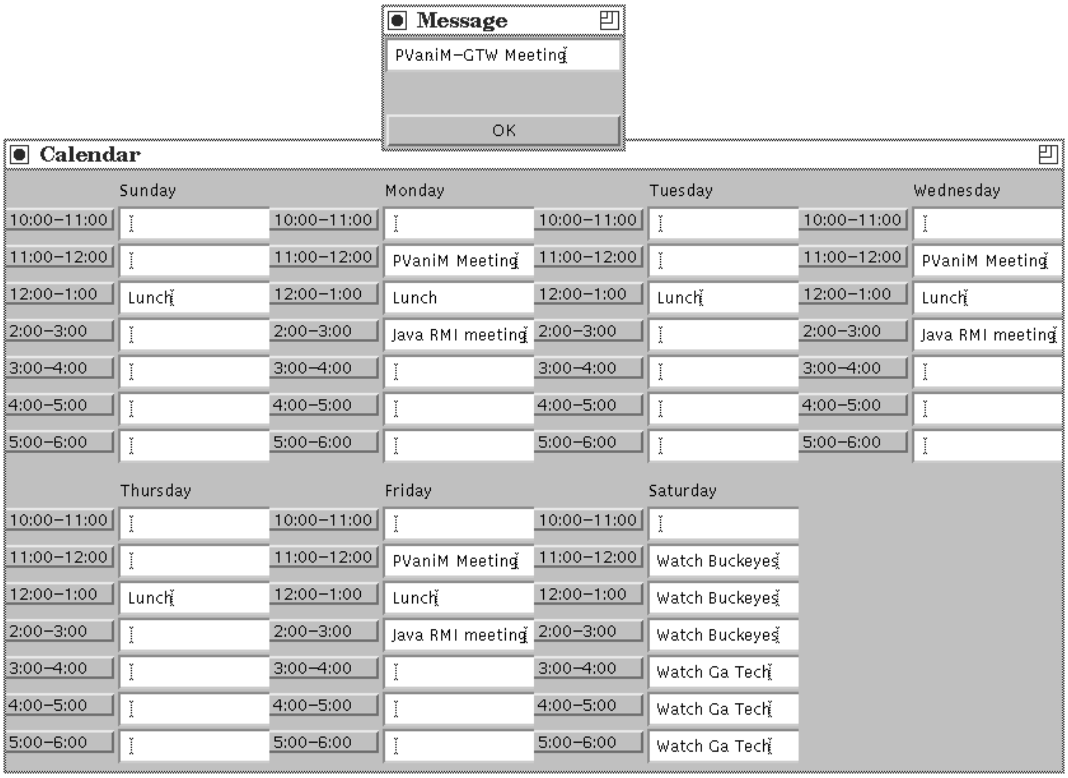

Figure 5: Mocha collaborative calendar application.

In the tightly coupled version of this application, there is one `ReplicaLock` for which all `StringReplica's` are associated. The shared calendars very frequently refresh their local GUI with the newest version of the `StringReplica's` available. Thus, after very short intervals they acquire the `ReplicaLock`, utilize the new versions of the StringReplica's to update the local GUI, and then release the lock. When the dialogue box is used to update a text field, the same procedure occurs with the following difference: the `StringReplica` associated with hourly interval that is being changed is first modified to contain the text from the dialogue box before the text fields are updated from the `StringReplicas`.

In the loosely coupled version of this application, the shared calendars are permitted to perform local updates without first obtaining the most recent version of the `StringReplicas`. This of course results in the updating of a locally cached copy and can significantly reduce the overheads associated with performing an update. After very large intervals of time (e.g., a day) the tasks perform the process of merging the individual copies of the shared calendars. Because the policy permits each task to update only its local copy, there can exist update conflicts that have to be resolved using a policy for merging[27, 20]. In our implementation, we associate timestamps with each text field's `StringReplica` and the timestamps are updated with the current time whenever a `StringReplica` is updated. Thus we support a merging policy whereby each `StringReplica` retains the most recent update.

The loosely coupled version of the shared calendar is slightly more complicated to program than the implementation for the tightly coupled calendar. In order to implement the loosely coupled calendar, each task associates a distinct `ReplicaLock` for its `StringReplica's`. Because each lock uses its own `ReplicaLock` when updating a `StringReplica`, the Mocha system permits the lock acquisition to occur without the need to find the most up-to-date copy

Lock Acquisition Local Area Network (Fast Ethernet)	5
Round Trip Null Message (LAN)	1.75
Lock Acquisition Wide Area (Internet)	19
Round Trip Null Message (WAN)	8.75

Table 1: Time to Acquire a Lock (with no data transfer) in milliseconds.

of the `Replica's`. However, when an update does occur, the timestamp associated with the `StringReplica` being modified is updated with the current time of day. Now, whenever a task begins the process of synchronizing (i.e., merging) the calendars it first makes a copy of its local `StringReplicas`. It then acquires a separate `ReplicaLock` which is used as a token to signify a task has begun the merging process. The task then iteratively acquires the `ReplicaLock's` that all the other tasks have associated with the `StringReplica's`. This allows the task to effectively pull versions of the `StringReplica's` from other tasks. These versions are then merged into the local copy of the `StringReplicas` using the timestamps to determine which version of the `StringReplica` should be considered most up-to-date and hence remain the current copy.

6 Evaluation

This section provides an evaluation of the state sharing support provided by our prototype system. We present the overheads associated with several core activities performed by the system to support object sharing. These include lock acquisition latency, data marshaling overheads, and communication overheads incurred transferring replicas in both local and wide area networks, as well as in the Georgia Tech Home Information Infrastructure Lab, a prototype home service environment that includes a Windows 95 PC connected to a Unix workstation hub via a digital subscriber line (ADSL) modem.

Table 1 presents the amount of time in milliseconds to perform lock acquisition using the methods of the `ReplicaLock` object as well as the time for a round trip null message for comparison purposes. The local area network results are from two SUN ULTRA 1 machines on Fast Ethernet. The wide area results are from a SUN ULTRA 1 and a SUN SPARCstation 20 connected via the Internet separated by a distance of approximately 6 miles. As shown in the Table, lock acquisition latency in wide area networks can be significantly greater than the latency experienced in local area networks.

Figure 6 presents the time for a SUN ULTRA 1 to marshal a Replica into a byte array to enable it to be transferred over the network. As shown in the Figure, this activity can be somewhat expensive for large replicas. This inefficiency is a result of the Mocha system relying on the generic data marshaling constructs provided by Java JDK 1.1. These constructs utilize dynamic arrays and marshal a single byte at a time. These factors result in marshaling being a relatively costly operation. In the future, we plan on providing a custom marshaling library that is more efficient for our needs.

For the transfer of replicas between hosts, we have developed two separate prototype systems. In the first system, all communication is performed using Mocha's network object library. This library implements reliable, sequenced, delivery of messages as well as performing fragmentation and reassembly. It is scalable in the number of hosts that communicate with the

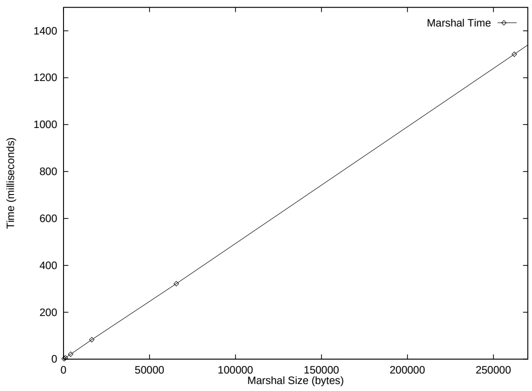

Figure 6: Time to marshal Replicas in milliseconds.

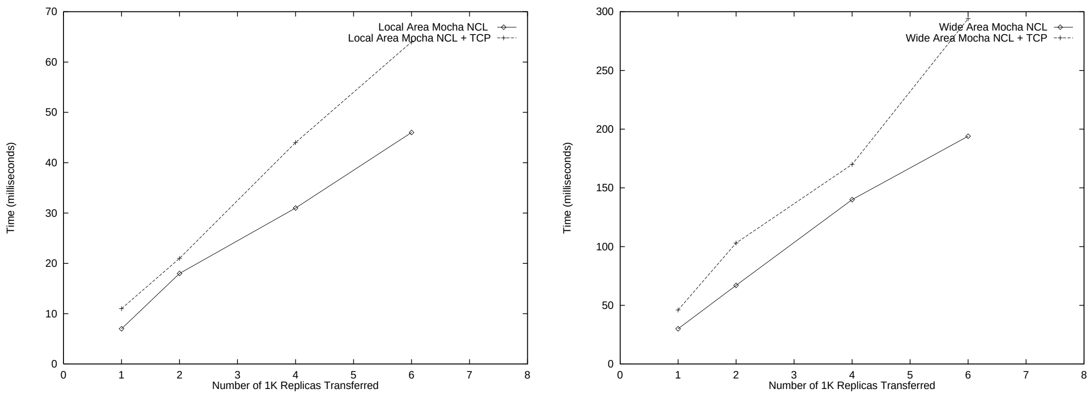

Figure 7: Time for local and wide area transfers of 1K replicas in milliseconds.

library because it performs its own upward multiplexing of packets. It is particularly well suited for sending small messages as it avoids the heavy connection and tear-down overheads associated with other transport protocols such as TCP. Empirically, we have found Mocha's network communication library to be approximately twice as fast as TCP for sending small (i.e., less than 256 byte) messages.

For the second prototype, small "control" messages used for lock acquisition and directing data transfers are sent using Mocha's network object library. For the actual transfer of replica data which typically involves sending large messages, we have developed a "hybrid protocol" approach whereby Mocha's custom network object library is used in conjunction with TCP. Here, Mocha's network communication is used for establishing a TCP connection (i.e., propagating TCP port numbers) and the actual transfer of replica data is done using TCP. Figure 7 presents the times in milliseconds to disseminate a 1K replica to several local and wide area network hosts. In both environments, solely using Mocha's network communication library is the more efficient approach. This is attributable to the higher connection and tear-down overheads associated with the hybrid approach.

As we increase the size of the replicas, the hybrid protocol approach provided by the second prototype begins to perform much better than simply using Mocha's network communication library for for all network environments investigated. Figure 8 presents the results for 4K replica transfers in local area networks, and Figures 9 and 10 illustrate the results for wide area and home service networks, respectively.

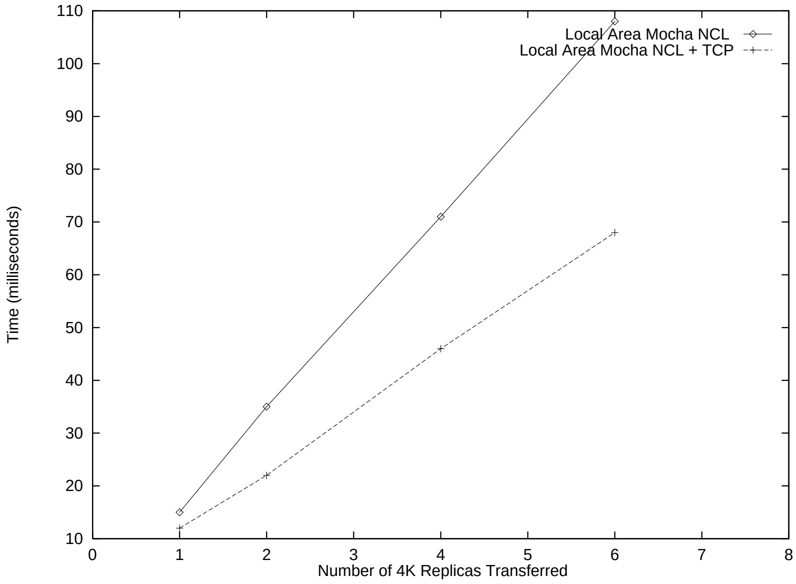

Figure 8: Time for local area transfer of 4K replicas in milliseconds.

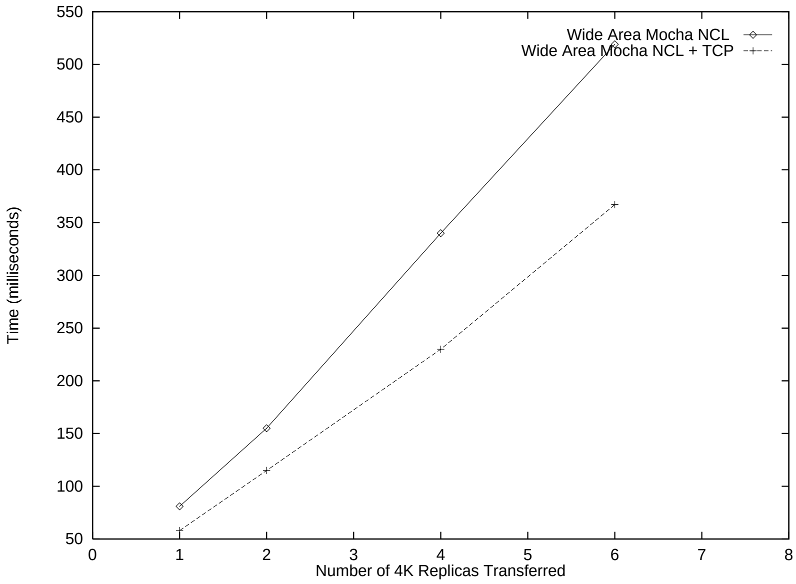

Figure 9: Time for wide area transfer of 4K replicas in milliseconds.

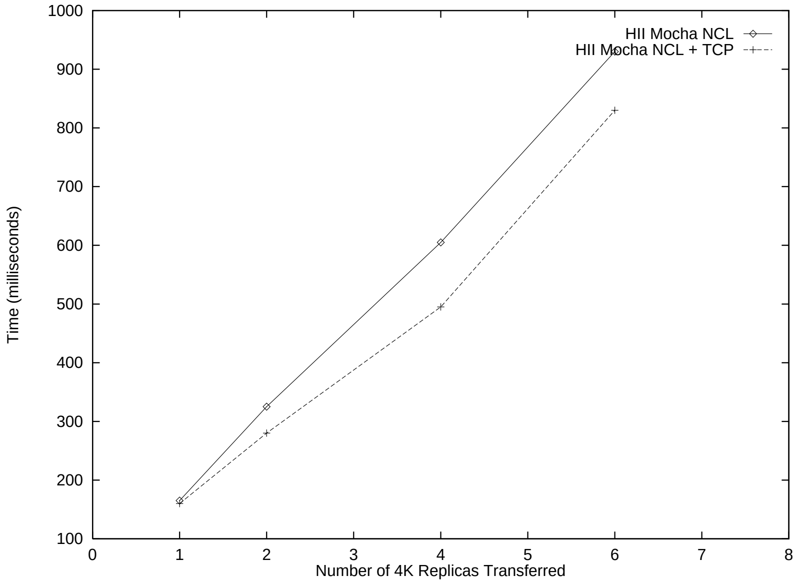

Figure 10: Time for home service transfer of 4K replicas in milliseconds.

Furthermore, as shown in Figure 11, 12, and 13, for larger replicas reaching sizes of 256K the superiority of the hybrid protocol becomes clear. This vast improvement is attributable to the speeds at which both protocols perform fragmentation and reassembly. Mocha's fragmentation and assembly executes at user level running as interpreted byte code. The TCP fragmentation and reassembly executes as native binary code at the kernel level. This vast disparity of execution speeds allows TCP to easily ameliorate its connection and tear-down overheads for large, multipacket messages.

In summary, several aspects regarding the cost of update dissemination for high availability as well as the relative efficiency of the two approaches for transferring replicas are apparent. As can be seen from Figure 9, when the number of moderately sized (i.e., 4K) replicas that are maintained up-to-date in a wide area environment is increased from 1 to 2, the overhead for consistency maintenance approximately doubles. Furthermore, as also depicted in Figure 9, the hybrid protocol approach can result in an improvement of approximately 30% over Mocha's basic protocol for transferring replicas as small as 4K to multiple (i.e., 6) sites. As shown in Figure 12, for replicas as large as 256K, the hybrid protocol can reduce transfer costs by as much as 70% over the basic protocol when transferring replicas to multiple sites in a wide area environment.

7 Application Evaluation

In this section, we provide an application level evaluation for all three classes of applications described in Section 5. For the electronic commerce class of applications, we show that Mocha's state sharing mechanisms are able support the response requirements of a highly interactive collaborative home shopping application.

We also provide empirical results for scientific TSP application that illustrate the benefits of the dissemination capabilities provided by Mocha as well as the overheads incurred by its failure detection and recovery mechanisms.

Finally, for the CSCW shared calendar application, it is shown that using Mocha's mecha-

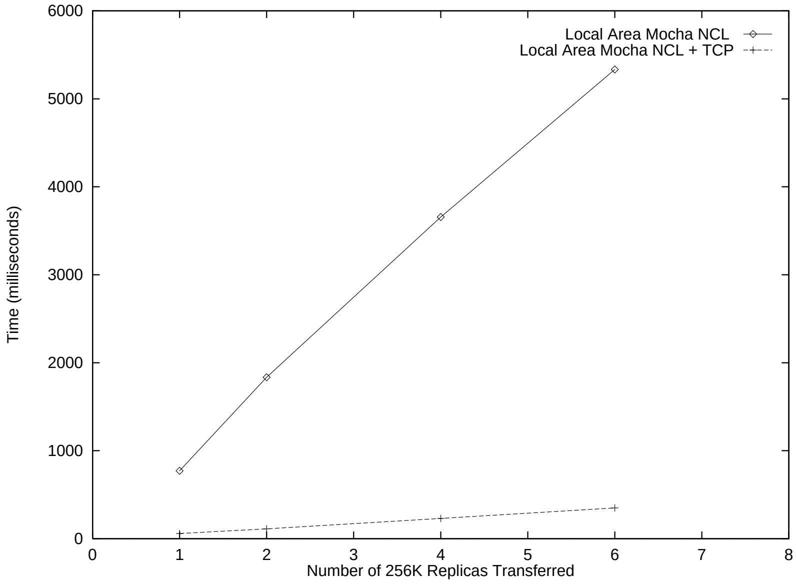

Figure 11: Time for local area transfer of 256K replicas in milliseconds.

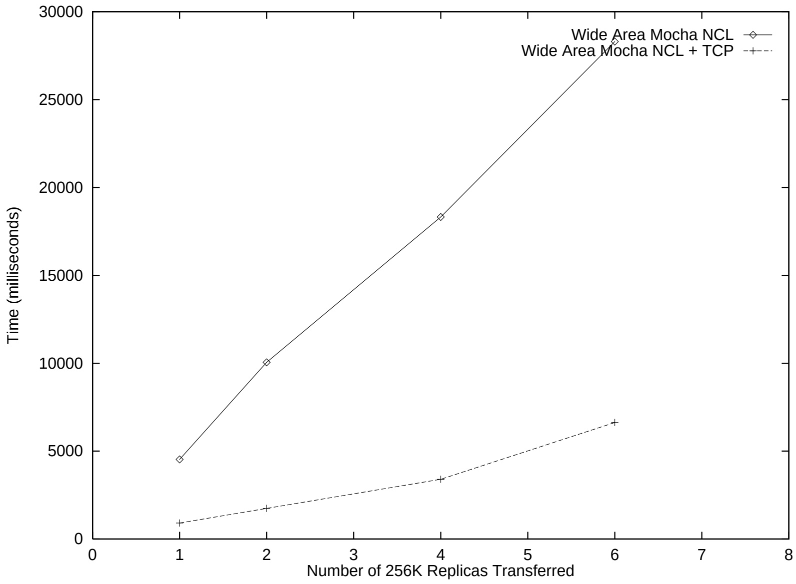

Figure 12: Time for wide area transfer of 256K replicas in milliseconds.

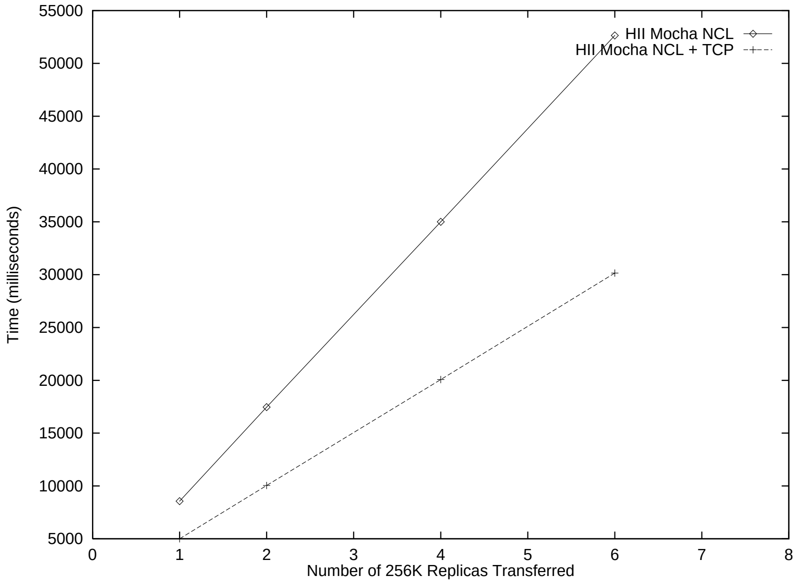

Figure 13: Time for home service transfer of 256K replicas in milliseconds.

nisms in a weakened consistency manner can yield significant performance improvements over collaborative applications that use Mocha's mechanisms in a more tightly coupled fashion.

7.1 Electronic Commerce Applications

With respect to performance of this application, we have measured the marshaling, lock acquisition, and transfer costs of keeping these three replicas consistent in a prototype home service environment. The home service environment results are from a Windows 95 PC to a SUN ULTRA 1 machine. These machines are physically located across campus from each other and are connected by way of a Unix workstation which serves as a hub. The Windows 95 PC is connected to the hub with an ADSL modem and the hub is connected to the SUN ULTRA 1 via an ATM network. The application performs shared state transfers of three `Replicas`, each of which stores an integer index variable. In this environment the transfer costs for sharing these replicas were as follows: marshaling required 2 milliseconds, lock acquisition and release overhead was 31 milliseconds and transfer costs were measured at 38 milliseconds. Overall, the total cost of maintaining consistency is 71 milliseconds, a latency value that we feel is suitable for interactive response time in this type of application.

7.2 Scientific Applications

Figure 14 presents the completion times in milliseconds for TSP with a problem size of 17 cities and which executes on a local area network cluster of SUN ULTRA 1 workstations. As illustrated in the Figure, completion times decrease as the number of worker tasks is increased. Thus, significant speedup is achievable with the system. Furthermore, the graph of completion times for this problem size of TSP is remarkably similar to graphs of completion times of other systems[21] that implement a distributed shared memory TSP application.

Another well known behavioral characteristic of TSP applications implemented using distributed shared memory is a decrease in completion times when an update or push-based dissemination implementation is utilized. This is due to the fact that tasks are better able to

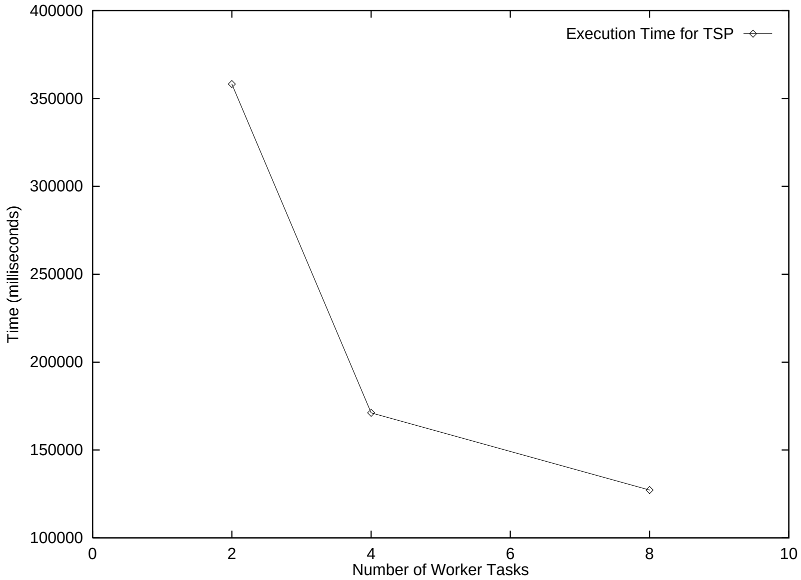

Figure 14: Completion times of TSP (no dissemination) in milliseconds.

Failure Type	Execution Time	Execution - Detection Time
No Failures	97 seconds	97 seconds
Failure in 1st Half of execution	198 seconds	176 seconds
Failure in 2nd Half of execution	206 seconds	173 seconds

Table 2: Completion time of 8 node TSP with full replica dissemination (with and without failures) in seconds.

bound or "short-circuit" the investigation of a partial tour as it becomes obvious the tour has no chance of being the best tour. Figure 15 presents the completion times for the TSP problem using eight task nodes with increasing amounts of push-based dissemination. As illustrated in the Figure, for the most part, Mocha's implementation of TSP also displays this behavioral characteristic. Note that there is an anomaly when the number of tasks to receive dissemination is equal to four. It is difficult to say exactly why this anomaly occurs because the TSP application is somewhat nondeterministic with regards to which tasks perform specific jobs. It appears however that the extra overhead required to disseminate to four tasks instead of two does not result in tighter bounding of tour investigation. Thus, completion times are not less when the number of tasks to receive dissemination is increased from two to four.

The TSP application is also useful for investigating the overheads that result when a failure occurs and the system must recover from the failure and complete execution. In the experiments that follow, we fabricate the most difficult failure for Mocha to mitigate: a failure in which a node dies while holding a lock. This requires Mocha to detect the failure, break the lock, find an up-to-date copy of the replica and transfer it to the next node that desires it. Table 2 presents completion times for TSP without failures as well as with failures in the first half and second half of execution. Table 2 also shows completion time including and excluding the time required by Mocha to detect the failure. Note that the time to detect a failure may be dictated by the timeout value set to break overdue locks. In this particular experiment the

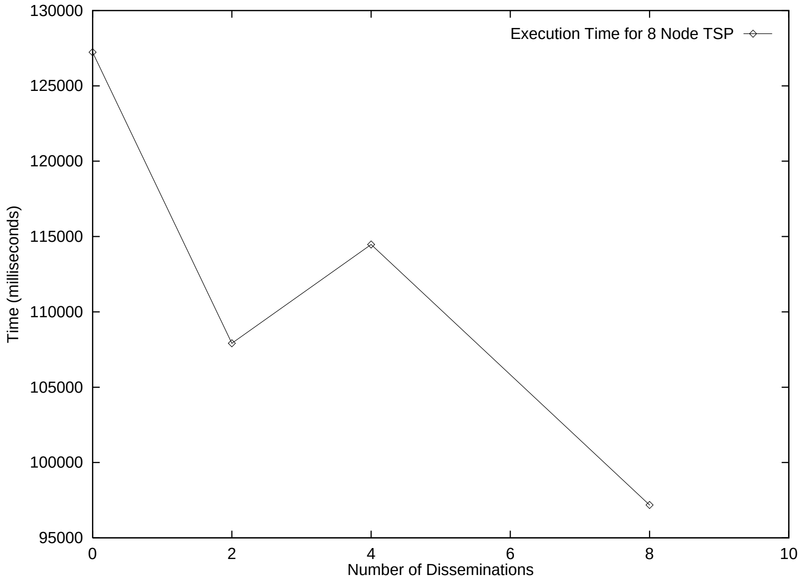

Figure 15: Completion times of 8 Node TSP (variable dissemination) in milliseconds.

timeout value was set to 20 seconds, a value that would be appropriate for many wide area computing environments. As presented in the Table, the overall completion time was slightly greater for a failure in the latter half of execution than a failure in the first half of execution. This anomaly was determined to be due to the synchronization thread requiring additional time to determine a lock was overdue. As the application progresses, there appears to be slightly more contention at the lock server, thus increasing the latency of detecting an overdue lock. When we remove the time associated with failure detection overhead, we obtain the more intuitive result of a failure occurring later in the execution yielding a smaller completion time. This of course is expected as a failure that occurs later in the execution has the benefit of extra computing effort from the failed node.

In order to measure the effect on performance when a failure occurs, it is useful to compare the completion times from executions in which a failure occurs to the completion times for various number of nodes in which no failures occur. This is presented in Figure 16.

As illustrated in the Figure, the effect on completion time of eight node TSP when a failure occurs increases the completion time to approximately that of a four node execution. By plotting the completion times after subtracting out the times for failure detection, we are still left with a somewhat substantial difference in execution time. While some of this is due to failure recovery overheads as well as the reduction of computing resources, another significant contributor to the reduction in performance is redundant work that needs to be completed. This redundant work not only resulted from redoing work from a failed node but also from redoing work at the end of execution for a node that appears to be failed but in reality has not completed its last job. The master task, after determining all jobs have been dequeued, typically encounters a will structure for a node, assumes it has failed, and reexamines the partial tour. One approach to mitigating this behavior is to have the master task wait a period of time once it determines the job queue to be exhausted to reduce the chance of incorrectly assuming a node has failed.

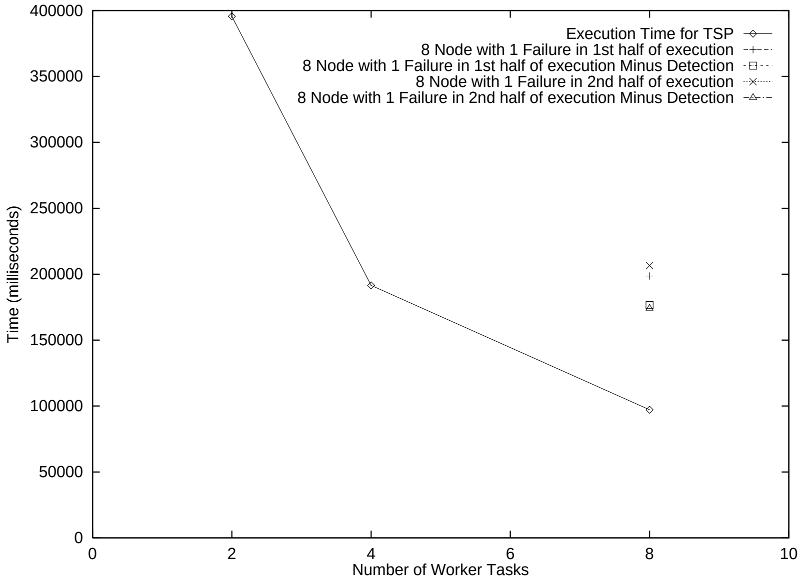

Figure 16: Execution times of TSP (with and without failures) in milliseconds.

Loose Update	Tight Non-Cached Update	Loose Refresh Update
10.25 milliseconds	122 milliseconds	146.5 milliseconds

Table 3: Replica update times for loosely and tightly coupled shared calendars.

7.3 CSCW Applications

Table 3 presents performance results for both the loosely coupled and the tightly coupled shared calendar applications, whose implementations are described in more detail in Section 5. The applications execute in local area network cluster of SUN ULTRA 1 workstations. There were two tasks participating in the sharing of calendars. As shown in the table, by allowing updates to occur locally without the acquisition of the most recent version of the `StringReplicas`, the time to perform an update is reduced by a factor of twelve. However, there are some drawbacks to this approach. In addition to the fact that the calendars do not stay tightly synchronized, when a task wishes to begin the process of merging calendars and does need to pull a new version of the replicas, the time for this operation has increased by approximately 20%. This is due to the extra marshaling and transport overheads associated with sending timestamps along with the `StringReplicas`. It is also worth reemphasizing that as the number of tasks sharing calendars increases so does the number of pull operations that must be performed during the merge process. Nonetheless, applications that need to perform a merge very infrequently and do not require tight coupling may strongly benefit from a loosely coupled approach.

7.4 Discussion

We have explored several classes of applications that Mocha is able to support. The first class, electronic commerce home service applications, illustrate that Mocha is able to support an interactive electronic commerce application that utilizes Mocha's state sharing facilities to enable collaborative home shopping. Furthermore, we show that Mocha is capable of supporting heterogeneous environments networked to the home that are in many ways similar to what we expect to exist in the near future.

For the second class of applications, scientific applications, we illustrate how the traveling salesperson problem, a parallel branch and bound application that requires state sharing, can be developed in a failure resilient manner using the mechanisms provided by Mocha. We also provide empirical results for this application that illustrate the benefits of the dissemination capabilities provided by Mocha as well as the overheads incurred by its failure detection and recovery mechanisms.

The third and final class of applications we explore is the more traditional CSCW collaborative applications such as a shared calendar. For these applications, we illustrate that Mocha's mechanisms may be utilized to provide a variant of weakened consistency. Furthermore, we demonstrate that using Mocha in this manner can yield significant performance improvement in the cost to perform an update over collaborative applications that use Mocha's mechanisms in a tightly coupled fashion.

8 Related Work

Mocha's goals as well as the techniques employed by it are related to several research areas. These include metacomputing systems, DSM systems, and systems that support object mobility. We describe systems from each of these categories.

Wide area computing is closely related to metacomputing. A number of metacomputing systems are currently being implemented. Some systems rely on message passing instead of shared objects to allow tasks to cooperate. Examples of these systems include Chandy's worldwide distributed system[8] and IceT[15]. For the Mocha system, we chose to utilize shared objects instead of message passing because they provide a model that is simpler to program than standard message passing. Furthermore, we provide replicated copies for failure handling which in a message passing model would require group communication and virtual synchrony support[5] and is not currently addressed by the above message passing systems.

Another non-shared object approach utilized by metacomputing systems to allow task cooperation is the use of remote procedure call (RPC) or its object-based equivalent, remote method invocation (RMI). Atlas[3], WebWork[13], NetSolve[7], and Legion[16] all rely on forms of RPC/RMI for task interaction. In some cases, a RPC/RMI model's performance suffers from the clients need to repeatedly contact a server to perform distributed computation. This of course depends on the type of remote computing activities being performed as well as the type of caching strategies employed by the RPC/RMI system. For a more thorough comparison of RPC/RMI and shared memory please refer to [29].

Several metacomputing systems are currently providing shared memory. The TIE design[10] supports shared objects via object caching and entry consistency. The developers of TIE believe that in the future, available network bandwidth will be limited (due to growing popularity of the internet) and aggressive caching must be performed to avoid server bottlenecks. The TIE design is in some ways similar to Mocha but the emphasis of the two systems is quite different. the TIE system currently focuses on mobile objects and security while Mocha focuses on high availability and advanced support for sharing general purpose Java objects. We are also not aware of an implementation of TIE.

The ParaWeb system[6] modifies the Java interpreter to provide a global shared address space using distributed shared memory techniques pioneered by systems such as Munin and Treadmarks. In the ParaWeb implementation, the Java interpreters have been modified to permit them to cooperate and maintain the illusion of global shared memory. ParaWeb utilizes Java's built-in synchronization facilities to monitor when remote memory must be updated to maintain the illusion of consistent global memory. In a similar approach, Yu and Cox[29] are currently implementing a parallel Java Virtual Machine layered on top of the TreadMarks page-based distributed shared memory system. Currently, they are addressing problems such as data type conversion between machines of different architectures as well as garbage collection. Mocha's approach towards supporting shared state differs from these two systems as it supports it at the object level while these systems support sharing at the page level. The difference in the two approaches results in the need to solve different types of problems. With the shared sequence of bytes approach, these systems typically must modify the Java virtual machine, must compensate for different byte orderings in heterogeneous environments, and mitigate false sharing. Although Mocha's shared object approach does not encounter these types of problems, it must deal with issues such as how to support complex objects.

Java Shared Data API (JSDA)[18] provides shared variable support using its own multipoint data delivery service. With JSDA, an update to a shared variable is sent to a session server which then sends the update to other threads that are sharing the variable. In contrast, the Mocha system attempts to exploit locality by sending shared state changes directly to the next thread that needs access to the data. Mocha also allows the number of updated replicas to be configured whereas JSDA updates all copies.

PageSpace[9] relies on a Linda-like coordination technology. Essentially, PageSpace supports a global tuple space which nodes may insert or remove tuples without any regard for where the tuples are stored.

Mocha's consistency actions are driven by synchronization operations. In certain systems and applications, synchronization for all operations may not be desirable. For example, systems such as Bayou[27], Coda[20], and Rover[17] which address mobility avoid synchronization and instead rely upon conflict detection and resolution to maintain consistency. Our future work will explore non-synchronization based consistency models that are suitable for supporting shared objects.

In summary, Mocha's state sharing support distinguishes itself from the above systems by combining advanced distributed shared object techniques with failure handling support that allows its overheads to be controlled based upon the level of availability needed for shared objects. Additionally, Mocha's runtime exploits Java's method overriding and serialization capabilities to support the sharing of complex objects without requiring modifications to the standard Java interpreter. Moreover, we have a broad application focus including applications directed to the home.

9 Conclusions

In this article, we have presented a robust shared object model for wide area distributed applications that has been implemented as part of the Mocha wide area computing infrastructure we are currently developing. Our model provides support for shared objects on heterogeneous platforms, and utilizes advanced distributed memory techniques for maintaining consistency of shared objects. Moreover, our system provides fault tolerance support that allows its overhead to be controlled based on the level of availability needed by an application. We have investigated a combination of protocols approach for improving the efficiency of shared state transfer between hosts and performed an empirical evaluation of two versions of our prototype system in local area, wide area, and home service networks. Furthermore, we have shown that several classes of applications may be written using the mechanisms provided by the system.

Currently, we are focusing on providing support for applications which require non-synchronization based solutions for maintaining consistency. We also plan to investigate the impact of techniques that improve the performance of Java (i.e., JIT compilation) will have on Mocha, and plan to evaluate Mocha's capabilities in the context of wireless network environments.

References

[1] H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *OOPSLA*, 1993.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in orca. In *Proceedings of the International Conference on Computer Languages*, pages 79–89, 1990.

[3] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 165–172, Connemara, Ireland, September 1996.

[4] 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.

[5] Kenneth Birman et al. Lightweight causal and atomic group multicast. *ACM TOCS*, 9(3):272–314, 1991.

[6] Tim Brecht, Sandhu Harjinder, Meijuan Shan, and Jimmy Talbott. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 181–188, Connemara, Ireland, September 1996.

[7] Henri Casanova and Jack Dongarra. NetSolve: A network server for solving computational science problems. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996. To appear.

[8] K. Mani Chandy et al. A world-wide distributed system using Java and the Internet. In *Fifth IEEE International Symposium on High-Performance Distributed Computing (HPDC-5)*, Syracuse, NY, August 1996.

[9] P. Ciancarini and R. Tolksdorf. Using the web to coordinate distributed applications. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[10] Michael Condict, Dejan Milojicic, Franklin Reynolds, and Don Bolinger. Towards a world-wide civilization of objects. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 25–32, Connemara, Ireland, September 1996.

[11] Symantec Corporation. Just-in-time compiler for windows 95/NT, 1996.

[12] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steve Tuecke. Multimethod communication for high-performance metacomputing applications. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996. To appear.

[13] Geoffrey Fox et al. WebWork: Integrated programming environment tools for national and grand challenges. Technical Report NPAC SCCS-715, Syracuse Univesity, Syracuse, NY, June 1995.

[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 Seventeenth International Symposium on Computer Architecture*, pages 15–26, May 1990.

[15] Paul Gray and Vaidy Sunderam. IceT: Distributed computing and Java. In *Proceedings of ACM 1997 Workshop on Java for Science and Engineering*, Las Vegas, Nevada, June 1997.

[16] Andrew S Grimshaw and William A. Wulf. Legion flexible support for wide-area computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 205–212, Connemara, Ireland, September 1996.

[17] Anthony D. Joseph et al. Rover: A toolkit for mobile information access. In *Proc. Fifteenth ACM Symposium on Operating Systems*, pages 156–171, Copper Mountain Resort, CO, dec 1995.

[18] http://www.javasoft.com/people/richb/jsda/, 1997.

[19] 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.

[20] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computing Systems*, 10:3–25, 1992.

[21] Prince Kohli, Mustaque Ahamad, and Karsten Schwan. Indigo: User-level support for building distributed shared abstractions. *Concurrency: Practice & Experience*, 10(1):1–29, January 1998.

[22] R. Riggs et al. Pickling state in Java. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 241–250, Toronto, Ontario, June 1996.

[23] Brian K. Schmidt and Vaidy S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice & Experience*, 6(1):1–33, February 1994.

[24] Softway. Introduction to guava, 1996.

[25] John Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[26] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[27] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 1994 Symposium on Parallel and Distributed Information Systems*, September 1994.

[28] Brad Topol. *A Framework for the Development of Wide Area Distributed Applications.* PhD thesis, Georgia Institute of Technology, 1998.

[29] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of ACM 1997 Workshop on Java for Science and Engineering*, Las Vegas, Nevada, June 1997.

Appendix–Basic Object Consistency Algorithm

The implementation of Mocha's shared object replicas utilizes a daemon thread at each site manager, and a single synchronization thread at the home site. The home site in our system is simply the site at which the initial application thread executes. All of the threads mentioned above are implemented using the standard Java threads library. All objects that the application threads wish to share are registered with the local daemon thread allowing it to directly access the shared objects themselves. The daemon threads execute at maximum priority which guarantees they may interrupt lower priority application threads when necessary. This behavior permits the daemon threads to perform the transfer of shared objects to remote sites as well to accept shared object consistency updates from remote sites when necessary. The asynchrony present in this architecture allows Mocha the flexibility to dynamically configure itself during execution from a streamlined shared object system to one that can handle common failures.

The synchronization thread handles *lock* and *unlock* requests from application threads. In addition, the synchronization thread directs daemon threads to perform operations such as transferring replicas to remote sites and is also responsible for deducing when such activities are necessary.

The Mocha shared object model consists of replicas which must be associated with a ReplicaLock object thereby supporting a variant of entry consistency[4]. When an application thread desires exclusive access to shared replicas, it calls the ReplicaLock's `lock()` method. The pseudo-code for this method is provided in Figure 17. As shown in the pseudo-code, when an application thread makes this method call, if any other *local* application threads have called this method, it must first wait until their calls have completed. After waiting, the thread sends the synchronization thread a lock `REQUEST` message which contains the identifier of the desired lock. When the lock is free, the synchronization thread responds with a `GRANT` message that contains the version number of the replicas associated with this lock and a flag which denotes whether or not a new version of the replicas need to be sent. If no new replicas need to be sent (i.e., the thread already has the most recent copy), the method may return and the application thread is free to access the replicas. If a new version of the replicas is coming, the application thread must wait for the new replicas to arrive and unmarshals them into the local copies before permitting access.

An application thread releases access to shared replicas by calling the ReplicaLock's `unlock()` method. This method call is responsible for sending the synchronization thread a message that indicates that the lock is being released. The method contains the identifier for the lock as well the updated version number associated with the replicas. Pseudo-code for this method is provided in Figure 17. Note that although it is possible that another local thread may be waiting for the lock, a local transfer is not permitted to insure lock acquisition proceeds in a manner that guarantees fairness.

The more complex aspects of Mocha's shared object support may be found in the operation of the daemon threads and the synchronization thread. Each daemon thread is responsible for

```
Lock Method

if (another local thread currently has
 this lock or waiting for it)

 wait();

send synchronization thread ReplicaLockId

receive GRANT Message M from
 synchronization thread;

newVersionNumber = unpackNewVersionNumber;
versionFlag = unpackVersionFlag;

if (versionFlag == VERSIONOK)
 // We have an up-to-date version number
 // and may simply return

else
 // we must wait for new version to be sent
 receive replicas from a remote daemon;
 unpackReplicas();

return;
```

```
Unlock Method

send synchronization thread ReplicaLockId
 and newVersionNumber of associated objects

if (another local thread currently
 is waiting for this lock)
 notify();

return;
```

Figure 17: ReplicaLock object's lock and unlock methods which are executed by application threads.

startup activities such as initialization, and responding to requests regarding the transfer and acceptance of replicas. As shown in Figure 18, when a daemon thread receives a request for its copy of replicas, the thread identifies the replicas associated with the lock identifier it receives, marshals those replicas and sends them to the mandated destination.

The synchronization thread at the home node is responsible for granting locks, queuing requests, and deducing whether a new version of replicas must be sent to an application thread. Figure 19 presents pseudo-code for the synchronization thread's operations. Upon receiving a request to acquire a lock, the synchronization thread determines if the lock exists and creates a `Lock` object if necessary. The thread then determines if the lock is free or currently owned by another thread. If the lock is not free the request is queued. If the lock is free, the acquire request is granted by sending a `GRANT` message to the requesting thread. This message contains the new version number for the replicas as well as a flag indicating whether or not new replicas will also arrive. The synchronization thread relies on the method `lastLockOwner()` to determine the value of the flag. If new replicas need to be sent, the synchronization thread sends a message to the daemon associated with the last owner of the lock and directs it to transfer a copy of its replicas to the application thread which desires them.

When the synchronization thread receives a request to release a lock, it updates state information such as the new version number and the identifier of the daemon that now has the most recent copy. The synchronization thread then dequeues the current owner of the lock and checks to see if another lock request is queued. If another request exists, this request is granted via a `GRANT` message and the appropriate daemon thread is instructed to transfer a copy of its replicas to the application thread which desires them.

```
while(true)  {

   Receive message M from anyone;

   if (M.type == REGISTERREPLICA) {
     // perform startup and
     // initialization activities
   }
   else if (M.type == TRANSFERREPLICA) {
     lockId = unpackLockId();

     // unpack destination information
     host   = unpackDestinationAddress();
     port   = unpackDestinationPort();

     replicaLock = replicaLockVector.find(lockId);

     replicaLock.packReplicas();

     send packed replicas to destination;
   }
}
```

Figure 18: Pseudo-code for a daemon thread.

```
while(true) {
   Receive message M from anyone;
   if (M.type == ACQUIRELOCK) {
      lockId = unpackLockId();
      if (!lockVector.exists(lockId)) {
          create new Lock object
      }

      lock1 = lockVector.find(lockId);
      if (!lock1.isFree()) {
          lock1.queueRequest();
      }
      else {
          lock1.queueRequest();
          packNewVersionNumber(lock1.returnVersionNumber());
          if (lock1.lastLockOwner() = M.source) {
              packVersionFlag(VERSIONOK);
              send GRANT message to M.source;
          }
          else {
              packVersionFlag(NEEDNEWVERSION);
              send GRANT message to M.source;
              send TRANSFERREPLICA to lock1.lastLockOwner().daemon;
          }
      }
   }
   else if (M.type == RELEASELOCK) {
      lock1 = lockVector.find(lockId);
      lock1.updateVersionNumber();
      lock1.updateDaemonId();
      //dequeue the current owner
      lockRequest = lock1.dequeueRequest();
      lock1.lastLockOwner = lockRequest.requestingHost;
      if (!lock1.isFree()) {
         nextlockRequest = lock1.examineQueueHead();
         packNewVersionNumber(lock1.returnVersionNumber());
         packVersionFlag(NEEDNEWVERSION);
         send GRANT message to nextlockRequest.source;
         send TRANSFERREPLICA to lock1.lastLockOwner().daemon;
      }
   }
}
```

Figure 19: Pseudo-code for the synchronization thread.