Mobile Aleph:
A System for Distributed Mobile Applications

Michael Coglianese

Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

Signature (Prof. Maurice Herlihy, Project Advisor)

13 Dec 2000

Date

Masters Project **Michael Coglianese**

Abstract

My paper introduces Mobile Aleph, a distributed shared object system that addresses the needs of mobile group-oriented applications. These applications need to work when the mobile client is connected to an application server, when offline, or when combined with other clients in an ad hoc network. In addition, a distributed system for mobile applications must have a high degree of fault tolerance and must deal with the limitations of resource-poor clients. The system employs an optimistic concurrency model that combines the approaches of client/server and peer-to-peer architectures. The paper outlines the Mobile Aleph approach, the algorithms used for maintainance of global data, and the experimental results of simulations of the system.

1 Introduction

1.1 Overview

Mobile and wireless computing is becoming more and more prevalent in today's society. In several years, the number of wireless devices in use is expected to exceed the number of devices that use wired connections. PDAs, handheld PCs, laptops, and cell phones are all being enabled to communicate wirelessly on the go. PDAs such as the Palm VII now have built-in wireless Internet connectivity. Wireless networking cards and emerging technologies such as Bluetooth [2] will further enable mobile users to intercommunicate regardless of the presence of an Internet connection.

While mobile environments afford a growing degree of network connectivity, such connectivity is often intermittent and of low quality. Applications that want to take advantage of mobile environments must give clients the ability to update shared application data when the network is unavailable or when application servers can not be reached. Optimistic data replication has emerged as the logical way of allowing clients to read, write, and share this data. In optimistic schemes, each client keeps a cache of the data, and reads and writes to this data without needing to contact other machines. If two clients update the same data and those updates conflict, then the updates need to be merged somehow or one update needs to be discarded.

A major design issue that must be confronted by systems using optimistic data replication is how updates can be transferred between machines. There are two main architectures that have been used to deal with this issue: *client/server* and *peer-to-peer.* In a client/server system, mobile clients cache data from the server when they disconnect, update that data offline, and then integrate their changes when they revisit the server. Since clients can not exchange data with each other, these systems inherently do not support server-independent group communication. In a peer-to-peer system, clients can exchange their updates with other clients. However, often these systems do not provide strong enough guarantees about when a group of clients will have the same set of updates.

To solve these problems, Mobile Aleph combines the architectures of traditional client/server and peer-to-peer systems. By taking this approach, Mobile Aleph enables effective group communication in order to address the needs of interactive distributed mobile applications. While many of today's mobile-enabled programs are restricted to occasional data synchronization, we designed our system for applications that rely on frequent data exchange among multiple mobile peers. Examples of such applications include collaborative software for the classroom or the conference room, multiplayer interactive games, and interpersonal communication such as instant messaging and group chat. We believe that the ever-growing processor speeds and memory in mobile devices will makes these kinds of applications increasingly viable.

In the rest of this first section, we will describe the requirements placed on such a system, and then we will present Mobile Aleph's objectives and contributions.

1.2 Requirements

Applications that want to take advantage of the increasing prevalence and connectivity of mobile devices must meet requirements driven by several sources. The first of these sources is the user of the application.

- *Users want to be able to use the application in varying states of connectivity.* Principally, mobile applications demand the ability to access application resources regardless of the device's outside connectivity. The user wants to run the application as seamlessly as possible throughout different connection modes. The device the user has could be connected to the application server, which may be located on the device's LAN or at some location on the Internet. Alternatively, the device might not be connected to the server, leaving it only able to connect with other nearby mobile devices. This connection scenario is called an *ad hoc network*, in which a group of mobile clients form a isolated, self-organizing network with no predefined structure. Another possiblity is that the user is offline, and therefore not connected to any other machines. Mobile applications should be expected to be in multiple of these connectivity modes at different points during the application's execution.

- *Users want to have control over the connectivity of the application.* Deciding on the connectivity level of the system is a tradeoff between having access to the most recent data and limiting the client's communication costs. On one hand, the client might want to be connected to the server whenever possible in order to have access to the most recently installed global information. On the other hand, the user may want to be connected only to locally available clients because the server is unavailable or the connection speed to the server is too slow. Or, the user may choose to be offline if the user is unwilling to join the network in order to save compute power or battery life.

- *Users want their application to have the most recent global data that is available.* In the vast majority of mobile applications, the user wants to be able to access and modify the most recent versions of that data. For instance, in a global calendering system, the user wants to see the most recent appointments that have been made in order to schedule new meetings. Indeed, the user would expect to sent the most recent data that any of the computers connected to it have. This requirement holds whether the computer is connected to the application server or in an ad hoc network with a group of mobile clients.

- *All clients connected to each other should have the same data, within a factor of the roundtrip communication cost.* Clients that are connected to each other, whether in an ad hoc network or through the server, expect to have the same view of the data. When changes are made to the data, these changes must be made at all clients in the time it takes to communicate those changes.

In addition, the mobile devices themselves put a set of requirements on applications. These requirements include the following:

- *The system must have a high degree of fault tolerance to handle devices that frequently and unexpectedly enter and leave the network.* Mobile devices are more likely than workstations to become disconnected from a distributed application. While workstations remain available for long periods of time, mobile devices are often being turned on and off in order to save battery life. Therefore, applications should be able to effortlessly handle the case when other clients on the network become suddenly unavailable. Also, the application should be easily restartable: if the device running the application is turned off (ending the application's execution), the global data in the application should be available the next time the application is run.
- *The system should take into account the reduced storage capacities of mobile devices.* Many mobile devices, such as the Windows CE computers used in our simulation, have only several megabytes of RAM and do not contain a hard drive. Therefore, applications need to make sure that they limit the amount of storage used for data maintainance.
- *The system must be able to resolve conflicts between updates to global data made on different devices.* Inherent in optimistic data replication is that conflicting updates will be made on different clients, and those conflicts will at some point need to be reconciled. Often, the application should be involved in resolving these conflicts since it knows best how updates can be merged or when one update must be rejected.
- *The system should have reasonable performance despite the limited computing power and slower networking capabilities of these devices.* Although the CPU and network speeds will continue to increase, for the time being they are still considerably slower than their workstation counterparts. Mobile applications must try to make effective use of these resources whenever feasible.

1.3 Objectives and Contributions

The main technical contribution of the Mobile Aleph system is the design and implementation of a new mobile directory manager for the Aleph Toolkit. The main challenge in developing this system is to allow mobile clients to make updates to shared data in a variety of connection scenarios. When clients form ad hoc networks or connect with application servers, their updates must be propagated to these machines with the greatest possibility of being successfully integrated. Furthermore, mobile clients have a small amount of available memory, so their caches must limit the information they store about the shared data. Finally, clients need to be able to perform group communication, but the overhead of clients frequently entering and leaving the group must be minimized relative to the group.

Our solution addresses the challenges above by seamlessly integrating the client/server and peer-to-peer models of communication. Our modeling of shared objects as a primary copy plus a list of tentative updates keeps client caches reduced when changes are integrated. Mobile Aleph uses application specific merge procedures that are applied both on the client and server sides, resulting in the earlier evaluation of updates and further reducing cache sizes when merging can not be performed. In addition, the system supports client-to-client group communication, such that joining a group has little cost to the group as a whole and leaving the group has no cost.

Mobile Aleph also extends the Aleph Toolkit to facilitate the development of mobile applications. It provides a nameserver that long-lived application servers can register with on a short-term, renewable basis. Clients can query the nameserver for the address of an application server of a particular class. Using this address, the client can then contact the application server. In addition, Mobile Aleph supports *restartable PEs*. This means that clients that use the mobile directory manager cache the global data that they receive during their application's execution. The PE can therefore be shut down and restarted, and it will be able to access the cached data without needing to contact other clients or servers.

We will now outline the contents of the remainder of this paper. Section 2 discusses work related to systems for distributed and mobile computing. Section 3 introduces the Mobile Aleph system, including its general infrastructure and the algorithms it uses to manage shared data among clients. Section 4 details the simulation of this system and its experimental results. Section 5 examines why we made particular design decisions in creating this system, and what we learned by working on this project. Finally, section 6 discusses future directions in this area.

2 Related Work

Mobile Aleph was derived from the Aleph Toolkit [8], a distributed shared object system developed at Brown University. Aleph is a collection of Java packages that support distributed computations on a heterogeneous set of workstations. These packages include a communication manager that provides transport-level communications (UDP, TCP), a directory manager that handles the sharing of and access to global data, and an event manager that enables group communication. Using these packages, logical processors (called Processing Elements, or PEs) can send messages to each other, signal events, and read and write to shared global data. Several other distributed shared memory systems have been created using Java, including Kan [10], Infospheres [4], Java/DSM [22], Mocha [20], and Sync [14].

Previous directory managers developed for the Aleph Toolkit have used pessimistic locking schemes. In such schemes, clients need to acquire read and/or write locks on global data before being able to view or modify that data. One example is Aleph's home directory manager, where each shared object is assigned a home PE. The home is in charge of keeping track of what PE currently holds the object, as well as a queue of the PEs that are waiting to access it. PEs that want to access the object contact the home, get added to the queue, and block until the object becomes available. This protocol is commonly used in DSM systems, and it works well up to a moderate number of processors.

The home-based protocol, however, encounters problems when used in a mobile environment. One issue is fault tolerance. If a PE leaves the network while it is holding on to an object, then the other processors that are waiting for that object will block forever. Object leasing could be incorporated into the system to ameliorate this, but the other clients would still suffer from having to wait for a timeout. More troublesome is if the home for the PE leaves the network. In that case, the queue of waiting clients would be lost, and these clients would not know the object's current location. A more fundamental problem with a home-based approach is that it does not provide support for disconnected operation. If the home for a particular object is not available, the client can not access that object.

For these reasons, most systems for mobile computing use optimistic concurrency schemes instead. As mentioned above, the architecture of such systems can be generally classified as either client/server or peer-to-peer. As one of the first systems to deal with disconnected operation, Coda [3, 12, 13] employs a client/server architecture. Coda views the data shared between servers and clients in the form of a file system. Before a client becomes disconnected from a server, it hoards any files which it thinks that it will need to access. When disconnected, a Coda client records all updates to the file system in a log that is flushed to disk. When the client reconnects with the server, the update log is replayed, thus synchronizing the file system on the server and the client. In the case of conflicts where two clients have modified the same file, Coda employs application-specific resolvers to merge the two versions. A number of other distributed file systems [9, 18] that support disconnected operation have also been created.

The Rover Toolkit [11] is another system that relies on a client/server architecture for mobile applications. Instead of using a file system abstraction like Coda, Rover uses objects to store shared data. Each object is stored as a primary copy with a list of tentative updates. (This approach is suggested and analyzed in a paper [5] by Gray et. al.) Mobile clients prefetch data before disconnecting with the server. Messages are sent between mobile clients and servers through queued remote procedure calls, meaning that if the client is not connected to a server, it will send those RPCs when connection is reestablished. When the client does reconnect, the server detects and resolves any conflicts that occured by incorporating the client's tentative updates. In addition, Rover allows for the development of both application-aware and application-transparent applications.

The Bayou system [15, 19] uses weakly consistent storage that permits updates to be exchanged on a peer-to-peer basis. Each Bayou server keeps a record of all writes either performed on that server or retrieved from other machines. Each server contains a database that is the result of replaying all of the writes. Servers share their writes with each other on a pair-wise basis, and in doing so eventually propagates writes to the other servers in the system. It allows for incremental progress in achieving this consistency, in that there are no adverse effects if clients become suddenly disconnected. In addition, Bayou uses a primary replica to commit certain writes as stable writes that can not be rolled back. In this way, Bayou shares with Rover the idea of having primary data with tentative updates.

Mobile Aleph shares several standard concepts with these and other distributed systems. Clients in our system prefetch data as it is updated while connected to the application server. On mobile clients, global data is stored as an installed version plus a list of tentative updates. And Mobile Aleph allows application-specific merge procedures to resolve update-update conflicts. Mobile Aleph combines the client/server and peer-to-peer architectures described above to enable mobile clients to maintain small, up-to-date caches of global data.

We should note the popularity of recent peer-to-peer applications like Napster and Gnutella. These applications enable clients to share files without routing them through a centralized server. In these applications, the peer-to-peer model presents no difficulty in reconciling differences between files because these files are not being changed by the clients. In our system, we are dealing with *mutable* objects that are being continually modified. Therefore, we need some way of deciding how different versions of an object get merged together.

Lastly, Mobile Aleph adds the ability to have group communication and consistency in ad hoc networks of mobile clients, while preserving the ability of those clients to connect with application servers as desired. Mobile Aleph achieves group communication using ordered multicast within a group of clients. This is similar to the notion of *virtually synchronous* group communication for distributed systems [1]. In virtually synchronous systems, each node in the group is guaranteed to receive all group events in the same order, including messages and events for nodes entering and leaving the group. This means that the system must track when a node enters or leaves the group. The difference between virtually synchronous systems and our directory manager is that we do not care when the node group changes; indeed, no events are propagated to the group when a client joins or leaves.

3 The Mobile Aleph System

3.1 Types of PEs

There are two types of PEs in Mobile Aleph: long-lived servers and mobile clients. In order for mobile clients to find an application server to connect to, they must first contact the nameserver. The nameserver runs as a standalone PE on a workstation. Upon starting up, the nameserver publishes its address (host name and port) to a well known, globally-accessible URL. (This last point is hidden from applications that use the nameserver's services.)

Once a nameserver is running, application servers running as standalone PEs can register with the nameserver. A server sends its address to the nameserver, and the nameserver adds the address to the list of addresses of servers for that application. In this way, the nameserver supports multiple application servers for a given application. As inspired by leasing in Sun's Jini technology [21], the nameserver considers each address to be available for only a fixed amount of time. After that time, the address is considered to be invalid and is purged from the nameserver's records. Soon before this happens, the nameserver attempts to contact the application server to remind it to reregister its address. If it does reregister, then its address remains valid.

Mobile Aleph provides support for creating these long-lived application servers through the `PE.runAsServer()` method. This method handles the registration of the PE with the nameserver, as well as making sure that the PE stays alive until it is explicitly shut down. A long-lived server can be shut down by calling the `PE.stopAsServer()` method.

Mobile clients can contact the nameserver to obtain the address of one or more application servers for the clients' particular application. A client can request the list of all available application servers, a single application server at a specified host, or a single application server that resides anywhere on the network. If the client requests a single server to be returned when multiple are available, the nameserver returns the server addresses in a round-robin fashion. This facilitates server load-balancing if multiple servers are available that provide the same services.

3.2 Inter-PE Communication

In a mobile environment, clients will be frequently entering and leaving the network. For this reason, we provide the following types of communication among mobile clients and servers:

- Clients can unicast messages to the server. This is always possible because the server's address is available from the nameserver.
- The unicast addresses of mobile clients are assumed to be valid only immediately after they are received. For instance, if client A sends a message containing its address to client B, then client B can use that address to immediately send one or more messages back to client A. This enables one PE to request and receive data from another PE. However, after the exchange takes place, we assume that the address is no longer valid. The reason for this is that after client A is done receiving that data, there is no guarantee that it would be available for further communication.
- The server can use ordered multicast to send messages to all clients that are listening to it. In Aleph, this is implemented by having clients register with an Aleph `Event`. Although ordered multicast requires more overhead than unordered multicast, the overhead is acceptable for the types of applications that we're considering, and it simplifies the algorithms used for data consistency.
- In an ad hoc network, clients can use ordered multicast to communicate with the other clients in the network.

3.3 Mobile Data Management

Data is shared among PEs through the use of the `GlobalObject` class. Aleph directory managers allow access to global objects through the following interface:

```
public abstract class DirectoryManager {
  /* ... */
  public abstract void newObject (GlobalObject key, Object object, String hint);
  public abstract Object open (GlobalObject object, String mode);
  public abstract void release (GlobalObject object);
  /* ... */
}
```

The global object is a key that allows the directory manager to retrieve the Java `Object` that contains the application data. When we refer to global data, we area talking about this Java `Object`, not about the key to this data. Global objects are considered to be mutable, in that properties of the objects are changed by clients. For example, the price a stock is selling for, the contents of a document, or the position of a player in a game are all properties of a global object that would be changed by the application. In our system, a global object may be created on the client or server side. Once the object has been created, a mobile client can then open a global object in either read or write mode. When the client is done accessing the object, the client releases it. This is how object properties are modified in Aleph.

We have developed two implementations of the abstract `DirectoryManager` class for use by mobile applications. Workstation-based application servers use the `MobileServerDirectory` directory manager, and mobile clients use the `MobileClientDirectory`. The internal storage of global objects for each of these directory managers is pictured in Figure 1 and is discussed below.

3.3.1 Mobile Client

Each mobile client contains a cache of global objects. Each object is stored as an *installed version* of the object plus a list of *tentative updates.* A tentative update is a change that has been made on a client. Tentative updates may or may not become installed at a later point. An installed version of an object is an update that has been accepted by the server, and it can not be rolled back.

The client contains a record in its cache for every object's installed version and tentative updates (see Figure 1). Each record contains the ID of the object that was changed; the new object data; a timestamp that the update was made; the timestamp of the update that this was based on; and the version number of the installed version that this update came from. A timestamp consists of the time and machine ID, thus uniquely identifying any tentative update made in the system.

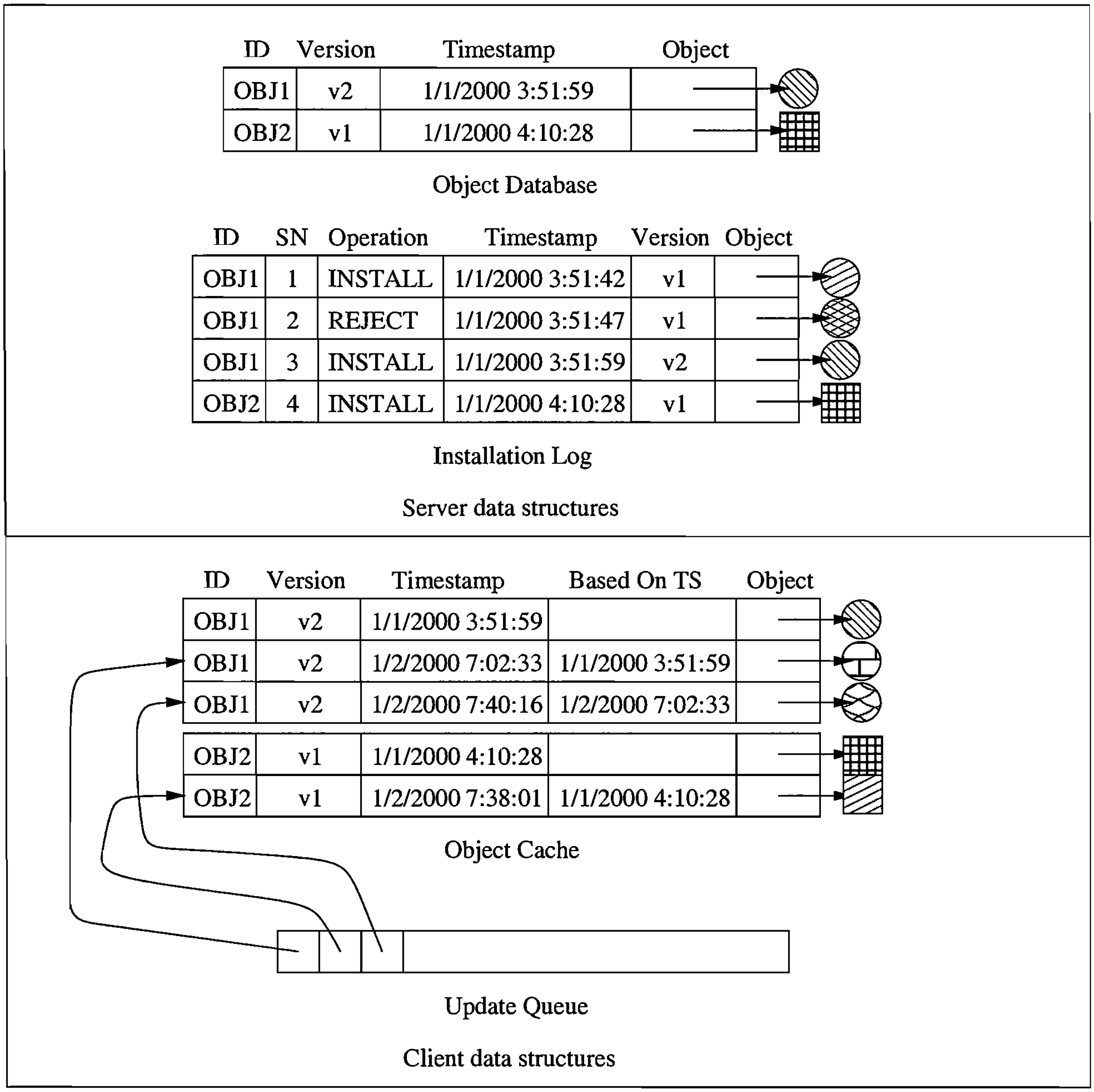
When the client application wants to open a global object, the directory manager returns the object with all of the tentative updates applied to it. The client makes a tentative update by opening the object for writing and then releasing it. A record for the tentative update is then appended to the list.

Besides the cache of global objects, the client also keeps an update queue which lists all tentative updates stored by the client. When the client connects with an application server, it sends all tentative updates from the queue to the server. See section 3.5.1 for more information.

When changes are made to the client's cache or update queue, the altered data is lazily flushed to the file system. If the client is shut down and restarted, the client will read the data from the file system and restore its previous state. This allows mobile clients to be restartable PEs.

3.3.2 Server

Each mobile application has an application server. An application server is a long-lived PE that runs on a workstation. Since mobile clients do not have much memory, the server is in charge of maintaining a database of all the permanently installed versions of every global object in the system. The server receives tentative updates from clients and decides whether these updates should be installed or rejected. (See section 3.5.1 for how the server decides whether an update should be installed.) When the server installs an update, it increments the version number of the global object and stores the object in its database. As mentioned above, installing a tentative update means that the update is committed and can not be undone.



ID	Version	Timestamp	Object
OBJ1	v2	1/1/2000 3:51:59	
OBJ2	v1	1/1/2000 4:10:28	

Object Database

ID	SN	Operation	Timestamp	Version	Object
OBJ1	1	INSTALL	1/1/2000 3:51:42	v1	
OBJ1	2	REJECT	1/1/2000 3:51:47	v1	
OBJ1	3	INSTALL	1/1/2000 3:51:59	v2	
OBJ2	4	INSTALL	1/1/2000 4:10:28	v1	

Installation Log

Server data structures

ID	Version	Timestamp	Based On TS	Object
OBJ1	v2	1/1/2000 3:51:59		
OBJ1	v2	1/2/2000 7:02:33	1/1/2000 3:51:59	
OBJ1	v2	1/2/2000 7:40:16	1/2/2000 7:02:33	
OBJ2	v1	1/1/2000 4:10:28		
OBJ2	v1	1/2/2000 7:38:01	1/1/2000 4:10:28	

Object Cache

Update Queue

Client data structures

Figure 1: An example of data management on the client and server sides. Here, the client and server hold versions of two objects, OBJ1 (represented as the circle) and OBJ2 (the square). Different data in each object are represented as different patterns in each shape. Here, the server's installation log shows that a tentative update to OBJ1 made at 1/1 3:51:47 was rejected by the server, and an update made at 1/1 3:51:59 was installed as version 2. The client then received this version from the server, disconnected, and made two more tentative updates on 1/2.

The server also maintains an installation log of all installations and rejections of tentative updates submitted to the server. Each entry into the log gets assigned an sequence number, which is incremented for each installation or rejection entry that is added.

3.4 Application-System Interaction

Both the mobile client and server applications present an interface to the directory manager so that the application can be more involved in and aware of the actions of the manager. The mobile application running on the server and client sides implement the following interface:

```
public interface MobileApplication {
  public Object merge(GlobalObject key, Object tentative, Object current);
  public void objectInstalled(GlobalObject key, Object installed);
  public void objectRejected(GlobalObject key, Object rejected, Object current);
}
```

The interface has the two callbacks from the directory manager to allow the application to be informed when a tentative update is installed or rejected. The most important part of this interface, however, is the merge method. When the directory manager can not resolve a conflict between two tentative updates, it asks the application for assistance. Many applications have different semantics for how to resolve these conflicts. For instance, a room reservation system could merge two reservations that were made for different times, but would not be able to merge them if they were for the same time. Other applications might always or might never be able to merge two updates. In any case, the application knows best how to perform this merge procedure, so the system delegates it to the application.

The application should return the merged object if merging is possible; it should return null otherwise. While the application can use any logic it wants to determine the merging procedure, it must be deterministic. That is, all mobile clients and the application server, given the same data, must return the same merge result. Note that the default is to disallow merging, i.e. merge returns null if no application-specific merge procedure is set.

The mobile application running on the client also implements the methods in this interface:

```
public interface MobileClientApplication extends MobileApplication {
  public Class getServerClass();

  public static final int AUTO_MODE = 0;
  public static final int OFFLINE_MODE = 1;
  public static final int ADHOC_MODE = 2;
  public static final int SERVER_MODE = 3;

  public int getDesiredConnectionMode();
  public void connectionStatusChanged(int mode, Address serverAddress);
}
```

```
CLIENT-RECEIVE-ENTRY(logEntry)
if logEntry is not the next entry that we expect to receive
   then new logon is needed
        return
if logEntry is an INSTALL entry
   then $myRecord \leftarrow$ get my version of the object in logEntry from my cache
        if $myRecord = \text{NIL}$
           then CLIENT-INSTALL(logEntry)
           else if version of $logEntry >$ version of myRecord
                   then CLIENT-INSTALL(logEntry)
   else CLIENT-REJECT(logEntry)
```

Figure 2: Client algorithm for processing a log entry received from the server.

The client can operate under four different connection policies. The default is the auto mode. Here, the client connects to the server if it is available; else, it connects to an ad hoc network; otherwise it is offline. The other three modes indicate that the client wants that connection level, or offline if the network is unavailable.

The client is also informed when the connection status changes. This can be used to change the way the application acts when different connection levels are started. The other method, `getServerClass()`, lets the application tell the mobile directory manager which server it wants to connect to.

3.5 Directory Algorithms

As discussed above, the client can be either connected to the server, connected to an ad hoc network, or offline. Here we describe how a mobile client operates in each of these scenarios.

3.5.1 Server Mode

Motivation. When the client detects that the application server is available, the client wants to send its tentative updates to the server, so they can be evaluated for installation. In addition, the client wants to receive the latest entries in the server's installation log, so that the client can incorporate any changes that other clients have made since this client last disconnected.

Algorithm. The client maintains the sequence number of the last entry it received from the server's installation log. When the client detects that the server is available, it sends this number in a logon message to the server. The client also includes its address in the message, so that the server can send messages back to it. The server then sends the client each entry in the installation log after the number it was given.

Figure 2 gives the algorithm for how incoming log entries are processed on the client side.

```
SERVER-RECEIVE-UPDATE(tentUpdate)
installedRecord ← get my version of the object in tentUpdate from my database
if installedRecord = NIL
  then ▷ object was created on client, so add to database
       SERVER-INSTALL(tentUpdate)
  else if tentUpdate is based on the object in installedRecord
          then SERVER-INSTALL(tentUpdate)
          else mergedObject ← MERGE(tentUpdate, installedRecord)
               if mergedObject ≠ NIL
                 then SERVER-INSTALL(tentUpdate with data in mergedObject)
                 else SERVER-REJECT(tentUpdate)
```

Figure 3: Server algorithm for deciding whether a tentative update received from a client should be installed or rejected.

First, if the entry is not the next one sequentially, then the client must have become disconnected momentarily, causing it to miss an intermediate entry. In this case, we need to log on again. Otherwise, if the entry is for an installation, we want to install the object in this entry. If we don't already have this object in our cache, then we add it to the cache by installing it. If we already had a version of the object, then we install the new one if the version number of the incoming object is greater. Alternatively, if the log entry is for a rejection, then we reject this entry.

Installing an entry means that we make the object the primary copy in our local cache. For both installation and rejection entries, we look through our tentative updates for that object. If this entry is for an update that we have, then it has been processed by the server, so we remove the tentative update from our cache.

After the client has received all the entries in the server's installation log, the client sends all of the tentative updates that are in its update queue along to the server. For each update the server receives, it uses the algorithm given in Figure 3 to determine whether that update should be installed. The server installs the update automatically if it is for a new object, or if the update was based on the currently installed version of the object. Otherwise, the update was based on an earlier version, so it may be incompatible with the current version. In this case, the system asks the application to resolve the conflict. If the two updates can be merged, then the merged version is installed; else, the update is rejected.

When the server installs a new version, it increments the version number of the object, it adds the new object data to its database, and it puts a new install entry into the installation log. For rejections, the server adds a reject entry into the installation log. In either case, it multicasts the message to all clients that are logged on. Clients then process these log entries just as they did during the logon process.

Analysis. This algorithm satisfies several requirements outlined at the start of the paper. First, after a client's initial logon process is complete, its cache is synchronized with all other connected clients, within the time of a roundtrip client-server communication round. After the client joins, it has received the same list of entries from the server's installation log as other clients. Then it sends its tentative updates to the server, and the resulting install or reject entries are then multicast to all connected clients. So all clients have the same data. In addition, the data is clearly the most recent installed data available. Of course, other clients who are not logged on to the server may be making more recent tentative updates, but the server-connected clients have no way of getting those updates until the other clients are able to be contacted.

3.5.2 Ad Hoc Network Mode

Motivation. When a client joins an ad hoc network, it needs to synchronize its cache with the caches of the other clients in the network. This means it must receive data from the clients in the network and then integrate any installed versions and tentative updates that are new to the group.

Algorithm. Let client A be the client that wants to be added to the network. Client A starts by getting the address of a random client already in the network; call it client B. Client A joins the multicast group, and it sends a logon message to client B. Client B responds by sending client A its cache, which is composed of a list of all the installed versions of its objects plus their tentative updates.

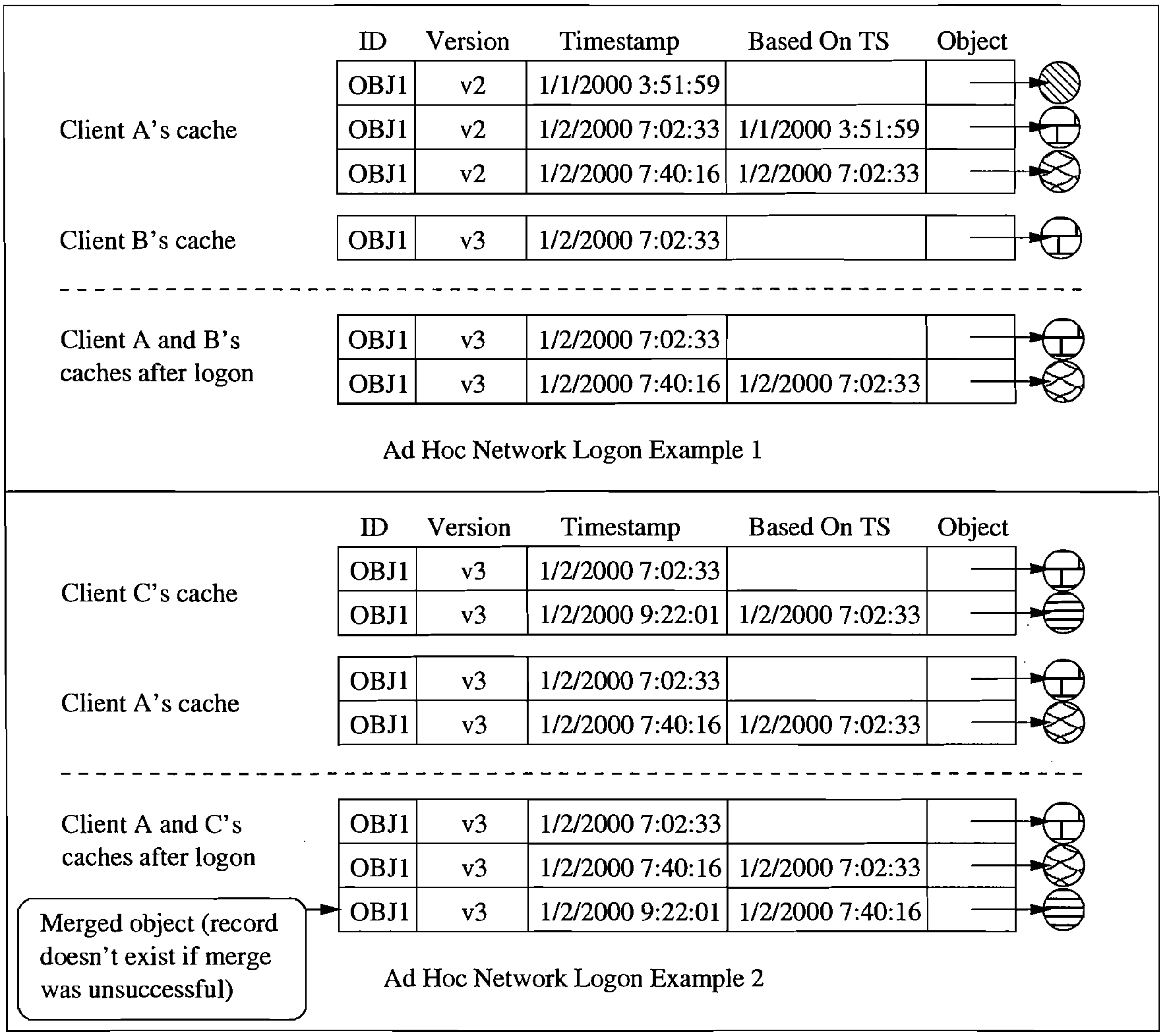
Client A first replaces its cache with the one received from client B. A then examines its old cache, looking for newer installed object versions and additional tentative updates. For each object where client A had a newer installed version, it multicasts that object and its updates to the group. For other objects, it multicasts all tentative updates that the incoming cache did not contain.

When any client receives an object and its tentative updates via multicast, it compares the installed version number with the object in its cache. If the incoming object has a later version, then it stores that object and updates in its cache, and merges its old tentative updates onto it (if merging is necessary).

When any client receives a tentative update, it checks to see if it is based on the last update it has stored in its cache. If so, it appends its tentative update list with the incoming one; otherwise it must try to merge in the update.

Analysis. It is clear that, after a client logs on to the system, it synchronizes its cache with the rest of the clients in the network. Furthermore, all clients in the ad hoc network contain the latest installed version of all objects. Objects with later installed versions always replace the current object in the client's cache. When a client logs on, all of its objects with later versions are sent to the group. Therefore, all clients get the benefit of getting the most recent server installed data.

This feature also means that if a client goes to the server, installs some updates, and returns to the ad hoc network, it will reduce the cache size of the clients in the network. By sending out the newly installed version of the object, clients in the network can remove all updates before the one that was installed.



	ID	Version	Timestamp	Based On TS	Object
Client A's cache	OBJ1	v2	1/1/2000 3:51:59		
	OBJ1	v2	1/2/2000 7:02:33	1/1/2000 3:51:59	
	OBJ1	v2	1/2/2000 7:40:16	1/2/2000 7:02:33	
Client B's cache	OBJ1	v3	1/2/2000 7:02:33		
Client A and B's caches after logon	OBJ1	v3	1/2/2000 7:02:33		
	OBJ1	v3	1/2/2000 7:40:16	1/2/2000 7:02:33	

Ad Hoc Network Logon Example 1

	ID	Version	Timestamp	Based On TS	Object
Client C's cache	OBJ1	v3	1/2/2000 7:02:33		
	OBJ1	v3	1/2/2000 9:22:01	1/2/2000 7:02:33	
Client A's cache	OBJ1	v3	1/2/2000 7:02:33		
	OBJ1	v3	1/2/2000 7:40:16	1/2/2000 7:02:33	
Client A and C's caches after logon	OBJ1	v3	1/2/2000 7:02:33		
	OBJ1	v3	1/2/2000 7:40:16	1/2/2000 7:02:33	
	OBJ1	v3	1/2/2000 9:22:01	1/2/2000 7:40:16	

Merged object (record doesn't exist if merge was unsuccessful)

Ad Hoc Network Logon Example 2

Figure 4: Two examples showing the integration of client caches in an ad hoc network logon. In the first example, client A joins a network that client B is already in. At some point, A's tentative update made at 7:02:33 was installed as version 3. B has this installed version, but A has not yet visited the server. When their updates are integrated, A adds the new installed version and contributes its later tentative update. No merging is necessary here. In the second example, at some later point client C joins the same network. C has the same installed version as A, but the two clients have different tentative updates. Since A is on the network already, its tentative update is used first, and C's tentative update is merged in if possible.

3.5.3 Offline Mode

An offline mobile client makes tentative updates to objects in its local cache. If it wants to be connected to other networks, it will occasionally check to see if those networks are available. Otherwise, it will not try to send out any tentative updates that it makes. This allows the client to save battery power while offline.

3.6 Issues

There are several issues with the above algorithm. First, it would be preferable to allow mobile clients to prefetch and store updates only for those objects that they are interested in. Each client could define a profile that specifies exactly which objects the client cares about. When connected to the server, the server would only send it installation log entries for those objects specified in the client's profile. In an ad hoc network, however, using these object profiles complicates things slightly, as the client in the network that it logs on to would not be guaranteed to have all objects of interest. In this case, the client would instead have to advertise its profile, and have one or more clients in the ad hoc network reply with their cache entries.

Another issue is when the server's installation log may be truncated. Several options are available. If the complete list of possible clients can be specified, then the server can remove a log entry once all clients have received it. Another possibility is that the server can assume that all clients must connect with the server during a certain period of time, say a week or two. If a client hasn't connected during that time, then it must download the entire server cache, and it can not commit any tentative updates that it holds. However, server log truncation is not a major concern, since each client only needs to read each entry once, and since the server is assumed to have sufficient disk space to hold the log.

An issue not discussed in the above algorithms is what happens when other clients are making updates when one client is trying to log on to the server or to an ad hoc network. In this case, the client logging on must queue up incoming updates, and when the logon process is over, it integrates those updates in order. This solves the problem of potential inconsistency between clients.

Lastly, when a client joins an ad hoc network, it would be preferable if that client only had to receive those updates that it does not currently contain, instead of receiving an entire cache. Doing this would greatly reduce the amount of data exchanged between clients, especially when the objects are large. While this improvement has not yet been incorporated into the system, implementing it does not complicate the algorithm at all.

4 Simulation

4.1 System Details

Application servers for our simulation ran on Sun Ultra 10 workstations running Sun Solaris 7. Mobile clients ran on NEC MobilePro 800s running Windows CE 2.11. The MobilePro 800 has a MIPS 4000 processor and 32MB of RAM that is used for both the file system and for application execution. The servers used JDK 1.1.6 with PersonalJava compatibility classes, and the clients ran

Sun's PersonalJava Runtime Environment for Windows CE, version 1.0. This client environment uses an implementation of the PersonalJava 1.1.3 specification, a subset of JDK 1.1.x.

Mobile clients were connected to our LAN through a Proxim RangeLAN2 wireless network. Each mobile client used a RangeLAN2 7410 CE PC card to connect to this wireless network. This network did not support multicast or broadcast between mobile clients. Therefore, multicast in ad hoc networks used an Aleph Event stored by a "beacon" PE running on a workstation. This beacon was responsible for transmitting multicast messages from a mobile client to all other clients in the network. The beacon also implemented the anycasting used in the network logon process. As for the Aleph components used, all tests were run with the UDP communication manager, the simple event manager, and the simple transaction manager.

4.2 Comparison with Home Protocol

In our simulation, we compared the existing home directory manager with the new mobile directory manager when clients are connected to the application server. In each, we had one mobile client periodically writing to a global object and another mobile client reading that object. We compared these two protocols on the amount of time needed to access (open and release) the object, and on the number of messages sent and received by the two PEs.

We ran a number of trials using different update intervals for both the writer and the reader PEs. For example, a reader update interval of 2 seconds means that the reader opened the global object for reading at a random time in every 2 second interval. In one set of experiments we varied the writer's interval, and in another we varied the reader's. We ran these tests using both directory managers. Each trial lasted for one minute.

The results are shown in figures 5 and 6. For the home directory manager, the ratio of the mean access times for the writer and reader is proportional to the ratio between the writer's and reader's update interval. In other words, when the writer was accessing the object more frequently than the reader, the writer took less time on average to access the object. This result is explained directly by the home protocol. After a PE receives an object, it stores that object until another PE tries to access it. When that happens, the PE's home (here, the application server) invalidates that copy of the object. Therefore, if a PE is frequently accessing an object, it will be less likely to need to request that object from its home. This also explains why the number of messages sent and received by both PEs decreases as their update intervals increase. Of course, when the writer's update interval increases, the number of messages decreases because fewer changes are being made to the object.

When the mobile directory manager was used, the writer's access times and number of updates made did not depend on the reader's update interval. This result was no surprise: the lack of read locks in the mobile protocol's optimistic data replication eliminates any need for the writer to wait to access the data. We expected that access times on the readers would also not depend on the update intervals. However, the reader's access times increased as its update interval increased.

Further analysis revealed that the mean access time increased only because the first read always triggers a logon with the server. The time to do a logon is considerably longer (on the order of a couple seconds) than for a read that does not require a logon. In the trials with larger update intervals, that first read that required a logon made up a greater percentage of the number of reads performed. Therefore, it contributed more to the mean access time. The time spent performing reads which didn't need to logon to the server stayed at the same low value seen when the reader's update interval was constant - less than 100ms.

In comparing the two protocols, we first note that the greater network latency in mobile environments hampers the use of locking in the home directory manager. When a PE needs to acquire a lock on an object, it takes a greater amount of time before the PE gets to access the object. The optimistic scheme in the mobile directory manager avoids this problem by receiving updates asynchronously, minimizing the amount of time needed later when the application decides to access the shared data. When logon costs are disregarded (or become insignificant in the case of a long-lived client), the mobile directory manager is considerably faster at reading and writing data. Even for clients that run for a short amount of time, as in our simulation, the mean access time for reading and writing in the mobile protocol beats the home protocol when reads are more frequent than writes.

Second, we would expect that the mobile directory manager would incur greater message counts because updates are broadcast to PEs regardless of whether they will soon read or write using the updated data. As shown in figure 7, this was case for messages received when writes were much more frequent than reads (ex. writer rate = 1, reader rate = 4 or 6). However, when writes occured less than twice as frequently as reads, the mobile directory manager had the lower message count. For each data request and release in the home protocol, the reader needs to send 2 messages and receive 2 messages. A reader using the mobile protocol just receives 1 message per update. The reader never has to send messages except for an initial logon.

That readers don't need to send messages is an advantage for the mobile protocol in terms of power conservation. Our wireless cards use twice the energy to send a message as they do to receive one. Therefore, writes would have to be over 6 times as frequent as reads for the home directory manager to use less power in sending and receiving messages, assuming that all messages are the same size. This assumption is not entirely accurate, since most messages in the mobile protocol include the shared data itself, whereas many messages in the home protocol are simply requests for or invalidations of data. However, if the mobile protocol sent updates that were actually bit-wise diffs of the change, or an operation to be performed at the client, then that would cut down on much of the message costs.

4.3 Proposed Evaluation of Ad Hoc Networking

Since the home directory manager does not support disconnected operation, we could not compare it with the mobile directory manager when clients were in an ad hoc network. Therefore, we intended to run a second simulation in which a group of clients formed an ad hoc network and periodically wrote to a global object. However, due to resource limitations such as the inability of mobile clients to perform true multicast and to the lack of devices available, we were not able to perform a meaningful simulation on mobile clients.

If we were to run this simulation, we would want to vary the number of devices in the ad hoc network from around 5 devices up to 30 or 40 devices. This would attempt to simulate some of our target applications, such as collaborative classroom software or multiplayer games. As in the previous simulation, we would put a number of readers and writers in the network, varying their

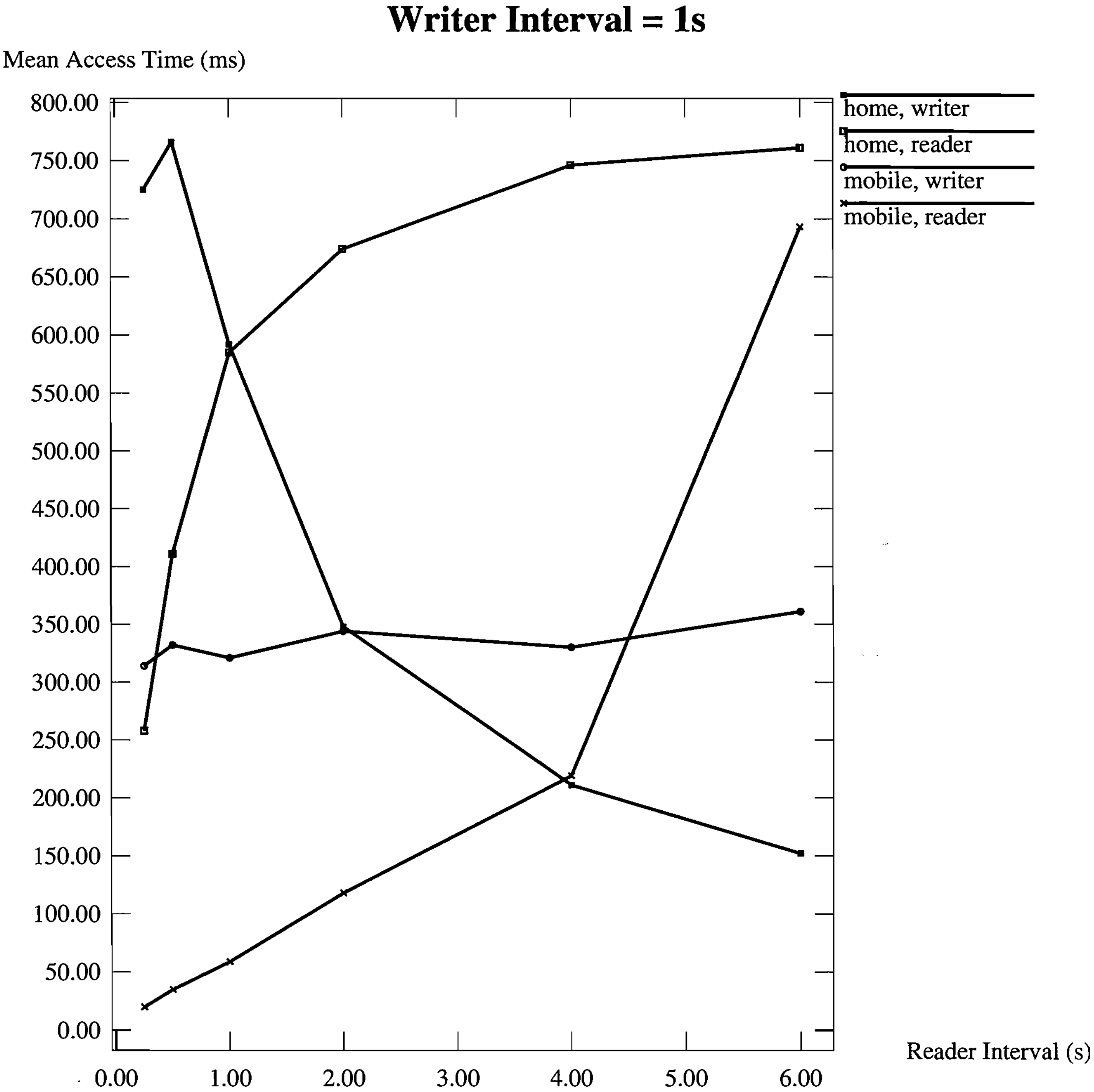


Figure 5: Experimental results for home protocol comparison, where the writer's update interval is set at 1 second and the reader's update interval varies from 0.25 to 6 seconds.

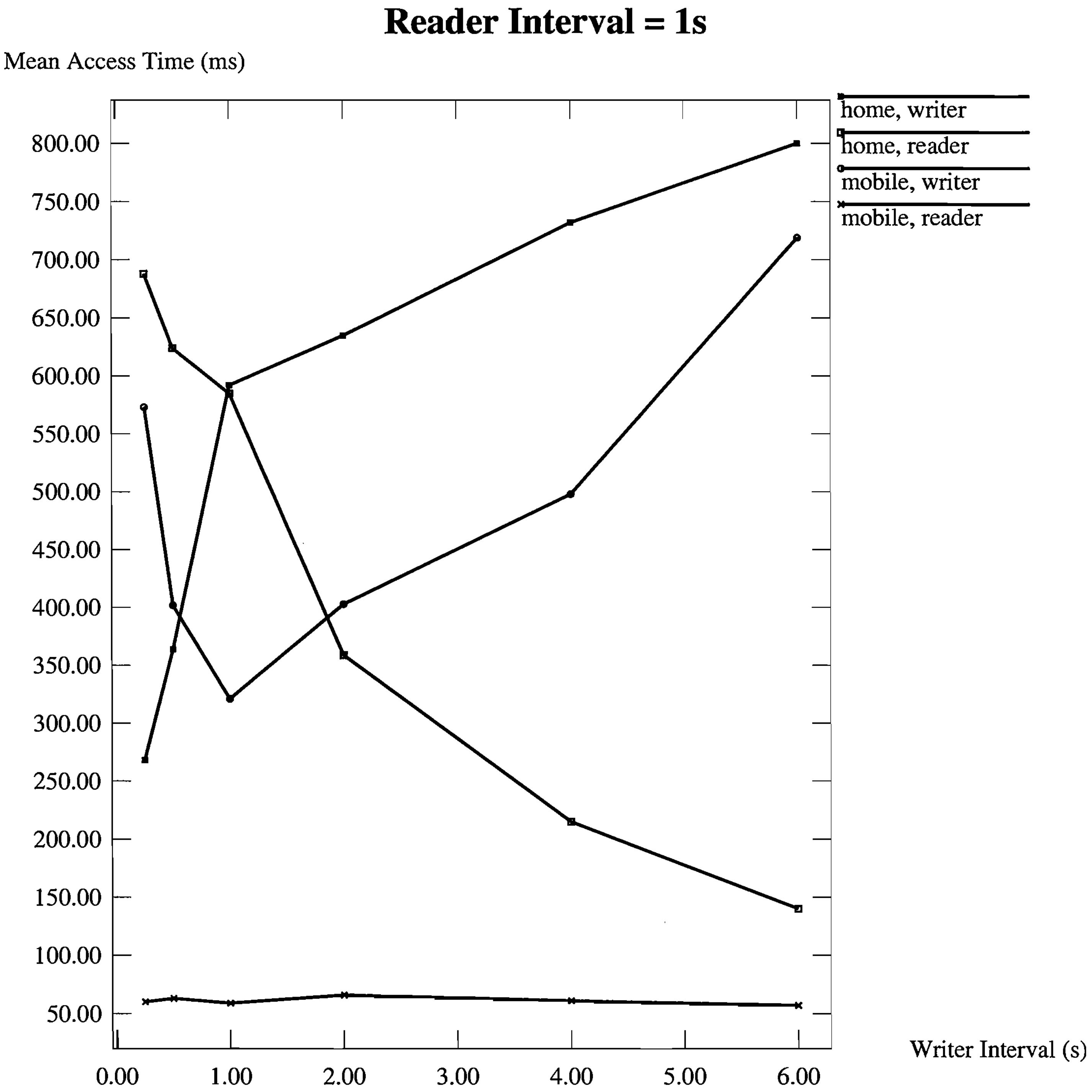


Figure 6: Experimental results for home protocol comparison, where the writer's update interval varies from 0.25 to 6 seconds and the reader's update interval is set at 1 second.

protocol	w. rate	r. rate	w. msgsent	w. msgsrcd	r. time	r. msgsent	r. msgrcvd	r. time
home	1	.25	66	66	725	67	67	258
home	1	.5	59	59	766	60	60	411
home	1	1	55	55	592	56	56	585
home	1	2	41	41	348	40	40	674
home	1	4	27	27	211	26	26	746
home	1	6	18	18	152	17	17	761
home	.25	1	67	67	268	66	66	688
home	.5	1	66	66	364	67	67	624
home	1	1	55	55	592	56	56	585
home	2	1	43	43	635	43	43	359
home	4	1	25	25	732	25	25	215
home	6	1	17	17	800	17	17	140
mobile	1	.25	44	45	314	1	45	20
mobile	1	.5	42	43	332	1	42	35
mobile	1	1	42	43	321	1	42	59
mobile	1	2	44	45	344	1	44	118
mobile	1	4	44	45	330	1	44	219
mobile	1	6	41	42	361	1	41	693
mobile	.25	1	65	66	573	1	66	60
mobile	.5	1	58	59	402	1	59	63
mobile	1	1	42	43	321	1	42	59
mobile	2	1	24	24	403	1	24	66
mobile	4	1	15	16	498	1	15	61
mobile	6	1	10	11	719	1	11	57

Figure 7: Table of all experimental results for home protocol comparison, including message counts. Writer and reader rates are in seconds. Mean access times are in milliseconds.

update intervals. Clients would occasionally leave and reenter the network. To evaluate the system, we would record access times and message counts. We would also examine the number of merge operations needed at the clients to get an idea of the percentage of updates that needed merging.

We would expect, of course, that as the writers decrease their update interval, message counts and the number of merges needed would increase. The amount of time needed to join an ad hoc network should increase as the number of updates increase. (Leaving the network never has a cost.) In addition, access times should remain constant since the amount of communication needed to open and release an object does not depend on other clients in the system.

The main goal of such a simulation would be to find the point (update interval) at which the percentage of updates requiring merges becomes too large. The threshold percentage would depend on the merge procedure employed by the type of application. In addition, we would need to have acceptable system performance on the mobile devices.

5 Project Reflections

5.1 Design Decisions

In the course of creating Mobile Aleph, we made several design decisions that had a major impact on how the system turned out. Below is a list of some of these design decisions, along with a description of the considerations that made us choose one solution over its alternatives.

One of the first problems we had to confront was how to deal with mobile clients that did not have much memory, and that the memory contents would be deleted if the client's battery ran out of power. Clearly, the mobile clients could not store the definitive copy of any shared object; we would somehow need to maintain a permanent record of every shared object in stable storage. This requirement led to the use of server workstations whose job it is to maintain this object database. Since all mobile clients would be able to find the address of the server through the nameserver, it was clear that the server should be the place where versions of the global objects are installed. If the clients decided when updates should be committed, then if a client went offline for a while, the caches of other clients would grow as they maintained an increasing number of tentative updates to submit.

The biggest design issue was to determine what level of consistency we wanted the directory manager to support. Mobile Aleph allows shared data to be updated anywhere and anytime, regardless of the connectivity of the machines involved. Mobile clients may be switching back and forth between the server and an ad hoc network, other clients could be connected to the server all the time, while still others could be almost always offline, only synchronizing their cache with the server occasionally. In these usage scenarios, where the interaction pattern between clients is unclear, it can be tricky to determine when a set of clients can be sure that they have the same data.

For group-oriented applications, we have the requirement that clients which want to work together need to have the same view of the data, and that they need to somehow notify each other when updates are made. For this reason, occasional pair-wise data synchronization was not enough; we needed to establish the concept of a group of clients that were connected and that had the same data. One part of implementing this in both ad hoc and server-based networks was to use ordered multicast to transmit updates as they are made. When combined with a deterministic merge procedure in the case of ad hoc networks, we ensure that all clients process the updates in exactly the same way, and therefore end up with the same data. As noted in section 2, this is similar to the idea of virtual synchrony for group communication.

The other part of achieving data consistency within a group of mobile clients was determining how a client incorporates its updates when it joins an ad hoc network. One guiding principle behind our design was to make sure that when a client received an installed version of an object from the server, it would never go back to a previous version. Because installed versions can not be rolled back on the server, it wouldn't make much sense if they were effectively rolled back on the client. For this reason, when a client joins a group, the group uses the later of the installed versions held by the client and the group.

This leaves the task of integrating the tentative updates. While these updates are "tentative" and may end up being rejected, we still need to give them their best chance for eventual installation.

When a client introduces a new installed version, we apply the tentative updates from the group after the updates from this new client. The updates from the client will certainly not need merging, while the ones from the group may or may not. If we did it the other way around, then all the updates would potentially need to be merged.

When the group and the joining client have the same installed version, the question is what the ordering of their combined tentative updates should be. One potential solution would be to order the updates by timestamp. An advantage of this solution is that whereever an update is made, the update that is preferred is the one that was made first. This seems like a good property to have. However, this strategy has a major disadvantage. When a client joins the group, the timestamps of its updates could cause them to be inserted into the middle of the list of updates that the group has. This would necessitate that later updates be merged and therefore possibly discarded. We would prefer that an incoming client not be able to disrupt the group like this, since the group may have been working together for a while, possibly creating a long list of tentative updates. Therefore, we force updates from new clients to be merged onto the group's updates when the updates are based on the same installed version of the object.

5.2 Lessons Learned

We have learned a number of things from working on this project, both about Aleph and about designing optimistic data replication systems in general. First, there are a couple changes to Aleph that could be made that would facilitate the further development of this and other directory managers. One such change is in the way that applications access shared data. Currently, an application must first acquire the `GlobalObject` key before it can ask the system for the data that it is associated with. How this key is acquired is left up to the application; usually it gets it by sending a message to the server asking for that key. The mobile directory manager contains a non-standard `getIDs()` method that returns all `GlobalObject` keys stored in its cache. This is a start to pushing the responsibility for getting these keys down to the system. However, applications still have to read all shared objects using these keys to find out what those objects hold.

It would be better if the directory manager presented a database interface to the application. The application could then query the database for the types of objects that it wanted to access, and if the directory manager had them, those objects would be returned. In addition, instead of opening and releasing an object to read and write it, these operations could be made through the database interface. Such an approach would allow the system to use operation-based logging instead of simply recording the new data. Recording operations made would further assist application-specific merge mechanisms to determine how to reconcile conflicting updates.

An additional Aleph improvement would be to add a shutdown procedure to the directory manager interface. When an application exists normally, Aleph could notify the directory manager (and possibly other components) that they should complete their execution. This would help the mobile directory manager by allowing it to cache all its global object data immediately before exiting. By doing this, the directory manager would be able to rely less on occasionally flushing the cache to the file system, which turned out to be a major resource drain for the mobile clients.

With respect to optimistic data replication algorithms in general, we have gained an appreciation for the difficulty in ensuring the dual goals of cache coherency and that the greatest number of tentative updates are eventually installed. Properly defining the requirements placed on such a system, including the set of target applications, was essential to making the design decisions which attempt to acheive these goals. Despite the best efforts of the system to resolve conflicts internally, it seems much of the responsibility still falls on proper application design. Applications need to choose the granularity of their shared objects carefully, since the applications' merge procedures will need to reconcile conflicting updates to these objects.

6 Future Work

The main area for future work is the development of applications using the Mobile Aleph system. Through this development, we should learn what additional features are needed in the system to enable applications to have correct behavior. In addition, through application testing we can learn the actual performance of the system since we would have real-life connectivity and usage patterns. One issue in application development will be figuring out the best way to divide up application data into discrete global objects such that the above consistency model is effective.

Another issue is how to combine tentative updates into one transaction that must be committed or rolled back in full. How to integrate transactions with the mobile directory manager algorithms is an important area of future investigation. In addition, we expect that a more sophisticated merging facility will be needed to assist applications in resolving conflicts. Finally, we need to address scalability concerns so that the system can be used effectively when a large number of clients are connected.

7 Acknowledgements

First, profound thanks to my advisor, Maurice Herlihy, for his guidance, insight, and patience. Many of his suggestions are reflected in this paper, and without his assistance this work would not have been possible. In addition, I would like to thank Microsoft for their generous donation of equipment. I also want to thank Benety Goh for developing an initial version of parts of the software, and to Benety and Luis Vega for helping to discover the networking capabilities of Java applications on our mobile devices. Lastly, my appreciation goes to Tom Doeppner, Scott Lewandowski, and the Brown CS technical staff for their support of the mobile wireless network.

References

[1] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November, 1987.

[2] The Bluetooth Special Interest Group. The Official Bluetooth SIG Website. http://www.bluetooth.com.

[3] P. J. Braam. The Coda Distributed File System. In *Linux Journal* #50, June 1998.

[4] K. M. Chandy, J. Kiniry, A. Rifkin, and D. Zimmerman. II: The Infospheres Infrastructure User Guide. January 1998.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[6] R. Guy, P. Reicher, D. Ratner, M. Gunter, W. Ma, and G. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-peer Replication. In *Proceedings: ER'98 Workshop on Mobile Data Access*, 1998.

[7] R. Guy. Ficus: A Very Large Scale Reliable Distributed File System. Ph.D. dissertation, University of California, Los Angeles, June 1991.

[8] M. Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, January 1999.

[9] L. Huston and P. Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, August 1993.

[10] J. James and A. Singh. Design of the Kan Distributed Object System. To appear in *Concurrency: Practice & Experience.*

[11] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile Computing with the Rover Toolkit. In *IEEE Transactions on Computers: Special issue on Mobile Computing*, 46(3), March 1997.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *ACM Transactions on Computer Systems*, February 1992.

[13] Y. Lee, K. Leung, and M. Satyanarayanan. Operation-based Update Propogation in a Mobile File System. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[14] J. P. Munson and P. Dewan. Sync: A Java Framework for Mobile Collaborative Applications. In *IEEE Computer*, June 1997.

[15] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[16] G. J. Popek and B. J. Walker, The LOCUS Distributed System Architecture, MIT Press, Boston, 1985.

[17] Sun Microsystems. Java^{TM} 2 Platform, Standard Edition, v1.2.2 API Specification. Available from http://java.sun.com/products/jdk/1.2/docs/api/index.html.

[18] C. Tait. A File System For Mobile Computing. PhD thesis, Columbia University, New York, NY, 1993.

[19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[20] B. Topol, M. Ahamad, and J. T. Stasko. Robust State Sharing for Wide Area Distributed Applications. In *18th International Conference on Distributed Computing Systems*, May 1998.

[21] J. Waldo. Jini Technology Architectural Overview. Available from http://www.sun.com/jini/whitepapers/architecture.html.

[22] W. Yu and A. L. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.