Object Sharing Over Mobile Devices

Sharad Agarwal, Ioannis Mavroidis, Alec Woo
Department of Electrical Engineering & Computer Science
{sagarwal, maurog, awoo}@cs.berkeley.edu
May 1999

Abstract

With the advent of hand-held computing devices, the need for collaborative applications has become more urgent. In order to allow for the design of collaborative applications, a common infrastructure for the sharing of data is required for mobile devices. Such an infrastructure should be built for devices that have low compute capabilities and have low bandwidth, high latency network connections. The infrastructure should also allow for such devices to periodically disconnect from the network and reconnect, as is typical of the use of hand-held computing devices. Typical software implementations of distributed shared memory are not suitable for mobile devices. Thus this study implements and explores a software based shared object system for mobile devices. An application was built on top of this infrastructure. The design, implementation, results and limitations of the whole system are described.

1. Introduction

With the advent of hand-held computing devices, the need for collaborative applications has become more urgent. Essential functionality such as being able to schedule meetings, read up to date address book entries, describe concepts on shared whiteboard spaces and to disseminate data is now required by the users of such devices.

In order to allow for the design of such collaborative applications, a common infrastructure for the sharing of data is required for mobile devices. Such an infrastructure should allow any application to create shared data that any mobile device can read and modify, irrespective of the mobile devices' locations in the interconnection framework. Such an infrastructure should be built for devices that have low compute capabilities and have low bandwidth, high latency network connections. The infrastructure should also allow for such devices to periodically disconnect from the network and reconnect, as is typical of the use of hand-held computing devices.

Typical software based DSMs, distributed shared memory, are not suitable for mobile devices because they were not designed to meet these constraints. Thus this study implements and explores a software based shared object system for mobile devices.

The rest of this paper is organized as follows. Section 2 discusses existing software based DSMs. Section 3 describes the design requirements of shared memory for such devices. Section 4 describes our implementation. Section 5 presents an application built on this infrastructure. Section 6 describes the limitations of this study and its implementation while section 7 concludes the paper.

2. Related Work

Several software based shared memory systems have been explored by researchers in the past. This section will describe the main published works in this area.

Treadmarks [K94], [K95], [KCZ92], [D93] is an excellent example of a software implementation of distributed shared memory. This system was designed for sharing data in an application running across multiple workstations connected by a high speed network. A model of release consistency is implemented on a multiple writer update protocol. This implementation is not suitable for sharing data across mobile devices. Firstly, the system is not light weight enough. It has heavy processing and bandwidth requirements due to the complex protocol that the system implements. Also, data is shared at the granularity of pages. There is no uniform concept of pages across different mobile devices. Performance and ease of use can be improved by allowing the applications programmer to pick the right granularity of sharing.

Orca [B98], [HAB96] is a software based DSM that allows the applications programmer to pick the granularity of sharing. However, it is based on an update protocol system, and updates are implemented by function and argument shipping where each sharer executes the functions shipped by the multiple writers to modify the

data. This system would not be ideal for mobile devices given both the limited bandwidth and the lack of compute power.

CRL [JKW95] is a software based DSM system that also allows the applications programmer to pick the granularity of sharing. However, it may not be portable across all mobile devices. Sharing is achieved at the granularity of programmer specified regions of bytes, rather than programmer specified objects with type information which is a much more useful abstraction for applications programmers.

TSpaces [TSPACES] is a network communication buffer with database capabilities. It enables communication between applications and devices in a network of heterogeneous computers and operating systems. T Spaces provides group communication services, database services, URL-based file transfer services, and event notification services. However, TSpaces does not provide coherency and consistency guarantees, requiring applications to build in their own coherency mechanisms.

The inapplicability of the existing software based DSMs described above can be formulated into a set of design requirements for the ideal system.

3. Design Requirements

There are several requirements that need to be satisfied by a system that will allow data to be shared across mobile devices.

3.1 Software Solution

One of the main goals of this study is to explore systems that will permit the sharing of data across all mobile devices. There does not appear to be anything inherently special about any one set of mobile devices that warrants a specialized implementation of data sharing. Furthermore, the interconnection between mobile devices is extremely diverse. Some devices may be connected via wireless modems, wireless pager systems or via immobile docking stations. However, one common communication medium that is accessible across all devices and their interconnections is the Internet.

Given this goal, it is apparent that a software solution is called for. A hardware solution would require an enormous effort to support the variation in interconnection and devices that share data with little added benefit.

3.2 Shared Data Storage

The persistent storage of shared data can occur at different levels in a shared data system. This data can be stored in the mobile devices that create them. This is an unsatisfactory solution given both the limited memory space on mobile devices and the lack of communication bandwidth to handle coherence traffic of heavily shared data. Additionally, if a device were to become disconnected from the network, the availability of its data would be compromised. Alternatively, memory servers can be provided in the communication network that reside on traditional server machines. This is a more attractive solution since server machines have relatively infinite memory and much higher bandwidth. A third option that some shared memory systems have explored is the removal of persistent storage by ensuring that the very last remaining cached copy of shared data is never victimized. It is unclear what benefit the added complexity of such a system would provide in this design space. Thus persistent memory storage should be provided on memory servers in the network.

3.3 Language & Communication Infrastructure

At the time of this study, it is not clear if any particular vendor's products will dominate the mobile device market. Thus a solution that will work and perform well on any platform is required. It is also not clear that any common platform exists among all mobile devices. However, we claim that it is a reasonable conjecture that a JVM, Java Virtual Machine, will be available on all mobile devices in the foreseeable future. Thus the Java programming language should be used to design and provide the shared data infrastructure.

Given the choice of using memory servers as described in section 3.2, it is necessary to design memory services on remote servers and allow for the communication of both data and commands between servers and mobile devices. Since the Internet is a common communication medium available across all mobile devices and server machines, the shared data system should perform this communication over the Internet.

3.4 Data Sharing Granularity

Hardware systems for data sharing have fixed granularities of sharing, which are often the size of words, cache blocks or pages. Since a software solution is being explored, the granularity of sharing can be varied. If sharing is provided at the granularity of user defined objects, programmers for such systems are given complete control over this granularity. Objects can be defined to be as big as 1 word or large numbers of words, each with different types, including user defined types. Such a system has the benefits of ease of use and higher performance by allowing programmers to eliminate false sharing and excessive coherence traffic (via the use of carefully sized shared objects). Thus sharing

should be implemented at the granularity of user defined objects.

3.5 Consistency Model

The two main consistency models that are of relevance to the proposed system are sequential consistency (SC) and release consistency (RC). Other forms of relaxed consistency such as total store ordering, processor consistency and all the other variants exist in the uninteresting spectrum between SC and RC. They relax the strong ordering constraints that SC imposes, but not as far as RC does and RC provides the simplest implementation requirements by relaxing all but the most important consistency constraints.

The main lure of RC is higher performance through aggressive compiler optimizations and relaxing constraints on the out-of-order execution of modern microprocessors. However, SC provides the simplest model for programmers to follow. Without aggressive compiler optimizations and speculative execution on processors, a shared object system for mobile devices should not relax the SC model.

3.6 Disconnected Operating Modes

Given that mobile devices experience periodic connectivity to the Internet, it is necessary for a shared memory system to allow disconnected operation. This entails the continuation of sharing of data in the network when a device disconnects and for the disconnected device to continue to use potentially stale data during its disconnection. However, the exact requirements of disconnected operation are unclear. Some applications may prefer to not allow a disconnected device to read stale data. It may be unacceptable to some applications to roll back data that has been modified but not flushed by a disconnected device. However, some applications may demand a stronger level of consistency during disconnection, requiring all sharing of exclusive data to stop until the disconnected owner reconnects. An infrastructure that allows for varying disconnected operating semantics is required.

Given these requirements of a shared object system for mobile devices, an implementation was designed and explored in this study. This implementation does not meet all of the requirements laid out in this section.

4. Implementation

A shared object infrastructure has been designed while attempting to meet as many of the above requirements as possible.

The DSM system that has been designed meets the requirements of being an all software implementation (in Java) with shared data storage on memory servers distributed across the network. Data is replicated locally in caches on the mobile devices at the granularity of user defined objects. A model of sequential consistency is adhered to in the design. However, only a simple model of disconnected operation is currently provided by the infrastructure. The design of this infrastructure has been influenced by the need to satisfy the performance requirements of differing applications.

4.1 Caching

Applications that exhibit patterns of multiple writes to shared data without intervening external coherence to the same data will have better performance on write back caches due to temporal locality. However, applications that exhibit patterns of writes to shared data with intervening reads by other sharers will perform better on write through caches by saving on the coherence traffic associated with invalidating a dirty copy. Furthermore, write through caches allow for ease of implementation of disconnected operating modes because the latest version of data will always be available on the memory servers (if writes during network connectivity loss are not allowed on mobile devices). Thus in the system designed, both forms of caching are allowed by the shared data memory servers and both kinds of caches have been designed and tested.

As with write through / write back caches, some applications will perform better in an update protocol system and others will perform better in an invalidate protocol system. Given the time constraints on the study and the inherent protocol complexity of update protocols, only an invalidate based protocol was explored in this study. However, the implemented infrastructure is general enough to allow for an update protocol via some minor modifications.

A locking mechanism is required by almost every distributed shared memory application to ensure that certain critical sections of the application will not overlap in time across multiple devices. This functionality has been provided via a primitive atomic swap operation in the client cache. In performing a swap on two objects, one of which is a shared object, the cache executes a series of operations atomically. It obtains exclusive access to the shared object, reads its value, writes the value of the other object into the shared object and returns the old value of the shared object while not allowing any intervening invalidate messages to be serviced. A lock acquire and release mechanism has been built on top of this atomic swap function.

4.2 Waba's VM & Ninja's iSpace

Another requirement of the infrastructure is to use a language and communication infrastructure that is common to all devices.

At present, [WABA] provides an unstable and incomplete implementation of the Sun Microsystem's JVM on both Windows CE based devices and 3Com's Palm devices. No other VM is known to exist for such devices. An SDK is also provided to facilitate communication between Java programs running on a Waba VM on a mobile device and Java programs running on standard Sun JVM's on server machines. Thus the Waba VM was used in the implementation of this system.

[NINJA] has released a Java based package called iSpace that provides an infrastructure for deploying services in a network. This infrastructure also contains ActiveProxy which includes an ongoing implementation of Java RMI, Remote Method Invocation, for the Waba VM (the Waba SDK does not implement RMI). This system was employed to facilitate communication in our implementation.

4.3 Consistency Model & Disconnected Operation

As described in section 3.4, sharing granularity is maintained at the object level. The consistency model maintained between accesses to different objects is sequential consistency. Due to the limited time and scope of the project, compiler modifications for improving performance under RC were not explored. Thus without the performance advantages of RC, there is no reason to relax the constraints imposed by SC. The exact guarantees on consistency provided by the system are further defined in section 4.6.

A limited amount of disconnected operation support is provided in the system. When using write through caches, up to date data is always available in the memory servers. When devices are disconnected, they are allowed to continue reading stale data from their caches but are not allowed to write data (because this would require a write through operation which requires a connection). Upon reconnection, all cache lines are invalidated. Thus at the expense of coherency on disconnected devices, limited disconnected operation is allowed in the system.

4.4 System Architecture

[Figure 1] shows the overall architecture of the system. During the design of this system, heavy emphasis was placed on scalability.

One or more mobile devices (which can be any devices that support a JVM) communicate with one memory server running on a server machine through any form of communication that gives these devices access to the Internet. Mobile devices can be scattered across multiple memory servers at start-up for scalability.

The communication between mobile devices and their memory server consists of requests for operations on shared objects and coherence traffic. A device that requests the creation of a shared object results in the memory server allocating this shared object and returning an address for this object. Each memory server maintains a bank of shared objects along with the coherence states of these objects and a directory of sharers. The memory server handles read requests, write requests and coherence traffic as described in section 4.5. Coherence traffic messages destined for mobile devices cannot be sent directly to these mobile devices because the Waba VM that these clients run their applications on is not multi-threaded and thus would not be listening for such messages. Instead, a poll queue is maintained for each device at its home memory server which buffers these coherence messages. These devices are required to check these poll queues on every access to a shared object, unless an atomic swap operation is being performed, as described in section 4.1. Even though this is an expensive requirement, it is merely a temporary solution to the immaturity of the underlying technology.

These memory servers communicate with each other, which in turn communicate with their devices. No memory server communicates with a mobile device that is not in its group. No mobile device communicates with a memory server that it does not belong to. The memory servers handle such communication amongst themselves. They achieve this through the use of the memory service nameserver.

The memory service nameserver binds network locations to names. Upon creation, each memory server is given a name and each mobile device is given its name, the name of its memory server and the location of its memory server. Each memory server registers its name and location with the memory service nameserver and each mobile device registers its name with its memory service.

For each shared object that is created on a memory server, an address is assigned, part of which contains the name of the memory server on which the object resides. If a device requests a copy of this object, it sends its request to its memory server. If its memory server is responsible for this object, it will handle the coherence traffic and send back a copy of the data. If it is not responsible for this object, it will find which memory service is responsible. Using the name of the memory server encoded in the address of the data, the local cache

of memory service locations is accessed. If this memory service is not found, the memory service nameserver is queried. The result is cached at this memory service and then the request is forwarded to the responsible memory service. The result is then forwarded back to the device that requested the transaction.

Since addresses are assigned on creation, application programmers cannot know at compile time how to name a shared object. Thus an application may employ a nameserver of its own, similar to the memory service nameserver, to map application object names to the runtime addresses generated by the memory servers. This nameserver was implemented and used, as described in section 5.

Also shown in the figure is a masquerader. Any program that can use the cache that communicates with a memory server can run on a server machine anywhere in the network, masquerading as an application running on a mobile device. Some applications may use this to continuously feed up to date data to the mobile devices or to perform compute intensive tasks. In the application designed, as described in section 5, a masquerader was used to initially create and set up the application's shared objects.

As shown in the figure, the applications on the mobile devices communicate through a cache to the memory server. The cache handles all the memory server communication and shared object coherence, thus providing a clean abstraction to the applications programmer, void of memory server details. The coherence protocol used between the caches and the memory servers is described in section 4.5.

4.5 Coherence Protocol

[Figure 2] attempts to describe the directory based coherence protocol used by the memory server. Several modifications have been made on top of the standard MSI protocol.

As with most shared memory systems, a choice must be made between implementing a MSI, MESI, MOESI or some other variant of cache coherency. The need for minimizing coherence traffic is especially acute in this design space given the low bandwidth, high latency connection between a device and its memory server. Influenced by the design of the SGI Origin 2000 memory coherence system, a MSI system is explored in this study with extra transitional states to avoid deadlock. As always, some applications may exhibit sharing patterns where a different coherency protocol would outperform the MSI protocol. A thorough analysis of this issue is not attempted in this study.

The protocol has been modified to allow both write through and write back caches to work with the same data on the same memory server. The write through section of the protocol has been optimized to minimize excessive coherence traffic (for example, a sharer of data in the S state is allowed to do a write through to the data without first requesting an upgrade provided it is the first request that is received at the serialization point of the memory server; the memory server sends out the

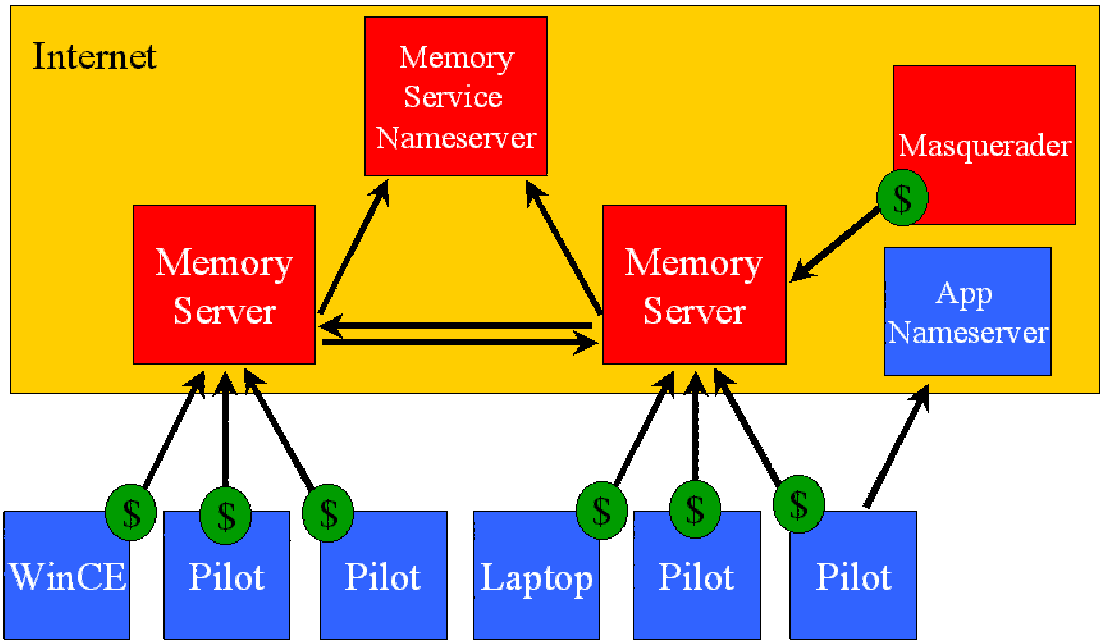

Figure 1: Shared Data Architecture

invalidates to the other sharers).

Two transition states have been added. These have been added to avoid deadlock in the coherence system. When leaving the M state to the S state due to a non-sharer requesting a copy, it is necessary to have the requester retry the request after the invalidate for the dirty copy has been sent out to avoid long dependence chains in the protocol, thus minimizing the chance of a deadlock occurring. The M_T transition state is provided to correctly handle the write back case and W_T is provided to correctly handle a conflicting write through with an invalidate causing a write back.

4.6 Consistency Guarantees

Sequential consistency is guaranteed at the memory server level. Each shared object has a single home which is the memory server which created it. Memory servers are multi-threaded, but array based synchronization has been added to allow accesses to different objects to proceed in parallel but to serialize multiple accesses to the same object, thus providing a single point of serialization. Also, communication is implemented via RMI, remote method invocation, which is blocking on the client side, thus not allowing accesses by one client to be satisfied out of order. Thus sequential consistency is enforced at this level.

However, both the Java compilers and VMs do not make strong guarantees about sequential consistency and thus sequential consistency cannot be guaranteed at the client application level.

5. Experimental Results

Two important goals of this study are to demonstrate the proof of concept that shared memory over mobile devices is feasible and that this infrastructure enables the design and use of collaborative applications. A MUD, multi-user domain, was designed to demonstrate these goals. Additionally, latency measurement experiments were conducted using small hand written micro-benchmarks.

5.1 Multi-User Domain

A MUD is a system that allows each user to control a virtual character that can move around a large arena of rooms, interacting with other characters (some of which may be controlled by a program, not a user) and picking up / dropping objects and using them. Characteristics of the character may alter during the course of the application, such as the objects that are being carried by a character or the strength of the character.

This is an example of a distributed application that users may wish to run on their individual mobile devices, each controlling a character. Shared data objects such as room characteristics and character locations and profiles need to be read and modified by these devices. This is an example of an application that could not have been easily designed without having a shared object infrastructure in place.

The MUD that has been designed consists of the MUD server and multiple MUD clients. The MUD

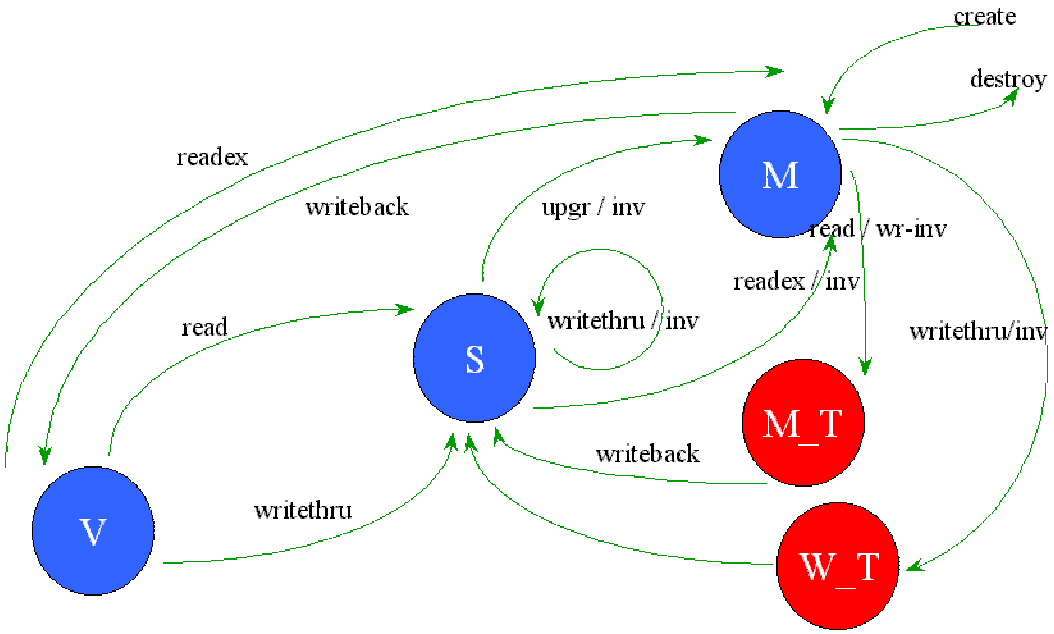

Figure 2: Memory Server Coherence Protocol

server is an application that runs on a masquerader (as shown in [Figure 1]) on a server machine. It initializes the grid of rooms that the characters will live in by creating the shared objects associated with these rooms. Each room has an object that contains a description of the room, an object that lists the characters in the room, an object that lists the virtual objects lying around in the room and an object that represents a lock for the room (in order to serialize modifications to the room). These objects' addresses are registered with an application level nameserver by the MUD server, using names that are known to the MUD clients.

Each MUD client runs on a mobile device and is assigned a user ID. The client requests the MUD server to place its virtual character in a room at initialization. At that point, the user can control the character by moving it from one room to another, looking at the virtual objects in a room, dropping and picking up virtual objects and seeing which other characters are in the room. A linear lock ordering scheme is used to acquire the shared locks associated with each room when a character moves from one room to another. The lock associated with the room with the lower ID is acquired first and then the lock for the room with the higher ID is acquired. The character is transferred from one room's list of characters to the next and the locks are released. This ordering solves the standard dining philosophers deadlock problem.

Sample automated bots have also been designed that control characters by continuously moving them around the arena from room to room.

[Figure 3] shows a screen shot of the MUD client running on a mobile device. The proof of concept of being able to create collaborative applications over mobile devices has been demonstrated. Some timing measurements were also made to identify the performance costs of this initial implementation of the framework.

5.2 Measurements

[Table 1] gives some timing measurement data that was collected on the infrastructure designed. Experiments were run using both the Xcopilot simulator and on real Palm III mobile devices. The measurements from both are presented in the table. Unfortunately, due to the inaccuracy of the timer available on these devices and the Waba VM, the data is accurate only to the nearest 20 ms. Also, the Palm III measurements were taken while the mobile device was connected to a Windows NT machine via a serial port and Windows NT RAS, Remote Access Service. In general use, a wireless form of connection will be used which will not incur the latency overhead of going through Windows NT RAS.

The "ReadMiss" rows indicate the latency seen at a mobile device at the application level in requesting a read of a shared object that is not present in the local cache but is present in valid form at the memory server. This latency includes the latency of having the cache check the device poll queue on its memory server (which is an RMI request from the mobile device to the

Figure 3: MUD Client Screen Shot

XCOPILOT	Min (ms) ± 20ms	Avg (ms) ± 20ms
ReadMiss	200	261
ReadHit	90	129
WriteThru	180	200
WriteBackMiss	190	210
WriteBackHit	110	134

PALM3	Min (ms) ± 20ms	Avg (ms) ± 20ms
ReadMiss	2050	2603
ReadHit	800	887
WriteThru	1600	1762
WriteBackMiss	1500	1812
WriteBackHit	700	972

Table 1: Timing Measurements

iSpace manager running at a remote location in the network and having the iSpace manager spawn a thread which executes the poll queue method on the memory server and sending back the result along the same way). The latency of checking the cache (which results in a miss) and the latency of sending a read request to the memory server (another RMI call, similar to the poll queue RMI call) and forwarding the result back up to the application level is also included.

The "ReadHit" rows indicate the latency cost of having an application perform a read on a shared object that is present in the cache. This not only includes the latency of checking the cache and returning the value, but also the latency of checking the remote poll queue (which is an RMI call as described in the "ReadMiss" case).

The "WriteThru" rows indicate the latency cost of having an application perform a write on a shared object while using a write through cache which contains a copy of the object in shared state. This involves checking the remote poll queue, updating the cache and updating the remote memory server.

The "WriteBackMiss" rows indicate the latency cost of having an application perform a write on a shared object while using a write back cache which contains a copy of the object in shared state. This involves checking the remote poll queue, requesting an upgrade from the memory server and updating the cache.

The "WriteBackHit" rows indicate the latency cost of having an application perform a write on a shared object while using a write back cache which contains a copy of the object in modified state. This involves checking the remote poll queue and updating the cache.

6. Future Work / Design Limitations

There are several limitations of the shared object infrastructure that has been designed in this study which should be improved given more time and better tools. They are outlined in this section.

6.1 Ninja ActiveProxy

As mentioned, the ActiveProxy infrastructure is still under development. There are several limitations of the current system, which, if solved, could drastically improve both the performance and the functionality of the designed system.

Firstly, sockets on the server side of ActiveProxy were not being reused correctly across multiple RMI calls. An extensive amount of time and effort was placed in changing ActiveProxy to destroy and re-create sockets across RMI calls during this study. However, as described in section 6.2, this resulted in other problems at the client side, as well as performance issues.

The RMI system in ActiveProxy only allows strings and integers to be passed as arguments in RMI calls. Thus shared objects can only be strings or integers, not any arbitrary object that can be serialized across a network pipe. Once this limitation is solved, the infrastructure will truly allow the programmer to use any granularity of sharing.

The RMI system does not allow null objects to be passed into or out of RMI calls. Thus for corner cases and error situations, a certain dead value was passed for the data string instead of null, further limiting the values that the shared objects can use.

There is a heavy marshalling overhead of setting up input arguments and output results for RMI calls, which once streamlined will improve the performance of the system.

6.2 Waba VM

The Waba VM is also currently under development. Several sections of the standard Sun Java API are not yet available in the Waba VM, making the explored implementation heavy and cumbersome.

The Waba VM does not support multi-threading. Thus it was not possible to send invalidates directly to clients to remove their copy of shared data. Instead, an intricate system of polling was devised, with poll queues for devices residing on memory servers. These devices have to check these poll queues for invalidate messages each time their local cache is accessed and each time a remote request is made. This heavily limits the performance and scalability of the system. Once a multi-threaded JVM is available and multi-threaded devices are available, invalidate messages can be sent directly to the caches residing on these devices.

A major problem encountered in the design of the system is the lack of debugging tools and details on errors thrown by the VM. Compounded by the lack of print space for debugging purposes on the Xcopilot interface, testing and debugging of client side applications boiled down to careful inspection of code rather than run time identification of faulty code. Once this aspect of the VM is adequately addressed, client side application development can proceed at a faster pace.

The API also does not provide adequate functionality to allow user inputs in client applications. Thus only buttons could be used for user applications and run time configuration parameters had to be hard coded into the application and re-compiled for each device.

A fundamental problem with the Waba VM is that the garbage collection system is not functioning correctly. Due to the socket reuse issue described in section

6.1, new sockets are created for each new RMI call, and not all of them are garbage collected by the VM at the termination of RMI calls and eventually the VM will run out of memory. This severely limits the size and running time of an application.

6.3 Xcopilot

The devices simulated in the design of the shared object system were Palm devices [PALM] and they were simulated via the use of the Xcopilot simulator [XCOPILOT]. The main limitation of using this simulator is that it is hard to simulate large number of devices because one simulator needs to be run for each device and using an automated script to coordinate all the devices is non trivial. A built in system for achieving this task would facilitate the testing and scalability measurements of the system across a large number of devices.

6.4 Disconnected Operation / Fault Tolerance

The current system supports disconnected operation in a minimal manner if write through caches are used. When disconnected, the disconnected device will be allowed to read stale cached data but will not be allowed to write data. However, the issue of application level locks is not addressed. The application halts if a device acquires a lock and then disconnects or fails because other devices that are connected cannot acquire the lock. A system of lock leasing coupled with transactional updates and rollbacks on lock boundaries will address this issue. This option was not explored in this study due to the limited time available.

7. Conclusions

In this study, the need for a shared memory system over mobile devices has been justified by the need for collaborative applications over such devices. The ideal characteristics of such an infrastructure have been identified as being an all software solution built around a cross platform language such as Java, using the Internet as the communication infrastructure, relying on persistent storage on memory servers in the Internet, allowing the granularity of sharing to be user defined objects and providing support for disconnected operations.

A shared memory infrastructure has been built for mobile devices that meets these requirements and the latency costs of using the designed infrastructure have been experimentally found and reported. A distributed collaborative application has been built on top of this infrastructure to demonstrate the scope of applications that can now be designed and used. The limitations of the current implementation have been discussed and ways of alleviating these shortcomings have been proposed.

8. Acknowledgments

We would like to thank the mentorship of Professor Dave Culler. We would like to thank the Ninja group for their ActiveProxy infrastructure and the use of their Linux machine for development and testing.

References

[A96] Sarita Adve et al. ``A Comparison of Entry Consistency and Lazy Release Consistency Implementations", 1996 HPCA

[B98] Henri Bal et al. ``Performance Evaluation of the Orca Shared-Object System", Feb 1998 TOCS

[C94] Alan Cox et al. ``Software Versus Hardware Shared-Memory Implementation: A Case Study", 1994 ISCA

[C99]Alan Cox et al. ``A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory", 1999 HPCA

[CBZ95] John Carter, John Bernnett, Willy Zwaenepoel ``Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems", Aug 1995 TOCS

[CDK94] Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 1994 Addison Wesley, 2nd edition

[D] Alan Demers et al. ``The Bayou Architecture: Support for Data Sharing among Mobile Users", Computer Science Laboratory, Xerox Palo Alto Research Center

[D93] Sandhya Dwarkadas et al. ``Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology", 1993 ISCA

[FB] Armando Fox and Eric A. Brewer, ``Harvest, Yield, and Scalable Tolerant Systems", University of California, Berkeley

[GWS96] A. Grimshaw, J. Weissman, W. Strayer ``Portable Run-Time Support for Dynamic Object-Oriented

Parallel Processing", May 1996 TOCS

[HAB96] S. Hassen, I. Athanasiu, H. Bal ``A Flexible Operation Execution Model for Shared Distributed Objects", 1996 OOPSLA

[JKW95] Kirk Johnson, M. Kaashoek, D. Wallach ``CRL: High-Performance All-Software Distributed Shared Memory", 1995 SIGOPS

[K94] Pete Keleher et al. ``TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Jan 1994 Usenix Conference

[K95] Pete Keleher ``An Evaluation of Software-Based Release Consistent Protocols", Oct 1995 Journal of Parallel and Distributed Computing, Special Issue on Distributed Shared Memory

[KCZ92] Pete Keleher, Alan Cox, Willy Zwaenepoel ``Lazy Release Consistency for Software Distributed Shared Memory", 1992 ISCA

[KS] J. Kistler, M. Satyanarayanan ``Distributed Operation in the Coda File System", TOCS

[L95] Honghui Lu ``Message Passing Versus Distributed Shared Memory on Networks of Workstations", 1995 SC

[L96] S. Lucco ``Parallel Programming in a Virtual Object Space", May 1996 ACM TOCS

[LH89] Kai Li and Paul Hudak, ``Memory Coherence in Shared Virtual Memory Systems", Nov 1989 TOCS

[M90] S. Mullender, Distributed Systems, 1990 ACM Process

[NINJA] Ninja Project, UC Berkeley, http://ninja.cs.berkeley.edu/

[PALM] Palm Devices, 3Com Corporation, http://palm.3com.com

[P96]K. Petersen et al. ``Bayou: Replicated Database Services for World-wide Applications", 1996 EuroSIGOPS

[P97]K. Petersen et al. ``Flexible Update Propagation for Weakly Consistent Replication", 1997 SOSP

[SB] W. Speight, J. Bennett ``Reducing Coherence-Related Communication in Software Distributed Shared Memory Systems"

[T] Marvin Theimer et al. ``Dealing with Tentative Data Values in Disconnected Work Groups", Computer Science Laboratory, Xerox Palo Alto Research Center

[T95] Douglas B. Terry et al. ``Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", 1995 SOSP

[TSPACES] TSpaces, IBM, http://www.almaden.ibm.com/cs/TSpaces/

[WABA] Wabasoft's VM for PDAs, http://www.wabasoft.com

[XCOPILOT] Xcopilot Palm simulator, http://xcopilot.cuspy.com/