Experience Using a Globally Shared State Abstraction to Support Distributed Applications *

Sai Susarla *Anand Ranganathan* *John Carter*

UU-CS-TR-98-016

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

In this paper, we evaluate the effectiveness of basing distributed systems on a persistent globally shared address space abstraction, as implemented by Khazana*. Khazana provides shared state management services to distributed application developers, including consistent caching, automated replication and migration of data, location management, access control, and (limited) fault tolerance. We report on our experience porting three applications to Khazana: a distributed file system (KFS), a distributed name service (KNS), and a cooperative drawing program based on* `xfig` *(KFIG). The basic Khazana abstraction of persistent shared memory made it easy to create fairly efficient distributed services with flat data abstractions (KFS and KNS), but our port of* `xfig` *made it clear that a higher level abstraction is preferable for applications with more structured state. As a result, we extended Khazana to support a limited set of object-like functions (reference swizzling, event upcalls, and update propagation). We present herein the current Khazana design and implementation, and discuss the lessons learned from our initial evaluation of it.*

1 Introduction

Distributed systems involve complicated applications with complex interactions between disparate components. The environment in which these applications operate introduces additional challenges in terms of fault tolerance and security. To ease the chore of building distributed applications, researchers have developed a wide variety of systems. The earliest distributed systems provided support for interprocess communication, via message passing [10, 28] or remote procedure calls [3], but provided little support for transparent distribution of data and execution or for fault tolerance. Systems that provide such support have tended to provide one of three basic abstractions: *distributed files* [5, 12, 22, 27, 29], *distributed objects* [13, 17, 18, 26] or *distributed shared memory* (DSM) [1, 7, 21, 23]. Each abstraction works well for certain classes of applications, but none is dominant, because none is a good fit for all classes of applications.

We have built a prototype of *Khazana*, a distributed system that uses a mix of technologies from each of the above types of systems to support the development of distributed applications and services. Khazana was originally envisioned as a DSM system that supported *distributed* applications, rather than the shared memory *parallel* applications supported by conventional DSM systems. This design worked well for a pair of applications that employed fairly "flat" data structures (a clustered file server and a distributed name server). However, when we ported a uniprocessor version of `xfig` to Khazana to create a cooperative drawing program, we found that adding a limited set of functions like those found in distributed object systems significantly eased development. The focus of this paper is on the design and performance of Khazana and the lessons we learned from the initial prototyping effort that led to the current mixed-mode shared-memory/shared-object abstraction.

At its core, Khazana is an extension of a conventional DSM system that supports distributed applications by letting them create and share *persistent* regions of shared state. Khazana is designed to handle the common problems associated with sharing state between applications, including replicating and caching data, keeping copies of the data coherent, tracking the location(s) of data, avoiding loss of data due to node or network failures, managing distributed system resources, and enforcing security restrictions on data access. At this core level, Khazana supports a persistent

*This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Email: {sai,anand,retrac}@cs.utah.edu
Khazana URL: `http://www.cs.utah.edu/projects/khazana`

DSM-style programming interface that allows different applications to communicate in much the same way that different threads of a shared memory program communicate. When an application wishes to access a piece of shared state, it performs a `kh_lock()` operation, specifying a range of global addresses to be locked and a lock mode (e.g., read, write, or read-with-intent-to-write). The application can then explicitly `read` or `write` the data, and when it is done accessing the data, it unlocks it. The Khazana core does not interpret the data – some clients may store structured data (e.g., CORBA objects) while others may simply use Khazana for distribution of raw data (e.g., files).

A significant difference between Khazana and conventional DSM systems is that Khazana supports a 128-bit address space that does not directly correspond to machine virtual addresses. This is necessary to support the amount of data that can exist in a large distributed system. When global addresses do not directly correspond to process virtual addresses, applications can not longer add or follow references to global shared data by writing/following pointers.

For two of the applications that we ported to Khazana, the mismatch between global addresses and process virtual addresses was not a serious problem. These applications (the clustered file server and distributed name server) naturally built their data structures using opaque references. However `xfig`'s primary data structure consists of a pointer rich tree of graphical objects, and manually converting these pointers to Khazana addresses (and vice versa) proved to be quite tedious. The resulting distributed version of `xfig` worked quite well, but it was clear that Khazana needed to be enhanced if we want to easily port (as opposed to write from scratch) applications of this type.

Based this experience, we extended Khazana to provide a limited set of object-like functions. We call this extension the Khazana Object LAyer or *KOLA*. The term "object" is somewhat of a misnomer, since the objects managed by KOLA lack many of the properties normally associated with objects in a distributed object system, but it is not entirely inappropriate. KOLA provides support for three basic object-style operations that allow programmers to build applications with highly structured data and still exploit Khazana's core functionality. These operations are *reference swizzling*, *event notification*, and *update propagation*. As part of adding the KOLA support, we extended the original Khazana API [9] to allow clients to register interest in certain Khazana-internal events via *upcalls* and to allow clients to trigger events (upcalls) on remote nodes caching an "object."

As a result, Khazana currently exports both a flat data (DSM) programming model and a simple distributed object model. For applications where the overhead and complexity of a distributed object system is inappropriate, such as a distributed file system or name service, programmers can interact with Khazana's core storage layer directly. When a somewhat higher level programming abstraction, e.g., automatic reference swizzling and a mix of function and data shipping communication styles, is appropriate, programmers can exploit KOLA in addition to the core Khazana functionality. As such, Khazana is similar to the PerDis [15] system, which aims to provide large-scale, efficient, secure data sharing by means of a persistent distributed object system. It is interesting to note that although Khazana originated as a persistent distributed shared memory system and Perdis as a persistent distributed object system, they have both evolved towards providing a combination of both programming styles. Thus, it appears to make sense to separate the basic persistent shared data management layer from the distributed object layer, and provide clients access to both layers.

In this paper, we present the design of Khazana (Section 2) and its implementation (Section 3). The primary focus of the paper is a description of our experience porting three test applications to Khazana (a distributed file server, a name server, and a groupware-like version of `xfig`) and the lessons that we derived from this effort (Section 4). We then compare Khazana to previous and contemporary systems (Section 5) and draw conclusions (Section 6).

2 Design of Khazana

In this section, we describe the high-level structure of Khazana (Section 2.1), the Khazana core operations (Section 2.2), and the Khazana Object LAyer (Section 2.3). We conclude with an example of how client programs use Khazana to manage their shared state (Section 2.4). Details on how this design is implemented appear in Section 3.

2.1 Structure of a Khazana System

The basic abstraction Khazana exports is that of a persistent globally shared storage space addressed by 128-bit global addresses. There is no direct correspondence between Khazana addresses and an application's virtual addresses. The basic storage abstraction is that of a *region*, which is simply a contiguous range of global addresses managed using common mechanisms (e.g., coherence protocol, minimum storage granularity, security key, and replication policy). To Khazana, regions are simply a range of bytes – it is up to higher level software (e.g., KOLA or client code) to interpret the contents of a region. In effect, the base Khazana layer can be thought of as a globally accessible disk against which distributed applications read and write data, similar to Petal[22]. As will be described below, however, Khazana provides a finer level of control over how individual regions are managed than Petal, and Khazana's DSM-like

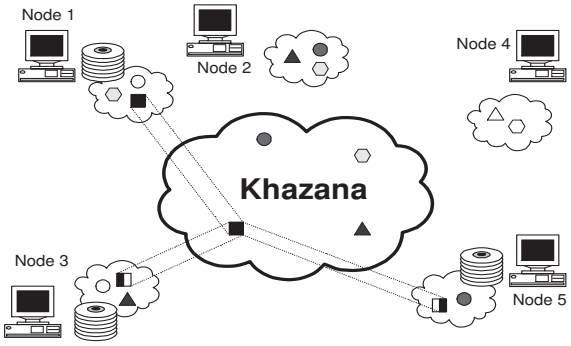

Figure 1: Typical Khazana-Based Distributed System

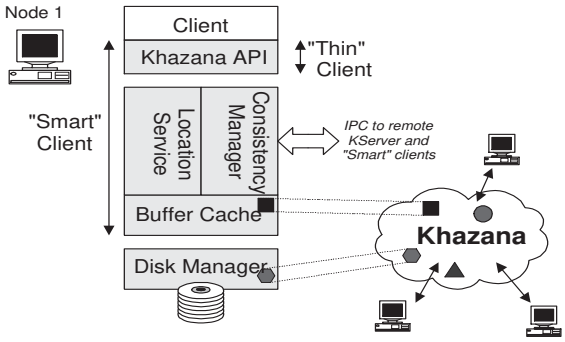

Figure 2: Typical Khazana-Based Distributed System

naming scheme allows applications to embed references to other structures in their data.

Figure 1 presents a high-level view of a 5-node Khazana-based distributed system. The cloud in the center of the picture represents the globally shared storage abstraction exported by Khazana. Each object (e.g., the square) represents a single Khazana region. Khazana's persistent storage is an amalgam of disk space spread throughout the system. Nodes can dynamically enter and leave Khazana and contribute/reclaim local resources (e.g., RAM or disk space) to/from Khazana. In this example, nodes 1, 3, and 5 are providing the disk space for storing persistent instances of Khazana regions; nodes 2 and 4 can access regions and cache them in local DRAM, but not store them on the local disk. In a sense, Khazana's persistent storage is similar to that of a clustered file server [19], although it is not as tightly integrated as a typical cluster nor is it as tuned for high performance.

Khazana's global shared storage abstraction is implemented by a collection of cooperating daemon processes (KServers) and client libraries. KServers run on some (not necessarily all) machines of a potentially wide-area network. Each node running a KServer is expected to contribute some portion of its local disk space to the global storage. Note that although we use the term "server", these daemon processes are in fact peers that cooperate to provide the illusion of a unified resource. Applications (clients) interact with Khazana through library routines. The basic set of operations exported by Khazana allow clients to allocate, access, and delete regions of data.

The shaded portions of Figure 2 represents Khazana's major functional components: the Khazana API, the location service, the consistency management service, the RAM buffer cache, and the disk buffer cache. These functions are divided between client libraries and KServers in a number of ways, as follows.

We currently provide two different client libraries, referred to as the *thin client* and the *smart client*. The thin client simply bundles up client requests and forwards them to a nearby KServer – it is in essence just an RPC client stub. The smart client, in contrast, implements most Khazana functions and protocols. In particular, it aggressively caches Khazana data and locks associated with this data locally in its internal buffer cache and consistency manager. To keep this data consistent with copies stored elsewhere in the system, the smart client exchanges consistency protocol messages with the KServers and other smart clients. In many ways, the smart client is a peer to the KServers for regions its client is accessing. Smart clients do not manage a disk cache, because they only execute as long as their client application does. Also, for security purposes, smart clients are not given access to Khazana-internal metadata, so they must contact a KServer to acquire the initial copy of a region's data and the lock token(s) associated with a given region. As will be discussed in Section 4, the ability to cache region data and locks within the application's address space significantly improves performance compared to the thin client.

KServers themselves implement all of the major Khazana functions. They can be invoked by either thin client RPCs, which arrive at the Khazana API layer, or by protocol messages sent from other KServers or smart clients. KServers also manage both volatile (DRAM) and persistent (disk) region caches.

Khazana is free to distribute region state across the network in any way it sees fit, subject to resource limitations, perceived demand, and the specified replication and consistency policies for the object. Portions of regions can be stored on different nodes. The minimum unit of storage is specified on a per-region basis and is referred to as a *page*. The default page size is 4-kilobytes, which matches the most common machine virtual memory page size. Consider the region represented by the square in Figure 1, which has a client-specified minimum of two persistent replicas. In this example, one copy of the entire region is stored on Node 1, while the second replica is divided between Nodes 3 and 5. Presumably Khazana chose these nodes because they access the region most frequently, are the most stable, or have

the most available resources. A goal of our work is to develop caching policies that address the needs for load balancing, high performance, high availability, and constrained resources.

2.2 Khazana Core Operations

The Khazana storage layer exports a set of simple programming APIs via which applications can operate on the shared storage space. Details of these APIs can be found elsewhere [9], but the basic operations can be summarized as follows:

`kh_reserve()`/`kh_unreserve()`: These operations reserve (unreserve) a *region* of Khazana's 128-bit address space, without allocating physical storage for it.

`kh_allocate()`/`kh_free()`: These operations allocate (free) physical storage for the specified region, or portion thereof. A region cannot be accessed until physical storage is allocated for it.

`kh_lock()`/`kh_unlock()`: Once storage for a region has been allocated, an application gains access to its contents by locking it. Applications can specify a number of locking modes – including read, write, and read-with-intent-to-overwrite. Khazana enforces particular lock modes in different ways depending on the consistency protocol used to manage the region.

`kh_read()`/`kh_write()`: Once an application has locked a region, it can access its contents via explicit read and write operations. Our design calls for clients to be able to "map" parts of global memory to their virtual memory space and read and write to this mapped section (akin to a memory-mapped file or conventional DSM system), but this functionality has not yet been implemented.

`kh_getattr()`/`kh_setattr()`: Applications can query and modify each region's attributes. Among the attributes currently supported are the number of persistent replicas of the region maintained by Khazana, the coherence protocol used by Khazana to keep the region consistent, and a security key used to control access to the region.

The first two sets of functions give applications the ability to allocate large contiguous pieces of the shared address space from which they can later allocate subregions of storage. For example, a distributed file system might `kh_reserve()` enough address space to contain an entire file system, but only `kh_allocate()` the disk space to back files as the file system fills. We found that the separation of address space allocation and storage allocation made it easier for applications and library routines to manage their own storage at a fine grain.

Prior to accessing the contents of a region, a client must lock the region in an appropriate mode – obtaining a read lock prior to reading it and a write lock prior to modifying it. The precise semantics of the various lock modes is entirely dependent on the particular coherence protocol being used to manage the region. The consistency manager is free to interpret "read lock" and "write lock" events as it sees fit. The default coherence protocol enforces conventional mutual exclusion semantics, but alternate coherence protocols can choose to use the lock events as hints about how and when to update data.

Client applications can specify a number of per-region attributes, either as part of the `kh_reserve()` operation that created the region or via the `kh_setattr()` routine. The most important user-visible attributes currently supported by Khazana are the *consistency protocol*, an *access control key*, and the *minimum number of replicas* that should be maintained.

2.3 The Khazana Object LAyer (KOLA)

The Khazana Object LAyer (KOLA) is an attempt to simplify the use of Khazana for building object-style distributed applications. Its design evolved as we gained experience converting the `xfig` drawing program into a groupware application (KFIG) built on Khazana. Porting `xfig` directly to the flat global storage abstraction exported by the Khazana core services was unsatisfying in a number ways.

First, applications with complex data structures need to be able to embed references within their shared data. Khazana's address space is much larger than a single machines virtual address space, so references to Khazana data cannot be directly mapped to virtual addresses. Without some level of runtime support, applications will need to hand swizzle Khazana addresses into local virtual memory addresses, and vice versa, by hand - a tedious chore that can be automated with a little runtime support.

Second, a purely pull-based coherence model, where changes to data are only detected when a client re-locks the data, is a poor match for interactive groupware-style applications that wish to be notified when the shared data they care about changes. In a shared memory system, this is equivalent to needing *signals* in addition to locks. In general, such applications can benefit from some form of signaling or push-based coherence.

Finally, applications like `xfig` tend to have a large number of objects of varying sizes, many of which are fine-grained. As such, a 4-kilobyte page may be able to hold many individual objects. Implementing fine-grained variable-sized objects on a page-based system like Khazana without adequate support for sharing can lead to inefficient

space utilization and false sharing, which can result in poor performance.

To alleviate these problems, we added support for the following operations:

- **Basic reference swizzling**: We have written a simple client library that can swizzle Khazana references to virtual addresses local to the client, and vice versa, on demand. The current support is quite rudimentary, in that it is not integrated into a language/compiler system, and thus the location of references within "objects" must be specified to the library routine as opposed to inferred from the object's type.

- **Event notification**: Khazana allows applications to register interest in Khazana-internal events such as updates to specific regions (or portions thereof), changes to region attributes such as the security token, Khazana node membership, and changes to the set of nodes caching a particular region copy location. In KFIG, for example, each instance of the KFIG application is notified when another node modifies the drawing, so it can redraw the screen without having to poll, an expensive operation across a network.

- **Flexible update propagation**: Khazana clients can invoke arbitrary "update" operations to "objects" cached on remote nodes. This functionality piggybacks on Khazana's consistency management protocols and and the basic event notification mechanism. It is useful when the object is itself not mobile (e.g., you wish to send a command to a fixed sensor) or when it is more efficient to update all instances of an object at once than to force them to all reload the data. In addition, KOLA's update mechanism is sufficient general that it can be used to perform a form of *remote object invocation*.

Note that KOLA does *not* provide certain services that might be expected from a full blown distributed object system. In particular, it provides no level of automatic garbage collection nor any form of automated data format conversion to support heterogeneous execution environments. Also, the operations described above are not automated, as is often the case in an integrated distributed object system,

2.4 Using Khazana

Client programs can access shared data stored within Khazana without needing to know where the desired data is physically stored at any given instant. As such, programming on top of Khazana is similar to programming a multithreaded uniprocessor application or programming on top of a DSM system. Data structures that an applications wishes to share with other applications or other instances of itself

```
addr_t put_in_global(buf, sz, key)
    char *buf;
    size_t sz;
    key_t key;
{
    addr_t addr;
    lock_context_t lock;

    kh_reserve(&addr, sz, key, WT_INV);
    kh_allocate(addr, sz);
    kh_lock(addr, sz, WRITE, &lock, key);
    kh_write(addr, lock, buf);
    kh_unlock(lock);

    return(addr);
}
```

Figure 3: Simple Khazana Programming Example

running on other nodes are allocated and stored in Khazana regions. A clustered application, such as the file system described in Section 4.2, can start multiple instances of itself, each of which independently can access and modify the same application "objects" by mapping, locking, accessing, and unlocking the objects' constituent Khazana regions. Depending on the consistency protocol selected for the region(s), Khazana can enforce strict mutual exclusion-style locking semantics, such as is required by KFS (Section 4.2), or a loosely coherent update protocol, such as is appropriate for a groupware application like KFIG (Section 4.4).

Figure 3 presents a bare bones example of using the basic Khazana operations. In this example, the application wishes to store a buffer into Khazana space, and protect it by a particular secret key. It first must reserve enough address space and allocate enough physical storage to hold the data. As part of the reserve operation, it specifies the secret key that must be used to access the region in the future, as well as indicating that the region should be managed using the write invalidate protocol. The application then acquires exclusive access to the region, initializes its contents, and signals completion of the write operation by unlocking the region. In this case, locking the data is superfluous, since no other node knows of the region's existence and thus there can be no data races on access to it. However, it is also true that the lock will be held in the application's address space (in the smart client), so the overhead of executing these superfluous operations is small.

Once data is placed in Khazana space, any number of clients can access it. They only need to knows its address and access key. They coordinate access to the data using fairly conventional locking mechanisms, which enforce the specified level of concurrency control. In the case of the write-invalidate protocol, locking is strict, meaning that ei-

ther one node can hold a write lock or an arbitrary number of nodes can hold a read lock. Clients need not concern themselves with how many other nodes are accessing the region (or portion of a region) nor where the data is physically stored or cached. As long as they obey the locking semantics associated with their selected consistency protocol, they are guaranteed correct behavior. It is up to Khazana's caching policies to provide efficient performance, for example by automatically replicating or migrating data to nodes where it is being accessed frequently or that historically respond to requests most quickly.

3 Implementation of Khazana

In this section, we describe the implementation of Khazana and KOLA – a more detailed description of the Khazana core's implementation can be found elsewhere [9].

Khazana is designed to scale to a WAN environment, with nodes within a single LAN forming a *cluster*. Each such cluster designates a single node as the *cluster manager*, which maintains hints about what regions are cached in the local cluster, a set of free address ranges from which local nodes can allocate storage, and other such non-critical information. The cluster manager is selected using a voting scheme based on process ids. If the cluster manager fails, the remaining nodes vote on a replacement. The cluster manager collects hints on locally cached regions via messages sent to it asynchronously by cluster members when they create or lock regions.

Managing the Global Address Space: Khazana maintains two primary data structures: a globally distributed *address map* that maintains global information about ranges of Khazana addresses and a collection of per-region *region descriptors* that store each region's attributes (i.e., security attributes, page size, and desired consistency protocol). A particularly important attribute associated with each region is its *home node*. A region's home node is responsible for maintaining a current copy of its region descriptor and tracking the set of nodes maintaining copies of the region's data. The address map also contains information regarding what regions of address space are reserved, allocated, and free.

Khazana daemon processes maintain a pool of locally reserved, but unused, address space, which they obtain in large chunks (multiple gigabytes) from their local cluster manager. Whenever a client request uses up the locally reserved address pool, the local KServer pre-reserves another large region to be locally subdivided. Once space is located to satisfy the reserve request, reserving a region amounts to modifying address map tree nodes so that they reflect that the region is allocated and where. Unreserving a region involves reclaiming any storage allocated for that region. For simplicity, we do not defragment (i.e., coalesce adjacent free) ranges of global address space managed by different Khazana nodes. We do not expect this to cause address space fragmentation problems, as we have a huge (128-bit) address space at our disposal and do reclaim storage. (just not address space).

Tracking Data: To initiate most operations, Khazana must obtain a copy of the region descriptor for the region containing the requested range of addresses. There are three methods by which Khazana can locate the region descriptor, each of which it tries in turn: (i) by examining a node-local cache of recently used region descriptors, (ii) by querying the local *cluster manager*, and, when all else fails, (iii) by performing a distributed tree walk of the address map data structure.

To avoid expensive remote lookups, Khazana maintains a cache of recently used region descriptors called the *region directory*. The region directory is *not* kept globally consistent, but since regions do not migrate home nodes often, the cached value is usually accurate. If this cached home node information is out of date, which will be detected when the queried node rejects the request, or if there is no local cache entry, Khazana queries the local cluster manager.

If neither the region directory nor the cluster manager contain an up to date region descriptor, Khazana resorts to searching the address map tree, starting at the root tree node and recursively loading pages in the tree until it locates the up to date region descriptor. The address map is implemented as a distributed tree itself stored in Khazana, where each subtree describes a range of global address space in finer detail. If the region descriptor cannot be located, the region is deemed inaccessible and the operation fails back to the client. This distributed tree walk is expensive, and thus avoided whenever possible.

Consistency Management: Khazana maintains consistency via program modules called Consistency Managers (CMs) running in the KServers and smart clients. The CMs cooperate to implement the required level of consistency among the replicas using Brun-Cottan-style decomposable consistency management [4]. CMs are somewhat independent from the rest of a KServer or smart client – their only role is to determine *when* locks can be granted and *when* data needs to be updated or invalidated. They do not perform the actual data or lock transfers – that is left to the Khazana communication services. By decomposing the entity that determines when consistency operations must occur (the CM) from the rest of the system, Brun-Cottan argues that you can more easily build protocols that exploit information specific to a given application or class of applications.

A lock request represents a request for permission to perform a certain operation on a region (or portion thereof). The application encapsulates all semantics of the requested operation affecting consistency in an application-defined object called an *intent* (following Brun Cottan's terminology [4]). When a CM receives a lock request and intent,

it checks to see if the request conflicts with ongoing operations given the semantics of the particular coherence protocol. An application can define a set of functions that the CM can invoke to make decisions affecting consistency, such as whether two operations conflict or whether two operations can be applied to different copies in different order. If necessary, the CM delays granting the lock request until the conflict is resolved.

Once a lock is granted, Khazana performs the subsequent permitted operations (e.g., reads and writes) on the local replica itself, notifying the CM of any changes. The CM then performs consistency-protocol-specific communication with CMs at other replica sites to inform them of the changes. Eventually, the other CMs notify their Khazana daemon of the change, causing it to update its replica.

Khazana provides an efficient default implementation of intents that helps provide traditional memory read-write sharing, with sub-range locking ability within a region page. This implementation proved useful in avoiding false sharing in case of the cooperative `xfig` application where the majority of the objects were fine-grained. Khazana's consistency management mechanism is highly flexible in that it enables application to tune their consistency management protocols based on application characteristics It also paves the way for building reusable protocol components, as they are independent of the objects whose consistency they are maintaining.

In Khazana's consistency management framework, an application that obtained a lock can influence how Khazana propagates updates to remote page copies. For this, Khazana provides an interface operation called `kh_update` by which an application can supply an uninterpreted byte stream called *update packet* to the CM. The CM can keep track of these packets, and send them at appropriate times to peer CMs on other nodes. Those CMs eventually hand over these packets to application instances running there, via a special Khazana event notification interface.

Event Notification : Khazana provides a simple interface for applications to register to be notified of various Khazana internal events. These events can be used by applications to make distribution-aware optimizations for efficiency and performance. To register, an application supplies a callback routine which is called on occurrence of the specified event.

One very useful event is an "object" update at a node. By registering for a certain kind of object update by other copies, the application instance at a copy site can ensure that it has the data available locally when it needs it, and also avoid expensive polling. This is especially useful for interactive groupware applications where real-time updates are desirable.

The `kh_update` mechanism can also be used to invoke an operation on remote copies of an object. To accomplish this form of "remote object invocation," application instances caching an object register for "updates" to their object. The application instance on the sending side of the RPC marshals parameters into a packet and hands it over to Khazana CM via the `kh_update` interface. To the CM, it looks like a regular object update, which it propagates to the receiving node, which hands it over to the instance of the application running there.

Storage Management: Node-local storage in the form of both DRAM and secondary storage is treated as a cache of global data indexed by global addresses. Each node's local storage subsystem maintains a *page directory*, indexed by global addresses, that contains information about global pages cached locally, including their local location (DRAM page or file location) and a list of other nodes caching the page. Like the region directory, the page directory is node-specific, not stored in global shared memory, and the list of nodes caching a page is only a hint. The local storage system provides raw storage for pages without knowledge of global memory region boundaries or their semantics. There may be different kinds of local storage (e.g., main memory, disk, and tape) organized into a storage hierarchy based on access speed, as in xFS[2]. In response to data access requests, the local storage system simply loads or stores the requested data from or to its local store (either RAM or disk).

Fault Tolerance: Khazana handles partial system failures by (optionally) replicating regions of data. In particular, address map pages are always replicated on at least two distinct nodes. The use of a local region directory cache and a cluster-level directory cache make Khazana less sensitive to the loss of address map nodes, but if a tree search is unavoidable, a region's availability depends on the availability of the address map tree nodes in the path of the tree search. Should all copies of an address map tree node become unavailable for an extended period of time, the address map can be recreated via a (slow) global recovery algorithm in which each node forwards a summary of the contents of its local storage system to a designated recovery node, which uses this information to rebuild the global address map. Similarly, Khazana allows clients to specify a minimum number of primary replicas that should be maintained for each page in a Khazana region, which allows them to trade off availability for performance and resource consumption.

To make it easier to recover locks when nodes fail or network partitions occur, we are modifying Khazana to implement locks as *leases*. Currently clients can hold locks indefinitely, so we cannot automatically recover locks lost due to node failures unless we are sure that the problem is not a temporary network partition. Using leases to implement locks would simplify lock token recovery immensely – you simply refuse to reissue the lock token until the lease expires, at which time you can reissue it safely. We will need

to experiment with various default lease times to tradeoff failure-free performance, when long leases are preferable, against fault recovery time, when short leases are preferable.

Security: Since we envision Khazana being used to store sensitive system state, it provides a simple access control mechanism on which higher-level software can enforce a variety of access control mechanisms. Khazana's built-in access control mechanism is based on secret keys — when a client creates a region, it can specify a 128-bit key that must be presented as part of any future `kh_lock()` operation on the region. Khazana provides no key management support – we assume that services manage their own keys. These key distribution services can, of course, themselves be built on top of Khazana. KServers never exports keys to clients, so a key cannot be compromised by insecure applications.

In a secure environment, we assume that authentication and encryption mechanisms will be available. Using such services, KServers will be able to authenticate themselves to one another, thereby ensuring that they are not exporting Khazana metadata to clients masquerading as a KServer. In addition, all inter-node communication between KServers and clients could be encrypted to make physical packet sniffing ineffective. The current Khazana prototype enforces the basic access control mechanism, but is not integrated with an authentication service nor does it encrypt its communications.

Client Libraries: After our initial experience with the thin proxy client, we soon realized that offloading functionality from the Kservers to client processes would significantly improve performance. Our smart client was motivated by our belief that data access operations should be far more common that storage/address management operations (e.g., `kh_reserve`). Thus, applications can benefit immensely by caching Khazana region pages in their address space and cooperating to manage their consistency.

The smart client library provides `fetch-release` style access to pages of data. Unlike the thin client, which simply acts as an RPC stub to invoke Khazana operations on remote Kserver nodes, the smart client contains the entire Khazana consistency management subsystem. This enables it to cache both region pages as well as frequently held locks on locally accessed pages. For security reasons, smart clients still must contact a KServer to perform address space range lookups, examine or modify attributes of reserved regions, reserve/free address space, or allocate/free storage.

The smart client interface has been carefully defined to isolate buffer management from Khazana consistency management functionality. Thus the smart client can be used both to manage consistency of application-level buffers or to provide a default buffer cache implementation that can physically share pages with a Khazana server running on the local machine. One motivation for this was the observation that some applications (e.g., file systems and databases) prefer to handle their own buffer cache management issues like cache size and replacement policy.

The smart client library's interface consists of the following operations:

`kh_buf_state_init`: Initializes Khazana-specific state of a buffer.

`kh_bcache_fetch`: Fetches given Khazana region page with specified locking mode (intent).

`kh_bcache_update`: Supplies an update packet to the CM. Usually this is to inform the CM that the page was modified locally, but as described above, it can also be used to perform a form of remote object invocation (via `kh_update`).

`kh_bcache_release`: Releases the lock on the region page and makes the update packets specified with the update call available for propagation to other nodes.

`kh_buf_state_uninit`: Cleans up Khazana-specific state of a buffer.

4 Evaluation

In this section, we describe a series of experiments we have performed on Khazana to determine how well it performs and how effective it is at making distributed application development (or porting) easy. In Section 4.1, we report the performance of fundamental Khazana operations (e.g., the time to lock and access regions of data under varying circumstances). We also present the results of three whole application experiments: the Khazana file system (Section 4.2), a Khazana-based name service (Section 4.3), and a port of the `xfig` drawing program to Khazana.

We ran all experiments on dedicated PentiumPro workstations running FreeBSD, connected by a 100Mbps Ethernet. All the benchmarks were performed on regions with 4-kilobyte pages.

The buffer cache in the smart clients were configured to be large enough to hold all of the experimental data. By configuring the buffer cache so, we avoid spurious conflict misses and can easily control when a request would hit in the local client buffer cache and when it would be required to invoke a remote server.

4.1 Microbenchmark Performance

Table 1 presents the results of a series of microbenchmarks performed on Khazana. The *Null RPC* and *Send Page* times were measurements of the Khazana IPC subsystem, with no Khazana operation is involved. Thus, they represent the raw performance lower bound.

The first set of numbers reported in Table 1 are for a single client and a single KServer running on a different node. `kh_reserve` requires little more than a null RPC time. The time to `kh_allocate` storage for a region, however, increases with the number of pages to be allocated. This is because this operation must perform disk I/O. The *Cold fetch* numbers refer to the time taken by `kh_bcache_fetch` to fetch data stored in the remote KServer. The times track fairly closely with the twice raw data transfer times – the lack of performance tuning for large transfers is evident. The overhead represents the overhead of receiving and manipulating the data in page-sized chunks. `kh_release` takes very little time because it is a local operation in the "smart client". The smart client will normally simply keep the data in its local cache in the expectation that the local client will access it again soon. It does periodically flush its cache, but this is an asynchronous operation, and thus its impact is not reflected in these results. `kh_flush` operations take slightly longer than the corresponding "cold fetch" owing to the extra overhead of ensuring that all the data is reliably flushed to the server.

Warm fetch refers to a `kh_bcache_fetch` that does not have to fetch the data from the server. In this case, the client already had the data read locked, and wished to upgrade the lock to a write lock. In that case, the client merely upgrades its lock with the parent and does not incur the overhead of fetching the data since it is already cached locally. It is in the case of the "warm fetch" and `kh_bcache_release` that the smart client's complexity results in significant performance wins compared to the thin client library. Recall that all data fetch operations translate into a data send over the network in the thin client since the library does not offer any caching. The smart client thus effectively exploits locality in data access to improve performance.

The bottom of Table 1 presents our measurements for the same operations when they are spread over two servers by the same client. In particular, the client performs the `kh_reserve` at one server and approaches the other server for all other operations. This is analogous to the case of applications like KFS where one instance of the application `kh_reserve`'s an entire region and individual instances of the program `kh_allocate` and access parts of that region as the need arises. In this case all the numbers match those of the single client and server case except for the time taken by `kh_allocate`. In this case, the additional delay may be ascribed to the fact that Khazana attempts to allocate storage for a region as close as possible to the request for storage. As a result, the `kh_allocate` operation is implemented similar to a 2-phase protocol with the home node where the region was reserved acting as the coordinator.

4.2 Khazana File Server (KFS)

In this section we describe our experience porting the Linux file system (`ext2fs`) to use Khazana for its persistent storage. Our goal was to evaluate how easily an existing single-machine service can be ported on to Khazana and thus automatically make it a distributed service. We also wanted to evaluate how well the resulting *clustered* file service performed under load.

We started out with an OSkit [16] port of `ext2fs` that implements `ext2fs` as a user-level library. We modified `ext2fs`'s buffer cache to use the smart client's `kh_lock` operation to lock each buffer cache entry in exclusive mode before accessing it and to use `kh_unlock` when it was through with the cache entry. Since the smart client caches page locks, after the initial communication with a KServer to fetch a copy of a file system block, subsequent lock requests can be handled locally. We modified `ext2fs` to fetch the superblock in read-only mode except when modifying. We refer to the modified user-mode file system as KFS.

We used the `userfs` [24] public domain library package to allow KFS to be mounted via the Linux kernel's VFS (virtual file system) interface via the `mount` system call. We also modified the Linux `mke2fs` program to reserve, allocate and format a Khazana region as an `ext2` filesystem. By running multiple instances of KFS on different nodes in the system, we get what is in effect a clustered file server. After making these fairly modest changes to the `ext2fs` library, we had a working clustered version of `ext2fs`.

Unfortunately, this initial implementation suffered serious performance problems because it always locked buffer cache blocks in exclusive mode. Since `ext2fs` was written as a single-node file system, there was no reason for it to distinguish between *read* locks and *write* locks. So, despite the fact that most buffer cache accesses only involve reading the data, cache entries were write locked, which led to extensive shuttling of file system blocks between the KFS instances. To attack this problem, we hand modified `ext2fs` to indicate the mode of access explicitly while fetching blocks from the buffer cache. Thus, in the common case, when the file system operation merely needed to read the contents of the buffer cache, we only needed to acquire a shared lock, which significantly reduces the amount of communication Khazana needed to perform to keep data consistent.

In summary, porting single-machine `ext2fs` to run as a clustered file system on top of Khazana was relatively simple and involved very localized modifications to `ext2fs`. The set of modifications needed to reduce communication overhead, viz. changing `ext2fs` to specify access modes during block access, though more extensive, was conceptually straightforward.

Configuration	Operation	Number of 4kB Pages			
		1	4	10	35
One Server One Client	null RPC	0.469			
	Send page	1.101	2.596	5.277	18.346
	`kh_reserve`	0.600			
	`kh_allocate`	0.856	0.999	1.896	4.324
	Cold fetch	1.141	4.641	11.573	20.930
	`kh_bcache_release`	0.315	1.278	3.081	10.639
	`kh_bcache_flush`	1.686	6.611	16.976	n/a
	Warm Fetch	1.200	1.275	1.373	1.979
Two Servers One Client	`kh_reserve`	1.187			
	`kh_allocate`	3.319	3.568	4.691	7.890

Table 1: Khazana Microbenchmark Results (All times are in milliseconds)

4.3 Khazana Name Server (KNS)

The Khazana nameserver is a B+-tree of fixed-page-size nodes that maps a key that includes a variable length symbolic name to a set of values.

The B+-tree is an efficient data structure for indexing a set of tuples based on a key value. For instance, it is used extensively in databases. It is often used as the primary data structure when implementing a centralized directory/name service that maintains mappings from symbolic names to directory information (e.g., mappings from machine names to IP addresses, as in DNS).

Typical directory services are characterized by frequent lookups, but relatively infrequent updates. To evaluate the performance of a directory service with a B+-tree as its central data structure on Khazana, we took a straightforward single node B+-tree implementation and ported it to run on Khazana using the smart client interface.

In our implementation, every B-tree operation locks two levels worth of B-tree nodes at a time in exclusive mode as it descends the tree. The tests we ran on the B-tree were deliberately designed to generate high contention for locks to test the worst-case performance of Khazana.

Table 2 shows the average and total times to insert (name, value) pairs into an empty B+-tree implemented using a file on a Unix filesystem and the same B+-tree implemented on top of Khazana. To obtain the numbers in the first two rows, 621 names were inserted into a B-tree of node-size 512 bytes. The Khazana version of the test was run with a KServer process and B-tree process on the same machine (a 133 Mhz pentium PC, with little other activity).

The B-tree running on top of Khazana is approximately 8 times slower than the local filesystem version. This is mainly due to the context switch overhead between the B-tree and Khazana server processes.

To obtain the next three rows of results, fifty (name, value) pairs were randomly picked up from a static file with 621 entries and inserted into an empty B-tree by each of one, two, and three B-tree processes respectively. The unusually high average insert time on some of the entries was due to the transient starvation for exclusive locks on B-tree nodes.

4.4 Cooperative `xfig` (KFIG)

`xfig` is a freely available drawing tool for X Windows on Unix written to run on a single processor machine. Our port of `xfig` to Khazana, called KFIG, allows users on multiple machines to share and collaboratively edit a drawing.

KFIG differs from KNS and KFS in the following ways:

- It is a collaborative "groupware" application that makes extensive use of pointers in its data structures.
- It manipulates objects at a finer grain than the page size of the regions involved. In fact since the objects that it manipulates vary in size from as small as 4 bytes to about 100 bytes, there is no common object size that one may choose as the region page size.

The difficulties we faced in our port of `xfig` to Khazana motivated the design of KOLA, as described in Section 2.3.

`xfig` maintains linked lists of objects (lines, arcs, points, etc). Every time an object is created or modified, these lists are modified to reflect the change. While this design works well as a uniprocessor program, it presents several challenges when converted into a distributed application. The lists of objects are now shared by all instances of the application, so the (shared) object references can no longer be virtual memory pointers – they must be Khazana addresses. However, to be used by the existing `xfig` code, these references must be swizzled into virtual memory pointers prior to their use. In addition, since objects are no longer modified by just one instance of the program, there

		Client 1		Client 2		Client 3	
		Average Time(ms)	Total Time(s)	Average Time(ms)	Total Time(s)	Average Time(ms)	Total Time(s)
Non-Khazana	Local filesystem	0.821	0.509				
Khazana	1 Client	06.57	4.074				
	2 Clients	26.2	1.3079	30.5	1.526		
	3 Clients	52.6	26.310	46.5	2.327	48.9	2.446

Table 2: KNS Performance

must be a way for remote instances of the program to determine when the object list is changed that they will update their display. Finally, since many small objects usually reside on the same page, care must be taken to ensure that false sharing does not ruin performance.

To create a distributed version of `xfig`, we made a series of source-level changes to the original program. We converted all object references in shared data structures into Khazana addresses, which required that we swizzle these references into virtual memory pointers in place. As a side effect of converting all object references to Khazana addresses, we eliminate the need to post-process or "flatten" a drawing before storing it.

Next, we replaced the default versions of `malloc` and `free` with versions that allocated objects from a persistent Khazana region, and replaced all calls to `malloc/free` with calls to these new routines.

Then, to enforce concurrency control when multiple clients attempt to modify the drawing, we surrounded all accesses to shared data structures with locks – setting the mode (read or write) appropriately. Normally, all pages were locked in "listen" mode. In this mode clients are informed, via an upcall, whenever there is a change (update) to the page. Read locks were used when the program navigated the data structure (e.g., when it was updating its display) and write locks were used when the drawing was being changed. All data structure navigation has to be done with a read lock on the data As in KFS, the determination of what lock operation was appropriate at any given use of the shared data was determined manually

The final change involved adding a callback routine that invokes the generic `redisplay_canvas()` function whenever a change event occurs. One can imagine more sophisticated upcalls being written that act appropriately based on the change itself. Since KOLA is designed to support remote invocations, the update routine may eventually just invoke the same operations on the local copy of the data as were invoked on the remote machine. Eventually, the port involved changes to almost all the source files to ensure that pointers are swizzled and `kh_lock()`/`kh_unlock()` statements inserted, and 500 lines of new code in all for the allocator and the glue code between Khazana and `xfig`.

The results of the port are encouraging. As Table 3 shows, the time taken for a change on one KFIG screen to be propagated is tiny compared to the time required to update the display. The numbers reported here are for modifying a single object. However, the results will not change significantly with a change in the number of objects, since the overhead is so small compared to the rate of display updates and the ability for human's to notice visual changes. Additionally, since Khazana only transmits the changed object to all sharers, the number of objects on the screen would not impact the time required for data to be available to all participants. Overall, the performance impact of porting `xfig` to Khazana is an increase in redisplay time of about 50%. We expect that those numbers would remain largely constant for all drawing sizes. Users also do not experience any unreasonable delay in reading a saved picture from Khazana as opposed to that from a file.

5 Related Work

The goals behind the original Khazana design closely resemble those of the Apollo Domain system developed over a decade ago [21]. Domain provided a system-supported abstraction of a shared address space spanning multiple workstations that could be used to communicate between applications. For example, clients could invoke file system operations by mapping the appropriate file system metadata into their address space and operating on this metadata themselves. Although this "experiment" into the value of using shared memory for communication between elements of a distributed system was brought to a premature close by Hewlett-Packard's purchase of Apollo, a number of lessons were learned [20]. In particular, directly exposing the internal representations of operating system data structures made it difficult to evolve the system over time – some form of data hiding, such as is present naturally in object systems, is useful.

Distributed object systems (e.g., Clouds [13], Emerald[18], Monads [17], and CORBA[26]) provide uniform location-transparent naming and access to heterogeneous networked objects. In these systems, services can export a well-typed set of object interfaces to clients, which can invoke operations on service objects by *binding* to a particular instance of a service interface and invoking

	Client A	Client B	Client C	Original xfig
Data available (ms)	2.197	3.361	0.991	
Redisplay time (ms)	77.5	90.933	90.780	52.013

Table 3: KFIG Performance (milliseconds)

said service. The addition of *object brokers*, such as are present in CORBA [26], provided a degree of location transparency, and the addition of an object veneer made it easier for servers to change their internal implementation without impacting their clients. However, these systems proved to be effective primarily when used to support the same type and granularity of services previously supported by ad hoc client-server systems: large servers exporting coarse grained operations on large datasets (e.g., mail daemons and file servers).

Building distributed applications and services on top of Khazana is analogous to building shared memory parallel applications on top of a software distributed shared memory system. DSM systems provide a purely shared memory abstraction [23, 7, 1], which can support the efficient execution of shared memory parallel programs [11]. However, these systems are not well-suited for supporting *distributed systems* applications, such as file systems and name servers. They do not support a number of elements critical to such applications: data persistence beyond the execution of a single program, security, efficient high availability, and the ability for multiple independent applications to access and modify shared data [8].

Many contemporary projects closely resemble Khazana – we can only touch on a few of them here.

Perdis [15] and Globe[30] provide functionality similar to Khazana but use distributed shared objects as their base abstraction. Like Khazana, however, both systems expose the lower level storage layer to applications that wish to avoid the overheads associated with a strictly object-based model. We are investigating the possibility of porting their more sophisticated object system technology on top of Khazana in an attempt to get the best of both worlds (models).

Petal[22] exports the notion of a distributed virtual disk. It has been used to implement Frangipani[29], which is similar to KFS. Petal works at a lower level than Khazana, in particular it provides no means of consistency management. Petal was conceived as a globally accessible, distributed storage system. On the other hand, Khazana attempts to provide infrastructure for the development and deployment of distributed services. In general, many distributed services with fine-grained objects are likely to find the file abstraction to be too heavyweight.

Like Khazana, xFS [2] rejects the use of servers and instead use a collection of peer processes to support a distributed system service. However, like Bayou, xFS was designed to meet the restricted goals of a filesystem, and as such is inappropriate for supporting general system services and applications.

GMS[14] allows the operating system to utilize cluster-wide main memory to avoid disk accesses, which could support similar single-cluster applications as Khazana. However, GMS was not designed with scalability, persistence, security, high availability, or interoperability in mind, which limits its applicability.

Bayou[12] is a system designed to support data sharing among mobile users. Bayou focuses on providing a platform to build collaborative applications for users who are likely to be disconnected more often than not. It is most useful for disconnected operations and uses a very specialized weak consistency protocol. In the current implementation, Khazana does not support disconnected operations or such a protocol, although we are considering adding a coherence protocol similar to Bayou's for mobile data.

Finally, object databases [6, 25] provide the necessary distributed storage abstraction, but most are implemented in a client-server environment and the systems we know of were not implemented with wide-area networks and high scalability in mind. Therefore, they do not have the degree of aggressive caching and replication provided by Khazana.

6 Conclusions

The goal of Khazana is to provide persistent globally shared state as the basic abstraction for developing distributed applications. Khazana provides a simple DSM like interface to a flat, globally addressed, persistent storage. Khazana provides all the common requirements of distributed systems, *viz* replication, caching, fault tolerance, security, consistency management, data location, etc while the programmer is free to concentrate on the distributed service being developed. This abstraction was powerful enough to develop efficient ports of important applications like KFS and KNS easily.

During our port of KFIG, however, we found that an object-like extension to Khazana could benefit applications with lots of small data structures interconnected via pointers. This observation motivated the design of KOLA, the Khazana Object LAyer. KOLA provides basic object layer functionality including pointer swizzling, event upcalls, and

a form of update propagation that can be used to perform remote object invocations. We have successfully ported `xfig` to Khazana to create a cooperative whiteboard-like program that can used to create shared drawings. KOLA is still in the early stages of implementation, but the porting of `xfig` demonstrated its usefulness. We expect that Khazana will evolve in the future to provide both a DSM interface and a more complete object system style interface. Various micro and macro benchmarks indicate that the flexibility of Khazana has been achieved at a reasonable cost.

Future work in Khazana will include but is not limited to the development of appropriate caching policies, load balancing algorithms, improved fault tolerance mechanisms, sufficient consistency mechanisms to provide applicability to a wide variety of application, support for disconnected operations, active replication and migration policies, wide area scalability.

Acknowledgements

We would like to thank everybody who helped with this research. First, we'd like to thank Bart Robinson for helping make the `userfs` port possible. We would also like to thank Jay Lepreau and the rest of the Flux group, Marc Shapiro and the rest of the PerDis group, and Maarten van Steen and the rest of the Globe group for their enlightening and inspiring technical discussions that helped motivate and drive this work.

References

[1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. In *IEEE Computer*, January 1996.

[2] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, and R.Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[3] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] G. Brun-Cottan and M. Makpangou. Adaptable replicated objects in distributed environments. *Second BROADCAST Open Workshop*, 1995.

[5] B. Callaghan. WebNFS: The filesystem for the internet. http://www.sun.com/webnfs/wp-webnfs/, 1997.

[6] M. Carey, D. Dewitt, D. Naughton, J. Solomon, et al. Shoring up persistent applications. In *Proceedings, of the 1994 ACM SIGMOD Conf*, May 1994.

[7] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[8] J.B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, May 1995.

[9] J.B. Carter, A. Ranganathan, and S. Susarla. Khazana: An infrastructure for building distributed services. In *Proceedings of The Eighteenth International Conference on Distributed Computing Systems*, pages 562–571, May 1998.

[10] D.R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[11] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, May 1994.

[12] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Wksp on Mobile Computing Systems and Applications*, December 1994.

[13] P. Dasgupta et al. Distributed programming with objects and threads in the Clouds system. *Computing Systems Journal*, 3(4), 1991.

[14] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy, and C.A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[15] P. Ferreira and M. Shapiro et. al. PerDis: Design, implementation, and use of a PERsistent DIstributed Store. *Submitted to The Third Symposium on Operating System Design and Implementation*, February 1999.

[16] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, December 1995.

[17] D. Henskens, P. Brossler, J.L. Keedy, and J. Rosenberg. Coarse and fine grain objects in a distributed persistent store. In *International Workshop on Object-Oriented Operating Systems*, pages 116–1123, 1993.

[18] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[19] N. Kronenberg, H. Levy, and W. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, pages 130–146, May 1986.

[20] P. Leach. Private communication (at hotos-v).

[21] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson, and B.L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–857, November 1983.

[22] E.K. Lee and C.A. Thekkath. Petal: Distributed virtual disks. In *Proceedings 7th International Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[23] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[24] Linux user file system support (*userfs*). http://sunsite.unc.edu/pub/Linux/ALPHA/userfs.

[25] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of SIGMOD '96*, June 1996.

[26] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1996.

[27] Microsoft Corp. Common internet file system protocol (CIFS 1.1). http://www.microsoft.com/workshop/progs/cifs, 1997.

[28] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[29] C.A. Thekkath, T. Mann, and E.K. Lee. Frangipani: A scalable distributed file system. In *Proceedings 16th ACM Symposium on Operating Systems*, October 1997.

[30] M. van Steen, P. Homburg, and A.S. Tanenbaum. Architectural design of globe: A wide-area distributed system. Technical Report IR-422, Vrije Universiteit, Department of Mathematics and Computer Science, March 1997.