Using Khazana to Support Distributed Application Development [*]

Sai Susarla Anand Ranganathan Yury Izrailevsky John Carter

UU-CS-TR-99-008

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

Abstract

One of the most important services required by most distributed applications is some form of shared data management, e.g., a directory service manages shared directory entries while groupware manages shared documents. Each such application currently must implement its own data management mechanisms, because existing runtime systems are not flexible enough to support all distributed applications efficiently. For example, groupware can be efficiently supported by a distributed object system, while a distributed database would prefer a more low-level storage abstraction. The goal of Khazana is to provide programmer's with configurable components that support the data management services required by a wide variety of distributed applications, including: consistent caching, automated replication and migration of data, persistence, access control, and fault tolerance. It does so via a carefully designed set of interfaces that support a hierarchy of data abstractions, ranging from flat data to C++/Java objects, and that give programmers a great deal of control over how their data is managed. To demonstrate the effectiveness of our design, we report on our experience porting three applications to Khazana: a distributed file system, a distributed directory service, and a shared whiteboard.

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. As a result, researchers have developed a wide variety of systems to ease the chore of building distributed applications. The earliest distributed systems provided support for interprocess communication via message passing [8, 27] or remote procedure calls [3], but provided little support for transparent distribution of data and execution or for fault tolerance. More sophisticated systems have provided such support via a variety of basic abstractions, including *distributed files* [5, 12, 23, 9, 28], *distributed objects* [20, 21, 19], and *distributed shared memory* (DSM) [1, 7, 22, 24]. Each of these models is useful for certain types of applications. For example, systems like Petal [23] that support a flat persistent storage abstraction are ideal for supporting distributed file systems and distributed directory services, systems with fairly simple persistent coarse-grained data structures. In contrast, distributed object systems such as CORBA [19] are useful for hiding the complexities of client-server systems, while distributed shared memory systems like Treadmarks [1] are useful for running shared memory codes on top of distributed systems.

Currently, distributed applications must implement their own data management mechanisms, because no existing runtime system can support the very different needs of each application efficiently. This approach has the advantage of allowing each system to optimize its data management mechanisms to suit its specific needs. However, it requires a great deal of redundant programmer effort to develop and maintain each such set of ad hoc mechanisms. It also makes it difficult to share state *between* applications or reuse code devel-

[*] 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,izrailev,retrac}@cs.utah.edu Khazana URL: http://www.cs.utah.edu/projects/khazana

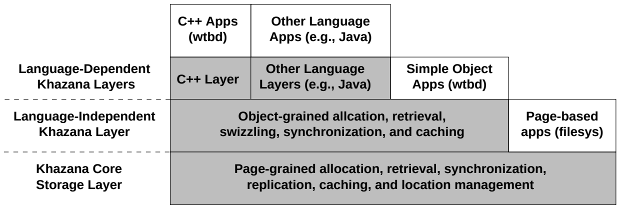
C++ Apps (wtbd)
Other Language Apps (e.g., Java)
Language-Dependent Khazana Layers
C++ Layer
Other Language Layers (e.g., Java)
Simple Object Apps (wtbd)
Language-Independent Khazana Layer
Object-grained allcation, retrieval, swizzling, synchronization, and caching
Page-based apps (filesys)
Khazana Core Storage Layer
Page-grained allocation, retrieval, synchronization, replication, caching, and location management

Figure 1: Khazana's Three Layer Organization

oped for one distributed application when implementing another.

We have built Khazana to demonstrate that a single distributed runtime system can support a wide range of applications with reasonable performance. Khazana is designed to make it easy to implement distributed applications quickly and then gradually improve their performance by refining the underlying consistency protocols and communication mechanisms. Programmers can access and manipulate shared data at a level of abstraction and with consistency mechanisms appropriate for their application's needs. Any specialized consistency protocols, fault tolerance mechanisms, etc., that are developed to support one application can be used to manage data for other applications with similar needs.

Khazana's internal structure is illustrated in Figure 1. It consists of three layers: (i) a base layer that exports a flat global address space abstraction, (ii) a language-independent distributed object layer, and (iii) a collection of layers that export language-specific distributed object abstractions (e.g., C++ and Java).

The base layer is intended to directly support applications with fairly simple "flat" data structures, such as file systems and directory services. It also exports sufficient "hooks" to the two upper layers for them to support a variety of object abstractions efficiently. For example, unlike Petal [23], Khazana's base layer provides mechanisms that give applications or other runtime layers control over how data is kept consistent.

The object layers provide functionality, such as automatic reference swizzling and remote method invocation, appropriate for applications with complex data structure like shared whiteboards. This functionality is unnecessary for applications like file systems or directory services, where its presence would only hurt performance. The difference between the two object layers is that the language-independent layer exports "raw" pointers and objects, while the language-specific layers hide the details of Khazana's object model behind that of an existing object-oriented language like C++ or Java.

To demonstrate that a variety of applications with quite different needs can be supported effectively by Khazana, we have built a distributed file system, a distributed directory service, and a shared whiteboard that use Khazana to manage their shared state. The base layer's persistent flat address space abstraction proved to be well-suited for the distributed file system and distributed directory service, both of which employ fairly flat data abstractions. However, depending on the mix of `lookup` and `update` operations on the directory service, it sometimes made sense to migrate the directory data to the clients (like a DSM system), while at other times it made sense to migrate the operation to the data (like an RPC system). Khazana's layered design and flexible interfaces made it easy to support both models, even to the extent of letting the application decide dynamically which model to use for each operation. The shared whiteboard program exploits the object layers' smart pointer and automatic swizzling capabilities to create pointer-rich data structures that can be shared between whiteboard instances and stored on disk. Manipulating its pointer-rich data structures via the core layer would have imposed a significant burden on the programmer.

Combining elements of distributed shared memory, distributed file systems, and distributed object systems into a unified runtime system for distributed applications had a number of benefits. First, we do not impose the performance overheads of an object system on applications and services where it is not warranted, such as a file or directory service. Second, we are able to exploit the location management, communication, consistency, and security mechanisms present in the core layer, thereby avoiding redundant development. Khazana's core layer provides a facility to deliver arbitrary "update" messages to one or all applications using a particular piece of data. The object layers use this facility to support location-transparent remote method invocations (RMI), whereby computation migrates to the data rather than vice versa. This facility could be used to support application-specific consistency mechanisms, such as employing a reliable multicast protocol [16] to manage updates to streaming multimedia images. Also, since the base layer tracks the locations of objects in the system and knows which ones are currently instantiated, there is no need for a separate object request broker (ORB), as in CORBA. Finally, if an object migrates, or one instance fails, the underlying consistency management routines will simply forward the *update* message (method invocation) to a different node that has registered its willingness to handle such operations. Thus, the Khazana object layer is inherently fault tolerant, due to the fault tolerance support in the core layer.

The remainder of this paper is organized as follows. We present the organization of a Khazana system in Section 2. We then describe the design and implementation of the core Khazana layer (Section 3), followed by the object layers (Section 4). Section 5 contains a description of our experience porting three test applications to Khazana (a distributed file server, a directory server, and a shared whiteboard) and the lessons that we derived from this effort. We then compare Khazana to previous and contemporary systems in Section 6 and draw conclusions in Section 7.

2 Organization of a Khazana System

Khazana is designed with the premise that most distributed applications and services at their core do roughly the same thing: manage shared state. What differs dramatically is what this shared state represents (e.g., files, database entries, game state, or interactive video images) and how it is manipulated (e.g., broadcast to a collection of cooperating applications, queried from a "central" server, or modified frequently in response to changes in the environment). We developed Khazana to explore the extent to which a carefully designed and sufficiently flexible runtime system can support the data management needs of a wide variety of applications.

Figure 2 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. The storage that Khazana manages consists of an amalgam of RAM and disk space spread over the nodes that participate in the Khazana 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 data; nodes 2 and 4 can access regions and cache them in local DRAM, but do not store them on the local disk. Each object in the figure, e.g., the square, represents a single piece of data managed by Khazana. In this example, node 1 is caching a complete copy of the "square" object, while nodes 3 and 5 are each caching a part of it.

Khazana's global shared storage abstraction is implemented by a collection of cooperating daemon processes (BServers) and client libraries. BServers run on some (not necessarily all) machines of a potentially wide-area network. Each node running a BServer 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.

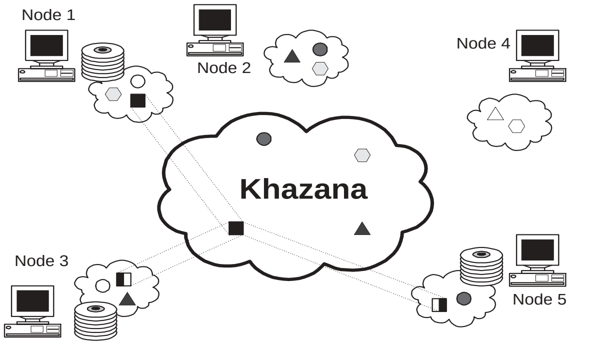

Figure 2: Typical Khazana-Based Distributed System: The cloud represents Khazana's global shared storage abstraction. Applications on any of the five nodes can manipulate shared objects, represented by the various shapes. The highlighted "square" object is currently being accessed by three nodes; a complete copy resides on Node 1, while nodes 3 and 5 combine to store a second replica.

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 data is 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.

Khazana is free to distribute 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, as illustrated in Figure 2. Presumably Khazana chooses these nodes because they access the region most frequently, are the most stable, or have the most available resources. Currently, our data placement schemes are quite simplistic (first touch, replicate on demand), but a goal of the project is to develop caching policies that address the needs for load balancing, high performance, high availability, and constrained resources.

3 Khazana Core Layer

The core Khazana layer exports the abstraction of a persistent globally shared storage space addressed by 128-bit global 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 the core layer, regions are simply a range of bytes – it is up to higher level software (e.g., the object layers or application code) to interpret the contents of a region. The minimum unit of storage managed by the core layer 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.

In effect, the core layer can be thought of as a globally accessible disk against which distributed applications read and write data, similar to Petal[23]. As will be described below, however, Khazana provides a finer level of control over how individual regions are managed than Petal, and Khazana's address-based naming scheme allows applications to embed references to other structures in their data. The core layer is designed to handle the common problems associated with sharing *generic* 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.

The basic operations supported by the core layer are:

`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_register()/kh_update()`: Khazana lets applications register to be notified of various Khazana internal events by supplying a callback routine and an event type as parameters to `kh_register()`. The callback routine is called on occurrence of the specified event. One useful event is an "object" update, initiated implicitly by `kh_unlock()` or explicitly via `kh_update()`.

`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.

Applications can use `kh_register()` to register callback routines to be invoked whenever certain events occur. For example, callback functions can be set up to respond when a remote node performs a `kh_unlock()` or `kh_update()` operation. `kh_unlock()` causes a Khazana-created update message to be sent to registered listeners, while `kh_update()` takes as input an arbitrary "update" message to be sent. Khazana does not interpret the contents of these update messages, so they can be used to support application-level consistency protocols or to send arbitrary messages to registered listeners.

Figure 3 illustrates how the core layer functionality is decomposed into five major components: the Khazana core API, the location service, the consistency management service, the RAM buffer cache, and the

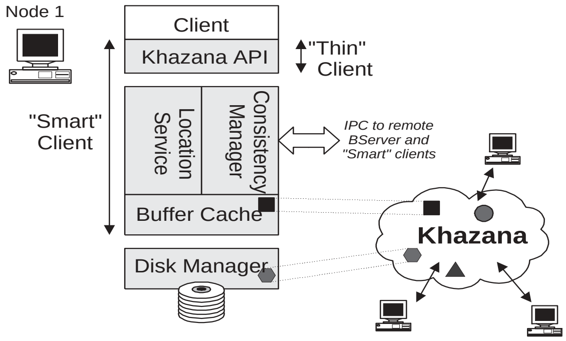

Figure 3: Khazana Core Layer Internals

disk buffer cache. These functions are divided between client libraries and BServers as follows.

Client Libraries: We provide two client core 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 BServer, which implements all of the Khazana core functions. The thin client is, in essence, simply an RPC client stub.

The smart client, in contrast, actually implements most core Khazana functions and protocols. In particular, it aggressively caches 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 BServers and remote smart clients. In most ways, the smart client is a peer to the BServers for regions its client is accessing. However, 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 BServer to perform address space range lookups, examine or modify attributes of reserved regions, reserve/free address space, or allocate/free space in the global storage hierarchy.

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.

Consistency Management: The core layer supports a 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 must lock a range of global addresses region in an appropriate mode, e.g., obtaining a read lock prior to reading it and a write lock prior to modifying it. The application can then explicitly `read` or `write` the data, and when it is done accessing the data, unlock it.

The semantics of the various lock modes (read, write, etc.) are entirely dependent on the coherence protocol being used to manage the region. A default coherence protocol is provided that enforces conventional mutual exclusion semantics on regions, but an important feature of Khazana's core layer design is that it exports consistency management operations to program modules called Consistency Managers (CMs) running in the BServers and smart clients. CMs are somewhat independent from the rest of a BServer or smart client, and are free to interpret "read lock" and "write lock" events as they see fit. 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. They cooperate to implement the required level of consistency among the replicas using Brun-Cottan-style decomposable consistency management [4].

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.

Several benefits result from decomposing the entity that determines when consistency operations must occur (the CM) from the rest of the system. Khazana's consistency management mechanism is highly flexible, so applications can tune their consistency management protocols based on specific application characteristics. More importantly, the hooks exported by this decomposed consistency management scheme can be exploited by the object layers in a number of ways. For example, one "consistency protocol" allows an application to register functions to handle particular events. This protocol allows the object layer to detect when an object is first loaded and transparently swizzle it. It also can be used to support remote method invocation (RMI), as will be described in detail in Section 4.

Location Management: Khazana's core layer 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 BServer 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. Deallocating 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.

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.

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. Although a 128-bit address space might itself seem to be sufficiently large that addresses would make good capabilities, Khazana addresses are not allocated randomly and thus make poor capabilities. 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. The key distribution services can, of course, themselves be built on top of Khazana. BServers 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, BServers will be able to authenticate themselves to one another, thereby ensuring that they are not exporting Khazana metadata to clients masquerading as a BServer. In addition, all inter-node communication between BServers 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.

```
addr_t put_in_global(buf, sz, key)
    char *buf;    /* Data to be stored    */
    size_t sz;    /* Size of data         */
    key_t key;    /* Security key         */
{
    addr_t addr;
    lock_context_t lock;

    /* Reserve address space for 'buf'    */
    /* and specify write-invalidate prot. */
    kh_reserve(&addr, sz, key, WT_INV);

    /* Allocate physical storage          */
    kh_allocate(addr, sz);

    /* Get exclusive (write) access       */
    kh_lock(addr, sz, WRITE, &lock, key);

    /* Store 'buf' in allocated region.   */
    kh_write(addr, lock, buf);

    /* Unlock region, which pushes data   */
    /* to persistent storage.             */
    kh_unlock(lock);

    return(addr);
}
```

Figure 4: Simple Khazana Programming Example: Stores `buf` in Khazana and returns 128-bit address of where it was stored.

Using Khazana's Core Layer Directly: Programming directly on top of Khazana's core layer is similar to programming on top of a DSM system. Data structures that an applications wishes to share with other applications or other instances of itself running on other nodes are allocated and stored in Khazana regions. A clustered application, such as the file system described in Section 5.2, 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 a filesystem or a loosely coherent update protocol, such as is appropriate for interactive groupware applications.

Figure 4 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 with 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 (`WT_INV`). 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 know 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 either 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.

4 The Khazana Object Layers

Khazana's flat shared storage abstraction works well for some services (e.g., file systems and directory services), but it is a poor match for applications that use "reference-rich" data structures or legacy applications written in object-oriented languages. The Khazana object layers are intended to simplify the use of Khazana for building these kinds of applications. Its design evolved as we gained experience converting the `xfig` drawing program into a groupware application built on Khazana. We found that the flat storage abstraction exported by the core Khazana layer needed to be extended in a number of ways to support a broader set of applications.

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 machine's virtual address space, so references to Khazana data cannot be mapped to virtual addresses. Without some level of runtime support, applications must swizzle Khazana addresses into local virtual memory addresses, and vice versa, by hand - a tedious chore that can be automated.

Second, a purely pull-based coherence model, where changes to data are only detected when a client locks the data, is a poor match for interactive applications that wish to be notified when state that 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.

Third, 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 hold many individual objects. Implementing fine-grained variable-sized objects on a page-based system without adequate support for fine-grained sharing can lead to inefficient space utilization and false sharing, which can result in poor performance.

Finally, moving data to the computation is not always the right execution model. When an application wants to make a small change to a large data structure, or when it will not reuse a data structure and thus would get no benefit from maintaining a local cached copy of it, it is better to move the computation to an existing instance of the data rather than replicating it. Thus, some form of transparent RPC or RMI support is needed.

Khazana's object layer addresses these problems by providing an efficient and largely transparent environment for manipulating persistent objects that are then stored and shared via Khazana's shared address space. It layer consists of two parts, a *language-independent layer* and a *language-specific layer*. The language-independent layer supports basic operations to manipulate arbitrary-sized (not just page-sized) data objects. These operations include mechanisms to allocate/free persistent objects, retrieve objects into virtual memory in a synchronized way, store them persistently, convert persistent references from/to in-memory virtual addresses, and manage an in-memory cache of objects. The language-independent layer does not address such vital issues as dynamic type identification and checking, object access detection and loading, class inheritance, or transparent concurrency control (via locking). Instead, these issues are handled by each language-specific layer.

At the core of our current C++ object-layer is a

```
struct Emp_Record {
    Date Start, Finish;
    int Salary; };
class Person {
    string Name, Address;
    Date DOB;
    ... };
class Employee: public Person {
    Emp_Record My_Record;

    /* Smart pointer that refers */
    /* to a >persistent< object. */
    Ref <Person> Supervisor;
    ... };
```

Figure 5: **Simple Khazana C++ Type Declarations**: The Emp_Record and Person classes are normal C++ class declarations. Employee, however, contains a reference to a persistent object, Supervisor. Whenever the program traverses such a reference, e.g., via emp->Supervisor, the object is loaded from Khazana automatically, if necessary.

preprocessor that parses conventional class declarations and augments them with overloaded new and delete operators, special constructors, and synchronization methods. Objects created from these augmented classes can then be stored and retrieved transparently to/from the shared store. In addition, the preprocessor generates support libraries that implement class metadata reinitialization and maintain a static cache for information relevant to the objects' state of consistency (e.g. their lock contexts and intents). Changes introduced by the preprocessor do not modify object layout and thus can be integrated transparently to other modules of the program that are not converted by the preprocessor.

Writing a Khazana-based C++ Program: To write a Khazana C++ program, a programmer first declares the classes whose objects will be stored and manipulated by Khazana. Figure 5 contains an example of such class declarations. In this example, there are three persistent types (Emp_Record, Person, and Employee). Emp_Record and Person contain only simple data that can be loaded and stored directly from the Khazana store without interpretation. Employee objects, on the other hand, contain a reference to another persistent object, Supervisor, which must be handled carefully.

Running this header file through our C++ preprocessor produces a set of header and C++ files that are used to support these classes. In addition, the preprocessor adds several non-virtual methods to the classes that have been declared. Figure 6 shows the methods that the preprocessor adds to the Employee class. Some of these methods, like the ones used to load and reinitialize the data, are transparent to the programmer. Others, like the lock and unlock methods, are provided for programmers to insert in their code to implement concurrency control, when and where appropriate.

```
class Employee: public Person {
  ...
  Employee (KhClass kh):
      Person(kh),
      MyRecord (kh),
      Supervisor(kh)
      {...};
  Bool lock(op_mode_t opmode);
  Bool unlock();
  void* operator new(size_t size, KhClass kh);
  void* operator new(size_t size, void *mem);
  void* operator new(size_t size);
  void  operator delete(void* local_ref);
  ...
}
```

Figure 6: **Methods added to Employee class by Khazana C++ preprocessor**. The new constructor, Employee(KhClass kh), is called when loading an existing persistent instance of class Employee into the local memory. The lock and unlock operations are for Khazana synchronization. The three added new operators are used to create a new persistent object, to load (reinitialize) an existing persistent instance of an Employee object, and to create a new local instance of an Employee object, respectively.

The set of support files produced by the preprocessor extends a number of object-oriented features of C++ to the persistent distributed environment of Khazana. It supports automatic class loading, which is accomplished statically. Persistent object creation and management is done transparently by utilizing special constructors, overloaded new and reimplemented delete operators. The C++ layer appends a special type information tag to each instance of an object stored in Khazana, and uses it for object loading and dynamic type checking. Persistent references (Object IDs, or oids) are generally hidden from the programmer and are swizzled and unswizzled automatically through the use of smart pointers. Concurrency control handlers are generated by the preprocessor for each class. Finally, the preprocessor generates additional code to

```
int kh_object_swizzle (void* obj, int type) {
    KhClass kh;
    switch (type_id) {
        case TYPE_CLASS_A :
             obj = new(obj) A(kh); break;

        case TYPE_CLASS_B :
             obj = new(obj) B(kh); break;
        ...}
    return 0;
}
```

Figure 7: **Object initialization:** When a persistent object is first loaded into memory, it is automatically initialized using the special `new` operator added by the C++ preprocessor. The type of the object is stored in a special tag with each persistent object instance.

support RMI. The details of these mechanisms are described below.

Class loading: Class usage detection and loading occurs at compile time. Instead of storing the class metadata (including the member and reference layout, method information, and class type and hierarchy information) as a separate persistent schema object, classes are linked at compile time. The C++ preprocessor parses user-specified class declarations and adds several methods that allow transparent object loading and synchronization.

Object loading: Objects are brought in from Khazana to virtual memory at the time of first access. Objects in Khazana are addressed by a unique Object-ID (oid). Currently, an oid is the same as the 128-bit Khazana address where the object has been stored. However, the language-independent layer may choose to prefetch objects in addition to the one requested. All objects that are brought in to local memory must be initialized, which is done via a callback mechanism. An example of how this is done is given in Figure 7. The language-specific layer registers a callback function, `kh_object_swizzle()`, that is called any time a new object is mapped into local memory. This function reinitializes the object (e.g., setting up vtbl pointers) so that it resembles an object of the same class that has been created locally. Similarly, the C++ layer provides a callback that is called every time an object is flushed. This design allows any conversion between the in-memory representation and the persistent (unswizzled) Khazana representation to be made as lazily as possible.

Reference swizzling: We implement references to persistent objects using a smart point template class, called `Ref<T>`, where `T` is the type of the object to which the reference points. The `Supervisor` reference in the `Employee` class is an example use of this facility. Smart pointers have been discussed previously, so we will not go into the details of the implementation here [13]. The overloaded dereference operators enable us to trap object access and thereby transparently convert the oid into a local memory reference. This may involve fetching and reinitializing the object.

While smart pointers ensure that programmers can use pointers to Khazana objects just as they would use pointers to regular objects, there is some performance overhead. In particular, every smart pointer access involves a table lookup. Programmers can avoid this problem by caching the pointer to the local copy of the object, rather than always accessing the object's contents via its persistent reference (smart pointer). While an object is locked, it is guaranteed not to move in local memory, so during this time the pointer value may be cached and a per-access table lookup can be avoided.

Remote method invocation: The object layers build on the underlying Khazana coherence management hooks, described in Section 3, to implement RMI. Recall that the `kh_register()` interface lets applications register a callback function to be invoked when certain events occur, while `kh_update()` allows a program to send an arbitrary application-level "update" message to one or more remote instances of an object. For each class that has been annotated to be "immobile," meaning that operations on it should be performed via RMI, the C++ preprocessor generates two classes: "stubs" and "real classes," similar to a normal RPC stub generator. Which sets of routines (stub or real classes) gets linked to an application depends on whether or not it indicated that the objects were immobile (stubs) or normal (real classes). To support RMI, server stubs register interest in "updates" to the object using the `kh_register()` interface. The client stub on the sending side of the RMI 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 and passes up to the registered callback function of the application running there – the server stub. The roles are reversed for propagating responses. In this way, Khazana is able to support both DSM-like "migrate the data" style distributed computing and RPC-like "migrate the computation" style distributed computing on top of a common storage management platform.

Event notification: Khazana allows applications to register interest in Khazana-internal events. In addition to supporting RMI, this facility can be used to signal applications in response to modifications to specific regions, changes to region attributes such as the security token, and changes to the set of nodes caching a particular region copy location. This facility is used by the shared whiteboard program described in Section 5.4. Each instance of the whiteboard 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. Notification services like this have proven useful in a variety of interactive, groupware-style applications [11]. Currently our update and notification mechanisms are serial, but we are considering exploiting a lightweight reliable multicast protocol, such as SRM [16], to make it more efficient.

Operations not supported: The language-independent layer 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. We also do not currently support atomic transactions as primitive operations at either the core or object layers, assuming instead that when necessary they will be built as part of a higher level library. These decisions may change as we gain more experience using Khazana.

5 Evaluation

In this section, we describe a series of experiments we have performed on Khazana to determine how effective it is at making distributed application development (or porting) easy. In Section 5.1, 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 5.2), a Khazana-based name service (Section 5.3), and a port of the `xfig` drawing program to Khazana.

We ran all experiments on dedicated 300MHz PentiumPro workstations running FreeBSD, connected by a 100Mbps Ethernet. All the benchmarks were performed on regions with 4-kilobyte pages.

The buffer caches 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.

5.1 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 IPC subsystem underlying, 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 BServer 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_read` to fetch data stored in the remote BServer. These results are consistently close to double the raw data transfer times, indicating that the performance of large data transfers needs to be improved. The difference between the *Cold_fetch* results and the raw data transfer times represents the overhead of receiving and manipulating the data in page-sized chunks. `kh_unlock` takes very little time because it is a local operation in the "smart client". The smart client will normally 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_lock/kh_write` pair that does not need to fetch the data from the server. In this case, the client already has the data read locked, and wishes 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_unlock` 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 the file server 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. All of 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.

5.2 Clustered File Service

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 could be ported to Khazana, thereby creating a distributed version of the service. We also wanted to evaluate how well the resulting *clustered* file service performed under load.

We started out with an OSkit [17] 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, lock requests after the initial copy of a file system block is fetched from a BServer 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 BFS.

We used the userfs [25] public domain library package to allow BFS 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 BFS 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[1].

Unfortunately, the 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 BFS 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.

To evaluate the performance of our clustered version of ext2fs, we ran the Andrew benchmark suite on a number of configurations of the system. First, to give us a baseline to compare against, we ran a single-client version of the Andrew benchmark against a single-node instance of ext2fs running directly on top of Linux. This experiment took 17.059 seconds to complete.

After determining a baseline for our performance,

we varied the number of clients from one to four. On each client node we ran an instance of the Khazana-ified version of ext2fs, which the client used as the file server for its run of the Andrew benchmark. In this experiment, the clients can exploit the caching ability of the Khazana smart client to avoid communicating with the (logically remote) "file server" when possible. The results of this set of experiments can be found in Figure 2. From this experiment we can see that Khazana-ifying ext2fs adds about 33% overhead compared to the non-Khazana version when running a single client and server (24.376 seconds versus 17.059 seconds). However, performance is quite stable as we increase the number of clients running the benchmark. Even when the load is increased by a factor of four, performance only degrades by 40% (from 24.276 seconds to as much as 38.264 seconds).

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. Our initial performance results showed that this port resulted in a version of the filesystem that was approximately 33% slower than the initial version, but with good scaling. We believe these results are quite conservative, since we have not yet put great effort into making Khazana highly efficient.

[1] We also found and removed quite a few bugs in the ext2fs and userfs sources, which took far more time to handle than the changes required to Khazana-ify them.

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_unlock	0.315	1.278	3.081	10.639
	kh_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)

	Client 1	Client 2	Client 3	Client 4
1 client	24.376	—	—	—
2 clients	32.315	31.524		
3 clients	35.881	32.547	30.293	
4 clients	38.264	36.578	35.365	33.492

Table 2: Khazana Clustered File Server Results (All times are in seconds): In this experiment, we ran the Andrew benchmark on from one to four nodes. We ran one BServer, on Client 1, one instance of ext2fs with the smart client per benchmark client. The baseline performance of a single Andrew benchmark client running directly on top of a user-level ext2fs file system is 17.059 seconds.

5.3 Directory Service

The Khazana directory server 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 3 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 BServer process and B+-tree process on the same machine. 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.

5.4 Shared Whiteboard

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 Khazanaallows users on multiple

		Client 1		Client 2		Client 3	
		Average Time	Total Time	Average Time	Total Time	Average Time	Total Time
Non-Khazana	Local filesystem	0.82	0.509				
Khazana	1 Client	6.57	4.074				
	2 Clients	26.20	1.308	30.5	1.526		
	3 Clients	52.62	26.31	46.5	2.327	48.9	2.446

Table 3: Directory Service Performance (milliseconds)

machines to share and collaboratively edit a drawing. This shared whiteboard example differs from the distributed file service and distributed directory service examples 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 run directly atop the core layer of Khazana motivated the design of object layers, as described earlier.

`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. Persistent references to shared graphical objects 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 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. First, we converted all object references in shared data structures into Khazana smart pointers, which automated the process of swizzling and unswizzling these references. Next, we overrode the default versions of `malloc` and `free` with versions that allocated objects from a persistent Khazana region. 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. This part of the porting effort could not be handled automatically by our object layer support. 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 our distributed file service, 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.

The original port of `xfig` to Khazana's core layer 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 for the allocator and the glue code between Khazana and `xfig`. The object layer support significantly reduced the amount of work required to handle the object issues, although we had to spend considerable time and effort to port `xfig`, a C program, to C++ so that our preprocessor could handle it.

The results of the port are encouraging. As Table 4 shows, the time taken for a change on one whiteboard 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, but 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. 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.

6 Related Work

The goal of Khazana closely resembles that of the Apollo Domain system from over a decade ago [22] – a universal distributed storage infrastructure on which large classes of applications and services can be built. 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. A number of lessons were learned from this effort. 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. We address this problem by not requiring that all distributed services use Khazana's shared storage abstraction. For example, Khazana's core layer abstractions can be used to support inter-fileserver clustering and client data caching, but directory operations could be buried inside of library routines or performed via RPC operations.

Distributed object systems (e.g., Emerald[21], Monads [20], CORBA[19], and Legion [18]) 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 said service. The addition of *object brokers*, such as are present in CORBA [19], 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 [24, 7, 1], which can support the efficient execution of shared memory parallel programs [10]. 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.

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

Khazana is closest in spirit and design to Perdis [15] and Globe[29]. Both 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. Their reasons for employing a multi-layer design is based on similar experience to our own – forcing all applications to use a single abstraction (in this case, objects) is ineffective and inefficient. We find it interesting that they came to the same conclusion despite starting from the opposite abstraction (distributed objects) as we did (distributed virtual disk).

Petal[23] exports the notion of a distributed virtual disk. It has been used to implement Frangipani[28], which is similar to our clustered userfs-based file system. 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.

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 wide area 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

	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 4: Shared Whiteboard Performance (milliseconds)

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.

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[12], xFS was designed to meet the restricted goals of a filesystem, and as such is inappropriate for supporting general system services and applications.

Finally, object databases [6, 26] 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.

7 Conclusions

Currently, distributed applications must implement their own data management mechanisms, because no existing runtime system can support the very different needs of each application efficiently. We have built Khazana to demonstrate that a single distributed runtime system can support a wide range of applications with reasonable performance. Khazana consists of three layers: (i) a base layer that exports a flat global address space abstraction, (ii) a language-independent distributed object layer, and (iii) a collection of layers that export language-specific distributed object abstractions (e.g., C++ and Java). Khazana provides programmer's with configurable components that support the data management services required by a wide variety of distributed applications, including: consistent caching, automated replication and migration of data, persistence, access control, and fault tolerance.

We reported on our experience porting three applications to Khazana: a distributed file system, a distributed directory service, and a shared whiteboard. As we showed via our experience porting `ex2fs` and building a directory service from scratch to run atop Khazana, the base layer's simple "flat" data abstraction is a good match for certain types of applications, such as file systems and directory services. For example, when we ran the Andrew filesystem benchmark on the Khazana version of `ext2fs`, it was only 40% slower on four nodes than on one node despite quadrupling the workload. The directory service demonstrated similarly solid performance. What makes these results particularly heartening is that Khazana is still fairly untuned, and we should be able to improve the performance of the core system substantially when it becomes a major focus.

While porting `xfig` to Khazana to create a shared whiteboard program, however, we found that an object-like extension to Khazana could benefit applications with small data structures interconnected via pointers. In particular, manually swizzling every pointer to a "shared" graphical object was very tedious when done by hand. This problem motivated Khazana's support for objects. The object layers provide functionality, such as automatic reference swizzling and remote method invocation, appropriate for applications with complex data structure like our shared whiteboard program. When we reimplemented `xfig` using our object layer support, performance remained the same as in the original hand-coded version, easily within the bounds of tolerance for human perception. In general, we believe that we have demonstrated that Khazana can be used to quickly implement a variety of distributed applications at a reasonable cost.

In summary, the contributions of this paper are:

- We demonstrate that a single distributed runtime system can support a wide range of applications with reasonable performance (within 50% of the performance of hand-tuned versions of the applications, even using a very untuned version of Khazana).
- We provide insight into how such a runtime system should be organized.
- We demonstrate the benefits of combining elements of distributed shared memory, distributed file systems, and distributed object systems.

Future work will be focussed in four areas: (i) improving the performance, scalability, and fault tolerance of the core Khazana layer, (ii) greatly expanding the number of distributed applications run on top of Khazana to identify issues that we may have overlooked in our initial design, and (iii) adding support for a Java object layer.

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] 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.

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

[10] 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.

[11] M. Day. What groupware needs: Notification services. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 118–121, May 1997.

[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 Workshop on Mobile Computing Systems and Applications*, December 1994.

[13] M. Ellis and B. Stroustrop. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[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. In *Submitted to The Third Symposium on Operating System Design and Implementation*, February 1999.

[16] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for lightweight sessions and applicatioon level framing. In *Proceedings of the Sigcomm '95 Symposium*, September 1995.

[17] 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.

[18] A.S. Grimshaw and W.W. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] Linux user-level file system support (*userfs*). http://sunsite.unc.edu/pub/linux/alpha/userfs.

[26] 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.

[27] 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.

[28] 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.

[29] 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.