University of Utah, Department of Computer Science, Technical Report UUCS-98-009

Khazana*An Infrastructure for Building Distributed Services†

John Carter Anand Ranganathan Sai Susarla

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

Abstract

Essentially all distributed systems, applications, and services at some level boil down to the problem of managing distributed shared state. Unfortunately, while the problem of managing distributed shared state is shared by many applications, there is no common means of managing the data – every application devises its own solution. We have developed Khazana, *a distributed service exporting the abstraction of a distributed persistent globally shared store that applications can use to store their shared state. Khazana is responsible for performing many of the common operations needed by distributed applications, including replication, consistency management, fault recovery, access control, and location management. Using Khazana as a form of middleware, distributed applications can be quickly developed from corresponding uniprocessor applications through the insertion of Khazana data access and synchronization operations.*

1 Introduction

Essentially all distributed systems applications at some level boil down to the problem of managing distributed shared state. Consider the following application areas:

- Distributed file systems (AFS, NFS, NTFS, WebNFS, CIFS, ...)
- Clustered file systems (DEC, Microsoft, ...)
- Distributed directory services (Novell's NDS, Microsoft's Active Directory, ...)
- Distributed databases (Oracle, SQL Server, ...)
- Distributed object systems (DCOM, CORBA, ...)
- Collaborative groupware (Lotus Notes, Microsoft Exchange, ...)

All of these services, and many more, perform essentially the same function, albeit in very different settings. That function is managing distributed shared state and providing a convenient way for users and other applications to access, update, and delete the information being managed. Unfortunately, while the problem of managing distributed shared state is shared by all of the above applications, there is no common infrastructure for managing shared data, so every system implements its own solution. The thesis of this paper is that it should be possible for distributed clients and servers to share state without each instance of sharing requiring specially written code. Just as TCP/IP hides many complex issues from programmers (e.g., handling link failures, routing, and congestion), there should be support for distributed state sharing that lets most applications remain oblivious to the many problems associated with managing shared state (e.g., heterogeneity, security, high availability, caching strategies, and coherence management). We are developing *Khazana*, a distributed service to provide this support.

Figure 1 illustrates a typical Khazana-based distributed system consisting of five nodes. Applications such as those described above can use Khazana to store shared data that can be accessed from any node connected to Khazana. The Khazana design assumes that some or all of the nodes may be connected via slow or intermittent WAN links. Khazana will use local storage, both volatile (RAM) and persistent (disk), on its constituent nodes to store data. In the example illustrated in Figure 1, the square represents a single piece

*Khazana (*kä-zä-nä*) [*Hindi*]: Treasury, repository, cache.

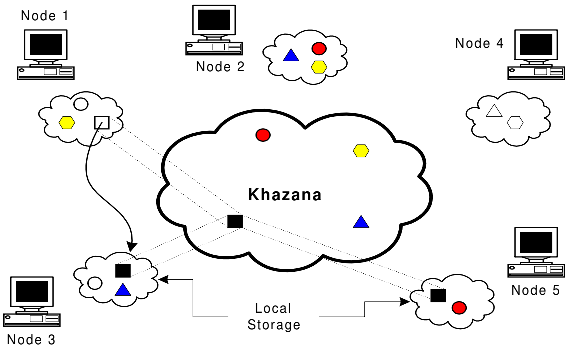

Figure 1: Typical Distributed Systems Based on Khazana

of shared data that is physically replicated on Nodes 3 and 5 (represented by the fact that the square is solid in the local Khazana storage on those nodes). When Node 1 accesses the data, Khazana is responsible for locating a copy of the data and providing it to the requester on Node 1. The details of how Khazana operates will be discussed in depth in Sections 2 and 3.

We envision Khazana as middleware on top of which a variety of distributed applications, servers, and runtime environments will be built. Most distributed applications have common needs (e.g., location transparency, high availability, and scalability), and it is Khazana's job to satisfy these needs. Although there are standard techniques available to achieve each of the above properties (e.g., naming, replication, and caching), the logic to deal with distribution usually is hardcoded into each application individually. This approach to building distributed applications and services, although capable of providing high performance, is suboptimal in several ways, including:

Complexity: The same technique may not perform well in all situations, so applications will either need to adapt their behavior or perform poorly in some environments. For example, a caching technique that performs well on fast LANs may not perform well on slow WANs.

Correctness: Programs that deal with networking and distribution explicitly tend to be complex and prone to errors. Requiring each application to "roll its own" support for distribution increases the amount of effort required to develop or modify a distributed application.

Interoperability: Independently developed services and applications cannot easily interoperate, because the way that they manage data differs. This has caused serious problems for developers trying to create unified directory and/or database services, for example.

Khazana is designed to solve these problems. Khazana does not interpret the shared 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). To improve performance, Khazana is responsive to guidance from its clients. Applications that use Khazana can specify their desired consistency, fault tolerance, security, and other requirements for individual blocks of data, and Khazana adapts the way that it manages the specified data. For example, a clustered file server that uses Khazana to store its file contents can specify that Khazana maintain at least N copies of each file block to ensure N-1 redundancy in the face of failures, and further specify that its data be kept strongly consistent. A clustered web server, on the other hand, would likely require fewer copies be maintained and allow a weaker (and thus higher performance) consistency protocol. A distributed object system built on top of Khazana might allow individual programmers to specify the sharing and replication semantics they desire for each of their objects. It also could use location information exported from Khazana to decide if it is more efficient to load a local copy of the object or perform a remote invocation of the object on a node where it is already physically instantiated.

Our work on Khazana is in its early phases. Our current focus is identifying the set of services that Khazana should support, the amount of flexibility that it must export, the extent to which it can support a wide variety of consumers, and the algorithms needed so it can scale efficiently. In particular, we are exploring the following questions:

- What interface should Khazana export to applications?
- What are the requirements of various classes of Khazana clients (e.g., file systems, directory servers, databases, object systems, etc.)?
- Can applications that use Khazana achieve high (scalable) performance?
- How much will existing distributed applications need to be changed to benefit from Khazana? Is the change worthwhile?

We discuss our current answers to these questions, and the resulting Khazana system design, in the remainder of this paper, which is organized as follows. In Section 2, we discuss the basic design philosophy of Khazana, and describe at a high level how a set of

Khazana nodes cooperate to provide the abstraction of persistent, globally accessible, secure shared state. In Section 3, we present the major data structures and inter-node protocols used by Khazana. Section 4 describes two sample uses of Khazana for different kinds of distributed applications. Section 5 describes the current implementation status of Khazana. We discuss the value of a Khazana-like middleware layer for distributed services in Section 6. Section 7 compares this work to related efforts. Finally, we draw conclusions in Section 8.

2 Approach

As described above, the basic abstraction exported by Khazana is that of shared state. Specifically, it exports the abstraction of a flat distributed persistent globally shared storage space called *global memory*. Applications allocate space in the global memory in much the same way that they allocate normal memory, except that Khazana regions are "addressed" using 128-bit identifiers, and there is no direct correspondence between Khazana addresses and an application's virtual addresses. Applications can access global memory either via explicit `read()` and `write()` calls that specify global addresses (similar to file reads and writes), or by mapping parts of global memory to their virtual memory space and reading and writing to this mapped section (akin to a memory-mapped file). In effect, Khazana can be thought of as a globally accessible disk against which distributed applications read and write data, similar to Petal[20]. Unlike Petal, however, Khazana explicitly supports consistency management and is designed as a middleware layer for arbitrary distributed applications.

Referring back to Figure 1, the Khazana service is implemented by a dynamically changing set of cooperating daemon processes running on some (not necessarily all) machines of a potentially wide-area network. Note that there is no notion of a "server" in a Khazana system – all Khazana *nodes* are peers that cooperate to provide the illusion of a unified resource. Typically an application process (client) interacts with Khazana through library routines. The basic set of operations exported by Khazana allows clients to reserve, allocate, and access *regions*. To Khazana, a region is simply a block of client data with common application-level characteristics accessed via contiguous Khazana *addresses*. Khazana also has the notion of a *page*, which is the minimum amount of data that Khazana independently manages. By default, regions are made up of 4-kilobyte pages to match the most common machine virtual memory page size. Related to regions, Khazana provides operations to:

- *reserve* and *unreserve* a contiguous range of global address space (region). A region cannot be accessed until physical storage is explicitly allocated to it. At the time of reservation, clients can specify that a region be managed in pages larger than 4-kilobytes (e.g., 16 kilobytes, 64 kilobytes, ...).
- *allocate* and *free* underlying physical storage for a given region of reserved global address space.
- *lock* and *unlock* parts of regions in a specified mode (e.g., read-only, read-write etc). The lock operation returns a lock context, which must be used during subsequent read and write operations to the region. Lock operations indicate the caller's intention to access a portion of a region. These operations do not themselves enforce any concurrency control policy on the Khazana region. The consistency protocol ultimately decides the concurrency control policy based on these stated intentions to provide the required consistency semantics.
- *read* and *write* subparts of a region by presenting its lock context.
- *get* and *set* attributes of a region. Currently, a region's attributes include:
  - desired consistency level
  - consistency protocol
  - access control information
  - minimum number of replicas

As can be seen from the list of operations above, Khazana provides basic data access operations, but does not provide or enforce any particular programming model. In particular, Khazana itself does not provide a notion of objects, transactions, streams, or other higher semantic level operations often found in distributed object systems or object databases[6, 7, 9, 18, 22]. We believe that these features of a global storage system are useful to many applications, but are heavyweight and unnecessary for many others. In contrast, Khazana concentrates only on storing and accessing data efficiently and robustly in a large distributed environment, and leaves higher level semantics to other middleware layers. Khazana has been designed to export a flexible set of operations and controls to support just this kind of layered use with other middleware layers.

One way to implement distributed application objects using Khazana is as follows. All object state is kept in global memory. A clustered application can start multiple instances of itself, each of which can access and modify the same object(s) by mapping, locking, accessing, and unlocking the object's constituent region(s). Unlike conventional distributed systems, instances of a clustered application do not need to interact directly with one another to keep their shared state consistent. In fact, unless the application explicitly queries Khazana, it will be unaware that there are other applications accessing and caching the shared state. The application can specify the object's consistency requirements[3]. Currently, Khazana can support strictly consistent objects[19]. The application can also specify that a minimum number of replicas be maintained for fault-tolerance. Currently all instances of an object must be accessed using the same consistency mechanisms, but we are exploring ways to relax this requirement so that each client can specify its specific requirements for each object that it maps.

Khazana is free to distribute object 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. In Figure 1, the square object has been physically replicated on two nodes (Node 3 and Node 5), presumably because these nodes are accessing the object most frequently, are the most stable, and/or had the most available resources when the object was last replicated. A major goal of this research is to develop caching policies that balance the needs for load balancing, low latency access to data, availability behavior, and resource constraints.

In summary, the main design goals of Khazana are:

Location transparency: Any client should be able access any region regardless of its current location(s) or degree of replication, subject to network connectivity and security restrictions.

High availability: If a node storing a copy of a region of global memory is accessible from a client, then the data itself must be available to the client. This requires that the internal state that Khazana needs to access or manage a region, so called *metadata*, must be available if the region is available.

Scalability: Performance should scale as nodes are added if the new nodes do not contend for access to the same regions as existing nodes. Data should be cached near where it is used, and the operations used to locate and access data must not be heavily dependent on the number or configuration of nodes in the system.

Flexibility: Khazana must provide "hooks" so that a wide variety of applications and higher-level middleware layers can use its data management facilities without undue loss of performance.

Robustness: The system should recover gracefully from node or network failures.

3 Design of Khazana

Global memory and the Khazana metadata used to locate and manage it are distributed among participating nodes. Machines can dynamically enter and leave Khazana and contribute/reclaim local resources (e.g., RAM or disk space) to/from Khazana. In this section we discuss Khazana's solution to the following problems:

- Global address space and storage management
- Locating global memory data and metadata
- Keeping shared data and metadata consistent
- Local storage management
- Handling failures gracefully

There are, of course, other issues such as authentication, node membership management, handling application failures, and backup/restore, but space precludes a detailed discussion of Khazana's solution to these problems.

3.1 Global Address Space and Storage Management

Each Khazana node attempts to cache copies of frequently accessed regions and the associated metadata nearby, preferably locally. Khazana maintains a global *region descriptor* associated with each region that stores various region attributes such as its security attributes, page size, and desired consistency protocol. In addition, each region has a *home node* that maintains a copy of the region's descriptor and keeps track of all the nodes maintaining copies of the region's data.

In addition to the per-region data structures, Khazana maintains a globally distributed data structure called the *address map* that maintains global information about the state of regions. The address map is used to keep track of reserved and free regions within the global address space. It is also used to locate the home nodes of regions in much the same way that directories are used to track copies of pages in software DSM systems [21]. The address map is implemented as a distributed tree where each subtree describes a range of global address space in finer detail. Each tree node is of fixed size and contains a set of entries describing disjoint global memory regions, each of which contains either a non-exhaustive list of home nodes for a reserved region or points to the root node of a subtree describing the region in finer detail. The address map itself resides in Khazana. A well-known region beginning at address 0 stores the root node of the address map tree. The address map is replicated and kept consistent using a relaxed consistency protocol, as it is not imperative that its contents be completely accurate. If the set of nodes specified in a given region's address map entry is stale, the region can still be located using a cluster-walk algorithm described below.

For scalability, the design of Khazana organizes nodes into groups of closely-connected nodes called *clusters*. A large-scale version of Khazana would involve multiple clusters, organized into a hierarchy, although the current prototype supports only a single cluster. Each cluster has one or more designated *cluster managers*, nodes responsible for being aware of other cluster locations, caching hint information about regions stored in the local cluster, and representing the local cluster during inter-cluster communication (if there are multiple clusters). Given the current lack of support for multiple clusters, we concentrate on the single-cluster design in this paper.

Khazana daemon processes maintain a pool of locally reserved, but unused, address space. In response to a client request to reserve a new region of memory, the contacted Khazana daemon first attempts to find enough space in unreserved regions that it is managing locally. If it has insufficient local unreserved space, the node contacts its local cluster manager, requesting a large (e.g., one gigabyte) region of unreserved space that it will then locally manage. Each cluster manager maintains hints of the sizes of free address space (total size, maximum free region size, etc) managed by other nodes in its cluster. 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.

3.2 Locating Khazana Regions

To initiate most operations, Khazana must obtain a copy of the region descriptor for the region enclosing the requested global range. The region descriptor is used to identify the region's home node(s). 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, and thus may contain stale data, but this is not a problem. Regions do not migrate home nodes often, so the cached value is most likely accurate, but even if the home node information is out of date, the use of a stale home pointer will simply result in a message being sent to a node that no longer is home to the object. If the region directory does not contain an entry for the desired region or the home node contacted as a result of the cache lookup is no longer a home node for the region, a node next queries its local cluster manager to determine if the region is cached in a nearby node. Only if this search fails does it search 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. If the region descriptor cannot be located, the region is deemed inaccessible and the operation fails back to the client. Otherwise Khazana checks the region's access permissions and (optionally) obtains copies of the relevant region pages. The location of data pages can be obtained by querying one of the region's home nodes.

3.3 Consistency management

Replicating global data and metadata introduces the problem of keeping the replicas consistent. Khazana maintains consistency as follows. Each Khazana node can independently choose to create a local replica of a data item based on its resource constraints. Program modules called Consistency Managers (CMs) run at each of the replica sites and cooperate to implement the required level of consistency among the replicas as is done by Brun-Cottan[4]. A Khazana node treats lock requests on an object as indications of intent to access the object in the specified mode (read-only, read-write, write-shared, etc.). It obtains the local consistency manager's permission before granting such requests. The CM, in response to such requests, checks if they conflict with ongoing operations. If necessary, it delays granting the locks 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. Given this consistency management framework, a variety of consistency protocols can be implemented for use by the Khazana to suit various application needs. For example, for the address map tree nodes, we use a release consistent protocol[15]. We plan to experiment with even more relaxed models for applications such as web caches and some database query engines for which release consistency is overkill. Such applications typically can tolerate data that is temporarily out-of-date (i.e., one or two versions old) as long as they get fast response.

3.4 Local storage management

Node-local storage is treated as a cache of global data indexed by global addresses. The local storage subsystem on each node maintains a *page directory*, indexed by global addresses, that contains information about individual pages of global regions including the list of nodes sharing this page. If a region's pages are locally cached, the page directory lists the local node as a sharer. The page directory maintains persistent information about pages homed locally, and for performance reasons it also maintains a cache of information about pages with remote homes. Like the region directory, the page directory is node-specific and not stored in global shared memory.

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 - main memory, disk, local filesystem, tape, etc., organized into a storage hierarchy based on access speed, as in xFS[27].

The local storage system handles access to global pages stored locally. In response to allocation requests, the local storage system will attempt to locate available storage for the specified range of addresses. If available, it will simply allocate available local storage. If local storage is full, it can choose to victimize unlocked pages. In the prototype implementation, there are two levels of local storage: main memory and on-disk. When memory is full, the local storage system can victimize pages from RAM to disk. When the disk cache wants to victimize a page, it must invoke the consistency protocol associated with the page to update the list of sharers, push any dirty data to remote nodes, etc.

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). It is the responsibility of the aforementioned consistency management routines to ensure that all of the locally cached copies of a region are kept globally consistent. The local storage subsystem simply provides backing store for Khazana.

3.5 Failure handling

Khazana is designed to cope with node and network failures. Khazana operations are repeatedly tried on all known Khazana nodes until they succeed or timeout. All errors encountered while acquiring resources (e.g., reserve, allocate, lock, read, write) are reflected back to the original client, while errors encountered while releasing resources (unreserve, deallocate, unlock) are not. Instead, the Khazana system keeps trying the operation in the background until it succeeds.

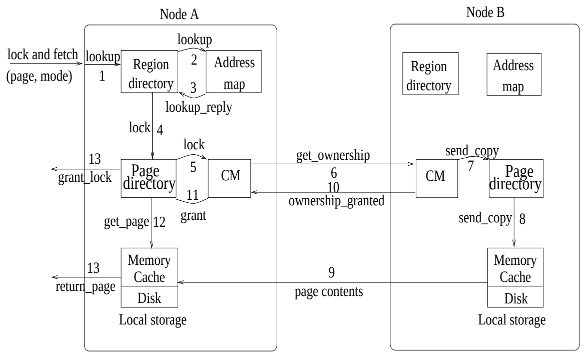

Figure 2: Sequence of actions on a lock and fetch request

Since the first step of many Khazana operations is address lookup, the success of many operations depends on the availability of the relevant address map tree nodes. To make Khazana less sensitive to the loss of address map nodes, the local region directory is searched first and then the cluster manager is queried, before an address map tree search is started. That way, if a lookup for a nearby address has recently been performed by another node in the same cluster, the tree search is avoided. 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. Such availability is also required to reserve and unreserve regions.

Finally, Khazana allows clients to specify a minimum number of primary replicas that should be maintained for each page in a Khazana region. This functionality further enhances availability, at a cost of resource consumption.

3.6 Example Operation

Figure 2 shows the steps involved in servicing a simple <lock, fetch> request pair for a page p at Node A, when Node B owns the page.

Step 1 Node A obtains the region descriptor for p's enclosing region.

Steps 2, 3 (Optional) Obtaining the region descriptor might involve an address map lookup.

Step 4 After sanity checks, p is looked up in the page directory. The page directory entry holds location and consistency information for that page.

Step 5 The Consistency Manager (CM) is invoked to grant the lock on p in specified mode.

Step 6 The CM requests its peer on Node B for credentials to grant the lock in the specified mode.

Steps 7, 8, 9 Node B's CM directs the local daemon to supply a copy of `p` to Node A, which caches it in its local storage.

Step 10 The CM grants ownership of `p` to Node A.

Step 11 The CM on node A then grants the lock.

Steps 12, 13 Node A supplies a copy of `p` locked in the requested mode to the requestor out of its local storage.

4 Example Uses of Khazana

In this section, we discuss two example applications that have been designed to run using Khazana. The first is a wide area distributed filesystem and the second a distributed object system. The discussion focuses on both the implementation issues involved and on how Khazana enables each application to be unaware of the fact that it is distributed.

4.1 Wide Area Distributed Filesystem

Recently the notion of Internet-wide file systems has become popular[5, 11]. Some of the desirable characteristics of such a filesystem are simplicity, support for efficient handling of files with diverse access patterns (e.g., temporary files, system files, program development files), scalability, availability, and easy load balancing. Ideally, the amount of effort required to modify a single-node filserver to make it a clustered file server should be small. In addition, it should be possible to alleviate server load by starting up additional instances of the server and transparently redirecting some of the load to these new instances. Using Khazana, we have designed a filesystem that takes a first step towards achieving these goals. A brief description follows.

The filesystem treats the entire Khazana space as a single disk, limited in size only by the size of the Khazana address space. At the time of file system creation, the creator allocates a superblock and an inode for the root of the filesystem. Mounting this filesystem only requires the Khazana address of the superblock. Creating a file involves the creation of an inode and directory entry for the file. Each inode is allocated as a region of its own. Parameters specified at file creation time may be used to specify the number of replicas required, consistency level required, access modes permitted, and so forth. In the current implementation, each block of the filesystem is allocated into a separate 4-kilobyte region. An alternative would be for the filesystem to allocate each file into a single contiguous region, which would require the filesystem to resize the region whenever the file size changes.

Opening a file is as simple as finding the inode address for the file by a recursive descent of the filesystem directory tree from the root and caching that address. Reads and writes to a file involve finding the Khazana address for the page to be read or written, locking the page in the appropriate mode, mapping it into local memory, and executing the actual operation. Closing a file releases the region containing the corresponding inode. To truncate a file, the system deallocates regions no longer needed.

This approach to designing a clustered file system satisfies all the criteria outlined above, with the possible exception of scalable performance. The same filesystem can be run on a stand-alone machine or in a distributed environment without the system being aware of the change in environment. Khazana takes care of the consistency, replication, and location of the individual regions. Specifying appropriate attributes at creation time allows the system to efficiently support different types of files. The default handling for each type of file can be changed if access patterns dictate a change in predicted behavior. Khazana provides high availability guarantees. The failure of one filesystem instance will not cause the entire filesystem to become unavailable, as is the case in a conventional distributed file system when the file server crashes. The filesystem maintainer can specify the desired degree of fault tolerance. New instances of the filesystem can be initiated without changes to existing instances of the filesystem, which enables external load balancing when the system becomes loaded. The initial prototype of Khazana performs poorly, but we have not yet spent enough time tuning performance to make a judgement about the inherent cost of building distributed systems on top of a Khazana-like middleware layer.

4.2 Distributed Objects

Another of the motivating applications for Khazana is an efficient implementation of a distributed objects runtime layer (e.g., CORBA[16] or DCOM[10]). To build a distributed object runtime system on top of Khazana, we plan to use Khazana as the repository for object data and for maintaining location information related to each object. The object runtime layer is responsible for determining the degree of consistency needed for each object, ensuring that the appropriate locking and data access operations are inserted (transparently) into the object code, and determining when to create a local replica of an object rather than using RPC to invoke a remote instance of the object. The Khazana-based distributed object system abstracts away many implementation issues that would arise, such as the need to keep replicated object contents consistent and ensuring availability in the face of failures. The object veneer would implement the more powerful semantics expected by users of distributed object systems, such as reference counting (or garbage collection) and transactional behavior. Khazana provides the hooks needed to support these higher level semantics, but does not implement them directly, since we anticipate that many users of Khazana will not require this level of support or be willing to pay the performance or resource overhead they will entail.

Depending on the size of the object, it might be allocated as a whole region or as part of a larger region. Khazana provides location transparency for the object by associating with each object a unique identifying Khazana address. All methods associated with the object need to be translated to the Khazana interface of reads and writes to the data contained within the object. Methods are invoked by downloading the code to be executed along with the object instance, and invoking the code locally. Khazana is useful in that it maintains consistency across all copies and replicas of the same object and provides caching to speed access. Currently, Khazana does not recognize object boundaries within a page. As a result, consistency management on fine-grain objects (small enough that many of them fit on a single region-page) is likely to incur a substantial overhead if false sharing is not addressed, although there are known techniques for addressing this problem[4, 9]. Khazana's CM interface adopts the approach of Brun-Cottan and Makpangou[4] to enable better application-specific conflict detection to address false sharing.

5 Implementation Status

We currently have a working, single-cluster prototype of Khazana ready. Cluster hierarchies are yet to be implemented. We have been able to test the algorithms and infrastructure. We are currently porting the BSD Fast File System available as part of the OSKit[14] to use Khazana. Concurrently, a simplified distributed object system is being implemented using Khazana.

The only consistency model we currently support is a Concurrent Read Exclusive Write (CREW) protocol[19]. However, the system was designed so that plugging in new protocols or consistency managers is only a matter of registering them with Khazana, provided they export the required functionality.

While the current implementation runs in a Unix environment, only the messaging layer is system dependent. Therefore, we expect that Khazana system can be ported to other platforms with little effort, but no ports are currently underway.

6 Discussion

One major concern that developers of distributed services have is that infrastructures that hide the distribution of data remove a lot of useful system-level information needed to efficiently control distribution. This is a valid concern, although we believe the tradeoffs to be reasonable, similar to the way in which high level languages and communication protocols are the norm. Even though services written on top of our infrastructure may not perform as well as the hand-coded versions, we believe that Khazana's flexible interface allows a lot of room for application-specific optimizations for performance, in addition to considerably simplifying their development. The extent of performance degradation compared to the hand-coded version depends on how well distributed services can translate their optimizations to shared state access optimizations. The widespread use of high level languages represents the tradeoff that developers are willing to make for rapid development and hiding of complexity. We are currently experimenting with various classes of applications using Khazana to validate this belief.

7 Related Work

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) [1, 8, 17, 21]. Just as parallelizing compute-intensive scientific programs using DSM simplifies the task of building and maintaining the parallel program, building distributed applications and services on top of Khazana will make them easier to build and maintain. Conventional DSM systems, however, lack several key features required to support distributed applications and services. In particular, conventional DSM systems are built for a single application, where all shared data persists only as long as the program is running, all data is mapped at the same address in every process accessing the data, failures generally lead to total system failure, and heterogeneity is not handled. Khazana addresses these limitations of conventional DSM systems.

Distributed object systems such as CORBA[16] and DCOM[10] provide uniform location-transparent naming and access to heterogeneous networked objects. Although these systems provide a convenient way for applications to access information in well-defined ways, they do not by themselves provide the basic functionality of managing shared state. In contrast, Khazana provides a more basic infrastructure of distributed storage on top of which distributed object systems and other less structured applications can be built. One drawback of the object abstraction is that it is difficult to provide efficient replication, fault-tolerance, caching, and consistency management for arbitrary application objects in an application-independent manner. The notion of state is hidden within the objects and cannot be made visible to the lower layer except by explicit object serialization, which can be costly and cumbersome in many applications. This argument applies equally to object systems that provide replication management[4, 23], but some of the techniques developed for explicitly managing replication and caching of individual objects would work well in a Khazana-like environment. For example, Brun-Cottan's approach to separating application-specific conflict-detection from generic consistency management[4] is used in Khazana as a modular consistency management framework. Object databases[6, 7, 9, 18, 22] 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.

There are many projects that closely resemble Khazana. Globe[26] provides functionality similar to Khazana but uses distributed shared objects as its base abstraction. We did not choose the same abstraction for the reasons outlined above. WebFS[25] is a global cache coherent file system to provide a common substrate for developing (distributed) Internet applications. Its goals are similar to those of Khazana, but it exports a file system as its base abstraction. It uses URLs for naming, and supports application-specific coherency protocols. In contrast, Khazana exports a shared global cache coherent shared storage abstraction. For many distributed services with fine-grained objects like object systems, a file abstraction may be too heavyweight. Petal[20] exports the notion of a distributed virtual disk. It has been used to implement Frangipani[24] which is similar to the filesystem we envisage in Section 4.1. 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. However, it is conceivable that Khazana could use Petal for storage management — in such a scenario, Khazana would be the middleware between Petal and the distributed service. GMS[13] 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 will limit 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.

Serverless file systems[2] utilize workstations on a closely coupled network, cooperating as peers to provide filesystem services. Like Khazana, serverless file systems reject the use of servers and instead use a collection of peer processes to support a distributed system service. xFS[27] is a wide area mass storage filesystem with similar scalability goals as Khazana. Both systems are designed to meet the restricted goals of a filesystem, and as such are inappropriate for supporting general system services and applications.

8 Conclusions

In this paper, we have motivated the need for developing a common infrastructure for building distributed applications and services. Most distributed applications require some form of distributed shared state management, but currently applications tend to spin their own mechanisms. We have developed Khazana, a distributed service that allows uniprocessor applications and services to be made into distributed applications and services in a straightforward fashion. We believe that this will greatly increase the number of distributed programs that are generated – Khazana handles many of the hardest problems associated with distribution, leaving application developers to concentrate on their real application needs.

Khazana exports the abstraction of 128-bit-addressable persistent shared storage that transparently spans all nodes in the system. It handles replication, consistency management, fault recovery, access control, and location management of shared state stored in it. It does so through a collection of cooperating Khazana nodes that use local storage, both volatile (RAM) and persistent (disk), on its constituent nodes to store data near where it is accessed. Our initial experience with Khazana indicates that it can support a variety of distributed services effectively. We will continue to refine Khazana and extend its API as we gain experience building applications and services based on the globally shared storage paradigm.

The work discussed herein represents only the beginning of this line of research. Among the topics we plan to explore are scalability to multiple clusters, resource- and load-aware migration and replication policies, more efficient region location algorithms, more sophisticated fault tolerance schemes, flexible security and authentication mechanisms, and a number of possible performance optimizations.

Acknowledgements

We would like to thank those individuals who helped improve the quality of this paper, and in particular the anonymous reviewers and members of the Computer Systems Laboratory seminar at the University of Utah.

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] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[4] G. Brun-Cottan and M. Makpangou. Adaptable replicated objects in distributed environments. *2nd BROADCAST Open Workshop*, June 1995.

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

[6] M. Carey, D. Dewitt, D. Frank, G. Graefe, J. Richards, E. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In *Proceedings of the 1st International Workshop on Object-Oriented Database Systems*, September 1996.

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

[8] 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*, August 1995.

[9] M. Castro, A. Adya, B. Liskov, and A.C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[10] Microsoft Corp. DCOM technical overview. http://microsoft.com/ntserver/library/dcomtec.exe.

[11] Microsoft Corp. Common Internet File System (CIFS). http://microsoft.com/intdev/cifs, 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.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.

[14] 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.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

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

[17] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[18] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *Communications of the ACM*, October 1991.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[20] 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.

[21] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, November 1989.

[22] 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.

[23] M. Makpangou, Y. Gourhant, J.L. Narzul, and M. Shapiro. *Fragmented Objects for Distributed Abstractions.* IEEE Computer Society Press, 1994.

[24] 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.

[25] A. Vahdat, P. Eastham, and T. Anderson. WebFS: A global cache coherent filesystem. http://www.cs.berkeley.edu/~vahdat/webfs/webfs.html, 1996.

[26] 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.

[27] R.Y. Wang and T.E. Anderson. xFS: A wide area mass storage file system. In *4th Workshop on Workstation Operating Systems*, October 1993.