Khazana: A Flexible Wide Area Data Store

Sai Susarla and John Carter
School of Computing
University of Utah
{*sai, retrac*}@*cs.utah.edu*

Abstract

Internet services increasingly need to operate over variable quality networks. It is challenging to provide a consistent user experience given the variable availability and latency of the underlying environment. Data caching is an effective technique for hiding variable network quality, but implementing caching in a new service requires significant effort due to the difficulty of reusing existing consistency management solutions.

In this paper, we present a wide-area data store, called Khazana*, that provides efficient coherent shared data management to a variety of internet services. We exploit the observation that although consistency needs vary widely from service to service, there is a common set of core mechanisms, combinations of which can support a variety of service needs. Khazana is unique in that it supports multiple consistency policies and gives applications control over the mechanisms used to implement these policies.*

We demonstrate the value of Khazana's flexible consistency management infrastructure by building and evaluating three diverse internet services: a replicated chat service that can be used for scalable data exchange, a distributed file store that supports multiple file sharing patterns, and a content distribution service that caters to a variety of delivery requirements.

1 Introduction

Many Internet-based services used for collaboration and information exchange are data-intensive. The variety in the capability of devices and networks over which users interact with these services is increasing rapidly. To provide a good end-user experience in this variable quality environment, a service should shield users from this variance as much as possible. One way to do this is to replicate or migrate portions of a service close to where they are used while managing their coordination internally. In this paper we focus on one specific aspect of this coordination for data-intensive services - namely shared access to service state and/or data.

Caching is an effective technique for improving data availability and access latency. A plethora of consistency schemes have been developed in the context of specific wide-area services with specific data access patterns [10, 13, 16, 21]. We surveyed a variety of distributed services and found that most employ different combinations of a small common set of consistency mechanisms [23]. However, services typically implement their consistency mechanisms as inseparable bundles that provide the needed service requirements. This bundling makes it difficult to reuse existing caching mechanisms in new services.

In this paper:

- We argue that behind the diverse consistency management needs of a variety of wide-area internet services there lie commonalities that can be met in a reusable yet efficient manner. We identify a set of consistency policies and implementation strategies, combinations of which can efficiently support a variety of service needs.
- We present a novel wide-area data store, Khazana, that exports these mechanisms through a familiar abstraction and gives applications control over their behavior.
- We demonstrate how Khazana's flexibility lets it efficiently manage the coherence of shared data for three distinct classes of wide-area internet services.

To achieve scalability and performance in the wide-area, Khazana implements a multi-level caching network similar to that proposed by Blaze [1]. Figure 1 illustrates a representative Khazana caching network. Unlike previous work, Khazana supports multiple coherence policies and our implementation of these policies takes into account the non-uniform communication costs between nodes. We provide multiple policy-independent

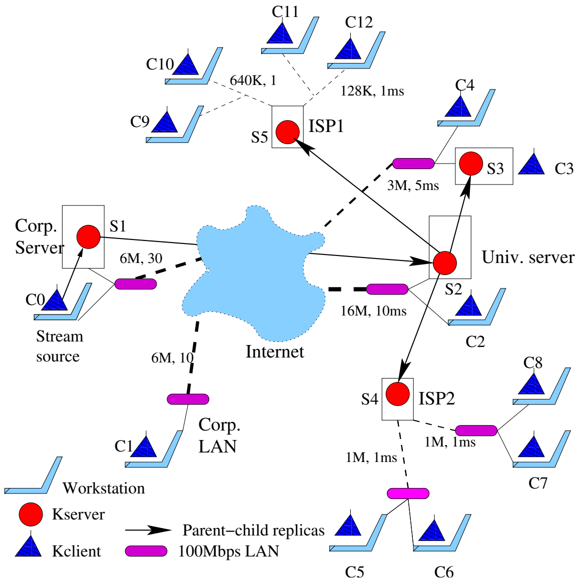

Figure 1. Khazana Topology for Kstream

update mechanisms that leverage the replica hierarchy for scalability. We delegate work among replicas based on network proximity, and resolve update conflicts at each level of the replica hierarchy to limit their spread. Khazana is unique in that it supports multiple policies and gives applications choice over the mechanisms used to enforce the policies. This flexibility enables Khazana to support a wide variety of applications efficiently.

The choice of policies and implementation mechanisms Khazana supports was driven by a detailed survey of a variety of distributed applications. In our study [23], we observed that certain key issues often recur in the design of shared data management in a variety of applications. They are: when access to a replica is allowed (e.g., optimistic vs. pessimistic access control), where updates are issued (e.g., single-master vs. multi-master), how updates are initiated (e.g., pull- vs. push-based), and the conditions that can trigger an update transfer (e.g., timer-driven, staleness-driven, resource-driven, eager, or lazy). These issues are common to many consistency policies, but existing systems bundle the choices inseparably into a policy's implementation.

We demonstrate that a reusable consistency management system can be built that efficiently supports a variety of data sharing needs. In particular, we show that three different applications with diverse needs and data access characteristics can be supported well by employing the above options in various combinations. We show how the same system can provide traditional file access with various consistency and availability tradeoffs, support scalable data exchange (which is traditionally implemented using message-passing), and enable a content distribution service that tunes the quality of data delivered to a recipient's resource budget. Our system completely offloads the complexity of managing coherent wide-area data access from each of these services while adding peer-to-peer service proxying capabilities to them for scalability and availability.

2 Background

Consistency mechanisms vary with respect to when they allow access to shared data, when and how they propagate updates to synchronize replicated data, and how they detect and resolve update conflicts (if they arise). To identify the consistency mechanisms that many applications care about, we have surveyed a variety of network applications involving shared data [23]. In our study, we observed that certain common issues recur in a variety of data access patterns. In this section, we classify consistency mechanisms along a few dimensions listed below and extend the taxonomy described by Saito [21].

When is access allowed? *Access control* determines whether an access/update request is satisfied after or before bringing it to the desired consistency level. *Pessimistic* schemes (either lazily or eagerly) propagate updates to achieve desired level of coherence before allowing access. *Optimistic* schemes allow immediate access to any data replica, but perform update propagation after data access in the hope that conflicting updates are rare and/or can be easily resolved. With optimistic schemes there is a risk of stale reads and conflicting writes, as updates may not propagate in time to prevent conflicting access on other nodes. The need for conflict detection makes them more complicated to implement than pessimistic schemes.

Pessimistic consistency schemes are attractive when updates are relatively frequent but cannot handle disconnections. Optimistic schemes provide high availability in the face of disconnections but perform poorly when update conflicts are frequent. One possible strategy (adopted by Coda [13]) is to employ pessimistic consistency during periods of good connectivity, while switching to an optimistic scheme at other times.

Where are updates issued? The *update transfer model* determines where an update can be issued. *Single master* systems statically designate a master replica from which all updates are made (e.g., DNS, Active Directory schema [15], or web content). They are simple to implement, but the master is a single point of failure. Single master systems are suitable when consistency is more important than availability. *Multi-master* systems allow updates to be issued at any replica, and coordinate to exchange each other's updates (e.g., Active Directory data [15], AFS [11], and Bayou [5]). Such systems are more complex and may introduce update conflicts. Hence they are suitable when data availability is more important than consistency.

Who triggers update transfer? *Pull-based* schemes make each replica responsible for polling others for updates (e.g., Coda, WWW), whereas *push-based* schemes make a replica with a pending update responsible for delivering it to other replicas (e.g., Usenet, Porcupine [20], and Active Directory). Pull-based schemes are suitable for read/write data sharing environments involving large scale but infrequent read-sharing. Push-based schemes are more suited to frequently read data and for data distribution.

When is an update transfer triggered? The *update transfer condition* determines when updates are propagated. Transfers may be *user-initiated* (e.g., when user presses the *refresh* button), *timer-driven* (e.g., every 3 hrs in Active Directory), *topology-based* (based on communication costs and connectivity situation), *data-driven* (based on application-specific tolerable data staleness [27]), *eager* (upon update arrival), or *lazy* (when an access request arrives).

These issues are largely orthogonal to one another – many combinations of choices are possible. The right policies to employ depends heavily on the application's data access patterns. By giving an application control over these choices, we better meet their consistency needs. For instance, an internet game and a streaming video service both need push-based updates. However the latter needs more control over its delivery rate to adapt to receivers of various capabilities, while the former needs reliable real-time delivery. Some services, e.g., Active Directory, have different update requirements for different pieces of their state (data versus schema). We have developed a shared data management system, Khazana, that gives applications the ability to specify useful combinations of mechanisms. We discuss Khazana in more detail in the following section.

3 Khazana Design

3.1 Overview

We are developing a wide-area data sharing framework, named Khazana, that provides coherent wide area file access at page-granularity. Khazana is differs from many systems in that it supports multiple consistency policies on a per-file basis. A typical distributed service can make use of Khazana by storing its shared state (persistent or not) in the Khazana data store and setting the consistency-related attributes for this state on a per-file basis. Distributed components of the service access shared data by contacting any nearby Khazana server. Khazana servers cooperate to locate and cache data close to where they are used while maintaining consistency according to the specified attributes. Because Khazana manages the shared state, services are no longer tied to the particular physical nodes where their data resides, and can more easily migrate or replicate.

We have two main design criteria for Khazana:

- *Reusability*: We support a feature only if it proves useful for multiple applications.
- *Efficiency*: We avoid features that cannot be efficiently implemented without hardwiring service-specific knowledge or without significantly slowing down other popular applications.

Khazana exports a file system-like interface that supports traditional read/write as well as operational update operations on files. The basic units of data in Khazana are flat files called *regions*[1] and Khazana provides session-oriented read/write access to these files at page-granularity. Khazana clients can control how the consistency of each region is maintained. In particular, clients have control over the four major issues discussed above, and in more detail in Section 3.2.

Khazana is organized as a collection of cooperating peer data server processes called Kservers. Each Kserver utilizes a portion of its persistent local store for permanent copies of some files and the rest as a cache of remotely stored files. Kservers can run on machines spread across the wide-area. Kservers discover each other through an external means such as a directory service. Each Kserver monitors its connection quality (latency, bandwidth, connectivity) to other Kservers with which it communicates and uses this information for controlling its protocol-related communication. Applications access Khazana files through a library called Kclient that encapsulates all communication with Kservers while exporting the Khazana session interface.

A file's permanent copy site is called its *home node*; there may be many replicas cached at other nodes. A Kserver creates a local file copy when a client contacts it to access a file that it does not store locally.

File Naming and Location Tracking

Khazana files are named by unique numeric IDs (called KIDs) assigned to them by their home node at the time of their creation. To simplify file location and management of ID space in our prototype, we employ a simple scheme wherein each Kserver manages its own local ID space and KIDs are a combination of its home Kserver's

[1] We use the terms *Khazana file* and *Khazana region* interchangeably in the rest of this document.

network address and its ID within the Kserver's local ID space. A Kserver finds a file's permanent copy site based on the address hard-coded in the file's KID. We choose this scheme for its simplicity and because naming and location management are not the focus of our work. Khazana's design allows this scheme to be easily replaced by more sophisticated file naming and dynamic location tracking schemes such as those of Pastry [18], Chord [22] and Tapestry [28].

Even though a file can be located by the home node address hard-coded in its KID, each Kserver maintains a cache of hints of nearby Kservers with copies of a file, which it contacts before the home node. This reduces access latency if the requesting Kserver is weakly connected to the file's home node relative to other copy sites. Our idea is analogous to cooperative proxy web caching [9], but works for write-shared data as well.

Creating and Destroying Replicas

When a Kserver queries another Kserver for a file copy, the responder can either supply the requestor a copy itself, or supply a list of other replicas to query in turn (e.g., if it is overloaded). In the latter case, the querying Kserver sends its request to one of these sites based on its perceived connection quality to them (e.g., by contacting nodes with high connection quality first). As a side-effect, the supplier Kserver adds the requestor to its list of child copy sites and the requestor makes the supplier its parent. This replica creation mechanism dynamically forms a hierarchical network of replicas rooted at the file's home node, similar to the dynamic hierarchical caching introduced by Blaze [1]. Figure 1 illustrates one such network. This network tends to be organized such that nearby nodes are directly connected to each other. The fanout of any node in the hierarchy (i.e., number of child copy sites) is limited by the load-handling capacity of that node or explicitly by the administrator. When a link or node in the hierarchy goes down, the orphaned nodes try to attach themselves to the hierarchy again, starting at a known copy site (home node by default), effectively repairing the replica hierarchy. A Kserver is free to delete locally cached copies of files to reclaim local store space at any time, after propagating their updates to other copy sites.

Khazana Interface

Khazana exports the following operations to its clients via its library interface:

`kh_alloc(page_size, other_attributes):` In response to this request, a Kserver allocates a region with given page size and attributes (described below), and returns its KID.

`kh_free(KID):` This operation destroys all the copies of the region, frees the region's contents and makes the KID available for reallocation.

`kh_getattr, kh_setattr(KID, region_attr):` This operation can be used to set various region attributes such as region size, consistency policy and various consistency control parameters on the region as a whole. It affects all replicas. We describe the various consistency options and control parameters that we support in Section 3.2.

`kh_open(KID, offset, size, mode):` This starts an access session for the given portion of a region in the specified mode, and returns a session ID. A *size* of zero indicates whole-file access. In response to an open request, the receiving Kserver obtains a copy of the requested portion of the region and performs access and concurrency control cbecks. We support traditional Unix open modes including file locks for isolation from other sessions. If required by the region's consistency policy, the CM brings the local copy up-to-date. A session is the point of concurrency control and isolation in Khazana. In the future we plan to add atomicity guarantees as an option.

`kh_read, kh_write(session_id, offset,` size, data): In response to these calls, region data is transferred to/from client buffers. These calls do not trigger any further consistency checks unless the user requests a per-access check explicitly via the snoop primitive described below. We support a rudimentary plugin mechanism to enable application writers to make *operational updates* to the data before it is returned/propagated, as described below.

`kh_snoop(session_id,cm_opt,opt_parm,` `update_callback):` This operation is used to specify the caller's desired session-specific consistency requirements. Choices include per-access consistency checks, asynchronous update notification, and the ability to control client's data quality. This call can be used by clients to receive remote updates asynchronously, i.e., to "snoop" on modifications of the data.

`kh_close(session_id):` This call terminates the session. Khazana propagates updates if required by the region's consistency policy.

3.2 Consistency Management

Khazana's consistency management (CM) subsystem is responsible for maintaining coherence among file copies created as described above. Khazana provides user choice over a set of consistency policies on a per-file basis, and over the various mechanisms to be used in implementing these policies. All the consistency schemes utilize a common set of underlying update propagation mechanisms that operate via the replica network. These mechanisms only affect performance, not the guarantees provided by the consistency policy. The decision on when to allow read/write access and propagate updates thus depends on the choice of consistency policy and the implementation strategy. In contrast to our approach, other existing systems (e.g., WebFS [25] and Fluid replication [4]) bundle these mechanisms as part of the implementation of each individual policy, and do not let applications choose among them.

Supported Consistency Policies

Khazana supports the following consistency policies on a per-file basis. None of these policies is new, nor is our design restricted to use only these policies. All replicas of a region enforce the same consistency policy for a region, in contrast to systems like Fluid replication [4] that enable different clients to see the same file at different consistency levels.

Strict pessimistic consistency: Only one writer is allowed at any time. A reader always gets the latest value written. This implies that in a partitioned network, access is disallowed in all but the partition that holds write privileges. We also implement a variation that allows read-write access in one partition, but read-only access in others.

Optimistic last-writer-wins consistency: Reads and writes are allowed on any replica, any time [4]. Updates are propagated based on control parameters. Write conflicts are resolved along the way to the root replica in the hierarchy. The write that reaches the root node of the copy hierarchy last wins and is propagated back to all nodes.

Append-only consistency: Appends are allowed on any replica, any time. They are never lost. Append conflicts (concurrent appends to the same version of a file) are resolved by applying both appends in the order they arrive at a common parent node in the replica hierarchy. This is implemented using optimistic consistency and treating appends as operational updates (see below) handled by an append plugin.

Supported Consistency Control Parameters

We provide three ways for applications to guide consistency implementation at a given replica site. Some of these parameters are settable on a per-region basis using *kh_setattr()*. Some can be set on a specific replica for the duration of a session using the *kh_snoop()* and affect only that replica.

Update transfer model: We provide both *single-master* and *multi-master* updates. In contrast, Blaze [1] supports only single-master updates.

Direction of update transfer: We offer *push-based* and *pull-based* updates.

Update transfer condition: We offer several update transfer options on a per-replica basis: *user-initiated*, *timer-driven* (with specified timeout), *topology-driven*, *eager* (as soon as an update arrives) and *lazy* (upon access request). We also offer two options based on different notions of staleness: maximum time elapsed since last change and number of unpropagated updates tolerated.

For instance, with single-master updates, the optimistic consistency scheme provides traditional Unix file semantics whereas pessimistic consistency provides strict database-style atomicity and isolation guarantees. The pessimistic scheme with multi-master updates provides Munin-style release consistency. The multi-master optimistic scheme with pull-based updates is equivalent to AFS semantics.

Each of our targeted applications can work with multiple combinations of the above options. However, some combinations perform better than others, as will be discussed in Section 4.

Operational Updates

Khazana distinguishes between absolute updates, which overwrite previous ones, and operational updates, which logically are "applied" to the data and thus need to be logged. For example, an operational update might specify "Add 1 to the data element at offset 1000". Operational updates can greatly reduce the amount of data transferred to maintain consistency and increase the amount of parallelism achievable in certain circumstances, e.g., updating a shared object or database. They can also be used to implement RPC-like services through the shared data store.

To support operational updates, code must be provided to interpret and apply the operation on the local copy of a region. We currently use simple plugins, implemented via callbacks, to handle operation updates with low overhead.

One way that we currently use operational updates is to give kservers the ability to control the quality of data sent to clients based on their resource constraints.

For instance, when a PDA user needs to access an image stored in Khazana via a nearby Kserver, the kserver can invoke a local plugin just before transmitting the file to tailor its quality to the resource limitations of the client. This does not affect consistency semantics, but can significantly improve the bandwidth utilization over the last hop to such clients. Similarly, a multimedia streaming service can maintain real-time stream delivery rates by dropping and/or transcoding data to fit the client's bandwidth limit. We demonstrate the value of operational updates and plugins in Section 4.

Update Propagation

Like Bayou [5], each replica maintains a small amount of versioning state based on node-local version numbers (common to all consistency policies) for each neighboring replica. This version state is used to identify when updates need to be propagated and detect update conflicts, Due to space limitations we refer the reader to a more detailed document describing our version management mechanism [24]. A per-Kserver log is used to store operational updates. Log entries are discarded once they have been propagated to all neighbors and registered clients.

In addition to the versioning state, a replica site also maintains the state of ongoing local sessions and any asynchronous update registrations, which are non-persistent. The amount of persistent state to be maintained at a replica site is thus proportional to the number of other replicas with which it directly communicates (i.e., the node's degree in the replica hierarchy), not the total number of replicas. This, coupled with the fact that replicas are restricted to communicate only along the hierarchy, ensures scalability to a large number of replicas. In contrast, systems such as Ficus [17] and Bayou [5] that employ more peer-to-peer replica topologies have to maintain state that is proportional to the total number of replicas, theoretically limiting their scalability.

The consistency manager at a replica site gets activated just before opening a session to perform admission control and to bring the local copy up-to-date for access, and just before closing the session to propagate updates. A replica is said to "commit" its local updates when it successfully sends them to its parent. How often and under what conditions a replica propagates local updates to its neighboring replicas (parent and children) depends on the consistency policy and the control parameters set for the replica. However, all the policies we implement use common underlying mechanisms for update propagation.

Once a child replica's updates have been accepted by its parent, the parent treats those updates as its own for the purpose of propagating them to other replicas. The child should not propagate those updates to any other new parent that it contacts later (e.g., due to disconnection and re-attachment to replica hierarchy) to avoid duplicate updates.

Reconnecting to the Replica Hierarchy

When a replica R loses connection to its parent and needs to re-attach itself to the copy hierarchy for a given region, it locates an accessible replica P. It then sets P as its parent, obtains a fresh copy from P (along with its version number) and re-applies any local uncommitted operational updates to this copy. If R simply propagated its local updates to P without resynchronizing, it might propagate a duplicate update.

Handling Update Conflicts

An update conflict arises when two replicas are independently modified, starting from the same version. Pessimistic locking-based consistency schemes prevent such conflicts by delaying conflicting operations, whereas optimistic schemes allow them, but employ versioning to detect and resolve conflicts. In our update propagation scheme, a conflict can be detected either while committing a child replica's updates at its parent, or when applying a parent's committed updates at a child replica site.

The action to take when a conflict is detected is consistency policy-specific. Conflict resolution can either require manual intervention, rely on resolution rules enforceable by the system itself (e.g., last-writer wins or a merge procedure), or need application-specific techniques. The rule-based approach is simple to implement, but may not successfully resolve all conflicts. For the last-writer-wins optimistic policy, incoming updates are incorporated into the local copy. For the append-only consistency policy, the parent's updates are applied first and then the child's.

Controlling Update Propagation

The frequency of propagation of updates from a replica is determined by the consistency policy and the parameters set for the replica. For pull-based update transfers, updates are sent to other replicas only when requested. For pessimistic consistency schemes, updates are issued before allowing read access to a replica, whereas in optimistic schemes, updates are performed periodically at a frequency settable on a per-replica basis.

Implementation of Consistency Policies

We implement the pessimistic strict consistency policy in Khazana as follows. At each replica site, we maintain an access token that indicates what operations (among read, update and append) are allowed on that replica. Strict consistency requires that only one replica hold the update access token at any time. In a single-master implementation, one replica site (the home node in our prototype) always keeps the update tokens and other replicas forward all update requests to this replica. In a multi-master implementation, this write token gets circulated among the replicas based on write requests arriving at their sites. Multiple replicas can hold read tokens, but they are all revoked (via the replica hierarchy) before allowing write access. When a replica site transfers its write token to another replica, it also triggers propagation of its updates to the receiver of the token, via the replica hierarchy.

We implement the optimistic policies (last-writer-wins and append-only) in Khazana as follows. All operations are allowed to proceed immediately on all replicas, after an effort is made to bring the local copy up-to-date (when connected) if updates are pull-based. Neighbors are informed of the intended access at this site. In response, they can supply outstanding updates, but multiple writers can proceed, unlike the pessimistic policy. To avoid polling other sites unnecessarily, a replica site remembers each neighbor's access mode and does not poll them further if they are readers. A subsequent write access notification breaks this trust, forcing a poll the next time updates are needed. This is analogous to the an invalidation message in AFS, but does not prevent access unless the pessimistic policy is chosen.

Updates requests are forwarded to to home node if single-master consistency is specified. When an update is made at a replica, it is propagated to others based on the update transfer condition set at various replica sites. Update conflicts are resolved as described above. Append operations are logged as operational updates.

3.3 Discussion

We described a prototype wide-area data store that we are building. Our design splits consistency management into a collection of core mechanisms that serve as building blocks for composing a variety of consistency schemes to suit specific application needs. Our hierarchical organization of replicas and multi-level update propagation and conflict resolution using the hierarchy limits the amount of state that needs to be maintained on a replica site, making our implementation scalable to a large number of replicas per object.

4 Evaluation

In this section we present our evaluation of Khazana's performance under a variety of data-intensive application scenarios. In particular, we evaluate Khazana's ability to minimize network utilization and response time, maximize application throughput, and meet application-specific requirements of consistency, scalability and availability over a variety of network conditions. To demonstrate the value of Khazana's flexible consistency management framework, we consider three applications with very different consistency requirements: a wide-area distributed file server, a chat server, and a streaming data (e.g., video) server.

4.1 Experimental Setup

For all our experiments, we used the University of Utah's Emulab Network Testbed [6]. The testbed provides a collection of PCs and the ability to specify a wide variety of interconnect topologies, latencies, bandwidths and loss rates of various links. Each of the PCs we used have an 850Mhz P3 CPU with 256MB of RAM running FreeBSD 4.3. They are manipulated via a 100Mbps control LAN isolated from the experimental network.

Kservers run as user-level processes and store their regions in the local file system as files named by their KID. For the chat and streaming experiments, we run the clients on different machines than the servers to avoid CPU contention. For the file system, we colocate each client with a kserver. The client's Khazana library interacts with the local kserver via RPCs, and exchanges data via references to the local files instead of transferring data over the socket. In this instance, the kserver is responsible for maintaining a consistent persistent cache of locally accessed shared files. The clients and servers log experimental output to an NFS server over the control network. We restrict data logging to the beginning and end of an experiment to minimize interference.

Next, we describe each of the experiments and their results in detail.

4.2 File Service

For our first set of experiments, we built a distributed file system on top of Khazana (kidfs) that uses Khazana to enable local persistent caching of shared files. We vary the degree of file sharing as well as the quality of the network between the clients, and compare kidfs's performance with that of NFS in a similar setting.

Kidfs is built as a naming layer on top of the flat file system that Khazana exports, with directories implemented in Khazana regions. Kidfs runs as a wrapper within a kserver process and exports itself to the local VFS interface as a Coda file system by interacting with the Coda kernel module via a device using the Venus-kernel interface. Khazana clients can access

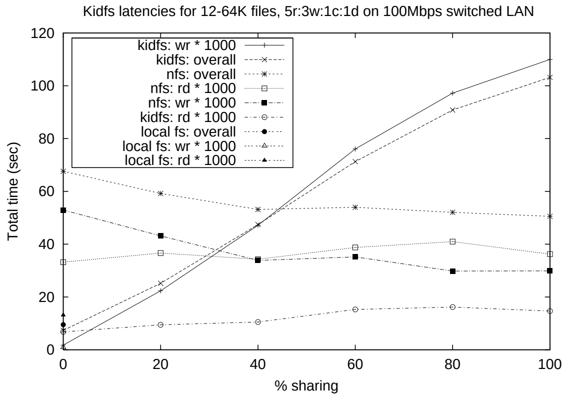

Figure 2. Kidfs vs. NFS latencies

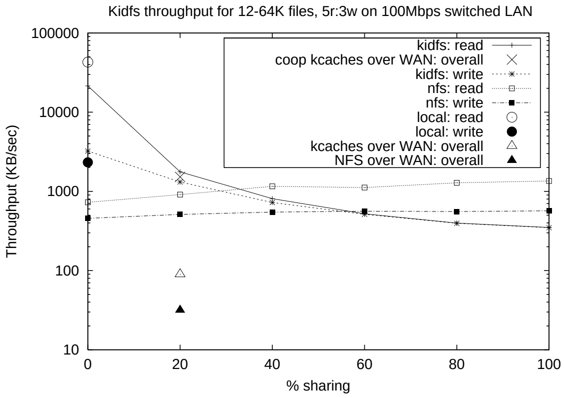

Figure 3. Kidfs vs. NFS throughput

kidfs' Khazana-specific functionality via its pioctl() interface. The kernel interacts with kidfs at file open and close times, and data is supplied via a local file just like with Coda. For the following experiments, however, the file system clients are linked with a special library that interacts with the local Kidfs server directly via IPC messages, while still accessing kidfs file cache contents via the local file system.

Our first experiment evaluates Khazana's suitability for traditional file access under various degrees of contention for shared files on a 100Mbps switched LAN. We set up a kserver and a client on each of 8 machines. We create a shared directory with 108 subdirectories. In each directory we store up to one file at a time, thus limiting the maximum number of files to 108 and inducing sharing. Creating files in subdirectories also removes contention for a single shared directory among clients. Each client considers 1/8th of these directories as private space and the other directories as shared space. Each client performs 1000 random file operations as follows. For each operation, it chooses either a private or global directory based on a specified sharing percentage P (it chooses amongst all directories P% of the time). If the directory is empty, it creates a new file of a random size in that subdirectory. Otherwise it selects a random operation to perform on the directory with the following distribution: 10% creates, 10% deletes, 50% reads and 30% writes. This distribution creates a modest amount write-sharing. With P = 0, each client accesses only private files, With P = 100, there can be substantial file sharing. We detect stale data during reads by tiling a file with its size value as pattern and matching the file size against this, in addition to noting stale NFS file handle errors. We use optimistic consistency with multi-master pull-based updates for all files in this experiment.

Figure 2 shows the average of observed overall runtime for 1000 file operations at each client, and the read

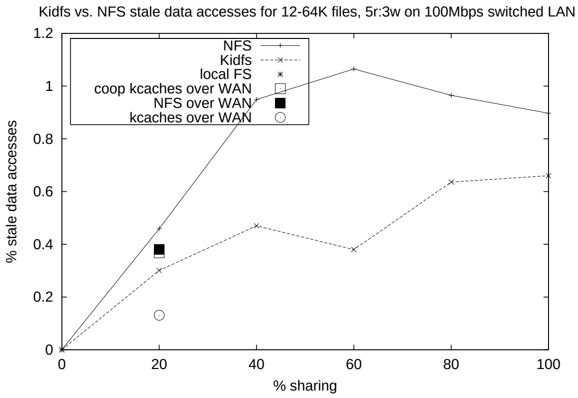

Figure 4. Stale data accesses in Kidfs vs. NFS

and write latencies per operation (in milliseconds but scaled up by 1000 to show them in the same graph). Figure 3 shows the read and write throughput in KB/s for each of NFS, Kidfs and the local file system. Below P=40, Kidfs provides latencies and bandwidths very close to the local file system. As P (and the amount of file sharing) increases, write invalidation message traffic increases in Kidfs, increasing its latency to more than twice that of NFS. This is because Kidfs provides a higher degree of coherence - NFS never sends invalidates to clients (sacrificing consistency). As shown in Figure 4, NFS suffers significantly more stale data accesses than Kidfs at modest sharing levels. However at 100 percent sharing, the loose consistency of the optimistic protocol is exposed. However, kidfs's read latency is much lower than NFS due to its persistent whole file caching.

Unlike NFS, kidfs persistently caches files near client nodes, Unlike NFS or AFS, kidfs supports cooperative client caching. We evaluated the affect of a high latency client-server link on kidfs under two conditions - with and without cooperative client caching - and compared it to NFS over the same links.

In this experiment, we ran a Khazana file server across a 1Mbps (40ms roundtrip, e.g., T1) link from a LAN of kservers serving as file caches (i.e., they cannot create files locally). We repeated the above file access experiments at 20% sharing and a read/write ratio of 6:1 (a typical ratio for read-mostly files). Even without cooperative caching, kidfs's whole-file caching and invalidation-based protocol helps it achieve median read latencies close to that of a LAN (i.e. about 1.89 milliseconds) while its average latency is around 0.54 seconds. In contrast, NFS provides a median latency of 1.1 seconds and an average of 1.4 seconds. However, the worst-case latency in kidfs is higher than NFS due to a design decision we made in our prototype - to hold off access to a file until the up to date version of the entire file is brought into the kserver's cache. With cooperative client (kserver) caching enabled and all clients on the same LAN, kidfs provides near LAN read latency (averaging 27 ms) while NFS remains worse. This experiment reinforces the value of cooperative multi-level caching of files in hiding WAN latencies (already demonstrated by Blaze [1]).

4.3 Internet Chat Service

Internet chat is an example of a many-to-many data exchange system involving N parties, each broadcasting data and listening to each others' data. Other services with similar data sharing characteristics include interactive voice- or video-conferencing, distributed whiteboards, internet games, group communication among distributed sensors for coordination, distributed data/event logging, and some publish-subscribe applications. Interactive applications need low end-to-end propagation delays, whereas other services need high throughput communication between a large number of endpoints with quick data posting rate at the senders. Such services are typically supported via special-purpose message passing protocols.

Although Khazana is not designed as a message-passing system, its update propagation mechanism is scalable and adaptable to handle data delivery over a variety of network speeds. Because caching is a fundamental part of Khazana, it can stage data at intermediate points along the delivery path, thereby improving network utilization and improving the end-to-end experience by amortizing variations in network quality. However, Khazana's data sharing-oriented design

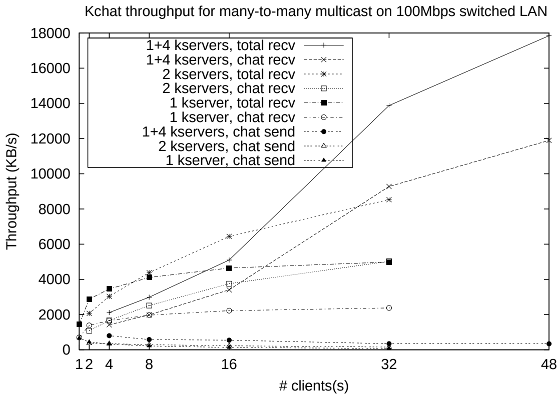

Figure 5. Kchat aggregate throughput over 100 Mbps switched LAN

incurs some inherent bookkeeping overhead relative to raw message-passing primitives such as IP multicast.

To evaluate Khazana's ability to support this type of application, we develop an Internet chat benchmark called `kchat`. The kchat "server" creates a region on a nearby kserver and specifies that it be kept consistent using multi-master optimistic consistency with eager push-based update propagation. Chat clients connect to a chat stream by opening this region in append mode. For this benchmark, each client sends 10,000 chat "messages" as fast as it can by performing 10,000 back-to-back one-way write calls, without waiting for acks from the server between writes. Each write is small (88 bytes), which maximizes server processing overhead. When a kserver receives an update packet, it appends the message to the local copy of the region and forwards it to other clients or replica sites as part of push-based consistency management. In effect, the Khazana caching network acts as a multicast tree, with each kserver acting as a multicast hub. It takes about 17 microseconds to store and apply an incoming packet and an additional 30 microseconds to send up to 20 packets in a single message to another client, excluding network transfer time.

Each client snoops its local kserver for updates and registers a callback routine with its local Khazana client library to be invoked when updates arrive. Clients stop snooping and close the session when they get a special sentinel packet, at which time they log the observed throughput, roundtrip time for their own packets, and total receive time. Khazana introduces 70 bytes of payload overhead for each update message sent to a client, but aggregates multiple updates into a single message when possible to reduce this overhead.

We first evaluate the throughput achievable by Khazana in a LAN, since the high bandwidth available on a

LAN will expose high server overheads. We varied the number of clients from 1 to 48, and considered systems with 1, 2, and 5 kservers on a 100Mbps switched LAN. The clients and servers were spread over 32 machines, given the resource availability limits on our testbed at the time. We measured the combined aggregate send and receive throughput observed by all the clients and plotted the results against the number of clients in Figure 5.

We present two metrics, the delivered throughput in terms of both raw bytes (including overhead bytes) and useful application data. Normally, a kserver sends out a packet as soon as it arrives to minimize propagation delay, which results in the useful throughput being roughly one-half the raw throughput (70 bytes of overhead per 80 bytes of application data). However, if messages become queued up at the server due to processing overhead, the kserver automatically combines multiple updates in a single message. Aggregation is more apparent in the multi-server case, because the server processing overhead for propagating updates to another server is about 70% higher per message.

On a LAN, this application makes the server CPU-bound. Thus the server's processing speed limits the client-perceived throughput. As a result, the aggregate receive throughput flattens out at about 5MB/sec beyond 16 clients, despite the 12.8 MB/sec link bandwidth per client-server connection . The sustained rate at which a client can send writes to the file also tapers off due to server processing delay. However, by adding one more server and redirecting half the clients to it, kchat can support twice as many clients and deliver twice the aggregate bandwidth.

For our second experiment, we employed a two-tier chat "server" hierarchy, with a root kserver serving 4 other kservers, each of which served an equal share of the clients. In this case, the extra per-message hop overhead keeps throughput low when the number of clients is small. However, the service is able to sustain high throughput (about 340 KB/sec) for a large number of clients due to efficient inter-server update propagation.

Figure 6 shows the individual raw receive throughput observed by up to 16 clients for various client-server configurations over a WAN. We first ran kchat between a LAN of 1, 2, 4 and 8 clients served by 1 kserver (S1). where the clients and server were separated by a 20Mbps, 40ms link (the observed characteristics of Internet-2 between the Univ. of Utah and MIT). In this configuration, the link is saturated by 8 clients, which results in an average per-packet roundtrip time of 7 seconds. When we added a kserver (S2) on the clients' LAN to serve 4 of them, S2's clients' throughput increased by a factor of 2.5 and their average roundtrip time dropped to 0.6 seconds. S1's client performance

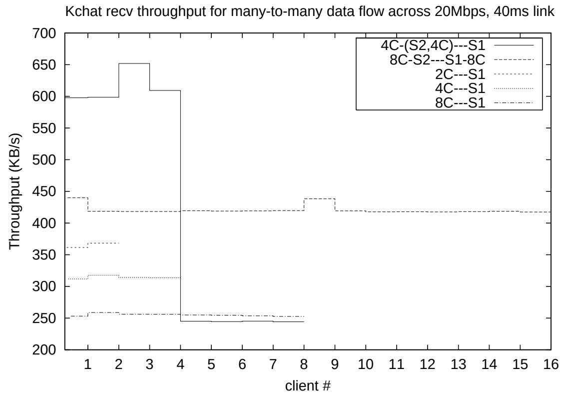

Figure 6. Kchat throughput across 20Mbps, 40ms link

dropped only by 3% due to the extra message overhead for inter-server communication between S1 and S2. The aggregate recv throughput increased to 1.4 times the 20Mbps link capacity. When we switched all 8 clients to S2 and added 8 more clients to S1, the aggregate throughput jumped to 2.7 times the link bandwidth, the average roundtrip time dropped to 1.8 secs, and all clients observed uniform throughput. One advantage of a Khazana-based solution to many-to-many data flow is that adding a proxy server to handle more clients or redirecting clients to a different server are very simple operations and can be done transparently during runtime.

4.4 Streaming Multimedia Service

Multimedia streaming is an example of a data dissemination service that needs to propagate data to a large set of clients with different receive capabilities over variable capacity links. Data dissemination differs from real-time data exchange (e.g., chat) in that its data is typically updated by only one entity, but monitored by a large number of clients with varying consistency requirements and tolerance to data staleness. For this class of services, client and network control over data quality and bandwidth utilization is an important consideration. We shall show how data flows with distinct delivery rates over channels with different capabilities can be achieved with little effort by the service author.

We built a data streaming application called `kstreams` to demonstrate customizable data streaming using Khazana servers as streaming agents. This experiment runs on the testbed topology depicted in Figure 1. This topology reflects a typical Internet scenario where a website with a moderately powerful Internet connection broadcasts a pre-recorded audio or video presentation on demand to registered listeners spread over the wide-area. Among the clients listening to it are a couple of home desktop users, University campus LAN users, 4 with individual 1Mbps lines, and 2 ISDN and 2 DSL subscribers. The bandwidths and latencies we chose are based on our observation of available network characteristics between various types of end hosts on the Internet on a typical day.

We start a kserver on the source website and a kserver on the University machine to serve as a proxy stream server for nearby clients. The ISPs also run proxy stream servers to cater to their local clients. When starting a kserver, a system administrator can specify for each file (stream), the maximum number of other replicas that it can serve based on the available bandwidth and server capacity. In this experiment, our root kserver restricts the number of replicas served to one, the Univ sets it to three, and the ISPs set it to zero.

To send stream data, the stream originator opens a stream file and appends to it packets representing video frames. To subscribe to the stream, clients need only determine the stream's KID, which we assume is done via an external mechanism. They then open the file and snoop on it as in kchat. With a snoop request, a client can specify a variety of criteria that control the quality of data they get, such as how stale data can be either in terms of time (e.g., within one second) or number of updates (e.g., no more than 100 frames old). One particular criterion of interest for multimedia users is the tolerable loss rate of updates. To limit the bandwidth consumed by a stream or the observed time lag of stream packets, a bandwidth-limited client can specify that intermediate frames (update packets) be dropped when a link is loaded.

Our prototype currently allows a client to specify with a snoop request that if the number of unsent update packets queued at the server for delivery to them increases beyond a number (N), the server should skip sending the packets to it at a given rate (R). This is not the same as having the server drop packets with rate R, because the server may be serving multiple clients. Setting N to zero results in all packets that would be queued being dropped, whereas a greater value causes packets to be skipped only in the presence of a given level of congestion.

In our streaming experiment, subscribers attached to low bandwidth connections (128Kbps and 640Kbps) use this selective "skip updates" feature to reduce the observed delay between when packets are sent by the source and received at the client.

Figure 7 shows the delay between when packets are sent and received for various clients. The gap between the ideal latency through kservers and the start lag time shows the delay introduced by kserver processing. As

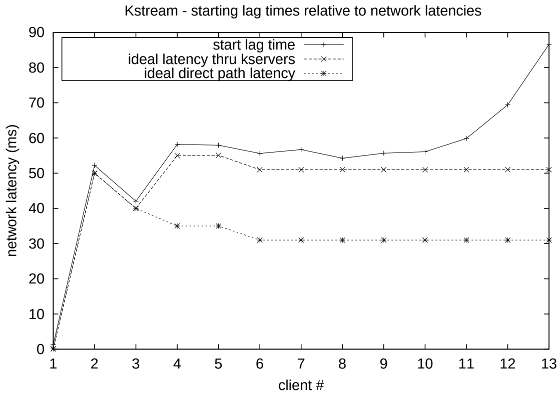

Figure 7. Kstream - start lag times

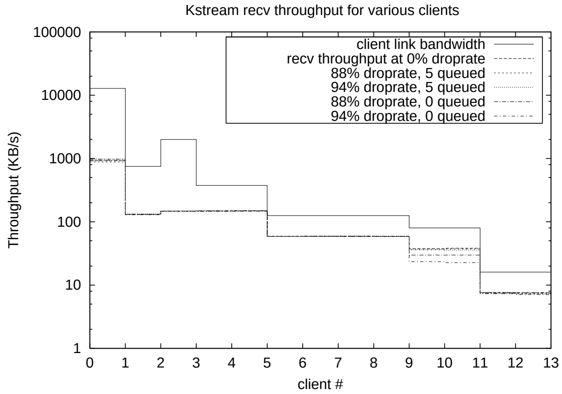

Figure 8. Kstream receive throughput

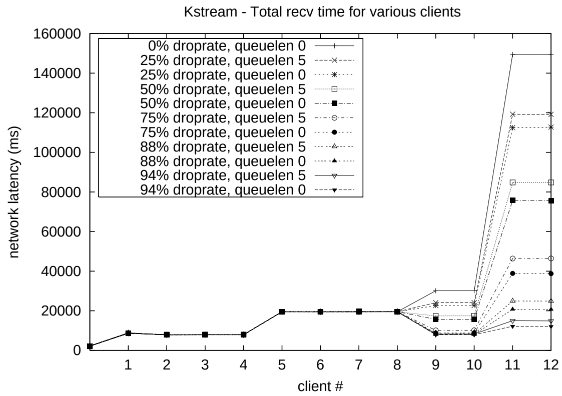

Figure 9. Kstream total receive time

can be seen, kserver processing times add about 2-5ms latency when bandwidth is plentiful, and a bit more when bandwidth is tight. The increasing delay is caused by queueing delays immediately in front of the low bandwidth client.

Queueing delays lead to poor user experiences in interactive multimedia applications, because it leads to lag and jitter. Selectively dropping packets can reduce these effects.

Figure 8 shows the perceived receive throughput of various clients relative to the bandwidth of their link to the network at various selected drop rates. Clients connected to 1Mbps and slower links are saturated by the update stream. In response, these clients can request that a fraction of the packets destined to them be "dropped", either when there are packets buffered (N=5) or always (N=0). For the case where 88% of packets sent to the 640Kbps clients are dropped, whether queuing or not, link bandwidth utilization drops by 160Kbps. Delivered bandwidth drops by 250Kbps when 94% of packets are discarded. However, when only queued packets are dropped (N=5), delivered bandwidth drops only slightly for either an 88% or 94% drop rate. Dropping packets has little impact on the bandwidth delivered to the 128Kbps clients, because the kserver at their ISP receives data at a much higher rate (i.e., 145 KB/s) than their link speed. This indicates that clients should ask servers to only drop packets when it observes some degree of queueing delay.

Dropping packets has a dramatic impact on the time it takes to deliver a stream to a low bandwidth client, as seen in Figure 9. It does so by providing a lower fidelity (fewer frames/second) version of the stream to the client. This dramatic decrease in lag is particularly important for interactive situations, e.g., teleconferencing. A more intelligent packet filter could use transcoding to reduce the bandwidth requirements of the stream by reducing the quality of the data (e.g., frame resolution) before forwarding it across a low bandwidth link. Transcoding could be easily incorporated into a multimedia delivery system built upon Khazana using its flexible plugin filter mechanism. Khazana's consistency management logic is not jeopardized by whatever filtering decisions the packet filter makes. Since the kservers would be an integrated part of the stream delivery mechanism, allowing the service provided to modify the data filter would introduce no security problems beyond those typical for servlets in webservers.

Discussion

The three applications achieve different individual goals due to the reusability and flexibility provided by Khazana. None of them need to implement any logic required for consistency management over variable speed links, thus simplifying their implementation considerably. We believe these results validate our research hypothesis that a consistency management system that provides a small set of customizable coherence mechanisms can efficiently satisfy the data coherence needs of a wide variety of distributed services. Such a system provides a sound basis for simplifying the construction of replicated internet services that need to operate in a variable quality network environment.

5 Related Work

Providing coherent shared data access across variable quality networks has been extensively dealt by work. However, previous solutions are aimed at particular application domains [13, 5], adopt a monolithic design unsuitable for reuse [17, 15], or lack customizability for efficiency [4, 25].

Flexibility in consistency management has often taken the form of supporting multiple selectable consistency options for application data. Munin [3] provides flexible/tunable consistency policies in a LAN environment for distributed shared memory-based parallel programming. WebFS [25] provides a useful set of coherence policies for wide-area file access and update multicasts. Fluid replication [4] provides intelligent data placement and multiple consistency policies for mobile file access across the wide-area. TACT [27] provides optimistic consistency with application-specified notion of coherence in a wide-area environment. The flexibility in consistency options provided by these systems enables their reuse for a variety of applications. However, these systems are limited in the hooks they provide to let an application customize the implementation of their policies. As seen in our evaluation, the customizability of the underlying implementation (e.g., pull- vs. push-based update propagation) is important for the performance of some applications. Moreover, these systems do not adequately address the issues of dealing with variable network latencies and/or scalability (to a large set of clients) that are typical in the wide area.

Providing data coherence across the wide-area while dealing with variable network latencies, disconnections and partitions is not new. However, most previous work has been done in the context of specific data access patterns. AFS provides last-writer consistency in a wide-area file system with traditional Unix-style file access. Coda [13] provides support for weakly connected and disconnected operation in a campus environment employing a client-server architecture. Ficus [17] provides replicated file access while handling intermittent connectivity in a more peer-to-peer setting. Bayou [5] provides optimistic consistency for a replicated database in a peer-to-peer mobile environment with highly intermittent connectivity and unstructured communication patterns. It requires application-specific conflict resolution procedures to be supplied, and replicas can take unbounded time to synchronize In contrast, we employ a hierarchical replica topology to achieve more bounded synchronization. Blaze's PhD thesis [1] showed the value of constructing dynamic per-file cache hierarchies to support efficient large-scale file sharing and to reduce server load in distributed file systems. We leverage techniques developed by these systems in a more general-purpose consistency management framework for the benefit of a wider variety of applications.

Numerous consistency options have been developed individually to handle the data coherence needs of specific services such as file systems, directory services [15], databases and persistent object systems [14, 8, 26]. For example, the OceanStore [7] project aims to provide a secure, global scale persistent data utility. Objects in OceanStore are immutable and consistency is maintained based on versioning instead of in-place updates. They employ update procedures to apply and propagate updates among replicas, similar to Bayou. Many algorithms have been proposed to maintain consistency among data replicas for fault-tolerance in the face of network partitions (e.g., Deno [12]). Saito presents a comprehensive survey [21] of optimistic consistency schemes with a taxonomy for classifying them based on their implementation choices. We extend his taxonomy to include pessimistic algorithms and used it to derive a set of parameters to provide flexible consistency management.

The growing popularity of Internet-based file sharing (e.g., Napster) has prompted research into peer-to-peer organization of distributed applications. Much of this work has focussed on anonymous information dissemination (e.g., Freenet) and/or immutable deep archival storage among peers. These systems do not typically support concurrent updates to replicated data. Pastry [18] and Chord [22] provide efficient object location and request routing across the wide-area based on hashed object IDs. PAST [18] and Farsite [2] provide cooperative file storage and retrieval, but do not address consistency among multiple replicas in the face of updates. Our consistency management work is complementary to these systems, as it can add consistent update functionality to them, making their benefits applicable for a wider range of services.

The application classes that we target have been explored individually by different systems. In particular, applications based on wide-area data dissemination and exchange currently employ message-passing using some form of multicasting support such as MBone or IP multicast. Our approach to achieve data multicast as a side-effect of update propagation among data replicas, though not as efficient as these highly tuned solutions, has several benefits. It allows intermediate data staging along the multicast path to cache data closer to its clients. It enables the quality of multicast-ed data and frequency of update transfer to be customized at intermediate hops, unlike IP multicast, where data quality is controlled only at the sender. Scribe [19] and Bayeux [29] construct multicast trees for efficient data/event dissemination among data replicas, by virtue of being built on top of topology-aware data location and routing schemes such as Pastry and Tapestry (respectively). Our focus is on making use of a multicast tree efficiently once it is built by staging data along the tree nodes.

6 Conclusions

In this paper we demonstrated the feasibility and value of implementing a wide area storage system flexible enough to serve the data access and delivery needs of a variety of wide-area applications. We show that behind the diverse consistency management needs of a variety of wide-area internet services there lie commonalities that can be met in a reusable yet efficient manner. In particular, we observed that certain key issues often recur in the design of shared data management in a variety of applications. They are: when access to a replica is allowed (e.g., optimistic vs. pessimistic access control), where updates are issued (e.g., single-master vs. multi-master), how updates are initiated (e.g., pull-based vs. push-based), and the conditions that can trigger an update transfer (e.g., timer-driven, staleness-driven, resource-driven, eager, or lazy transfer).

Khazana is a flexible wide area data store designed to support the data consistency needs of a wide variety of distributed applications. It does so by providing a scalable caching network and giving its clients control over how the network keeps data consistent. Khazana exports a simple file-like abstraction, but provides a variety of hooks to control the caching, replication, and consistency mechanisms used to manage data. We evaluate Khazana in the context of three diverse distributed applications: a file system, a chat server, and a streaming system. Despite the very different data management requirements of these three applications, Khazana is able to support efficient implementations in both LAN and WAN environments.

Khazana provides several benefits over existing systems. It enables reuse of consistency mechanisms for a variety of applications with coexistence of different consistency schemes in the same system. It enables customization of implementation for performance. It helps achieve scalable data delivery across the wide-area. Finally, it conserves bandwidth by enabling applications to customize data quality during update propagation.

References

[1] M. Blaze. *Caching in Large Scale Distributed File Systems*. PhD thesis, Princeton University, 1993.

[2] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the Int'l Conference on Measurement and Modeling of Computer Systems (Sigmetrics '01)*, pages 34–43, 2001.

[3] J. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, Sept. 1995.

[4] L. Cox and B. Noble. Fast reconciliations in Fluid Replication. In *Proceedings of the 21st International Conference on Distributed Conputing Systems*, Apr. 2001.

[5] 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*, Dec. 1994.

[6] Emulab. `http://www.emulab.net/`, 2001.

[7] J. K. et al. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

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

[9] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *5th International Web Caching and Content Delivery Workshop*, May 2000.

[10] A. Goel, C. Pu, and G. Popek. View consistency for optimistic replication. In *17th IEEE Symposium on Reliable Distributed Systems*, Oct. 1998.

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–82, Feb. 1988.

[12] P. Keleher. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, Apr. 1999.

[13] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 213–225, Oct. 1991.

[14] 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.

[15] Microsoft Corp. Windows 2000 server resource kit. Microsoft Press, 2000.

[16] K. Petersen, M. J. Spreitzer, D. B. Terry, M. T. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[17] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the 1994 Summer Usenix Conference*, 1994.

[18] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 2001.

[19] A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. SCRIBE: The design of a large-scale event notification infrastructure. Submitted for publication, June 2001.

[20] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable internet mail service. *ACM Trans. Comput. Syst.*, Aug. 2000.

[21] Y. Saito and M. Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Labs.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Sigcomm '01 Symposium*, August 2001.

[23] S. Susarla. A survey of implementation techniques for distributed applications. `http://www.cs.utah.edu/~sai/papers/app-survey/`, Mar. 2000.

[24] S. Susarla. Flexible consistency management in a wide-area caching data store. `http://www.cs.utah.edu/~sai/papers/proposal/`, Mar. 2001.

[25] A. Vahdat. *Operating System Services For Wide Area Applications*. PhD thesis, University of California, Berkeley, CA, 1998.

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

[27] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, Oct. 2000.

[28] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for wide-area fault-tolerant location and routing. Submitted for publication, 2001.

[29] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh Int'l Workshop on Network and OS Support for Digital Audio and Video*, 2001.