Khazana: A Flexible Wide Area Data Store

Sai Susarla and John Carter
{sai, retrac}@cs.utah.edu

UUCS-03-020

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

October 13, 2003

Abstract

Khazana is a peer-to-peer data service that supports efficient sharing and aggressive caching of mutable data across the wide area while giving clients significant control over replica divergence. Previous work on wide-area replicated services focussed on at most two of the following three properties: aggressive replication, customizable consistency, and generality. In contrast, Khazana provides scalable support for large numbers of replicas while giving applications considerable flexibility in trading off consistency for availability and performance. Its flexibility enables applications to effectively exploit inherent data locality while meeting consistency needs. Khazana exports a file system-like interface with a small set of consistency controls which can be combined to yield a broad spectrum of consistency flavors ranging from strong consistency to best-effort eventual consistency. Khazana servers form failure-resilient dynamic replica hierarchies to manage replicas across variable quality network links. In this report, we outline Khazana's design and show how its flexibility enables three diverse network services built on top of it to meet their individual consistency and performance needs: (i) a wide-area replicated file system that supports serializable writes as well as traditional file sharing across wide area, (ii) an enterprise data service that exploits locality by caching enterprise data closer to end-users while ensuring strong consistency for data integrity, and (iii) a replicated database that reaps order of magnitude gains in throughput by relaxing consistency.

1 Introduction

The development of scalable, high-performance, highly available Internet services remains a daunting task due to lack of reusable system support. Replication is well-understood as a technique to improve service availability and performance. The prevalence of replicable data in distributed applications and the complexity of replication algorithms motivates the need for reusable system support (middleware) for replicated data management. Typical distributed systems hardcode domain-specific assumptions into their consistency management subsystems (e.g., NFS [21], Coda [12], Pangaea [20], and ObjectStore [13]). This design makes them highly efficient for their problem domain, but adapting their consistency management mechanisms for different applications requires significant redesign. In general, Internet services must make tradeoffs between performance, consistency, and availability to meet application requirements [8]. Thus, any distributed data management system designed to support a wide variety of services must allow applications to customize the way replicated data is managed [25].

The ideal wide area data management middleware would have the following three features: *aggressive replication* to support large systems, *customizable consistency management* to enable applications to meet their specific performance and consistency requirements, and sufficient *generality* to support diverse application characteristics efficiently. Existing distributed data management middleware systems [4, 17, 20, 25] lack one or more of these features. Several peer-to-peer systems support aggressive replication of read-only data [9, 17] or rarely write-shared data [20], but such systems are not designed to handle frequent write-sharing. Though several reusable consistency toolkits have been proposed (e.g., Bayou [4] and TACT [25]), there exists no reusable middleware to exploit their power in an aggressively replicated environment.

Khazana is a wide area peer-to-peer data service that supports aggressive replication and highly customizable consistency mechanisms to support a wide variety of scalable distributed services. It exports a simple filesystem-like interface and a carefully chosen set of consistency management options that, when used in various combinations, yield useful consistency semantics on a per-file or per-replica basis. The choice of consistency-related options Khazana provides derives from a detailed survey of the consistency needs of distributed applications [18]. We found that the sharing needs of distributed applications could be described along five dimensions:

- *concurrency* - the degree to which conflicting read/write accesses can be tolerated,
- *consistency* - the degree to which stale data can be tolerated,

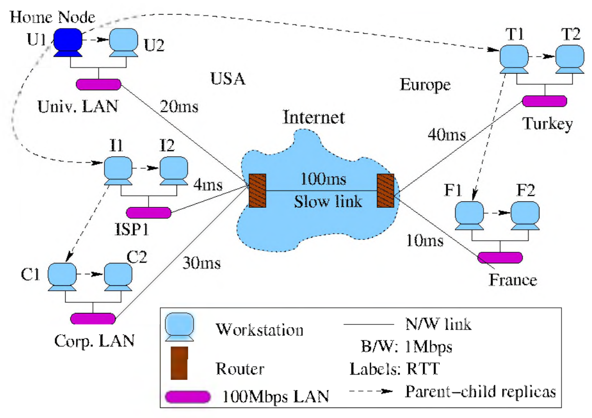

Figure 1: Typical Khazana Replication Topology

- *availability* - how data access should be handled when some replicas are unreachable,
- *visibility* - the time at which local modifications to replicated data must be made visible globally, and
- *isolation* - the time at which remote updates must be made visible locally.

There are multiple reasonable options for each of these issues. Khazana's approach to customizable consistency enables applications to make more precise tradeoffs between consistency, availability, and performance than is possible with other systems.

To achieve scalability and performance in the wide area, the Khazana servers (*Kservers*) caching a particular piece of data organize themselves into a self-repairing dynamic replica hierarchy. This replica hierarchy is used for all consistency-related communication and is organized in a way that attempts to minimize the use of slow links. Figure 1 illustrates a representative Khazana caching network consisting of ten nodes spanning five WAN sites and two continents. Each node accessing a file connects to the "nearest" node in the existing replica hierarchy, e.g., F1 connects via T1 to avoid using the intercontinental link.

In this paper, we outline Khazana's design, and show its value in replicating three different network-based services with diverse needs and data access characteristics. First, we show that *KidFS*, a distributed file system built on top of Khazana, can exploit locality more effectively than existing file systems, while supporting a broader variety of file usage modes with diverse consistency requirements. Second, we show that *Kobj*, an enterprise object proxy built on top of Khazana, can significantly improve its responsiveness to wide-area clients via intelligent caching. We demonstrate that Khazana's adaptive caching algorithm makes good decisions about when and where to replicate data, so that data with good locality tends to be aggressively migrated or replicated, while data with poor locality or heavy write sharing tends to be managed via a more efficient centralized mechanism. We further show how building Kobj on top of Khazana enables it to automatically adapt to shifts in data locality and sudden periods heavy of contention. Finally, we demonstrate how *Kdb*, a replicated version of the BerkeleyDB built on top of Khazana, performs when using consistency requirements ranging from strong (appropriate for a conventional database) to time-based (appropriate for many directory services). Our results show that relaxing consistency requirements even slightly can significantly improve throughput.

The goal of this paper is to motivate the value of Khazana's flexible consistency mechanisms and demonstrate the feasibility of building a shared data management middleware layer to support a broad variety of distributed services. As such, we do not present Khazana's failure resiliency mechanisms in detail, nor do we propose a particularly sophisticated distributed naming mechanism. In Section 2 we describe Khazana's design and implementation, including its flexible consistency mechanisms. In Section 3 we present the three experiments outlined above. Finally, in Sections 4 and 5 we compare Khazana with related systems and conclude.

2 Khazana Design

In this section, we briefly describe the Khazana prototype. We start with an overview of Khazana's basic organization in Section 2.1. In Section 2.2 we describe the various consistency options that Khazana supports, including some particular sets of options that are commonly useful. Finally, in Sectionsection:khazana-discussion summarize our discussion of Khazana's design.

2.1 Overview

Khazana provides coherent wide area file access at a page granularity. A typical distributed service can use Khazana to store its shared persistent state and specify consistency-related attributes for this state on a per-file basis. Khazana clients (Kclient) access shared data by contacting any nearby Khazana server (Kserver). Kservers cooperate to locate and cache data close to where they are used while maintaining consistency according to the specified attributes. 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 discover each other through 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 in forming efficient replica networks. 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.

Khazana manages flat arrays of bytes called (interchangeably) *files* or *regions*. Khazana can maintain consistency at the granularity of a 4-kilobyte page (when using physical updates) or the entire file (when using operational updates). Khazana clients can control how the consistency of each region is maintained per-file (affecting all replicas), per-replica (affecting its local sessions), or per-session (affecting only one session). A file's permanent copy site is called its *home node*. Replicas are created at other Kservers as a side-effect of local access.

Khazana exports a file system-like interface that supports traditional read/write as well as operational updates on files. Specifically, Khazana exports the following operations to its clients via the Kclient library. Kclients internally communicate with Kservers using socket-based IPC.

```
KID ← kh_alloc(attributes)
sid ← kh_open(KID, off, sz, mode)
kh_read/write(sid, off, sz, buf)
kh_snoop(sid, opts, callback_func)
kh_close(sid)
kh_free(KID)
kh_getattr/setattr(KID,attr)
```

`kh_alloc()` creates a new file with the specified consistency attributes and returns a unique Khazana ID (KID). `kh_open()` starts an access session for a portion (perhaps all) of the specified file, returning a session id. Depending on the file's consistency attributes and the requested access mode, the Kserver may need to obtain a copy of the data and/or perform operations needed to bring the local copy up to date. A *session* is Khazana's point of concurrency control and isolation. `kh_read()` and `kh_write()` transfer file contents to/from a user buffer. Depending on the consistency requirements, these operations may need to perform consistency operations. `kh_snoop()` can be used by clients to register to receive remote updates via a registered callback function, i.e., to "snoop" on updates arriving for the local replica. It is used to support operational updates, described below. `kh_close()` terminates the current session. `kh_getattr()` and `kh_setattr()` let the client inspect or modify region attributes, such as the consistency requirements. Finally, `kh_free()` deletes the specified file from Khazana.

File Naming and Location Tracking

Khazana files are named by unique numeric IDs (called KIDs). Because file location is not a focus of our work, and has well known scalable solutions [17, 23], we employ the following simple scheme for assigning KIDs. Each Kserver manages its own local ID space,and KIDs are a combination of its home Kserver's IP 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. Khazana's design allows this simple scheme to be easily replaced by more sophisticated file naming and dynamic location tracking schemes such as those of Pastry [17] and Chord [23]. To improve access latency, each Kserver maintains a cache of hints of nearby (i.e., strongly connected) Kservers with copies of a file, which it contacts before contacting the home node. This helps avoid crossing slow links for inter-replica communication.

Creating and Destroying Replicas

Here we briefly outline the method by which replicas are created and destroyed, and how the replica hierarchy is built and managed. We refer the interested reader to a more detailed description elsewhere [19].

When one Kserver queries another to request a copy of a file, the responder either supplies the requester a copy itself (if it has not reached its configured fanout limit), forwards the request to a few randomly selected children, or sends a list of its children to the requestor. In the last case, the querying Kserver selects the "nearest" child node (in terms of network quality) and sends it the request. The node that supplies the file data to the requestor becomes its parent in the replica hierarchy. This mechanism dynamically forms a hierarchical replica network rooted at the file's home node that roughly matches the physical network topology. It is similar to dynamic hierarchical caching as introduced by Blaze [1]. Figure 1 illustrates one such network.

Replicas continually monitor the network link quality (currently RTT) to other known replicas and rebind to a new parent if they find a replica closer than the current one. The fanout of any node in the hierarchy is limited by its load-handling capacity or explicitly by the administrator. When a link or node in the hierarchy goes down, the roots of the orphaned subtrees try to re-attach themselves to the hierarchy, starting at a known copy site (home node by default), effectively repairing the replica hierarchy.

To guard against transient home node failure, the root's direct children are called *custo-*

dians. Their identity is propagated to all replicas in the background. If the root node stops responding, these custodians keep a file's replica hierarchy together until the home node comes up, and prevent it from getting permanently partitioned. In the future we plan to make these custodians into redundant primary replicas for fault tolerance. A Kserver (other than a custodian) is free to delete locally cached copies of files to reclaim local store space at any time, after propagating their updates to neighboring copy sites.

Reconnecting to the Replica Hierarchy

When a replica R loses its connection to its parent, it first selects a new parent P as described above. Subsequently, it loads a fresh copy from P and re-applies any pending updates to this copy. (Khazana normally avoids this reload unless the replicas are "too far out of sync". See [19]). If R simply propagated its local updates to P without resynchronizing, it might propagate a duplicate update.

Update Propagation

Updates are propagated up (from clients to the root) and down (from the root to clients) the distributed replica hierarchy. Associated with each replica/update is a version. Each replica maintains the version numbers of its neighboring replicas. Each update message (physical or operational) contains a version number. The version numbers are used to identify when updates need to be propagated and to detect update conflicts. Due to space limitations we refer the reader to a more detailed document describing our version management mechanism [19].

A replica's responsibility for an update is considered over once it propagates it successfully to its parent replica. 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.

An update conflict arises when two replicas are independently modified, starting from the same version. Certain (lock-based) consistency options avoid conflicts by serializing conflicting operations, but other consistency modes allow conflicts. Update conflicts can be detected via object and update version numbers. Khazana supports three conflict resolution mechanisms. For operational updates, Khazana lets the client plugin resolve the conflict as it wishes, e.g., merging the updates. For physical updates, Khazana can be configured to quietly apply a "last writer wins" policy for updates arriving at the local replica, or to drop the update and inform the client of the conflict.

2.2 Consistency Management

In this section we describe how Khazana's consistency management subsystem is implemented, and the set of consistency options it supports.

Khazana clients access shared data via the Kclient library API. Before performing any operations, a Kclient must locate and bind to a nearby (often co-located) Kserver. To access a particular shared file, the client performs a `kh_open()`. Before responding, the Kserver must have a local replica of the file at the specified level of consistency. Typically this involves obtaining a copy of the file from a remote Kserver and attaching to the dynamic replica hiearchy. The Kclient can then perform `kh_read()` and `kh_write()` operations on the local replica – what the Kserver does in response to these reads and writes (if anything) depends on this file's consistency semantics.

Associated with each active replica of a file is a set of access privileges. The privileges determine whether the local replica can be read or written without contacting remote replicas. When no Kserver is currently serving a particular file, the file's root node owns all privileges. As clients request read and write access to a file, the access privileges are replicated or migrated to other replicas – the manner in and time at which privileges are exchanged between Kservers also depends on the file's consistency semantics.

Khazana maintains consistency via two basic mechanisms, *absolute updates* where updates to a file's data directly overwrites remote copies as in a distributed shared memory system, and *operational updates* where what is exchanged are logical operations that should be applied to each replica (e.g., "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. To use operational updates, the Kclient accessing each replica must provide callback functions that interpret and apply "updates" on the local copy of a region. We currently use simple plugins, implemented via callbacks, to handle operational updates with low overhead.

As described in Section 1, the consistency options provided by Khazana can be described along five dimensions: *concurrency*, *consistency*, *availability*, *visibility*, and *isolation*. In this Section we describe the various options that Khazana provides along each dimension. Kclients can specify almost arbitrary combinations of options, although not all combinations make sense. Other existing systems (e.g., WebFS [24] and Fluid replication [3]) bundle options as part of the implementation of individual policies, and do not let applications choose arbitrary combinations.

Concurrency Options

Concurrency control refers to the parallelism allowed among reads and writes at various replicas. Khazana supports four distinct access modes that can be specified when opening a session:

RD (snapshot read): In this mode, only reads are allowed. A read returns a snapshot of the data that is guaranteed to be the result of a previous write. Such reads do not block for ongoing writes to finish nor do they block them. Successive reads within a session may return different data. The semantics provided by this mode are similar to that of the Unix O_RDONLY open() mode.

RDLK (exclusive read): In contrast to **RD** mode, this mode ensures that the data is not updated anywhere for the duration of the read session. This provides conventional CREW locking semantics and strongly consistent reads. It blocks for ongoing write sessions as well as blocks future write sessions until the session ends.

WR (shared write): In this write mode, each individual write concurrent write sessions are allowed and may interleave their writes. These writes might conflict. For applications where write conflicts are either rare or can be easily resolved, this mode improves parallelism and write latency. The semantics provided by this mode are similar to that of the Unix O_RDWR open() mode.

WRLK (exclusive write): This mode provides serializable writes. It blocks and is blocked by other ongoing RDLK, WR, and WRLK sessions and hence trades parallelism for stronger consistency.

Consistency Options

Consistency refers to the degree to which stale data can be tolerated. Khazana allows clients to specify their consistency needs in terms of *timeliness* or *data interdependencies*.

Timeliness refers to how close data read from the file must be to the most current global version. Khazana supports three choices for timeliness: *most current*, *time-bounded* and *modification-bounded*. Time bounds are specified in tens of milliseconds. As shown in Section 3, even time bounds as small as ten milliseconds can signficantly improve performance of *most current*. Modicication bound specifies the number of unseen remote writes tolerated by a replica, similar to the numerical error metric provided by the TACT toolkit [25].

Khazana allows clients to specify whether the timeliness requirements are *hard* (true requirements) or *soft* (best-effort suffices). If a client specifies that it requires hard guarantees, then requests are blocked until the required timeliness guaranteed can be met. Hard guarantees are implemented using pull-based consistency protocols. For example, close-to-open consistency for files can be achieved by choosing to pull updates. If a client specifies that soft best-effort guarantees are adequate, Khazana uses push-based consistency protocols. Soft timeliness guarantees are adequate for applications like bulletins boards and email, and can lead to significantly better performance and scalability.

Data interdependence refers to whether an application requires that updates be propagated in any particular order. Khazana currently supports two data interdependence options: *none* – each update can be propagated independent of all other updates and can be applied in different orders at different replicas, and *total order* – all updates must be applied in the same order to all replicas. We are considering adding support for *causal* ordering and *atomic ordering of updates to multiple objects*, the latter of which is useful for distributed databases and must currently be implemented on top of Khazana. Khazana currently supports causality and atomicity for multiple updates to a single file, and thus within a single replica hierarchy, but adding support for multi-file causality or atomicity would require significant work and violate the end-to-end principle.

Availability Options

Many applications, e.g., mobile ones, must continue to operate during periods of intermittent connectivity. To support this class of applications, Khazana gives clients two availability choices: *optimistic* – the best available data is supplied to the user, even if it cannot be guaranteed to be current due to network or node failures, or *pessimistic* – accesses stall (or fail) if Khazana cannot supply data with the requisite consistency guarantees due to network or node failures. During failure-free performance, the optimistic and pessimistic availability options perform identically. Optimistic availability is only a viable option for data with "weak" consistency requirements.

Visibility and Isolation Options

Visibility refers to the time at which updates are made visible to remote readers. *Isolation* refers to the time when a replica must apply pending updates. Khazana enable clients to specify three options for visibility and isolation:

Session: This specifies that updates are visible to sessions on remote replicas after the local session ends. Session isolation means that checks for remote updates are made only at the beginning of a session not before each read operation. Session semantics prevent readers from seeing intermediate writes of remote sessions.

Per-access: This means that updates are immediately available to be propagated to remote replicas. When they will actually be propagated depends on the consistency desired and the isolation needs of remote readers. Per-access isolation means that a replica applies remote updates before each read operation.

Manual: Manual visibility means that a session's owner specifies when its updates should be made visible. They are always made visible at the end of the session. Manual isolation means that remote updates are only incorporated when the local client explicitly requests they be applied. These settings enable an application to hand-tune its update propagation strategy to balance timeliness of data against performance.

Discussion

The above options allow applications to compose consistency semantics appropriate for their sharing needs. Table 1 lists what we anticipate will be some common sets of options that correspond to well known "hard wired" consistency schemes. The first column denotes a particular well known consistency scheme, and the other fields denote the set of options that an application should specify to achieve this desired semantics. For example, "close-to-rd" semantics means that a session reads only writes by completed sessions, not the intermediate writes of still open sessions. If an application's updates preserve data integrity only at session boundaries, this set of options ensure that reads always return stable (i.e., internally consistent) data regardless of ongoing update activity. In contrast, "wr-to-rd" semantics does not provide this guarantee, but is useful when up-to-date data is preferred over stable data, or when write sessions are long-lived as in the case of distributed data logging, or live multimedia streaming. Finally, the rightmost column gives an example situation or application where that choice of consistency options is appropriate.

Access (rd & wr) semantics	**Check for updates**	**Wr Visible**	**Strength of guarantee**	**Availability in partition**	**Use/ Provider**
strong (exclusive)	on open	on close	hard(pull)	no	serializability
close-to-open	on open	on close	hard	app. choice	collaboration, AFS
close-to-rd	on access	on close	hard	app. choice	read stable data
wr-to-open	on open	on write	hard	app. choice	NFS
wr-to-rd	on access	on write	hard	app. choice	log monitoring
eventual close-to-rd	never	on close	soft(push)	yes	Pangaea
eventual wr-to-rd	never	on write	soft	yes	chat, stock quotes

Table 1: Some “reasonable” sets of consistency options

3 Evaluation

In this section, we present our evaluation of the benefits of using Khazana middleware to build three distinct data services.

In Section 3.1 we describe our experimental setup. In Section 3.2, we show that *KidFS*, a distributed file system built on top of Khazana, can exploit locality more effectively than existing file systems, while supporting a broader variety of file usage modes with diverse consistency requirements. Kidfs supports a variety of file access semantics ranging from exclusive file locking, close-to-open to eventual consistency on a per-file basis. In Section 3.3, we show that *Kobj*, an enterprise object proxy built on top of Khazana, can significantly improve its responsiveness to wide-area clients via intelligent caching. Khazana’s adaptive caching algorithm decides when and where to replicate data; data with good locality tends to be aggressively migrated or replicated, while data with poor locality or heavy write sharing tends to be managed via a more efficient centralized mechanism. Caching improves responsiveness by an order of magnitude even under modest locality. Finally, in Section 3.4, we demonstrate how *Kdb*, a replicated version of the BerkeleyDB built on top of Khazana, performs using five different consistency requirements ranging from strong (appropriate for a conventional database) to time-based (appropriate for many directory services). We find that relaxing consistency requirements even slightly significantly improves throughput.

3.1 Experimental Setup

For all our experiments, we used the University of Utah's Emulab Network Testbed [5]. Emulab allows us to model a collection of PCs connected by arbitrary network topologies with configurable per-link latency, bandwidth, and packet loss rates. The PCs had 850Mhz Pentium-III CPUs with 512MB of RAM. Depending on the experimental requirements, we configured them to run FreeBSD 4.7 (BSD), Redhat Linux 7.2 (RH7.2), or RedHat 9 (RH9). In addition to the simulated network, each PC is connected to a 100Mbps control LAN isolated from the experimental network.

Kservers run as user-level processes and store regions in a single directory in the local file system, using the KID as the filename. The replica fanout was set to a low value of 4 to induce a multi-level hierarchy. Clients and servers log experimental output to an NFS server over the control network. All logging occurs at the start or finish of an experiment to minimize interference.

3.2 Kidfs: A Flexible Distributed File System

We built a distributed file system called *Kidfs* using Khazana, that exploits Khazana's flexible consistency options to efficiently support a broad range of file sharing patterns. In particular, it efficiently provides on a per-file-replica basis: the strong consistency semantics of Sprite [22], the close-to-open consistency of AFS [10] and Coda [12] and the (weak) eventual consistency of Coda, Pangaea [20], and NFS [21].

From a client's perspective, Kidfs consists of a collection of peer Kidfs agents spread across the network. Each Kidfs agent is tightly integrated into a local Kserver process, which is responsible for locating and caching globally shared files.

Each Kidfs agent exports the global Kidfs file space to local users at the mount point '`/kidfs`' by taking the place of Coda's client-side daemon, 'venus'. When clients access files under `/kidfs`, Coda's in-kernel file module routes them to the Kidfs agent via upcalls on a special device. These upcalls are made only for metadata and open/close operations on files. Kidfs agent can perform consistency management actions on a file only during those times. The Coda module performs file reads and writes directly on the underlying local file, bypassing Kidfs. Kidfs agents create files in the local file system to satisfy client file creation requests or to persistently cache files created by other agents. In Kidfs, KIDs take the place of inode numbers for referencing files. Kidfs implements directories as regular files in Coda's directory format and synchronizes replicas via operational updates.

Users can set consistency options on individual files and directories via `ioctl()` system calls. By default, a directory's consistency attributes are inherited by its subsequently created files and subdirectories. We describe the supported consistency attributes below.

Unlike typical client-server file systems (e.g., NFS, AFS, and Coda), Kidfs exploits Khazana's underlying peer-to-peer replica hierarchy to access a file's contents from geographically nearby (preferably local) copies of the file.

To determine the value of building a file service on top of Khazana, we evaluate Kidfs under three different usage scenarios.

First, in Section 3.2.1, we consider the case of a single client and server connected via a high-speed LAN. This study shows that the inherent inefficiencies of adding an extra level of middleware to a LAN file system implementation has little impact on performance. Our second and third experiments focus on sharing files across a WAN. In Section 3.2.2 we consider the case where a shared set of files are accessed sequentially on series of widely separated nodes. This study shows that the underlying Khazana layer's ability to satisfy file requests from the "closest" replica can significantly improve performance. Finally, in Section 3.2.3 we consider the case where an shared RCS repository is accessed in parallel by a collection of widely separated developers. This study shows that Khazana's lock caching not only provides correct semantics for this application, but can also enable developers to fully exploit available locality by caching files near frequent sharers.

3.2.1 Local Client-Server Performance

To provide a performance baseline, we first study the performance of Kidfs and several other representative distributed file systems in a simple single-server, single-client topology. In this experiment, a single client runs Andrew-tcl, and scaled up version of the Andrew benchmark on a file directory hosted by a remote file server, starting with a warm cache. Andrew-tcl consists of five phases, each of which stresses a different aspect of the system (e.g., read/write performance, metadata operation performance, etc.). For this experiment, there is no inter-node sharing, so eventual consistency suffices.

We ran the Andrew-tcl benchmark on four file systems: the Redhat Linux 7.2 local file system, NFS, Coda, and Kidfs. For Kidfs, we considered two modes: *kcache*, where the client is configured to merely cache remotely homed files, and *kpeer*, where files created on the client are locally homed in full peer-to-peer mode. Figure 2 shows the relative performance of each system where the client and server are separated by a 100-Mbps LAN, broken down by where the time is spent. Execution time in all cases was dominated by

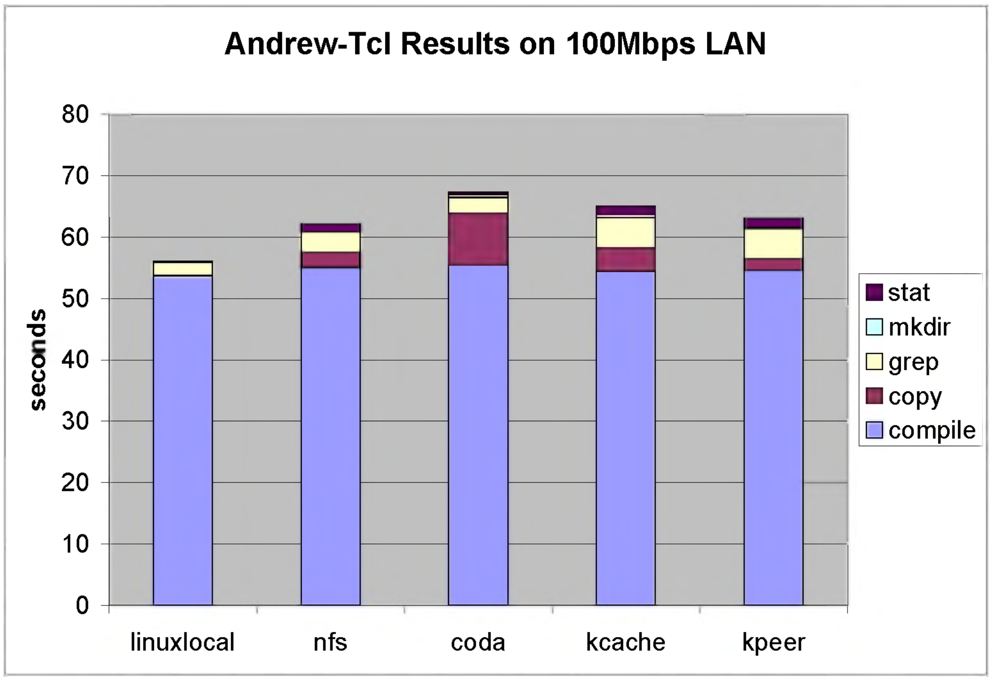

Figure 2: Andrew-Tcl Results on 100Mbps LAN

the compute bound compile phase, which is identical for all systems. Figure 3 focuses on the other phases. As expected, the best performance is achieved running the benchmark directly on the client's local file system. Among the distributed file systems, NFS performs best, followed by kpeer, but all distributed file systems performed within 10%. kpeer performance particularly well during the data-intensive `copy` and `mkdir` phases of the benchmark, because files created by the benchmark are homed on the client node[1]. Coda's file copy over LAN takes twice as long as kcache due to its eager flushes of newly created files to the server.

[1]The performance of Kidfs metadata operations, which are used heavily in the `grep` and `stat` phases of the benchmark, is poor due to a flaw in our current Kidfs implementation. We do not fully exploit the Coda in-kernel module's ability to cache directory data. When we enhance Kidfs to incorporate this optimization, Kidfs performance during the metadata-intensive phases of the benchmark will match Coda's performance.

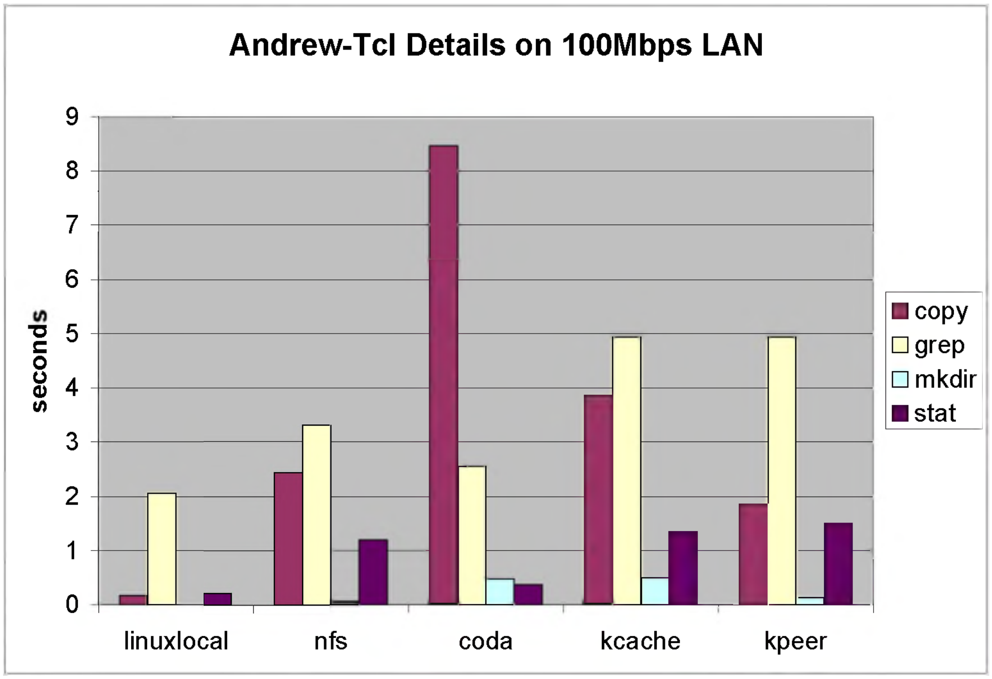

Figure 3: Andrew-Tcl Details on 100Mbps LAN

3.2.2 Sequential Wide Area Access (Roaming)

In our second experiment we focus on a simple scenario in which files are shared in migratory fashion across a WAN. For this experiment, we assume a network topology like that illustrated in Figure 1 consisting of widely distributed five sites, each of which contains two nodes connected via a 100Mbps LAN.

Each of the ten nodes run the Andrew-tcl benchmark in turn on a shared directory to completion, followed by a "`make clean`". First one node on the University LAN runs the benchmark, then its second node, then the first node in ISP1, etc., in the order University (U) $\rightarrow$ ISP1 (I) $\rightarrow$ Corporate (C) $\rightarrow$ Turkey (T) $\rightarrow$ France (F). Each node starts with a cold file cache. This scenario represents any situation where access behavior tends to be migratory, e.g., due to a single user roaming among nodes or an outsourced operational with 24-hour service provided by operators across the globe. The initial copy of the Andrew-tcl benchmark tree is hosted by a Kidfs agent on the node marked "Home Agent" on the University LAN.

We performed this experiment using three distributed file systems: Kidfs in peer-to-peer mode, Coda in strongly-connected mode (Coda-s), and Coda in adaptive mode (Coda-w). Kidfs was configured to provide close-to-open consistency. Coda-s provides strong consistency, whereas Coda-w quickly switched to weakly connected operation due to the high link latencies. During weakly connected operation, coda-w employed trickle reintegration to write back updates to the server "eventually".

Figure 4 shows the time each node took to perform the compile phase of the Andrew-tcl benchmark. As reported in the previous section, both Kidfs and Coda perform comparably when the file server is on the local LAN, as is the case on nodes U1 and U2. However, there are two major differences between Kidfs and Coda when the benchmark is run on other nodes. *First, Kidfs always pulls source files from a nearby replica*, whereas Coda clients always pull file updates through the home server incurring WAN roundtrips from every client. As a result, Coda clients suffered 2x-5x higher file access latency than Kidfs clients. *Second, Kidfs was able to provide "just enough" consistency to implement this benchmark efficiently but correctly, whereas the two Coda solutions were either overly conservative (leading to poor performance for coda-s) or overly optimistic (leading to incorrect results for coda-w).*

Kidfs had a number of advantages over coda-s. One was the aforementioned ability to read a source file from any replica, not just the home node. This flexibility was especially important when the benchmark was run either on the second node of a LAN or run for the second (or subsequent) time in "Europe". Also, file creation in coda-s is mediated by the home server, which leads to poor performance when the latency to the server is high, such as is the case for C1-F2. The net result is that the benchmark ran 2-4X faster on Kidfs than on coda-s on the WAN clients, as might be expected given that coda-s is not intended for WAN use.

The comparison between Kidfs and coda-w illustrates the importance of having user-configurable consistency policies. In adaptive mode, if Coda determines that the client and server are weakly connected, it switches to 'eventual consistency" mode, wherein changes to files are lazily propagated to the home server, from which they are propagated to other replicas. Unfortunately, in this scenario, that degree of consistency is insufficient to ensure correctness. Reintegrating a large number of object files and directory updates over a WAN link takes time. If a second benchmark run starts before all changes from the previous run have been pushed to the current client, conflicts occur. In this case, Coda reports an update conflict, which requires manual intervention. If these conflicts are ignored, delete messages associated with intermediate files created by earlier nodes are not integrated in time, which leads later nodes to incorrectly assume that they do not need to recompile the associated source files. In contrast, Khazana enforces Kidfs's desired close-to-open consistency policy on each file, thereby ensuring correct operation regardless of contention.

3.2.3 Simultaneous WAN Access

Some distribued applications (e.g., email servers and version control systems) require reliable file locking or atomic file/directory operations to synchronize concurrent read/write accesses. However, the atomicity guarantees required by these operations are not provided by most wide area file systems across replicas. As a result, such applications cannot benefit from caching, even if they exhibit high degrees of access locality.

For example, the RCS version control system uses the exclusive file creation semantics provided by the POSIX open() system call's O_EXCL flag to gain exclusive access to repository files. During a checkout/checkin operation, RCS attempts to atomically create a lock file and relies on its pre-existence to determine if someone else is accessing the underlying repository file. Coda's close-to-open consistency semantics is inadequate to guarantee the exclusive file creation semantics required by RCS. Thus hosting an RCS repository in Coda could cause incorrect behavior. In contrast, Kidfs can provide strong consistency that ensures correct semantics for repository directory and file updates required by RCS. This allows Kidfs to safely replicate RCS files across a WAN, thereby exploiting locality for low latency access. We evaluated two versions of RCS, one for which the RCS repository resides in Kidfs (*peer sharing mode*) and one in which the RCS repository resides on U1 and is accessed via ssh (*client-server/RPC mode*).

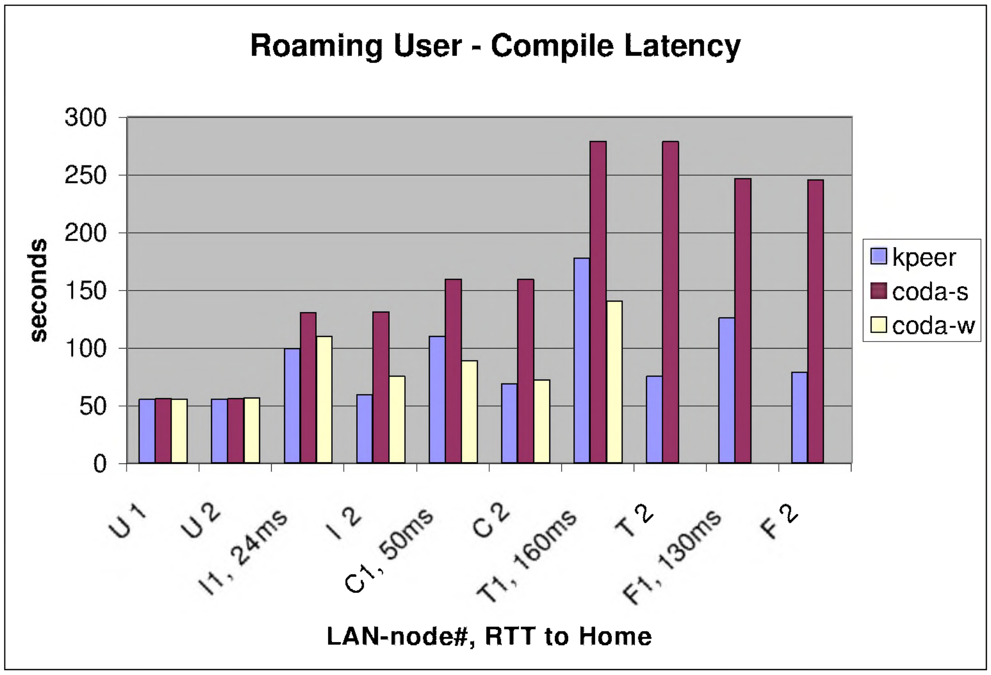

Figure 4: Compile Phase: Kidfs pulls files from nearby replicas. Strong-mode CODA behaves correctly, but exhibits poor performance. Weak-mode CODA performs well, but generates incorrect results on the final three nodes.

To illustrate how Kidfs performs in the face of concurrent file sharing, we simulated concurrent development activities on a project source tree using RCS for version control. We chose RCS, rather than CVS, because RCS employs per-file locking for concurrency control and hence allows more parallelism than CVS, which locks the entire repository for every operation. For this set of experiments, we used a simplified version of the topology shown in Figure 1 without the ISP1 LAN (I). The "Home Node" initially hosts three project subdirectories from the Andrew-tcl benchmark: `unix` (39 files, 0.5MB), `mac` (43 files, 0.8MB), and `tests` (131 files, 2.1MB).

Our synthetic software development benchmark consists of six phases, each lasting 200 seconds. In Phase 1 (widespread development), all developers work concurrently on the `unix` module. In Phase 2 (clustered development), the developers on the University and Corporate LANs switch to the `tests` module, the developers in Turkey continue work on the `unix` module, and the developers in France switch to the `mac` module. In Phases 3-6 (migratory development), work is shifted between "cooperating" LANs – the `unix` module migrates between the University and Turkey, while the `mac` module migrates between Corporate LAN and France (e.g., to time shift developers). During each phase, a developer updates a random file every 0.5-2.5 seconds from the directory she is currently using. Each update consists of an RCS checkout, a file modification, and a checkin.

Figure 5 shows the checkout latencies observed from clients on the University LAN, where the master copy of the RCS repository is hosted. Figure 6 shows the checkout latencies observed from clients on the "Turkey" LAN. The checkout latencies were fairly consistent at each node in client-server mode. Therefore, we plotted the average latency curve for each node on both graphs. The checkout latencies in peer sharing mode were heavily dependent on where the nearest replica was located and the amount of work needed to maintain consistency between replicas, so we provide a scatter-plot of all checkout latencies in this mode.

Overall, our results indicate that *Kidfs enables RCS developers to realize the performance benefits of caching when there is locality, while ensuring correct operation under all workloads and avoiding performance meltdowns when there is little to no locality*. This is shown by the fact that checkout latency under clustered development (i.e., phases 3 and 5 of Figure 5, and phases 2, 4 and 6 of Figure 6) quickly drop to that of local RCS performance observed by U1 (shown in Figure 5). At low locality (as in Phase 1 for all developers, and Phase 2 for U1-C2), RCS on Kidfs still outperforms client-server RCS. RCS on Kidfs' latency is close to two seconds or less for all developers, while that of client-server RCS degrades in proportion to the latency between the client and central server. This is because Khazana avoids using the slow WAN link as much as possible. Finally, Kidfs responds quickly to changes in data locality.

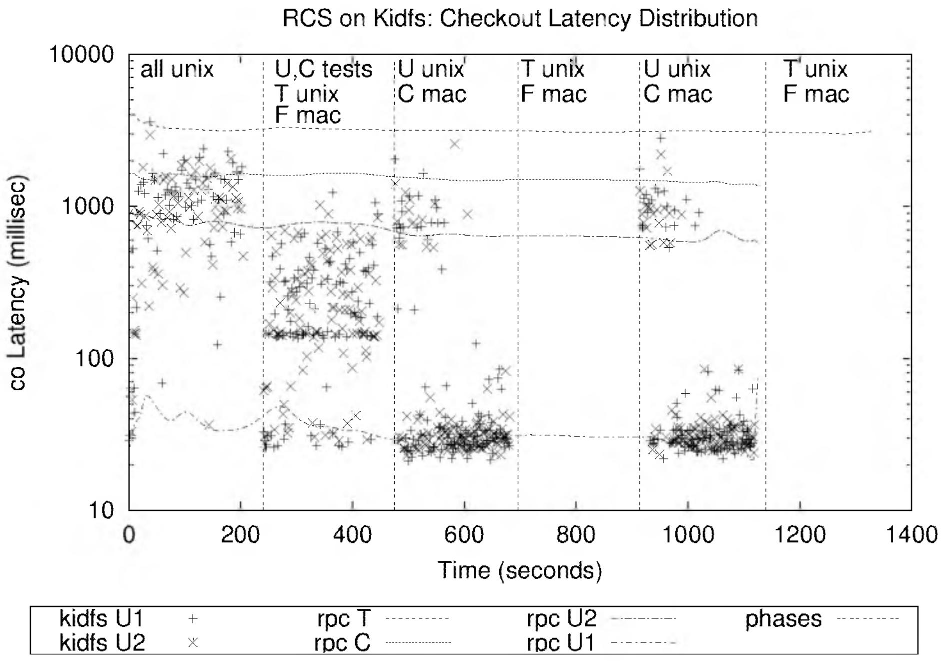

Figure 5: RCS on Kidfs: Checkout Latencies on the "University LAN"

When the set of nodes sharing a file changes, Khazana migrates the replicas to the new set of sharers fairly rapidly. This phenomenon is illustrated by the initial high checkout latency for each node during Phases 3-6, which rapidly drops to local checkout latency once Khazana determines that the new sharing pattern is stable and persistent. The time at the beginning of each phase change when nodes see high checkout latency represents the hysteris in the system, whereby Khazana does not migrate data with low locality.

3.3 Kobj: Wide-area Object Caching

Distributed enterprise services that handle business-critical information (e.g., sales or inventory data) could benefit from object proxy caching. However, they tend to have stringent integrity requirements that can lead to significant coherence traffic if data with poor locality is cached widely. In this section we demonstrate that Khazana can be used to build robust object proxy servers that operate on cached enterprise data without compromising integrity, by employing strong consistency and caching optimizations. For data with poor locality, Khazana automatically inhibits migration or replication of data to limit performance degradation.

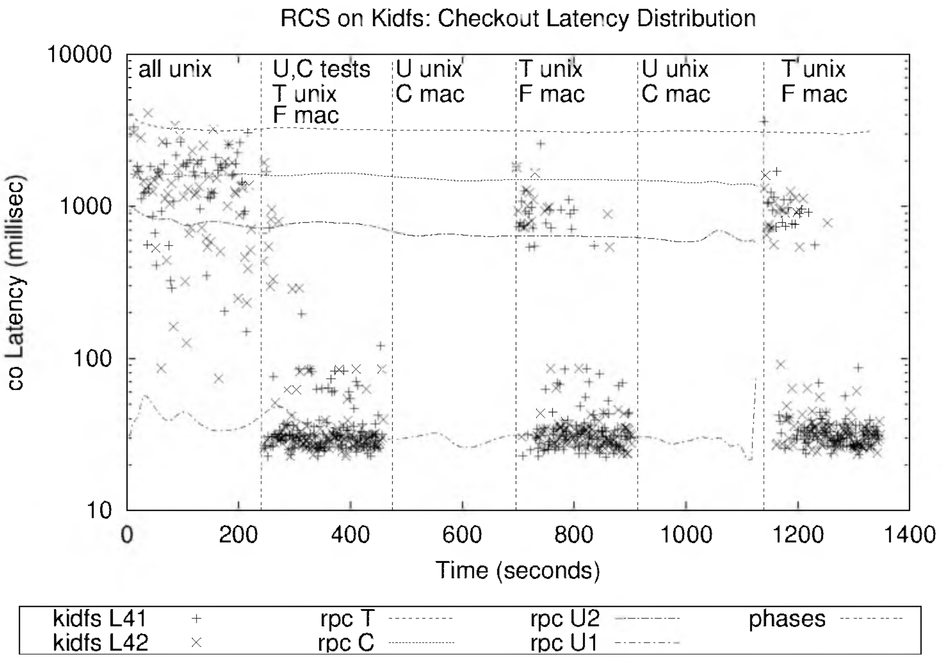

Figure 6: RCS on Kidfs: Checkout Latencies from “Turkey”

We simulate a simple three-tier enterprise service consisting of a collection of ASPs that accept input via the web and operate on data stored in a distributed object database. To enhance performance, we introduce an object proxy cache (Kobj) that caches enterprise objects near where they are being used if there is sufficient locality. Kobj consists of two data components: an index structure and a collection of 256 4-kilobyte enterprise objects, both of which reside in a single Khazana file managed as a persistent page heap. The index structure maps object IDs to offsets within this heap. We use a relatively small object space to ensure a reasonable degree of sharing and contention over the course of the experiment. For all Kobj experiments, we deploy 64 Kobj clients on 16 nodes (K1-K16) running RedHat 9, each residing in a separate LAN. Independent LANs are connected via 1Mbps WAN connections with a 40msec roundtrip latency.

The modeled workload is similar to TPC-A, where each client repeatedly (at full speed) selects a bank account at random on which to operate and then randomly chooses to query or update the account. We model a 50-50 mix of reads and writes to evaluate Kobj’s performance during periods of heavy write contention. We vary the degree of access locality as follows. Associated with each of the 16 nodes are 16 “local” objects. When a client randomly selects an object on which to operate, it first decides whether to select a local object or a “random” object. We vary the likelihood of selecting a local object from 0%, in which case the client selects any of the 256 objects with uniform probability, to 100%, in which case the client selects one of its node's 16 local objects with uniform probability. In essence, the 100% case represents a partitioned object space with maximal throughput because there is no sharing, while the 0% case represents a scenario where there is no access locality. Kobj specifies to the Khazana layer that it requires strong consistency for the heap file. Despite the strong consistency requirement, Kobj improves service throughput when clients exhibit moderate to high access locality; during periods of low locality and/or heavy contention, feedback from the Khazana layer causes Kobj it to inhibit caching and switch to RPC mode to avoid thrashing.

At the start of each experiment, all data is hosted by a single node K0 in its own server LAN. As the experiment progresses, we add a Kobj proxy (and associated Kserver) to a new LAN (K1-K16) every 50 seconds. Throughout the entire experiment, each client selects an object at random, using the locality distribution described above, and then contacts either the "home" server or a local Kobj proxy if one exists. The selected Kobj asks the local Kserver where the closest replica of the requested object is currently cached, and forwards the client request to the proxy on that node. The receiving proxy performs the operation locally and returns results directly to the initiating client. Processing a request involves walking the B-tree index in RDLK mode to find the requested object's offset and then locking the object's page in appropriate mode. As we add proxies, we increase Khazana's ability to cache data near where it is most often accessed, at the cost of potentially greatly increasing the amount coherence traffic needed to keep individual objects strongly consistent.

After the experiment has run long enough to start a Kobj proxy on each LAN, each client shifts its notion of what objects are "local" to be those of its next cyclical neighbor. We run this scenario for 100 seconds, a period denoted as "expt 2" in Figures 9 through 11. After this phase ends, clients 33-64 all treat client 1's "local" objects as their own "local" objects, which introduces very heavy contention on those 16 objects. We run this scenario for 100 seconds, a period denoted as "expt 3". We ran each of the above experiments in two modes, *eager replication mode*: where Kservers always create a local replica when an object is accessed locally, and *adaptive replication mode*: where Khazana adapts between replication and master-slave (RPC) modes to prevent thrashing.

Figures 7 and 8 show how aggregate throughput varies as we increase the number of Kobj proxies at different degrees of locality (0%-100%). Vertical bars denote the creation of a new proxy. Without adaptive lock caching, aggregate throughput quickly levels off as more proxies contend for the same objects. The higher the degree of locality, the higher the throughput, but even at 95% locality throughput using eager replication never exceeds 2500 ops/sec. Using adaptive caching, clients initially forward all requests to the root server on k0, but once locality exceeds 40%, Kobj proxies quickly cache their "local" objects after they are spawned. Under these circumstances, nodes will use RPCs to access "remote" objects, rather than replicating them, which eliminates thrashing and allows throughput to

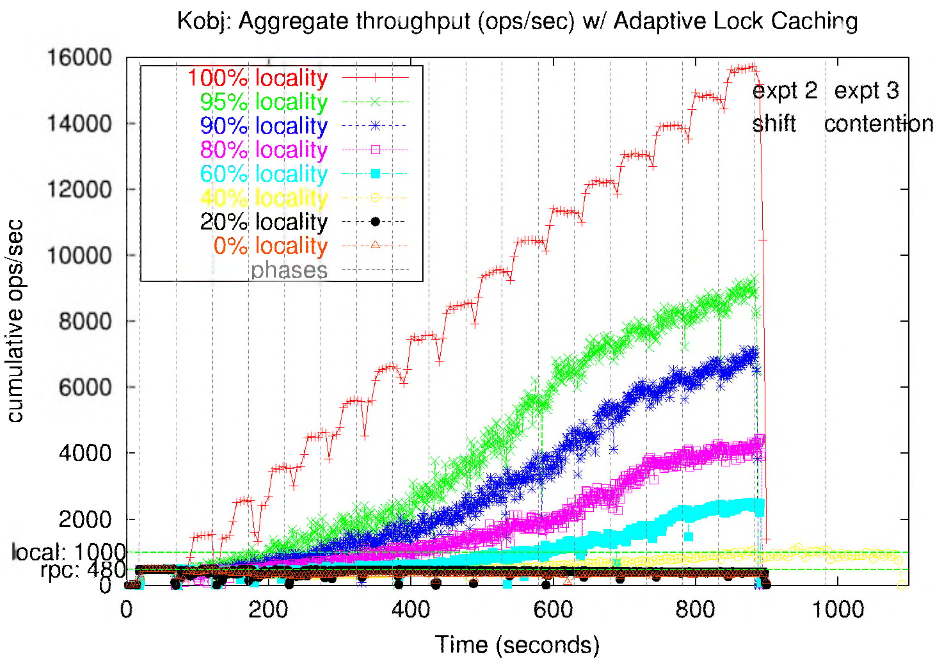

Figure 7: Kobj aggregate throughput (ops/sec) with adaptive replication.

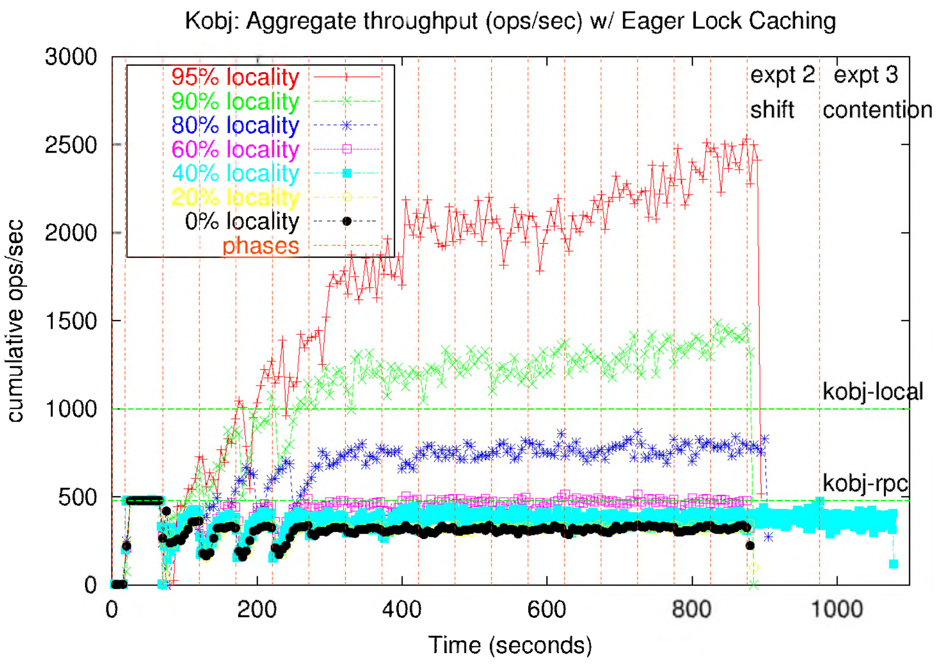

Figure 8: Kobj aggregate throughput (ops/sec) with eager replication.

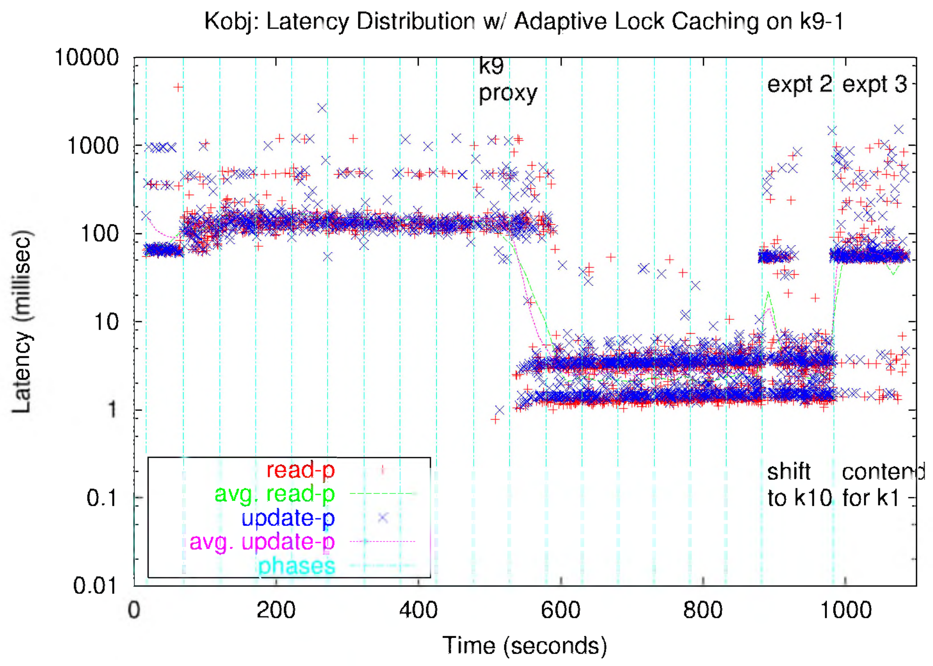

Figure 9: Kobj "local" object access latencies on node k9 with adaptive replication at 40% locality. Once a Kobj proxy is started on k9, Khazana's adaptive caching protocol inhibits k9's "local" objects from being replicated remotely.

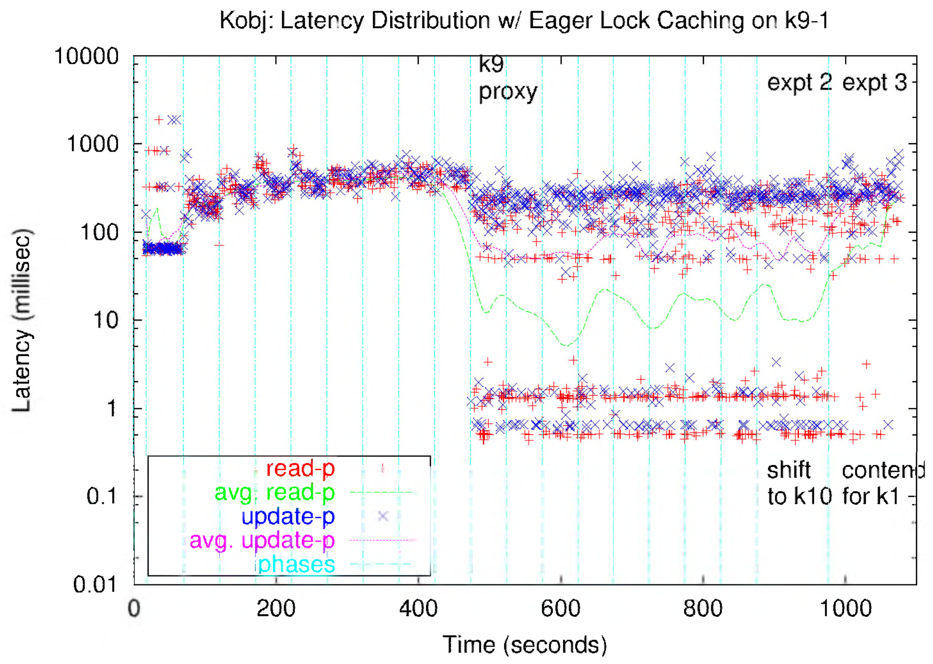

Figure 10: Kobj "local" object access latencies on k9 with eager replication at 40% locality. Locks on private objects keep shuttling to and from k9, reducing throughput.

continue to scale as proxies are added. With high ($> 95\%$) locality, the adaptive protocol can support almost 9000 ops/sec in the fully populated system.

Figures 9 through 11 provide a detailed breakdown of typical access latencies under different scenarios at a client on node K9. Figure 9 shows that even with modest (40%) locality, the adaptive protocol reduces the access latency to "local" objects from 100msecs to under 5msecs once a proxy is spawned. In contrast, Figure 10 shows that under the same circumstances the average access latency of local objects increases to over 200msecs using eager replication due to frequent lock shuttling. Figure 11 shows that the adaptive protocol also outperforms the eager protocol for accesses to "non-local" objects, despite never caching these objects locally, by eliminating useless coherence traffic.

When we have each node shift the set of objects that most interest it, the phase denoted "expt2" in the graphs, the adaptive protocol migrates each object to the node that now accesses it most often after about 10 seconds, as seen in Figure 9. Performance of the eager protocol does not change, nor does the average access latency of "non-local" objects.

Finally, when we induce extremely heavy contention for a small number of objects, the phase denoted "expt3" in the graphs, the adaptive protocol almost immediately picks a sin-

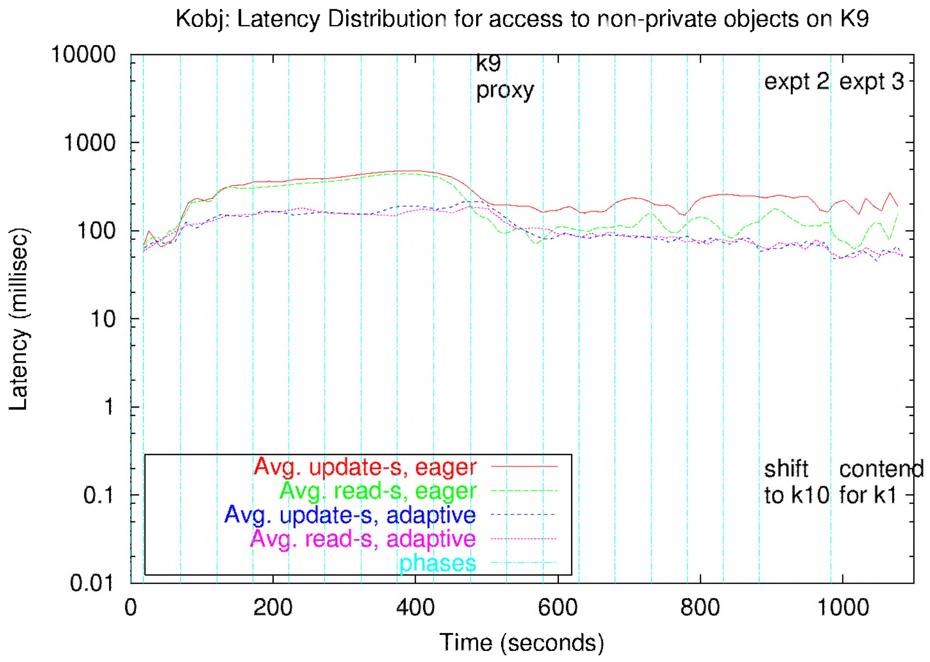

Figure 11: Kobj "non-local" object access latencies on k9 at 40% locality. Adaptive caching handles these accesses via RPCs rather than via replication, which results in an average latency of roughly 50% that of eager caching.

gle replica to cache the data and shifts into RPC mode. By doing so, it is able to serve even heavily accessed objects in roughly 100msecs. In contrast, the eager replication protocol often requires over 500msecs to service a request and only rarely is able to exploit the aggressive sharing (as seen by the small number of sub-100msec latency accesses during "expt3" in Figure 10.

In summary, we find that a wide-area object service built on top of Khazana can exploit available access locality to scale, while at the same time guaranteeing strong consistency requirements. Khazana's adaptive caching protocol inhibits data replication or migration when there is insufficient locality to warrant it, which dramatically improves access latency.

3.4 Kdb: Replicated BerkeleyDB

To demonstrate how Khazana can be used to add replication to an existing data service without altering its internals, we implemented Kdb, a BerkeleyDB database (representing a directory) stored in a Khazana file. Kdb aggressively caches the database file on all nodes, and uses operational updates to propagate modifications to all replicas. The same approach can be used, for example, to replicate a relational database such as mySQL across wide area. All update queries will then be intercepted and propagated among replicas as operational updates.

A common way to improve the performance of distributed databases is to allow objects to be read-only replicated close to clients, but require all writes to be sent to a central "master" replica. This approach to replication is relatively easy to implement and ensures strong consistency for updates. However, it severely limits scalability and does not perform well in the presence of heavy write traffic. Relaxing consistency for writes can significantly enhance system throughput, but can lead to update conflicts. By using Khazana to implement data replication, we can choose on a per datum basis how much consistency is required, thereby achieving high throughput when the consistency requirements are less strict, e.g., a directory service [15], while using the same code base as a conventional strongly consistent database.

We evaluated Kdb's performance and scalability using five flavors of consistency semantics (listed from strongest to weakest) and compared it to a client-server (RPC) version: (1) push-based peer-to-peer where writes are serialized before being propagated (*serialized writes, eventual reads*), (2) master-slave where all writes are sent to the root of the replica hierarchy and propagated (*single-master writes, eventual reads*, and (3) pull-based peer-to-peer with close-to-open semantics (*close-to-open*), (4) pull-based peer-to-peer where data

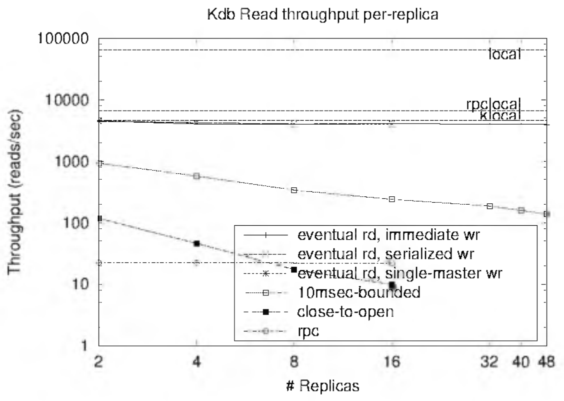

Figure 12: Kdb Per-replica Read Throughput

is synched upon access if more than 10msecs have passed since last synch (*time-bounded staleness*), (5) push-based peer-to-peer where writes are performed locally before being propagated to other replicas (*local-immediate writes, eventual reads*).

For our Kdb experiments, the DB stores 100 key-value pairs inside a Khazana file. The database size does not affect performance because we employ operational updates and the entire database file is treated as a single consistency unit. As in Section 3.3, we employ a small dataset to measure the system under periods of heavy contention – all of the implementations work well when sharing is infrequent. For each experiment, we run a directory server process on each node. We vary the number of nodes from 2 to 48. Nodes run FreeBSD 4.7 and are connected by a 1Mbps, 40-msec latency WAN link. Each server executes an update-intensive workload consisting of 10,000 random operations run at full-speed (i.e., no think time) on its local database replica. The operation mix consists of 5% adds, 5% deletes, 20% updates, 30% lookups, and 40% cursor-based scans. Reads (lookups and cursor-based scans) are performed directly on the database file, while writes (adds, deletes, and updates) are sent to the local Kserver. Each operation opens a Khazana session on the database file in the appropriate mode, performs the operation, and closes the session.

Figures 12 and 13 show the average throughput observed per replica for reads and writes. In addition to the Kdb results, we present baseline performance when directly operating on a database file stored in the local file system (*local*), when invoking RPCs to a colocated berkeleyDB server (*rpclocal*), and when accessing a local Khazana-based database file with no contention (*klocal*). Klocal represents the best throughput using Kdb on top of our Khazana prototype. The high cost of socket-based IPC between the directory server and Kserver processes account for the performance degradation compared to `local`.

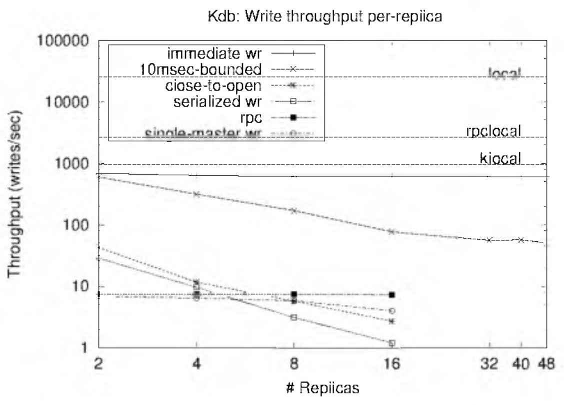

Figure 13: Kdb Per-replica Write Throughput

Figure 12 shows that read throughput scales well when we request soft (push-based) or time-based consistency guarantees, but not when we request hard (firm pull-based) guarantees, as expected. Due to the update-intensive nature of the workload, there is almost always a write in progress somewhere in the system. Thus the strict pull-based schemes are constantly pulling updates across the WAN, suffering the same high latency that the RPC-based solution inherently incurs.

Close-to-open and strong consistency do not scale beyond 16 replicas as expected, given the high degree of write-sharing in the workload. Eventual consistency scales well to large replica sets even when pushing updates eagerly. Timer-based consistency, wherein a replica pulls updates from neighbors only periodically, also significantly improves read and write performance, even with a very small staleness factor (10msecs).

In summary, different consistency options provide vastly different semantics and performance characteristics, even for the same workload. Khazana enables an application to choose the right semantics based on specific need at hand.

4 Related Work

Providing coherent shared data access across variable quality networks has been extensively studied by previous work. However, previous solutions either target specific application domains [12, 4], adopt a monolithic design unsuitable for reuse [16, 15], or lack customizability for efficiency [3, 24].

Numerous consistency schemes 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, 7], Distributed file systems such as NFS, Pangaea, Sprite, AFS, Coda and Ficus target traditional file access with low write-sharing among multiple users. Khazana supports the consistency flavors of all these systems in a peer-to-peer setting. Bayou [4] explored optimistic replication and epidemic-style propagation of operational updates in the context of database applications under ad-hoc connectivity. we employ a hierarchical replica topology in a more connected environment to achieve more bounded synchronization.

Flexibility in consistency management has often taken the form of supporting multiple selectable consistency policies for application data (e.g., Munin [2], WebFS [24], Fluid replication [3]). In contrast, Khazana offers primitives that can be combined to yield a variety of policies. Our approach is closer to that of TACT [25] but more conservative, as our goal is scalable performance under aggressive replication for several popular applications.

Several solutions exist to manage coherence of aggressively replicated data over variable quality network links. However, most previous work has targetted specific data access patterns. 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. Pangaea [20] provides eventual consistency among files hosted by wide-area peer file servers. They organize replicas as a graph. Our use of a hierarchy to organize replicas helps avoid version vector sizes proportional to the total number of replicas. PAST [17], Kazaa and several other systems manage large-scale peer sharing of read-only files such as multimedia, but do not address updates.

The OceanStore [6] 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. Khazana's goals are more specific, namely, to provide a flexible middleware for managing shared data in diverse services behind a simple API.

Lastly, many techniques have been proposed to maintain consistency among data replicas for fault-tolerance in the face of network partitions (e.g., Deno [11]). These are complementary to our work which is to provide a flexible middleware that enables these techniques to benefit a variety of services.

5 Conclusions

In this paper we demonstrated the feasibility and value of implementing a wide area storage middleware that effectively serves the data access needs of a variety of wide-area services with diverse characteristics.

We outlined the description of a middleware called Khazana that we implemented and showed that it exploits locality more effectively while accurately meeting consistency needs of a variety of services.

Khazana builds a scalable caching network and gives 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 and consistency mechanisms used to manage data. These hooks can be employed in various combinations to yield a rich variety of consistency semantics. We evaluate Khazana in the context of three diverse distributed applications: a file system, an enterprise object proxy and a replicated BerkeleyDB database. Despite the very different data management requirements of these three applications, Khazana is able to effectively detect and exploit locality while ensuring correct semantics.

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 provides aggressive replication, customizable consistency and significant generality in the same system.

References

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] A. Fox, , and E. Brewer. Harvest, yield and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Mar. 1999.

[9] Gnutella. `http://gnutella.wego.com/`, 2000.

[10] 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.

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

[12] 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.

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

[17] 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.

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

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

[20] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, pages 15–30, 2002.

[21] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the SUN Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

[22] V. Srinivasan and J. Mogul. Spritely NFS: experiments with cache-consistency protocols. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 45–57, Dec. 1989.

[23] 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.

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

[25] 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.