Segank: A Distributed Mobile Storage System

Sumeet Sobti* Nitin Garg* Fengzhou Zheng* Junwen Lai* Elisha Ziskind*
Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

This paper presents a distributed mobile storage system designed for storage elements connected by a network of non-uniform quality. Flexible data placement is crucial, and it leads to challenges for locating data and keeping it consistent. Our system employs a location- and topology-sensitive multicast-like solution for locating data, lazy peer-to-peer propagation of invalidation information for ensuring consistency, and a distributed snapshot mechanism for supporting sharing. The combination of these mechanisms allows a user to make the most of what a non-uniform network has to offer in terms of gaining fast access to fresh data, without incurring the foreground penalty of keeping distributed elements on a weak network consistent.

1 Introduction

In this paper, we study the construction of a mobile storage system designed to work on distributed storage elements connected by a network of non-uniform quality. The target environment of our system is one where all storage elements are connected with each other, but only some storage elements, typically those that are close to each other, enjoy high-quality links.

1.1 The Target Environment

This non-uniformly connected world is the reality today and it is continuing to evolve. There are three aspects of this development. First, low-cost short-range wireless technologies, such as 802.11 and Bluetooth, are proliferating. These technologies allow mobile elements in a small neighborhood to be spontaneously connected with each other at a level of quality that is quite good. Second, when a fast WiFi "gateway" into the Internet is not available, low quality wireless connectivity may still be available. Technologies, such as cellular modems, that provide ubiquitous connectivity in the wide area are now very affordable. Third, stationary storage elements are becoming increasingly wired, and they are "always on" the network. These may not only include computers in offices and server rooms, but also broadband-connected computers at home and hotels, and an increasing array of entertainment appliances, such as Tivo-like personal video recorders. The connectivity quality among these devices also exhibits a high degree of variance. A typical DSL-connected home computer, for example, may only have an up-link capacity around 100 Kbps.

Despite the high variance in connectivity quality, total disconnection is (or can be) increasingly rare, as those who own BlackBerry email devices, and those who experiment with Internet access on transcontinental flights are beginning to realize. While our system has provisions to cope with it, total disconnection is *not* our top focus. Instead, our focus is to cope with a non-uniform but always-on interconnect linking distributed and mobile storage elements.

1.2 Requirements

We begin by considering some example usage scenarios. A user owns several computers. Perhaps some of them are in his office, some at his DSL-linked home, and some in an "off-site" office in a different city, which he occasionally visits. Some of them are desktop machines, and others are laptops and PDAs that may accompany the user when he travels.

When the user arrives at his office, some of his latest work may have been done on a laptop that he carried home the night before and is still with him. At this time, the user should not be forced to wait for all the new data to propagate from the laptop to the office desktop before he is allowed to resume work on the desktop. He should be able to see and operate on the complete and latest view of his data from his desktop immediately. Also, the user should not have to remember where the latest copy of a particular piece of data is.

The next day, the user may run into a colleague on a train and the two spontaneously decide to share some files. In this case, the system should try its best to satisfy the requests using the ad hoc 802.11 link between the two laptops, and resort to a cellular modem to reach data that

*Department of Computer Science, Princeton University, {sobti, nitin, zheng, lai, eziskind, rywang}@cs.princeton.edu.
†Department of Computer Science, Yale University, arvind@cs.yale.edu.

Krishnamurthy is supported by NSF grants CCR-9985304, ANI-0207399, and CCR-0209122. Wang is supported by NSF grants CCR-9984790 and CCR-0313089.

is only available at the office or home. On a third day, when the user is again on the train, and a colleague in the office tries to read some of his files, the system would instead attempt to satisfy the colleague's requests using a copy stored on the office LAN, on a DSL-linked home machine, or on the cellular modem-connected laptop on the train, in that order of preference.

We summarize the requirements of the system. In our target system, data may be stored on, moved to, and replicated at any device for performance optimization and reliability purposes. No device necessarily houses all the data. A user sees a single image of name space spanning all the devices. A user experiences coherent semantics even when he sends read and write requests into the system from different entrance points of the network. Data and metadata propagations can happen in the background; but no foreground propagation is mandatory for the user to be able to start using a consistent system immediately. An additional requirement that we desire to fulfill is to provide a storage or file system-level solution that can transparently cater to most existing applications.

1.3 The Segank System

We call our system Segank (pronounced *see-gank*). It must solve three key problems: (1) how does the system locate data that can be stored on any subset of devices and how does it choose a best replica? (2) without costly mandatory propagation, how does the system ensure consistency across multiple devices as old data on these devices becomes obsolete? (3) how does the system ensure a consistent image across all devices for the purpose of sharing and backup?

Segank solves the first problem using a location- and topology-sensitive multicast-like solution (Section 3). The advantage of this solution is that it minimizes global state, allows autonomous data movement decisions, and can effectively exploit locality. The system solves the second problem using lazy peer-to-peer propagation of invalidation information (Section 4). The combination of the laziness element and the decoupling of the propagation of invalidation information from that of data minimizes the cost of bringing weakly connected devices up-to-date. The system solves the third problem using a distributed snapshot mechanism (Section 5). This solution allows one to flexibly trade off freshness of data against performance when facing weak connectivity.

2 Background

2.1 Naive Approaches

Solutions that indiscriminately tax a weak wide area connection are unlikely to be adequate, at least in the foreseeable future. The much anticipated 3G wireless networks, for example, are designed to ultimately achieve 384 Kbps, but industry observers agree that wide availability of such speeds is many years away. Today, most US 3G users can realistically expect data speeds of somewhere between 40 to 80 Kbps, a far cry from the hypothetical speeds of 144 Kbps and 192 Kbps [24]. Two users who meet on a train, for example, are unlikely to be able to communicate and collaborate productively by separately connecting to a remote stationary file-server via weak WAN connections.

The other extreme approach is to avoid using networks altogether. Instead, a user would rely exclusively on a mobile storage device to carry all of his data. This approach, however, is also unlikely to be adequate for several reasons. First, despite the capacity improvement of storage devices, the nature of new applications' appetite for storage is such that the capacity of a single portable device is unlikely to be sufficient for all of a user's storage needs. The capacity of the mobile devices is likely to continue to lag behind that of their stationary counterparts, and we expect much data, such as TV programs recorded on a "Tivo," to continue to reside on these stationary devices. Second, mobile storage devices tend to have poorer performance compared to desktop versions due to considerations such as energy consumption, noise, and form factor. Last, but not least, mobile storage devices, by themselves, provide little support for transparent data sharing among collaborating users.

2.2 Existing Systems

To understand the different challenges posed by non-uniform connectivity and disconnection, let us start by considering the Bayou system [18, 22]. Each Bayou device houses a complete replica of a database, and alternates between two distinct states of operation: "disconnected" and "merging." In the disconnected state, the user of the device only "sees" local state stored on this device. In the merging state, the device communicates with a peer device, and new updates made on each are played onto the other.

While the Bayou model may make sense in a disconnected environment, it is less appropriate for a non-uniform network of storage elements. First, the requirement of housing complete replicas on each device may be unnecessary, expensive, and in some cases, even infeasible. Second, being required to work on a device in a "disconnected" mode is overly restrictive when (potentially fresher) data stored on other devices could have been made available over a network. In Bayou, the only way to access data on other devices is to perform a merge operation with them, play their updates onto the local device, and then read data from the local device. Merging can be time-consuming as it propagates both meta-data and data, and forcing a user to wait until merging finishes can be inconvenient.

While the initial Coda system [11] shares Bayou's disconnected model of operation, and is equally problematic for a non-uniform network, later enhancements extend the system to work with a weak network [13]. The more serious problem with Coda is its lack of support of peer-to-peer interaction. Coda differentiates "clients" from "servers" and peer clients do not communicate with each other directly. Each data item has a fixed "home" on the server and clients are always required to "reintegrate" their updates back to the server. Requiring nearby devices to communicate only with a far-away server becomes too strict a constraint when peer-to-peer interactions could have worked well. One additional disadvantage that Coda shares with Bayou is its potential high cost of "merging:" any updates must be played to a server before they become visible to other Coda clients.

A class of existing file and storage systems that do address the missing elements of Coda and Bayou are the peer-to-peer systems: they do not require any machine to house a complete replica; they take advantage of an always-on network; they allow peer-to-peer interactions; and they do not mandate expensive propagations. They, however, exhibit their own problems when exposed to a mobile environment, a context that they have not been designed for. A key problem that a system like Gnutella [6] fails to address is consistency. For example, a mobile user may issue read and write requests into the network of devices from different entrance points, and the user is not guaranteed a consistent view, as data copies of different levels of freshness may coexist in different parts of the network. More recent wide-area peer-to-peer file systems employ distributed hash table-based (DHT-based) placement algorithms [2, 14, 20]. One problem with this approach is that the hash algorithms dictate the placement of data. On the other hand, in our target environment, we need to be able to control data placement and replication in a more flexible manner. (We discuss other issues of DHT-based systems in Section 9.)

Cluster file systems allow data to be stored flexibly in a fast LAN [1, 12, 23, 16]. These networks, however, have a simple, homogeneous topology that behaves more like a storage backplane, allowing these systems to freely manipulate cohesive distributed data structures. In our target environment, we must exercise care not to overuse weak networks. Data structures that are carelessly spread across many nodes separated by slow links, for example, are unacceptable. Also, the work of keeping distributed storage elements consistent needs to be pushed to the background as much as possible.

2.3 Minimizing Foreground Propagations

Segank allows data stored on distributed devices or owned by different users to be used without mandating expensive foreground propagation among different devices. This is one of the key features that differentiates Segank from systems such as Coda and Bayou. A Bayou client relies exclusively on a single device to satisfy its read requests. Similarly, a Coda client relies on its hoard (and the server in later enhancements.) In order to "see" new data written by other clients, mandatory propagation of all updates must occur to bring these devices up-to-date. Such propagation can be expensive on a weakly connected network. A Segank data consumer, on the other hand, is not dependent on any single device. Segank provides a fresh and consistent view of the entire system even when none of the individual devices is entirely "fresh" by itself.

3 Reading Data Using Segankast

Upon a read request, Segank needs to find out which devices have the desired data, and it needs to choose a device to retrieve the data from. In this section, we discuss the data location mechanism. We consider a single reader in this section and defer the discussion of multi-user sharing to Section 5.

A Segank user carries a small device that we call a MOAD (MObile Air-linked Disk.) Transparent to the user, the device plays four roles: (1) storing small amounts of invalidation information that can be quickly accessed to guarantee a consistent view of the system; (2) optionally caching and propagating data to improve performance; (3) providing short-range WiFi connectivity to peer devices (via 802.11 or Bluetooth) whenever possible; and (4) providing wide-area connectivity to far-away always-on devices (via a cellular modem) as a last resort when faster connectivity is not available. We conjecture that an industrial strength version of the MOAD can be packaged in a form factor that is not much larger than a wrist watch. There is, however, nothing special about the hardware requirements of a MOAD, and a PDA or a laptop can serve as a MOAD if it has the required communication capabilities. In our prototype, a Compaq iPAQ equipped with an IBM 1 GB Microdrive is used as the MOAD.

3.1 Drawbacks of Location Maps

One plausible solution to the data location problem is to maintain a mapping from an object ID to a list of devices where a replica of the object can be found. The map itself is too large to be stored on any one device or to be replicated on all devices, so the map needs to be distributed. To cope with a non-uniform network, the system needs to be able to store and replicate pieces of the map flexibly as it does data; so a higher-level map of map is needed. This leads to a hierarchical map solution where the highest level map should be compact and perhaps easier to manage.

This approach, however, has its drawbacks. Any data movements, such as caching data at, pushing data to, and evicting cached copies from devices, require reading and/or updating the multi-leveled location map. These operations may involve significant complexity as the system must exercise care to keep various pieces of distributed state consistent with each other. These operations may also introduce extra costs associated with extra network messages and I/Os. Furthermore, the location map approach, in itself, does not answer the question of which copy to actually read when there is more than one to choose from.

3.2 Segankast

Segank does not use location maps. Instead, it employs a mechanism that is similar to multicast: the system queries a number of devices until it locates one that has the desired data. We call this mechanism Segankast. Segankast is different from the data location mechanism used in Gnutella [6] in two important ways: it guarantees a consistent view of the system as a mobile user reads and writes at different locations of the network (described in Section 4.2); and it carefully controls the order, type, and parallelism of the requests to optimize performance (described in Section 3.3).

Segankast has several advantages over the use of location maps for our purposes. The system may freely place, move, replicate, or purge data on any device without having to update location information stored elsewhere. Each device is therefore autonomous. There is no risk of data and its map becoming inconsistent with respect to each other, and there are no complications resulting from, for example, attempting to access map information that is stored farther away than data, or map information that is not reachable although the corresponding data is.

Like user-level multicast systems, Segankast requests are issued over an overlay tree rooted at the current reader device. The tree includes only the devices owned by a single user. We do not envision this number of devices in a single tree to be massive. The tree is location-sensitive, so when a device is at a new location, a new tree rooted at the device is constructed. Figure 1 shows an example Segankast tree.

3.3 Optimizing Segankast Performance

There are two types of potential performance cost. The first is the latency incurred querying devices that do not contain the desired data. Segankast must carefully control the ordering of its queries. For example, if the desired data is found across a wide-area or a modem link, the extra time spent querying devices on a nearby fast LAN is relatively insignificant. The second type of potential cost is the network contention resulting from multiple data replies. It should be noted, however, that

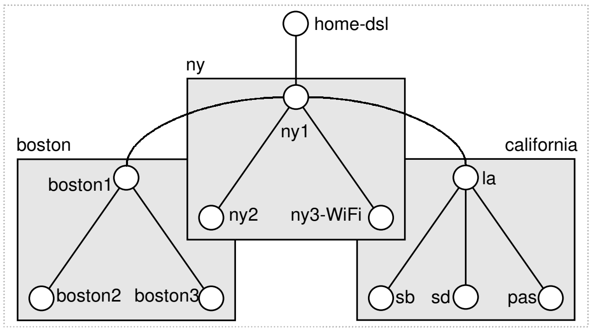

Figure 1: An example Segankast tree. A rectangle represents a "cluster" and the circles are machines.

not all types of contention necessarily can lead to visible Segankast cost. For example, suppose three hosts, A, B, and C share a single fast LAN; if A and B send reply data to C in parallel, while C forwards only one reply over a modem link to a requester R, even though A and B generate contention on the LAN, the contention is not visible to R. These two types of potential costs can be traded off against each other: for a small amount of data reply, for example, minimizing the latency of the request is more important than minimizing the contention of the replies, so one may choose to increase the degree of parallelism in Segankast. These goals make the optimization problem faced by Segankast quite different from that faced by traditional user-level multicast systems [9, 4].

The problem of optimizing Segankast performance has two sub-problems. One is determining the structure of the Segankast tree; and the other is deciding how queries are forwarded on the chosen tree. We note that the following Segankast strategies are only heuristics; better solutions to the optimal Segankast problem may be applicable to other contexts and they remain a research focus.

3.3.1 Construction of Segankast Trees

Trees are constructed based on probing measurements. The measurements and the resulting trees can be "cached" and reused across sessions to minimize probing. Lighter-weight and simpler probing can then be used to choose from appropriate existing trees.

Tree construction proceeds in two steps. Step one constructs *clusters*, whose members are close to each other and at approximately equal-distance from the single reader device at the root. Each cluster is a subtree. Step two connects the clusters to form a larger tree. The cluster-based two-step approach allows us to simplify the tree construction via divide-and-conquer.

We use the example in Figure 1 to illustrate the heuristics used in tree construction. The example includes a DSL-connected home machine in New York, two LAN-connected clusters in two offices in New York and Boston, and a WAN-connected cluster of machines in four cities in California (in colleagues' offices, for example).

In step one, we incrementally form clusters by considering the non-root nodes in increasing order of their latencies from the root node. We begin with a single cluster containing only the node (`ny1`) with the lowest latency from the root (`home-dsl`). Incrementally, we attempt to attach the next node to one of the previously-added nodes. The position of attachment is determined by a "cost" function that we describe below. In the example, the costs of attaching `ny2` and `ny3-WiFi` to `ny1` are low, so all these nodes are declared to be in the same cluster (`ny`). If regardless of where it is attached, the "cost" of attaching a node is high (i.e., exceeds a heuristic threshold), the node is declared to start a new cluster of its own. For example, the cost of attaching `la` to any node in the current set of clusters is high, so it starts a new cluster.

We now define the cost function used. Let r be the root node. Let $t(r \leftarrow x)$ be the time spent sending a block from x to r directly, and $t(r \leftarrow y \leftarrow x)$ be the time spent sending a block from x to r via y. We define the *cost* of attaching x to y to be:

$$\frac{t(r \leftarrow y \leftarrow x) - t(r \leftarrow x)}{t(r \leftarrow y)}$$

The numerator represents the penalty incurred by an extra hop (which can be negative if the overlay route is better than the Internet route), and it is normalized against the distance to the intermediate node in the denominator. At the end of step one, the heuristics produce the three disconnected clusters in Figure 1.

Step two connects clusters to form trees. We build separate trees to optimize for latency and bandwidth. Latency refers to the cost of fetching a single data object (a file block in our prototype). Bandwidth refers to the rate at which a stream of blocks are fetched. Hints available in softwares running on top of Segank can be used to determine which tree to use. During this step, we only consider the root nodes of the cluster subtrees (`ny1`, `boston1`, and `la` in the example), and the root of the tree (`home-dsl`). To form the Segankast tree designed to optimize latency, we consider the complete graph spanning these nodes. We annotate each edge by the round-trip time of fetching a block between a pair of nodes, and compute the shortest-path-tree rooted at the root node. To form the Segankast tree designed to optimize bandwidth, we annotate each edge of the complete graph by the inverse of the edge bandwidth, and compute the minimum-spanning-tree. In Figure 1, we show how to connect the cluster subtree root nodes with the reader device to form a complete tree designed to optimize bandwidth.

3.3.2 Forwarding Segankast Requests

Upon receiving a request, each node in a Segankast tree has two decisions to make: whether to forward the request in parallel to its children or sequentially, and the type of messages to send. In terms of the second decision, there are two choices: a direct *fetch*, or a *test-and-fetch* that first queries which children (if any) have the desired data and then issues a separate message to retrieve data from a chosen child. These two decisions can be combined to form four possible ways of forwarding: (1) parallel fetch, (2) parallel test-and-fetch, (3) sequential fetch, and (4) sequential test-and-fetch. Of them, the first three possibilities are viable.

We use the following strategy to choose how to forward requests at each node. If the message received from the parent is a parallel test-and-fetch, we simply propagate it in parallel to all children. If simultaneous replies from all children can exceed the bottleneck bandwidth to the reader device, we use parallel test-and-fetch. Otherwise, we use parallel fetch. In the near future, we plan to incorporate location hints of target data, and sequential fetch, which is not currently used in the prototype, may become appropriate.

4 Maintaining Consistency

When data is deleted or overwritten, devices that house obsolete copies need to be "informed" so the storage space can be reclaimed. Furthermore, we must ensure that Segankast does not mistakenly return obsolete data even when a mobile user initiates requests from different entrance points of the network. We desire to achieve these goals without mandating foreground propagations of either data or metadata. In this section, we consider operations involving a single owner/writer, who always has his MOAD device with him. We consider sharing (reading/writing by multiple users) in Section 5.

4.1 Propagating the Invalidation Log

A naive solution is to send invalidation messages to all devices belonging to the owner. Due to the non-uniform network, however, some of the devices may be poorly connected, so foreground invalidation is not always feasible. Lazy invalidation is especially appealing when the quality of connectivity may change significantly due to mobility. For example, it may be desirable to delay invalidating obsolete data stored on an office computer until its owner, who has been creating new data away from the office on his weakly connected laptop, returns. Upon his return, however, we should not force the user to immediately propagate either data or invalidation information to the office computer while he waits. Instead, the user should be able to start using the system immediately without risking reading obsolete information. We now examine the details of lazy invalidation.

Each write in Segank is tagged with a monotonically incrementing counter, or a *timestamp*. (This is a local counter maintained on the MOAD.) The sequence of write operations in the increasing timestamp order is called the *invalidation log* of the system. Specifically, the invalidation log entry of a write operation contains the ID of the object written and the timestamp. Each device stores its data in a persistent data-structure that we call a *block store*. The meta-data stored with each data object includes the timestamp of the write operation that created the data.

Each device also has a persistent data-structure to store the invalidation log. Segank, however, does not force any device (except a user's MOAD, as we discuss below) to store the complete invalidation log. In fact, the portion of the invalidation log stored on a device may not even be contiguous.

We assume that the MOAD houses the most complete invalidation log as it follows the user (who is the sole writer for the purpose of our present discussion). The head of the log can be truncated once it has been sent to all this user's other devices. The size of the log is bounded by the amount of new data writes performed in a certain period of time, which should be smaller than that of a location map, since a location map must map all the data in the system. Parts of the log can also be stored on other well-connected devices if the capacity on the MOAD becomes a premium. Since the MOAD is always with the user and it can communicate using at least the wireless modem link, the entire invalidation log should always be reachable. Also, it is easy to turn any other device (a laptop, for example) into a MOAD simply by transferring the invalidation log onto it, provided the device has the same communication capabilities as the MOAD. Therefore, to simplify the rest of the discussion, we assume that the device that a user works on is a MOAD, and that it contains the entire invalidation log.

As the user works on a MOAD device and creates data, new entries are appended to the invalidation log on the device. This new tail of the log is propagated to other devices in the background. Log propagation is a peer-to-peer operation that can happen between any two devices. If a device houses a piece of the log that another lacks, then log propagation can be performed. For efficiency, in the normal case, log propagation is performed along the edges of the Segankast tree, especially those that correspond to high-quality network links. It must be noted, however, that all propagation is performed in the background.

Having received a portion of the log, a device may decide to "play" the log entries onto its block store at any convenient time. Playing a log entry onto the block store means discarding any data that is overwritten by the operation in the entry. For correctness (especially of the snapshot design described in Section 5.2), log entries are played only in strict timestamp order. We define *freshness* of a device to be the timestamp of the last log entry played onto it. Devices other than the MOAD are not expected to store the log forever. Log fragments may be discarded at any time after having been played.

Note that only the invalidation records need to be propagated in the background and no data exchange is necessary to ensure a consistent view of the Segank system. The amount of the invalidation information should be at least three orders of magnitude smaller than that of data. This is in contrast to existing systems where data and metadata propagations are intertwined in the same logs [11, 13, 14, 18, 22].

All the devices in a Segank system are very much similar to each other. One difference between the MOAD and the other devices is that the MOAD is guaranteed to have the most complete invalidation log. (Although as we have said, even this difference is not strictly necessary.) As long as the MOAD is with the user, it ensures fast access to the invalidation information, which as we describe in the next section, is sufficient to ensure a consistent view of the system without mandating any type of foreground propagation.

4.2 Querying Invalidation Logs for Reads

The discussion so far may seem to have implied that it is necessary to bring a device up-to-date by first playing invalidation records to it before the system can allow the device to participate in the Segankast protocol to satisfy reads. Such a requirement would have violated our requirement of avoiding mandatory foreground propagations. We now explain how this is not the case.

Suppose a user is working on his laptop MOAD device with the most complete invalidation log. We maintain a sufficiently long tail of the invalidation log in a hash-table like data-structure that supports the following operation: given an object ID, it locates the last write (and the corresponding timestamp) to that object in the tail. We refer to this portion of the invalidation log as the "queryable log." Upon a read request, the system queries the queryable log to look for the latest write to the requested object. If an entry for the object is found, the system launches a Segankast request, specifically asking for an object with the timestamp found in the queryable log. When any device (including the local device) receives this Segankast query, without regard to its own freshness value, it queries its block store for the object with the specified timestamp. When the data is found on some device, the read request is satisfied.

If no entry for the object is found, it implies that the data has not been overwritten during the entire time period reflected in the queryable log. Suppose the head of the queryable log on the MOAD has a timestamp of t_0. The system then launches a Segankast request asking only devices with freshness at least $t_0 - 1$ to respond.

An invariant of the system is that all the devices reachable by Segankast at this moment must be at least as fresh as $t_0 - 1$. This invariant, however, does not imply that the complete invalidation log must be queryable: older portions of the invalidation log that are kept for currently-unreachable devices need not be queryable. A device can be beyond the reach of Segankast because, for example, it is currently disconnected, in which case there is no danger of reading obsolete data from it. When such a device later becomes reachable again, the system must restore the invariant by either making more of the older portion of the log queryable, or by playing this older portion of the log to the newly connected device to upgrade its freshness up to $t_0 - 1$. This invariant makes it possible to cache the queryable tail of the invalidation log entirely in memory, minimizing overhead paid on reads. Note that this protocol handles disconnection without extra provisions. It also allows the system to flexibly choose how aggressively the invalidation log should be propagated: a weakly-connected device need not receive such propagations while still being able to supply consistent data. The overall effect of this protocol is that a consistent view of the system is always maintained without any mandatory foreground propagation, even though individual devices are allowed to contain obsolete information.

4.3 Data Movement and Discard

As explained earlier, an important feature of Segank compared to some existing epidemic exchange-based systems is that the propagation of the invalidation records and that of data can be decoupled. Data movement is mostly a performance optimization, and it is largely decided by policy decisions. At one extreme, an aggressive replication policy effectively can also support disconnected operation. One goal of the Segank design is to allow individual devices or subsets of devices to autonomously make data movement/discard decisions without relying on global state or global coordination. There are, however, still some constraints.

One of them concerns data movement: data is only sent from fresher devices to less fresh devices. This constraint ensures that if the propagated data is overwritten, the corresponding invalidation record is guaranteed to not have been played to the data receiver prematurely. Interestingly, there is no constraint on the relationship between the timestamp of the propagated data and the freshness of the receiver device. Another constraint is that we need to exercise care not to discard a last lone copy of the data. We adopt a simple solution: when data is initially created, a *golden copy* is established; and a device is not allowed to discard a golden copy without propagating a replacement golden copy to another device.

5 Sharing with Snapshots

So far, we have only considered system support for a single owner of his devices and data. We now consider sharing using a "snapshot" mechanism—a *snapshot* represents a consistent state of an owner's data "frozen" at one point in time.

5.1 Requirements of Sharing

Let us examine an example scenario. On day 1, users A and B are collaborating in a well-connected office. A may wish to promptly see fresh data being continuously produced on a desktop by B. Over night, some of the data produced by B may be propagated to a laptop of B's. On day 2, B takes his laptop onto a train, where only a cellular modem is available, and he continues to modify some (but not all) of his data.

On day 2, A has a continuum of options available to him when accessing B's data in terms of how fresh he desires the data to be. (1) A may decide that the data produced by B on day 1 is fresh enough. In this case, A's read requests are satisfied entirely by B's office desktop without ever using the cellular modem. Again, this is effectively supporting disconnected operation. (2) A may desire to see a snapshot of B's data at, say, noon of day 2. For the data that B has not modified by noon, A's read requests may be satisfied by B's office desktop; but occasionally, A may need to use the cellular modem to access a piece of data produced by B before noon on day 2. (3) A may desire to see a new snapshot of B's data, say, every minute. A now uses the cellular modem more often.

We summarize some requirements. First, consider case (2) above. In order for A to read a piece of data written by B shortly before noon, A should not have to wait for B to flush all the data that B has produced by then. In fact, we should not even necessarily force B to flush its invalidation information. A should be able to read whatever data whenever he desires on demand. In other words, no mandatory propagation of any kind should be necessary to guarantee a certain degree of freshness. This requirement is not limited to case (2)—it is a general property of Segank. Second, facing a non-uniform network, a user should be able to precisely control when and how often a new snapshot is created for the shared data to trade off freshness against performance.

5.2 Snapshots

To support snapshots, we rely on a copy-on-write feature in the block store. A snapshot is created or deleted simply by appending a snapshot creation or deletion record to the invalidation log. These operations are instantaneous. The in-order propagation of the invalidation log among the devices ensures that data overwritten in different snapshots is not inadvertently deleted. Each snapshot is identified by a *snapshot ID* based on a monotonically-increasing counter kept persistent on the MOAD. (This counter is different from the counter used to assign timestamps to write operations.) The snapshot named by the ID contains the writes that have occurred between the creation of the previous snapshot and this snapshot. In the rest of this discussion, the timestamp of a snapshot is understood to be the timestamp of the creation record of the snapshot. Since a snapshot must be internally consistent, the decision as to when to create or delete a snapshot must be made by higher-level software (e.g., the file system) or the user.

5.3 Read Sharing

When user A reads data created by user B, we call A a *foreign reader*. The foreign reader first chooses a snapshot of a desired recentness. To do this, the reader device contacts a device belonging to the writer. To obtain the most recent snapshot ID, the reader must contact the MOAD owned by the writer.

Suppose the reader desires to read an object from a snapshot with timestamp T_S. In the cases where either the devices with the desired object have freshness at least T_S, or where the reader happens to have a long enough tail of the invalidation log ending at T_S, the description of Section 4.2 still applies. In other cases, more generally, we need to extend the earlier description.

Recall that the read algorithm given in Section 4.2 requires the reader to query the invalidation log to ensure consistency. This querying yields a timestamp of the desired data if it is written in the period covered by the invalidation log, or the timestamp of the head of the log. In this earlier discussion, when the reader and the writer users are the same, the complete invalidation log is available on the single user's MOAD for fast querying. In the case of a foreign reader, however, this assumption no longer holds. A simple solution is to first query the invalidation log stored on the remote MOAD owned by the writer for the timestamp. Once the reader obtains the timestamp, the rest of the read process remains the same as described earlier. The disadvantage of this approach is that the writer's MOAD device may be weakly connected, and querying it for all reads can be expensive.

To overcome the inefficiency, we note that fragments of the invalidation log may have been propagated to other devices (in the background), some of which may be better connected to the reader device. Indeed, some or all of the invalidation log fragments may have been propagated to the reader device itself. One possible improvement is to split each Segankast into two phases: a first phase queries the devices to obtain the target timestamp in the invalidation log fragments, and a second phase retrieves the data as described earlier. (A modified parallel test-and-fetch strategy given in Section 3.3.2 is best suited for these foreign reads.) In a more sophisticated improvement, it is possible to combine these two phases into a single one in certain cases. Due to lack of space, we omit the details of these improvements.

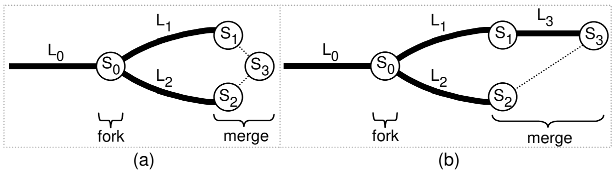

Figure 2: Snapshots for write-sharing. (a) Without conflicts, (b) with conflicts.

The mechanism described above ensures that a foreign reader can read a snapshot of certain recentness without having to wait for foreground propagation of either data or the invalidation log. However, as we see in Section 8, the data-less invalidation log can be propagated efficiently in the background on most networks. By default, Segank aggressively propagates the invalidation log.

5.4 Write Sharing

We start by considering two writers (illustrated in Figure 2.) The two users begin with a single consistent file system. To start concurrent write-sharing, they perform a *fork* operation, which names the initial snapshot as S_0, and allows the two users to concurrently write into two new snapshots, S_1 and S_2, in isolation. New updates by one user are not visible to the other, until when the two users desire to make their new data available to each other by performing a *merge* operation. Prior to the fork, the invalidation log is L_0. Prior to the merge, the two users' new writes result in two separate invalidation log fragments L_1 and L_2.

The first step of the merge operation is conflict detection. This is an application-specific process that should be dependent on the nature of the software running on top of Segank. The three snapshots S_0, S_1, and S_2 are available to the conflict detection process as inputs. In our prototype, we lay a file system on top, and the three snapshots manifest themselves as three distinct file systems. In theory, a possible way of implementing conflict detection in this case is to recursively traverse the three file systems to identify files and directories that have been modified and to determine whether the modifications constitute conflicts. This slow traversal, however, is not necessary. As modifications are made to S_1 and S_2, the higher-level software should have recorded book-keeping information to aid later merging.

In our prototype, we modify the file system running on Segank to capture path names of modified files and directories. This information is summarized as a tree of modification bits that partially mirrors the hierarchical name space. A node in this tree signifies that *some* objects below the corresponding directory are modified. This tree of modification information is logged separately to the MOAD device and is cached. The conflict detection is implemented by comparing the two trees. In what we believe to be the common case of modifications being restricted to a modest number of subdirectories, this comparison can be quickly made even on a weak connection. The amount of information exchanged between the two nodes should be far smaller than the invalidation logs. The details of such an application-specific conflict detection mechanism, however, are not central to the more general Segank system.

If no conflict is found (Figure 2(a)), the system automatically creates a merged snapshot S_3. Due to the lack of conflicts, the ordering of L_1 and L_2 is irrelevant. The invalidation log resulting in S_3 is logically simply a concatenation of L_0, L_1, and L_2. However, it is not necessary to physically transfer the log fragments among the devices. Since the ordering of log fragments in the system invalidation log is determined by the timestamps of the head entries of these fragments, a simple exchange and reassignment of timestamps is sufficient to effect the "logical concatenation". For instance, the invalidation log for the merged snapshot could be created by retaining the entries from L_1 and assigning timestamps to the new entries in L_2 such that they logically succeed the tail entry of L_1. The outcome of the merging process is a consistent snapshot and a merged invalidation log that both users can now use to satisfy their read requests in a way similar to what is described in Section 5.3.

If conflicts are found (Figure 2(b)), an application-specific resolver or user intervention is required. The user is, again, given the three distinct file systems. Without loss of generality, suppose the user starts with S_1 and performs a sequence of manual file system modifications to it, incorporating some desired modifications from S_2, and ending up with a new snapshot S_3, the result of resolving conflicts. These new modifications result in a new invalidation log fragment L_3. Although L_1 and L_2 contain invalidation log entries resulting from conflicting updates, their ordering is still unimportant, because any such entries should be superseded by entries in L_3. The outcome of this process is a single consistent invalidation log that is logically the concatenation of L_0, L_1, L_2, and L_3. Again, no physical transfer of log fragments is necessary to enable either user to read from a consistent snapshot S_3.

The above process can be generalized to more than two concurrent writers by performing repeated pair-wise merging. We note that the process shares some common goals with the read-sharing mechanism. No foreground propagation of data or invalidation is mandatory for the concurrent writers to share consistent snapshots. As a result, in the absence of conflicts, the latencies of all snapshot operations can be kept low, enabling collaborating users to perform these operations frequently when they are well-connected, and to trade off recentness of shared data against performance when they are weakly-connected.

6 Exceptional Events and Limitations

6.1 Disconnected Devices

The Segank system is primarily designed for always-on but non-uniform connectivity: disconnection is not a main focus. Nevertheless, the handling of disconnected devices has been mentioned throughout the previous sections. We now consolidate these descriptions and clarify the limitations. First, possible disconnection and reconnection events cannot cause consistency violations. Neither invalidation log entries nor Segankast requests can reach a disconnected device. Upon reconnection, in order for the previously disconnected device to be eligible to return data, we must first restore an invariant (described in Section 4.2).

Second, as explained in Section 4.3, how data is moved is a policy decision, decoupled from the basic data location and consistency mechanisms of Segank. An important design goal of the Segank system is to accommodate almost arbitrary policies. Under an aggressive data replication policy tailored for disconnected environments, a Segankast should reach alternative replicas instead of being limited by disconnected copies. Third, if there is not enough time to replicate the freshest data, the snapshot mechanisms described in Section 5 may allow an older consistent snapshot of the system to be read instead. If one exhausts these options, data on a disconnected device would be unavailable until reconnection. Although the Segank system is designed to allow flexible data movement and replication policies, this paper does not provide a systematic study of specific policies designed to minimize potential disruption caused by disconnection and to optimize performance for various usage scenarios and environments. We plan to address this limitation in future research.

6.2 Inaccessible MOAD

We intend to have the small MOAD device accompany its owner at all times if the owner desires access to the Segank system. Now suppose we would like to loosen this restriction so that the user is still allowed access at one of the Segank computers without his MOAD device. The MOAD is like one of the many computers managed by the Segank system, so in terms of the issues of data consistency and data accessibility, the above descriptions (Section 6.1) apply to the MOAD as well. The MOAD, however, is special compared to the remaining peer Segank devices, in that it houses the most complete invalidation log, and access to this log is necessary for ensuring data consistency.

As discussed in Section 4.1, one way of increasing the availability of the complete invalidation log is to replicate it on other well-connected devices so that the single forgotten MOAD does not become a problem. One can also designate an alternate device (such as a laptop) to be the replacement MOAD by copying the invalidation log onto it and starting to append new invalidation log entries to it. If the complete invalidation log is not sufficiently replicated to be accessible and only an older portion of the log is reachable, one may use this partial log and the snapshot mechanism (described in Section 5) to access data in an older snapshot. Failing these options, one would be unable to access the Segank system.

6.3 Backup and Restore

At least two factors complicate the handling of device losses. First, because the Segank system can function as a low-level storage system, the loss of a device can result in the loss of a fraction of data blocks managed by the system, making maintaining high-level system integrity challenging. Second, the mobile and weakly connected environments within which Segank operates mean that it may not be always possible to dictate replication of data on multiple devices. Therefore, our goal is not necessarily guaranteeing the survival of each data item at all times, which is not always possible; instead, our goal is to maintain system integrity and preserving as much data as possible when we face device losses. We rely on a backup and restore mechanism for achieving these goals.

The same snapshot mechanism (Section 5) that we use to enable sharing can be used to create consistent *backup snapshots* that are incrementally copied to backup devices. Backup devices can be introduced into the Segank system in a flexible manner that is very much similar to how regular devices are incorporated. At one extreme, each device or each site can have its own backup device; at another extreme, we can have a single backup device (such as a tape) that is periodically transported from site to site to backup all devices onto a single tape; an intermediate number of backup devices are of course possible as well. Data belonging to a backup snapshot can be deleted in a piecemeal fashion as they are copied onto backup devices. During restore, one needs to restore the latest snapshot that has all its participating devices completely backed up. There are various ways of finding out the name of this snapshot and examining the state recorded on all backup devices is an obvious alternative. More details of the backup/restore mechanism can be found in a technical report [5]. Note that the set of "backup devices" are not fundamentally different from the other storage devices managed by the Segank system, so for example, even in absence of tape drives, one can designate an arbitrary subset of the regular Segank devices as "backup devices," from which we can recover from the loss of remaining devices (such as the potentially more vulnerable MOAD).

6.4 Other Issues

Due to lack of space, we describe the following issues only briefly; more details can be found in a technical report [5]. Crash recovery in Segank is simple. Communications that can result in modification of persistent storage are made atomic. It is sufficient to recover the block store on a crashed device locally, without concerns of causing inconsistent distributed data structures. Segank appears to the file system built on top as a disk and provides only crash-consistent semantics. The file system must run its own recovery code (such as `fsck`). Adding or removing devices, locating names of devices belonging to a particular user, and access control utilize simple or existing mechanisms whose details we omit in this paper.

7 Implementation

This section gives some implementation details that are less fundamental: alternative choices could have been made. For example, while the current implementation works principally at the storage level, an entirely file system-level solution could have worked too.

7.1 Volumes

We have developed the system on Linux. The owner initially makes a slightly modified "ext2" file system on a virtual disk backed by Segank and mounts it. Separate file systems are mounted for each of the other users' data whose access has been granted. We call each of these file systems a *volume*.

We use a pseudo block device driver that redirects the requests sent to it to a user-level process via up-calls. The user-level process implements the bulk of the Segank functionalities. The unit of data managed by Segank is a block, so it is principally a storage-level solution. The object IDs used in Segank are 64-bit block IDs, and each block is 4KB. The timestamps are also 64-bit. The ext2 file system uses the lower 32 bits of a block ID, to which a prefix consisting of bits representing users and attributes is added by Segank. Modifications to the ext2 file system are needed for data sharing scenarios in terms of acquiring and releasing snapshots, flushing in-kernel caches, capturing the path names of modified files and directories for detecting conflicting write-sharing, and allocating Inode numbers and block IDs in a way that ensures that IDs allocated by different users do not collide.

Segank allows flexible data replication and migration policies. We have implemented some simple ones. Each owner may define several volumes, each with its own *attributes*: a volume can be *mobile*, *shared*, or *stationary*. If the volume is defined mobile, the owner hints to the system that she would like to have this data follow her, and Segank attempts to propagate the data in the volume to as many devices as possible. If the volume is defined shared, the owner hints to the system that others may need the data, and Segank attempts to propagate the data to a well-connected device and to cache a copy on the MOAD if possible. If the volume is defined stationary, the user hints to the system that the data is most likely to be needed only on this creator device, so no automatic propagation is attempted. We recognize that these simple policies are unlikely to be adequate for many usage contexts; and more sophisticated policies are part of our ongoing research.

7.2 Block Store

The block store is a log-structured logical disk [3]. Because of the use of 64-bit block IDs (forming a sparse address space), the block store employs an in-memory B-tree to map logical addresses to physical addresses. The block store accesses the physical disk using raw I/O via the Linux `/dev/raw/raw*` interface, bypassing the buffer cache. The invalidation log is stored in a designated subspace of the 64-bit address space of the block store.

7.3 Connectivity

A responsibility of the connectivity layer is to route to the MOAD regardless where it is and what physical communication interface it uses. A MOAD can be reached via: (1) an ad hoc wireless network encompassing both the requester and the target MOAD, (2) the Internet which connects to a remote ad hoc network within which the target MOAD is currently located, or (3) the target MOAD's cellular modem interface. In our implementation, the first two cases of communication to a MOAD use an identical and fixed IP address, while the third case uses its own IP address.

To reach a MOAD in the first two cases, we need a combination of an ad hoc routing mechanism and "Mobile IP." We have implemented our own version of the "Ad hoc On-demand Distance Vector" (AODV) routing algorithm based on the draft 9 specification of AODV [17]. The implementation runs at user-level and can support multiple network interfaces. For example, a computer equipped with both an ad hoc network interface and a wired ethernet interface can participate in Segankasts. We have used an existing Mobile IP implementation: Dynamics HUT Mobile IP version 0.8.1. For point-to-point communication channels during Segankasts, the implementation supports TCP and UDP sockets and SUN-style RPCs. The experimental results are based on TCP sockets.

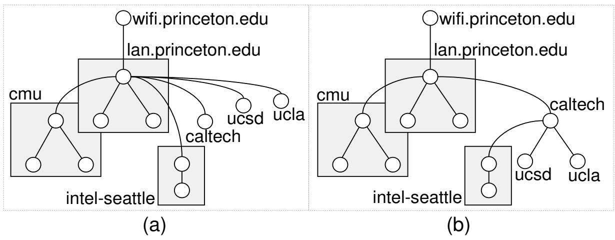

Figure 3: Segankast trees. (a) The tree for optimizing latency, (b) the tree for optimizing bandwidth.

8 Experimental Results

We first examine Segankast performance under a single user. We then study performance of data sharing by multiple users.

8.1 Segankast Performance

We now study the performance of reading data using Segankast under two usage scenarios. In the first scenario, the user is at work accessing data on a mobile device. The mobile device is connected to the wired network through an ad hoc 802.11 link. The user owns three other devices that are located in his office LAN, and eight other devices located at five different cities that are at varying distances from his current location. The mobile device along with the eleven other storage devices comprise the personal storage system for the user. We will refer to this scenario as `WiFi-Work`. In the second scenario (referred to as `DSL-Home`), the user comes home and connects to the Internet using a DSL connection. The eleven other devices are accessible over the DSL connection, but data accesses incur a substantially higher latency.

The mobile device we use in our experiments is a Dell Inspiron Laptop with a 650 MHz, Intel Pentium III processor, 256 MB of memory, and a 10 GB, IBM Travelstar 20GN disk. The remaining devices are PlanetLab [19] nodes, which are either Dell Precision 340 workstations or Dell Poweredge 1650 servers. Table 1 lists for each data source, the average latency for the reader to access a single 4 KB block from the source and the average bandwidth attained by the reader in accessing a stream of blocks from the same source under the two usage scenarios.

Figure 3 shows the trees determined by our tree building algorithm for optimizing latency and bandwidth in

Scenario	Data Source	Latency (ms)	Bandwidth (MB/s)
WiFi-Work	princeton	15	0.49
	cmu	45	0.42
	intel-seattle	157	0.32
	caltech	160	0.37
	ucsd	173	0.36
	ucla	177	0.32
DSL-Home	princeton	140	0.068
	cmu	160	0.065
	intel-seattle	189	0.065
	caltech	198	0.063
	ucsd	223	0.063
	ucla	243	0.062

Table 1: Performance of accessing data blocks directly from devices at different locations.

the `WiFi-Work` scenario. The tree built for optimizing latency is built using the shortest-path-tree calculation on a complete overlay graph, where each edge is annotated with the cost of fetching a block between the corresponding pair of nodes. The overlay paths from the mobile node through an intermediate `lan.princeton.edu` device perform better than the direct paths, and hence the mobile node has only one child. The resulting tree is still shallow since it is built to optimize for latency with the intermediate nodes capable of operating with large fan-outs due to the small payloads associated with fetching a block. On the other hand, the tree for optimizing bandwidth is built using the minimum-spanning-tree algorithm and is designed to minimize contention and maximize the use of high bandwidth links. The resulting tree organizes the devices into more levels. A parallel test-and-fetch strategy is used for propagating requests within the `lan.princeton.edu` and the `cmu` clusters since the nodes in the two clusters compete for bandwidth on a local area network. A parallel fetch strategy is used for the other connections since the cumulative bandwidth of the corresponding WAN flows was determined to be less than the bottleneck bandwidth of the parent nodes.

The Segankast trees for the `DSL-Home` scenario are similar to the ones constructed for the `WiFi-Work` scenario. The reader achieves the lowest latency and the highest bandwidth to a `lan.princeton.edu` device. Therefore, the `lan.princeton.edu` device is designated as its child in both trees. Furthermore, given the high latency and the low bottleneck bandwidth for the DSL connection, the reader can not sustain more than one child; if the reader has multiple children, two simultaneous replies to a parallel fetch of a single 4 KB block would be sufficient to overwhelm the bottleneck bandwidth, while the round-trip message overhead associated with a parallel test-and-fetch is equally unattractive. The `lan.princeton.edu` device is, therefore, the sole child of the reader, and the overlay paths from the reader to other devices could be more expensive than the direct connection.

We evaluate Segankast by running two experiments. The first experiment is designed to benchmark the performance of reading small and large files. The dataset consists of 25,000 small files each of size 4 KB and a single large file of size 50 MB. The files are initially created at the device that is farthest from the reader. We measure the cost of reading the files using Segankast under the following data distribution settings. (1) Remote: only the `ucla` device, which is farthest from the reader, has all the data, (2) Nearby: in addition to `ucla`, one of the nearby nodes at `lan.princeton.edu` has all of the files, and (3) Random: `ucla` has data, and each one of the files is replicated at three other randomly chosen locations.

For the second experiment, we use a disk trace collected on a Dell Dimension 8100 workstation running Windows 2000. During the monitoring process that lasted many days, the user was performing activities that are typical to personal computer users, such as reading email, web browsing, document editing, and playing multimedia files. We used some simple tools (such as `tracelog` and `tracedmp`) from Microsoft's Windows 2000 resource kits to collect these traces. We use a portion of this trace comprising of 787,175 I/O requests accessing a total of about 9.2 GB of data. We execute the first 760,000 requests on eleven devices in a round-robin fashion with a switch granularity of 20,000 requests. We then measure the performance of the read operations contained in the last 27,175 requests by executing them on the twelfth device that is connected to the remaining devices using a 802.11 connection (the `WiFi-Work` scenario) or a DSL connection (the `DSL-Home` scenario).

For both experiments, we also measure the cost of reading the files if there was an Oracle that provided the location of the closest device that contains a replica of the desired data, so that the reader can directly fetch the data. This allows us to bound the performance of alternative mechanisms such as location maps that track the location of the objects in the system. Recall that a lookup of the location map could involve communication if the entire location map is not locally available at the reader and that location maps require protocol traffic to maintain consistency as the data objects are propagated or discarded. Since these costs are not included in our Oracle mechanism, the resulting read performance is a generous upper-bound on the performance of an implementation that uses location maps.

The results of the two experiments are shown in Tables 2, 3, and 4. The performance measurements reveal that the overhead introduced by Segankast is minimal for

Data Layout	Segankast		Oracle	
	sread (s)	lread (s)	sread (s)	lread (s)
Remote	563	132	552	122
Nearby	76	109	74	108
Random	234	107	226	105

Table 2: Read performance for the `WiFi-Work` scenario. `sread` refers to reading the small files. `lread` is the large file read.

Data Layout	Segankast		Oracle	
	sread (s)	lread (s)	sread (s)	lread (s)
Remote	1266	711	1096	702
Nearby	781	689	778	687
Random	1126	693	1019	688

Table 3: Read performance for the `DSL-Home` scenario.

reading large files even when the file is fetched from a remote `ucla` node through two intermediate hops. The cost of routing data through higher latency overlay paths and the overheads of forwarding requests and replies are amortized by pipelining the block fetches. In the `WiFi-Work` setting, the cost of performing small file reads is only marginally worse than a direct fetch. However, in the `DSL-Home` setting, the tree designed to optimize latency has some overlay paths that are more expensive than the direct connections and the reads incur a 10-15% overhead when a copy of the required data is not available at a nearby `lan.princeton.edu` device.

8.2 Sharing Experiments

We now examine the performance of Segank when users share data. In the first set of experiments, the setup consists of a writer node sharing a Segank volume with a reader node. We measure the performance of the reader as the writer creates new data in the volume and exports a new read-only snapshot of the volume to the reader. We define the "snapshot refresh latency" at the reader to be the difference between the time when the reader expresses the desire to read from the new snapshot for the first time and the time when it is actually able to perform its first operation on the new snapshot. A key result is that the refresh latency is solely a function of the network latency between the writer and the reader. In particular, it is independent of the amount of new data created by the writer in the new snapshot. We perform several experiments where we vary (1) the type of the network link (Table 5), and (2) the amount of new data written by the writer before creating the new snapshot (Table 6).

Each experiment proceeds in three phases. The first phase initializes the Segank volume at the reader. In the second phase, the writer performs a number of updates on the volume and then, creates a new read-only snapshot of the volume. In the final phase, the reader performs a

Scenario	Segankast	Oracle
WiFi-Work	499	496
DSL-Home	1763	1714

Table 4: Execution time in seconds of a trace collected on a personal computer.

	LAN	WiFi	WAN	Modem
BW (MB/s)	11	0.5	2.2	0.01
RTT (ms)	0.2	1.4	10	110

Table 5: Observed bandwidth and latency characteristics of the network links used. LAN refers to a 100 Mb/s Ethernet. WiFi refers to using two wireless 802.11 cards in the ad-hoc mode at 11 Mb/s bit-rate. WAN corresponds to a wired connection between two machines in Princeton and Yale Universities respectively. Modem refers to connecting one of the machines via a dial-up modem.

"refresh" operation to express its desire to read from the new snapshot. Subsequently, a read-benchmark is run at the reader to read data from the new snapshot.

In the initialization phase, two identical, but distinct, directory trees are created. We name them T_1 and T_2. Each tree is 5 levels deep, where each non-leaf directory contains 5 sub-directories. In each directory, 10 files are created, each of size 8 KB. Thus, each tree has a total of 781 directories and 7810 files, comprising about 64 MB of data. The data and invalidation log created during the initialization are at the reader only.

The update phase at the writer overwrites all files in the first K levels in T_1. We perform three sets of experiments with K taking values 3, 4 and 5. We name these experiments "Small", "Medium" and "Large" respectively.

The read-benchmark run at the reader involves reading 1000 randomly-selected files from either T_1 or T_2. Thus, it reads about 7.8 MB of file data, in addition to reading some directory data and meta-data. In the rest of this section, "reading" a particular tree means running the read-benchmark for the given tree at the reader.

The left portion of Table 7 summarizes the results from this set of experiments. For the results in this table, each experiment is repeated 3 times, and the median values are reported.

The refresh latency (column 3) is observed to be determined only by the network latency between the reader and the writer, and not by the amount of new data or invalidation log created in the new snapshot. As shown in column 9, complete data propagation can take several tens of seconds or even minutes on slow networks. Thus, compared to alternatives which mandate complete data propagation and replay before allowing access to new data, Segank incurs significantly lower user-perceived la-

Experiment	Data (MB)	Log (KB)
Small	2.6	5.2
Medium	12.9	25.8
Large	62.1	124.1

Table 6: Amount of data and invalidation log generated at the writer in experiments of each type.

tency.

The columns labeled "Read Time" are listed to show the overhead of lack of invalidation log propagation on read performance of the reader. The base case, labeled "LL" (for "Local" log and "Local" data), is the time for reading any of the two trees immediately after the initialization phase. Here, the complete invalidation log and all the data to be read are present locally at the reader. The column labeled "DD" (for "Distributed" log and "Distributed" data) lists the time for reading the updated tree T_1 immediately after the refresh operation. In this case, the reader neither has the complete log, nor all the data. The "DL" case is for reading the non-updated tree T_2 immediately after the refresh operation. The "LD" case is for reading T_1 after the new log (but not the new data) has been propagated from the writer to the reader.

Comparing LL with DL, and LD with DD, we conclude that not propagating the log adds significant overhead to the read performance for all network types except the LAN. When the complete log is present locally, network communication is needed only when the required data is not present locally. When the complete log is not present locally, each read request invariably results in network communication. As column 8 illustrates, the log propagation can usually be achieved in significantly less time than data propagation. This validates Segank's default policy of being highly aggressive in propagating the log, although the propagation is only performed in the background.

A second set of experiments is performed to evaluate Segank in a scenario where multiple users write to the same Segank volume concurrently. The setup consists of two nodes A and B. As in the single-writer case above, each experiment here proceeds in three phases. In the first phase, the volume is initialized to contain two trees T_1 and T_2. The complete log and the data is propagated to both A and B. Then, in the second phase, both A and B write to the volume concurrently. Node A overwrites files in the first few levels of T_1 while B does the same in T_2. To allow this, the system creates two snapshots, one for each writer. During concurrent writing, new updates performed by a node are visible only to that node. In phase three, the two nodes decide to merge their private snapshots. After the merge, each writer is able to read the new data created by the other in the second phase.

In our experiments, nodes A and B update the volume in non-conflicting ways. So, the system is able to merge the two snapshots automatically. We define "merge latency" as the difference between the time when the two users express the desire to merge their versions, and the time when the system has successfully merged both versions so that the users can see each other's updates.

Results from this set of experiments are presented in the right portion of Table 7. The merge latency (column 10) in this case is observed to be independent of the amount of data. Note that the merge latency includes the time it takes for the system to detect if there are any conflicts between the two versions. Since in our experiments, updates of A and B are restricted to separate directories, it takes only a constant number of network messages to detect that there are no conflicts. Segank does not require any log or data propagation during automatic merging, unless there are conflicts in which case user intervention or application-specific conflict resolvers are needed.

Columns labeled "DL" and "DD" are times for reading T_1 and T_2 at A after the merge. Note that A does not have the complete log after the merge, so all reads result in network communication. These columns are similar to the "DL" and "DD" columns for the read-sharing case. The results for reading the trees at node B after merge are similar, and therefore, omitted.

9 Related Work

We have discussed some existing systems in Section 2.2. In addition to the issues already explored about Bayou [18, 22], another important difference is that Bayou is an application construction framework designed for application-specific merging and conflict resolution, while Segank is a storage/file system level solution. Segank allows many existing applications to run transparently, but it provides little support for merging and conflict resolution. An ongoing research topic is to investigate how the techniques that Segank employs to exploit a non-uniform network can be applied to a Bayou-like system to eliminate some of its limitations (such as its requiring full replicas).

Ivy [14] is a DHT-based peer-to-peer file system. Segank is similar to Ivy in querying logs of multiple users and using snapshots to support sharing. Segank's logs contain only object invalidation records, while Ivy's logs contain NFS operations and their associated data. Playing the invalidation logs and creating snapshots in Segank are light weight. Ivy effectively stores data twice, once in its NFS operation logs, and once again when the logs are played to create snapshots. Segank allows more flexible data placement, movement, and replication than Ivy's DHT-based approach. This level of flexibility is essential for the non-uniform network that it targets. As

		Read Sharing							Write Sharing		
Link	Expt.	Refresh	Read Time (s)				Log-Prop.	Data-Prop.	Merge	Read Time (s)	
Type	Type	(ms)	LL	DD	DL	LD	Time (s)	Time (s)	(ms)	DD	DL
	Small			17	17	18	0.002	2		18	17
LAN	Medium	0.4	16	16	17	16	0.004	3	0.8	15	16
	Large			21	16	21	0.014	8		19	23
	Small			27	25	19	0.014	6		29	30
WiFi	Medium	2.9	16	29	25	21	0.049	26	6.0	34	32
	Large			44	25	42	0.236	119		45	35
	Small			70	69	18	0.042	2		72	70
WAN	Medium	20	16	72	69	28	0.086	6	40	70	71
	Large			78	69	68	0.131	43		79	79
	Small			465	393	83	0.63	273		473	415
Modem	Medium	130	16	664	398	347	2.6	1356	355	643	394
	Large			1306	403	1212	12.3	6478		1312	402

Table 7: Timing results for the read-sharing and the write-sharing experiments.

a result of not maintaining precise location information, however, Segank may query a number of devices before reaching a device with the desired data. Segankast attempts to minimize the cost of extra queries. DHT-based querying may involve multiple network hops as well but it is more scalable than Segankast, although due to the way it searches multiple users' logs, Ivy is not meant to support a large number of users who share the same data.

Fluid Replication [10], an extension based on Coda, introduces an intermediate level between mobile clients and their stationary servers, called "WayStations," which are designed to provide a degree of data reliability while minimizing the communication across the wide-area used for maintaining replica consistency. We have explained that one useful feature provided by Segank compared to Coda is its treatment of all devices as equals. This treatment allows more flexible direct device-to-device interactions without relying on any server-based infrastructure. For example, two arbitrary devices on a speeding train should be allowed to interact spontaneously and directly. Fluid Replication is a middle point, in the sense that WayStations are equals to each other and this intermediate level allows clients to interact with each other without requiring them to connect to a single server. However, clients do not communicate with each other directly under Fluid Replication.

Distributed databases [7, 15], like Bayou, use update logs to keep replicas consistent. The purpose of the invalidation logs in Segank is not to replicate data. The invalidation log in Segank contains only invalidation records and the system does not need to propagate data to quickly bring other devices up-to-date.

JetFile uses multicasts to perform "best-effort" invalidation of obsolete data [8]. The mobile environment targeted by Segank is different from the wide-area stationary environment targeted by JetFile. Instead of using a "symmetric" solution for reads and writes, Segank uses an "asymmetric" solution: Segankast for reads and log-based lazy invalidation for writes. We believe that this delayed and batched propagation of invalidation records is more appropriate for a more dynamic and non-uniform network. In addition, Segank relies on no centralized components and has little distributed state.

Although we have called Segankast a "multicast-like" solution, it is actually quite different from overlay multicast systems [9, 4]. Typically, the goal of existing multicast systems is to deliver data to all machines in the target set. In contrast, the goal of a Segankast is to retrieve a single copy from several possible locations: the Segankast request need not always reach all possible locations, and one or more data replies may return. To accomplish this efficiently, we must carefully manage the ordering of requests, and determine what requests to send in parallel and what to send sequentially.

The PersonalRAID system [21] can be seen as a first step towards realizing many of Segank's goals, namely, decoupling data and meta-data propagation, and providing a single uniform name space across multiple devices. Segank, however, significantly differs from PersonalRAID in many respects. (1) PersonalRAID is designed to manage a *disconnected* set of devices, which forces it to maintain complete replicas at all devices (except the mobile device.) Segank, on the other hand, is able to use connectivity among devices to improve performance and availability. (2) PersonalRAID is primarily a single user system with no support for data sharing among multiple users. Segank includes a distributed snapshot mechanism to support sharing. (3) In PersonalRAID, when two devices are brought together, the system requires complete meta-data propagation to finish before the user is allowed to operate. Segank mandates no such propagation. (4) The use of a MOAD-like mobile device is critical in PersonalRAID, since it is the sole means of data and meta-data propagation. In Segank, however, the use of MOAD is largely a convenience and a performance optimization. (5) PersonalRAID provides data redundancy, while Segank does not. Segank falls back on backup/restore in case of a device loss.

10 Conclusion

We have constructed a mobile storage system that is designed to manage storage elements distributed over a non-uniform network. Like some systems for a wired network, it needs to allow flexible placement and consistent access of distributed data. Like systems designed for disconnected operation and/or weak connectivity, it needs to avoid over-using weak links. The Segank system must account for a possible simultaneous coexistence of a continuum of connectivity conditions in the mechanisms that the system uses to locate data, to keep data consistent, and to manage sharing. It allows a mobile storage user to make the most of what a non-uniform network has to offer without penalizing him with unnecessary foreground propagation costs.

Acknowledgments

We thank Charles Coglianese and Adam Wells for their contributions to the initial Segankast design and implementation. We also thank our shepherd Guillermo Alvarez and other FAST reviewers for their comments on this paper.

References

[1] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless Network File Systems. *ACM Transactions on Computer Systems 14*, 1 (Feb. 1996), 41–79.

[2] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-Area Cooperative Storage with CFS. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001), pp. 202–215.

[3] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[4] DEERING, S. E., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G., AND WEI, L. An Architecture for Wide-Area Multicast Routing. In *Proc. of SIGCOMM'94* (London, UK, August 1994), pp. 126–135.

[5] GARG, N., SHAO, Y., ZISKIND, E., SOBTI, S., ZHENG, F., LAI, J., KRISHNAMURTHY, A., AND WANG, R. Y. A Peer-to-Peer Mobile Storage System. Tech. Rep. TR-664-02, Computer Science Department, Princeton University, October 2002.

[6] Gnutella. http://gnutella.wego.com/.

[7] GORELIK, A., WANG, Y., AND DEPPE, M. Sybase Replication Server. In *Proc. ACM SIGMOD Conference* (May 1994), p. 468.

[8] GRONVALL, B., WESTERLUND, A., AND PINK, S. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation* (1999), pp. 251–264.

[9] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, J. W. Overcast: Reliable Multicasting with an Overlay Network. In *Proc. the Fourth Symposium on Operating Systems Design and Implementation* (October 2000).

[10] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, Visibility, and Performance in a Wide-Area File System. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[11] KISTLER, J., AND SATYANARAYANAN, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[12] LEE, E. K., AND THEKKATH, C. E. Petal: Distributed Virtual Disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 84–92.

[13] MUMMERT, L. B., EBLING, M. R., AND SATYANARAYANAN, M. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[14] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A Read/Write Peer-to-Peer File System. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (December 2002).

[15] ORACLE CORPORATION. *Oracle7 Server Distributed Systems: Replicated Data*, 1994.

[16] PEASE, D. A., MENON, J., REES, B., DUYANOVICH, L. M., AND HILLSBERG, B. L. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal 42*, 2 (2003), 250–67.

[17] PERKINS, C., BELDING-ROYER, E., AND DAS, S. Ad Hoc On Demand Distance Vector (AODV) Routing, 2001.

[18] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 288–301.

[19] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. First Workshop on Hot Topics in Networks (HotNets-I)* (October 2002).

[20] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Eighteenth Symposium on Operating Systems Principles* (October 2001).

[21] SOBTI, S., GARG, N., ZHANG, C., YU, X., KRISHNAMURTHY, A., AND WANG, R. Y. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[22] TERRY, D. B., THEIMER, M. M., PETERSON, K., DEMERS, A. J., SPREITZER, M. J., AND HAUASER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 172–183.

[23] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of the ACM Sixteenth Symposium on Operating Systems Principles* (Oct. 1997).

[24] WAGNER, J. Getting to Know Your 3G. http://www.internetnews.com/wireless/article/-0,,10692_964581,00.html, January 2002.