# A Peer-to-Peer Mobile Storage System

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

## Abstract

This paper presents the design and implementation of a peer-to-peer storage system that allows mobile users to transparently access and share data. The system employs a small networked portable storage device that is designed to compensate for weak wide-area connectivity by leveraging ad hoc peer-to-peer connectivity and an embedded storage element. Other key features of the system include the use of a location and topology-sensitive multicast-like solution for locating data, lazy peer-to-peer propagation of invalidation information for ensuring consistency across multiple devices, and a distributed snapshot mechanism for supporting sharing and distributed backup. A common theme of the design decisions is minimizing the amount of distributed state and global coordination, while still achieving the desired functionality and good performance. Initial experiences with a prototype implementation suggest that we have largely achieved our objectives.

## 1 Introduction

As the cost, form factor, and capacity of stable storage continues to improve dramatically, one consequence is the emergence of highly compact secondary storage technologies that can be seamlessly integrated into devices of all shapes and forms. Today, these devices are largely disjoint and users are expected to manually hoard, propagate, backup data on individual devices. As these devices rapidly proliferate in our surroundings, we are faced with an increasingly difficult challenge of managing this chaotic sea of "invisible bits".

In this paper, we argue that an effective way of managing this data is applying the peer-to-peer philosophy: instead of "powering" all these devices with an omnipresent external networked storage utility, these devices are by themselves able peer components of a mobile storage system and what is needed is a piece of system software that ties all these disjoint devices into a coherent whole. However, wide-area connectivity alone is not always sufficient for the purpose of coordinating all these devices. To compensate for this inadequacy, we introduce a small portable storage device equipped with several connectivity technologies. This device leverages ad hoc peer-to-peer connectivity and an embedded storage element to overcome the wide-area connectivity bottleneck in accomplishing its role as a coordinator of the other devices.

From a user's point of view, this portable device follows her wherever she goes (just like a BlackBerry email device today would). As long as the user has the device with her, she can (1) transparently access her own data regardless on which machine the bytes are physically stored and regardless where the user is, (2) transparently read other users' data whose access has been granted, and (3) use the portable device as a storage "adaptor" for other appliances so that their special purpose data can be readily integrated into the entire storage system. One important difference between this system and some existing application-specific solutions is that this system is designed to be a storage-level solution that can transparently support most existing file systems and applications.

Underneath the hood, the system must solve three hard problems: (1) How does the system locate data that can be stored on any devices? (2) 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 backup and sharing?

The system solves the first problem using a location and topology-sensitive multicast-like solution. The advantage of this solution is that it minimizes global state, allows for autonomous data movement decisions, and can effectively exploit locality. The system solves the second problem using lazy peer-to-peer propagation of invalidation information. As a result of decoupling this propagation from data propagation, it can quickly bring weakly connected devices up-to-date. The system solves the third problem using a distributed snapshot mechanism. The advantage of this solution is that it allows a user to continue to modify the storage system without interfering with either backup or sharing.

We have implemented a Linux-based prototype of the system in which the role of the portable storage device is played by a Compaq iPAQ equipped with storage and connectivity accessories. One goal of our design is to minimize the amount of distributed state and global coordination. This design goal has resulted in a relatively simple and robust implementation that, for example, has little complexity in its crash recovery mechanism. Our initial experience with the prototype leads us to conclude that a peer-to-peer model, when aided by a mobile device equipped with embedded storage and ad hoc connectivity, can effectively overcome wide-area connectivity weakness and enable mobile users to access and share data ubiquitously.

[*]Department of Computer Science, Princeton University, {nitin, yshao, eziskind, sobti, zheng, lai, rywang}@cs.princeton.edu.

[†]Department of Computer Science, Yale University, arvind@cs.yale.edu.

## 2 Two Naive Approaches

There are two broad categories of existing solutions to providing ubiquitously available mobile storage. One is relying on ubiquitous connectivity to a "storage utility" infrastructure; the second is relying exclusively on a small portable storage device. We briefly explore these simple alternatives and their limitations.

### 2.1 Connectivity Limitations

High-performance universal network connectivity remains an elusive goal. Even the typical so called "broadband" home DSL users typically only have access to an uplink capacity around 100 Kbps today. The much anticipated 3G wireless networks are designed to ultimately achieve 384 Kbps, but industry observers agree that wide availability of such speeds is many years away. Today, 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 [25]. At any instant, only a small number of devices may be strongly connected to each other; and a mobile storage user cannot always count on an omnipresent high-quality connectivity to a centralized storage service.

### 2.2 Limitations of Portable Storage Devices

Due to the difficulty of accessing data across a mobile wide-area network, we tend to resort to carrying bits with us. Today, when some of us travel, we are apprehensive about leaving our laptops behind, not necessarily because we fear that our travel destinations lack computers, or because our personal machines have any special capabilities. Instead, what makes a person's machine personal is the *data* stored on it. Carrying a laptop for this purpose, however, is cumbersome: among its many faults, one of the most serious is that the form factor of a generic computing device such as a laptop is unlikely to improve in terms of portability due to user interface considerations such as the size of a screen or a keyboard.

Recognizing this inconvenience, manufacturers have started to offer a wide array of mobile storage devices. The form factor of these devices can be as small as a key chain ornament [21]. Some hope that as storage density continues to increase, a day may come when all a user would have to carry is such a small device.

We share this vision: we believe that the notion of carrying computers will be as unnecessary as the notion of carrying television sets—television sets are commodity generic devices that are no different regardless which user "owns" them; and even though small portable sets are available, people have rarely found the desire to carry them, as sets with more comfortable form factors are widely available at public locations such as hotels and airports. Computers should be no different. They will be cheap, widely available, and generic. The only thing that a user needs to take advantage of these widely available generic computing devices is an easily portable storage device that houses their personal data, something that makes a user uniquely herself.

However, relying exclusively on such a device for all data storage needs has its pitfalls. 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 will continue to reside on these stationary devices. Second, mobile storage devices tend to have poorer performance compared to desktop versions due to energy consumption and form factor considerations. For example, an IBM Microdrive housed in a PCMCIA interface delivers an order of magnitude less bandwidth than a typical laptop disk. Third, mobile storage devices tend to be less reliable due to environmental and human factors such as exposure to shock, moisture, and theft. Last, but not least, mobile storage devices, by themselves, provide little support for transparent data sharing among collaborating users.

## 3 The Skunk-Based Peer-to-Peer Approach

Our approach has two key elements: (1) instead of relying on omnipresent access to a storage utility, the system is based on the coordination of users' existing peer devices and their embedded storage elements; (2) instead of relying exclusively on a portable storage device, the system employs such a device as a coordinator of the other peer elements; and exploiting this portable storage element and ad hoc peer-to-peer connectivity becomes a powerful means of overcoming the handicaps of wide-area connectivity. We shall refer to this portable device as the *Skunk device*, or simply the *Skunk*; and we shall refer to the entire peer-to-peer storage system as the *Skunk system*.

### 3.1 Peer-to-Peer Components

The storage managed by the Skunk system principally resides at the devices already owned by users. These may include one's computers at home, at work, or on the road; and various consumer electronic appliances already equipped with storage elements. In many cases, these devices are sufficiently powerful by themselves and they are the natural "homes" of some data; so a peer-to-peer system made of these existing storage elements makes good sense. One of the main roles of the Skunk system is to coordinate these peer devices to form a single coherent name space.

### 3.2 The Skunk Device

The task of coordinating these existing disjoint devices, however, may be difficult with existing connectivity capabilities. The role of the Skunk device is to facilitate this coordination across a potentially wide area. Some of the tasks performed by the Skunk include: (1) storing and propagating metadata that is used to drive other devices towards eventual consistency; (2) caching and propagating data to improve performance; (3) providing wireless peer-to-peer ad hoc short-range connectivity among peer devices; and (4) providing wireless wide-area connectivity among peer devices.

Physically, a Skunk consists of a processor, a storage element (such as the IBM Microdrive), an ad hoc connectivity interface (such as a Bluetooth or 802.11 card), and a WAN connectivity interface (such as a cellular modem).

We conjecture that an industrial strength version of the Skunk can be packaged in a form factor that is not much larger than a wrist watch; and just as a wrist watch, or the larger BlackBerry email device, the Skunk is a personal device and it accompanies the user at all times. As long as a user has a Skunk with her, she can access and share the storage system. We explore some of the Skunk use cases in greater detail next.

## 3.3 Personal Use Cases

A user works on her office computer, with her Skunk nearby, which communicates with the office computer with an ad hoc wireless interface. The user "sees" a single Skunk-backed file system whose data may be physically spread across any of her computers. As she creates new data on the office computer, some of the new data may be stored on the Skunk, and some of it may be pushed to some of her other computers in the background. When the user leaves the office at the end of the work day, she carries only the Skunk home, and she "sees" the same coherent Skunk file system on her home computer, which happens to have only a weak DSL uplink. As she reads data on her home computer, the Skunk system ensures that she always reads a fresh copy of the data, which may physically reside on her home computer, her Skunk device, her office computer, or any other devices that she may own. The next morning, the user carries the Skunk back to her office and the cycle repeats. The Skunk carries the metadata for ensuring consistency and caches much of the data to avoid over-stressing the weak wide-area link between home and office.

Once in a while, the user travels. Again, the user only brings the Skunk with her, fully expecting computers with comfortable form factor user interfaces to be ubiquitously available: at airports, on planes, and in hotels. Two factors make these locations different from her office and home: (1) computers at these locations may not be connected to the wide-area network; and (2) the user cannot leave her data behind on these computers if she expects to be able to retrieve them later. As the user reads data, some of the read requests are satisfied by data cached on the Skunk itself, while other requests may be satisfied by the user's home or office computers via the wide-area wireless interface built into the Skunk device. As the user writes data, it is principally stored on the Skunk. (Although pushing over the wide-area wireless interface is possible, the user may choose not to due to its poor performance and high cost.) If the Skunk fills up, the user may simply walk to the corner drug store and buy another six-pack of Skunk devices. When the user returns to her office or home from her travel, the data accumulated on her Skunk(s) will be gradually transfered to her office and/or home machines via the fast ad hoc interface.

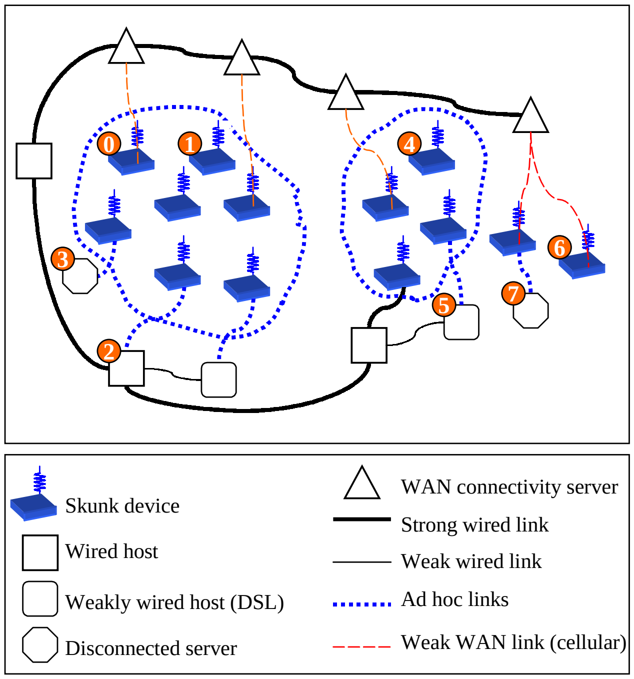

*Figure 1: Example route choices available to Skunkast.*

Note that as long as the user has the Skunk with her, she can access data stored on any of her machines. At any instant, the Skunk system routes the user request to the closest replica. We call this communication mechanism *Skunkast* (whose detail will be described in a later section). Also note that although we have so far assumed that the Skunk is a highly compact wrist watch-sized or iPAQ-sized device, nothing prevents the Skunk from being a fully functional conventional laptop. In fact, as we shall see later, in the peer-to-peer Skunk system, there is little difference among all the participating machines, and their roles are easily interchangeable so, for example, a laptop that usually resides in the office can be turned into a Skunk when the user travels.

## 3.4 Inter-Personal Use Cases

Two colleagues, Bob and Alice, bring their Skunks to a business trip. The night before their scheduled presentation, they need to collaborate on their slides as they work on the hotel computers. The Skunk system allows Alice to have read-only access to some of Bob's data, which, again, may be physically stored on any of Bob's devices. The system again uses Skunkast to route Alice's request to a closest device of Bob's that houses a desired replica. In the common case, the ad hoc wireless interfaces on their Skunks allow the two users to directly and quickly share data without resorting to a wide-area connection which may be either weak or nonexistent on the hotel computers. In the more general case, if the collaborators are separated by a large geographical distance, however, the route choices available to Skunkast may be more diverse and complex.

Consider the example shown in Figure 1. Let us assume that the Skunk with a label of (0) is the reader of a data item. The other devices with numbered labels represent devices that house a replica of the desired data item. The numbers roughly reflect the order of desirability as Skunkast chooses a replica. (1) a peer Skunk in the same ad hoc network; (2) a wired end host on the stationary "backbone"; (3) a disconnected end host; (4) a Skunk in a different ad hoc network that is reachable via the backbone; (5) a wired but weakly connected end host (such as a DSL host with limited uplink capacity); (6) a Skunk that is only reachable via the cellular link; (7) a disconnected end host reachable only via the cellular link of a Skunk.

### 3.5 Sharing Model

In the Skunk system, each device and the data stored on it have a single *owner*[1]. While the owner has full read/write access to her data in the "personal" use cases, in the "inter-personal" sharing use cases, the Skunk system only supports read-only access to data not owned by a user. In the example above, Alice can modify any of her own data stored on any of her devices, but Alice can only read from Bob's devices—overwriting Bob's data by Alice is not supported by the Skunk system, and Alice would have to make a copy of her own before she can modify it[2]. This design decision is driven by several considerations.

Our foremost consideration is simplicity. By legislating away the possibility of concurrent writes to the same data at different locations, we eliminate a large amount of the complexity associated with handling conflicts and maintaining data coherence. Indeed, the simplification is such that it has become possible to engineer the Skunk system entirely as a storage-level solution: the owner's data appears to a host computer as a read/write virtual disk, and others' data appears to a host computer as a read-only virtual disk. Existing file systems, operating systems, and most applications largely do not need to be altered and they transparently benefit from the Skunk system. This level of transparency is an important goal that we set out to achieve.

The second consideration has to do with the target environment within which we would like to see the Skunk system deployed, an environment that is not unlike the one targeted by the BlackBerry device and its service, or the one targeted by various peer-to-peer file sharing applications. In these contexts, there are clear and natural notions of ownership of devices and data.

The third consideration concerns the question of whether this simple model of sharing is sufficient for enabling meaningful collaboration. We believe the answer is yes. This model in fact corresponds closely to how some of us collaborate today: on our departmental file server, even though the authors' home directories reside in a single read/write file system, we rarely turn on write permission bits of the directories and files that we own and allow others to update them directly; instead, we typically only modify our local copies.

This is not to say that we believe that this is the only sharing model that one needs and there is no need for a shared read/write store. To the contrary, we believe that for more sophisticated collaborations, one needs both. For example, while maintaining separate local copies of Powerpoint slides in the collaborators' own home directories tends to be sufficient, collaborating on a large programming project typically requires a source control system (such as CVS) as the shared read/write store, in addition to the separate local copies. This shared read/write store, however, is best implemented above the file system at the application level (as is the case of CVS). The task of solving problems such as concurrent writes and conflict resolution does not belong in the file system layer. In this sense, we do not necessarily see a peer-to-peer storage-level solution (such as the Skunk system) as a replacement of or even a competitor to a peer-to-peer application-specific read/write store (such as the Bayou system [16, 23]). Each of these two classes of systems has its own reason of existence: the former allows many existing applications to run transparently in a distributed and mobile context and provides a simple model of sharing, while the latter provides richer sharing semantics at the expense of sacrificing generality. Neither type of system is suitable for all types of data, and a side-by-side coexistence delivers the best benefits of both worlds.

### 3.6 The Skunk as a Storage Adaptor

The data management needs addressed by the Skunk system are by no means limited to traditional desktop applications. As the data management functionalities are separated from cumbersome generic computing devices, and as these functionalities are cleanly encapsulated in modular small form factor devices (such as the Skunk), such devices can then readily interact with other consumer electronic devices in interesting and useful ways. For example, as a camera is coupled with a Skunk, the captured images can instantly be made sharable among family and friends; as an MP3 player is coupled with a Skunk, friends can swap music as ad hoc fan groups form; as a phone is coupled with a Skunk, one can store, retrieve, edit, embed, share voice messages just as one can with any other forms of data; and as an email device is coupled with a Skunk, one can integrate file sharing into emails smoothly. In these examples, the Skunk devices essentially serve as storage "adaptors" for other appliances so that their special-purpose data becomes accessible on a "backplane" connecting all peer-to-peer Skunk devices.

## 4 Design

The key questions that the Skunk system design must address are: (1) Upon a read request, how does the system find out which device has the desired data,

[1]It is possible to create multiple *logical* devices out of a single *physical* device to allow multiple Skunk users to share physical devices; but to simplify the discussion, we do not consider this possibility in this paper.

[2]The Skunk system can in fact be easily extended to accommodate concurrent but non-conflicting writes at different devices without user intervention by merging logical disks. A previous paper [20] on a primitive predecessor of the system describes this possible extension but we do not further consider this possibility in this paper.

and which device does the system choose to retrieve the data from? (2) Upon delete or overwrite operations, how does the system keep all the devices "consistent" so that they do not return stale data and the storage space occupied by obsolete data is freed? (3) How does the system support backup and sharing while the owner may continue to modify the system.

## 4.1 Drawbacks of Employing Location Maps

The types of connectivity technologies connecting the devices in a Skunk system may include fast LANs, ad hoc links such as Bluetooth or 802.11, high-latency wide-area Internet links, and even wireless modems. It is critical for the Skunk system to be able to flexibly place the data around the system and minimize the use of slow links. This requirement of flexible data placement precludes the use of simple distributed hash table-based schemes which dictate rigid placement algorithms of their own [22, 18]. And unlike peer-to-peer systems that attempt to maintain full replicas at all devices [16, 20, 23], a copy of the desired data may only be stored on a subset of the devices in the Skunk system.

One plausible solution to the data location problem is to maintain a mapping from block IDs to a list of devices where a replica of the block can be found. This mapping, however, scales with the total amount of data in the entire Skunk system; so it may be too large to store the entire map on a single device in general, or on the Skunk device in particular. Therefore, the map itself may need to be distributed across a number of devices and the placement decision of the map also needs to be carefully controlled to minimize the use of slow links. This is in contrast to existing cluster file systems where all the participating machines are located on a single fast LAN and the penalty of communicating with fixed location "managers" is not as severe [1, 11, 24]. As the system must be able to flexibly store pieces of the location map on any device, we now need a means of locating the location map itself; so a higher level of map of map is needed. This leads to a hierarchical map solution: the highest level map should be compact enough that it can fit on any single device and following the hierarchy in a top-down fashion allows the reader of any data block to uncover where the data is stored.

This hierarchical map solution, however, has some drawbacks. First, for each data read operation, the system may be forced to perform one or more map read operations, potentially across a network; and for each data write operation, the system may need to perform more map writes to update the location map. These extra I/Os can be costly. Second, data movements, such as caching data at, pushing data to, and evicting cached copies from nearby devices, must update the location map, and the cost of doing so may offset the benefit. Third, an implementation based on this design must exercise care to keep various pieces of distributed state, such as different levels of the map, consistent with each other; this approach may lead to considerable implementation complexity.

## 4.2 Locating Data Using Skunkast

Instead of employing location maps, the Skunk system uses a mechanism that is similar to multicast, which we call *Skunkast*. At each location, the system constructs a multicast tree rooted at the current reader device connecting all of a user's devices, which we call a Skunkast tree. We note the following properties of a Skunkast tree. First, the tree is on a per-user basis and it includes only the devices owned by a single user. We do not envision this number of devices to be massive at this stage so there is little scalability concern. Second, the Skunkast tree is an overlay tree. Third, the Skunkast tree is location sensitive so a new tree rooted at each new location is constructed. Fourth, the Skunkast tree is only for sending requests and the single data reply in response to a Skunkast request flows directly from the replier to the requester.

One feature that makes Skunkast different from traditional multicast is that a Skunkast request does not necessarily have to reach all the devices in the Skunkast tree: the Skunkast requests are divided into several *stages* and if an earlier stage succeeds in finding a replica, no further stage is executed. For example, the first stage request queries the single local device on which the read request originates. Only when this device fails to produce the requested data does the system initiate the second stage requests which would reach all the user's devices on a nearby ad hoc network. Only when the data is not found on any of these devices does the system cross a modem link once to initiate the third stage requests to all devices reachable only across this modem link.

The request messages within a single Skunkast stage may propagate in parallel but different stages occur sequentially. If the desired data is located in nearby devices, Skunkast introduces little extra overhead. If the desired data is found in farther away devices, such as those across a wide-area or a modem link, the extra time spent by Skunkast querying the nearby devices is likely to be relatively insignificant. To further improve data location speed, it is possible for the system to incorporate a small location hint table that is indexed by a hash of the block ID. Querying the hint table can proceed in parallel with regular Skunkast and incorrect hints have no negative impact on correctness or performance.

Compared to the location map-based approach discussed in the last section, Skunkast has several advantages. Skunkast relies on no distributed state so it has no complexity associated with maintaining the consistency of distributed state. The system may freely move, replicate, or purge data on any device without having to update location information stored elsewhere. Skunkast provides a built-in means of exploiting location proximity in both the data location phase and data read phase, as a nearby node on the Skunkast tree tends to receive the query first and supplies the data. In contrast, although a location map may pinpoint the exact locations of the data, it still leaves the question of which one to choose unanswered.

## 4.3 Invalidating Stale Data

When the user deletes or overwrites data, data stored on some devices becomes obsolete. Regardless the data location mechanism used, these devices need to be informed of the invalidation events so the storage space occupied by stale data can be freed up. Furthermore, under the Skunkast approach, a device should not respond to a Skunkast request if its copy of the data is stale, and the only way for the device to "know" that its copy is stale is for it to have received an invalidation message. The devices that should receive these invalidation messages, however, may be poorly connected to the user's current device (which is where writes occur). It is thus infeasible to require the invalidation messages to be sent to all the appropriate devices in the foreground. In the Skunk system, these invalidation messages are initially buffered at the writer device and propagated to other devices in the background. We now describe the details of this process and how it interacts with other operations of the Skunk system.

In the Skunk system, writes can only occur at the device that is physically with the user. Each write is tagged with a monotonically incrementing counter, which we shall refer to as a *timestamp*. Logically, a device in the Skunk system maintains three persistent data structures: (1) a *block store* that is keyed by block IDs and stores the block content and the timestamps of the writes; (2) an *invalidation log* that records, in time order, the block IDs and their timestamps of the writes; and (3) the *freshness* of a device, which is the timestamp of the tip of the tail of the contiguous invalidation log received by a device.

As a write occurs on the writer device $D_3$, data and its associated timestamp $t_3$ is added to the block store on $D_3$, the block ID and $t_3$ is appended to its invalidation log, and the freshness of $D_3$ becomes $t_3$. In the background, possibly at a much later time, $D_3$ attempts to send a portion of its invalidation log to a device $D_1$ that has a freshness value of $t_1 (t_1 < t_3)$. $D_3$ may choose to send only a convenient chunk of the log between $t_1$ and $t_2 (t_1 < t_2 < t_3)$. Upon successfully receiving this chunk, $D_1$ appends it to its own invalidation log, and advances its freshness to $t_2$. At possibly yet another later time, $D_1$ plays its invalidation log to its own block store to delete the stale data if the block ID contained in an invalidation record is indeed found in the block store and the time stamp of the corresponding block in the block store is older than the one contained in the invalidation log.

We assume that the Skunk device houses the most complete invalidation log as the Skunk follows the user, who is always the sole writer. The head of the log can be truncated once it has been sent to all this user's other devices[3]. Theoretically, log fragments stored on other devices can be discarded as soon as they are played to the local block store; but next, we will discuss another potential use of these log fragments.

[3]Parts of the log can also be stored on other well-connected devices if the capacity on the Skunk becomes a premium. Since the Skunk is always with the user and the Skunk has at least the wireless modem link, the entire invalidation log should always be reachable. To simplify the rest of the discussion, however, we shall not consider this possibility.

## 4.4 Peer-to-Peer Propagation of Invalidation Records

So far, we have assumed that the invalidation records are only sent from the Skunk device to another devices to bring them up to date. This restriction is not necessary: any fresher device can send an invalidation log fragment to any less fresh device to bring it more up-to-date. Theoretically, it is not even necessary for a device to receive invalidation records in the strict order of increasing timestamps—as long as a device conservatively announces its freshness $t$ so that $t < t'$ for all missing invalidation record timestamps $t'$, correctness can be ensured. For simplicity, however, we dictate that each device in the Skunk system receives invalidation records in strict time order. Because the single-writer restriction that we have imposed on the Skunk system, the invalidation record propagations are always one-way from a fresher device to a less fresh one. This feature leads to a few key differences between the Skunk system and other peer-to-peer systems based on the epidemic exchange model such as Bayou [16, 23].

First, the Skunk system is a storage-level solution. By imposing the single-writer restriction and replacing two-way exchanges with one-way invalidation propagations, the system legislates away the problem of conflict resolution, which intrinsically does not belong to the storage level. The benefit that one gains is a much more general system that supports a vast majority of the existing applications.

Second, to ensure correctness, only the invalidation records need to be propagated in the Skunk system and no data exchange is necessary to ensure a consistent view of the participating devices. The size of invalidation records should be at least three orders of magnitude smaller than that of data. This allows devices to be quickly brought up-to-date. This is in contrast to replicated databases where data and metadata propagations are intertwined.

This peer-to-peer model makes all the devices in a Skunk system very much similar to each other. One difference between the Skunk device and the other devices is that the Skunk device is guaranteed to have the most complete invalidation log. This property ensures that as long as the Skunk is with the user, the user sees a consistent system wherever she goes. This property also makes it easy for a user to turn other devices into the Skunk device: as long as a device, such as a laptop, has the proper hardware capabilities, the user can simply transfer the complete invalidation log onto it and take it with her in place of her usual wrist watch-sized Skunk device.

## 4.5 Querying Invalidation Logs for Reads

The discussion so far may have implied that it is necessary to bring a device up-to-date by first playing invalidation records to it before the system allows the device to participate in the Skunkast protocol to satisfy reads. This is in fact not necessary. We now describe how the invalidation log interacts with Skunkast.

Suppose a user is performing read and write operations on a particular device. This device contains the most complete invalidation log as writes are being recorded, so it is effectively the Skunk device of the moment. Suppose invalidations on this device are immediately performed to the block store on this device. So in every sense of the word, this device is up-to-date.

When this device receives a read request, it first queries the local block store. If the desired data is found there, the read request is satisfied and no further action is necessary. Otherwise, the system queries the invalidation log on this device. If an entry for this block is found in the invalidation log, it implies that this block has been recently overwritten but the data is not found on this device. The system then launches a Skunkast to query the rest of the devices, specifically asking for a block with the timestamp found in the invalidation log. When the other devices receive this Skunkast query, regardless of their own freshness value, they query their own block store for the block ID with the specified timestamp. When the desired data is found in any block store, the read request is satisfied.

If no entry for this block is found in the invalidation log of the original read device, it implies that the block has not been overwritten during the entire time period reflected in the invalidation log on this device. Suppose the head of the invalidation log has a timestamp of $t_0$. The system then launches a Skunkast to query the rest of the devices, asking only the devices whose freshness is larger than $t_0$ to respond. When such a device receives the Skunkast request, it may also need to query its own invalidation log to be sure that the data contained in its own block store is fresh up to $t_0$.

This read algorithm requires at least a portion of the invalidation log to be queryable. Again, suppose the oldest queryable log record has a timestamp of $t_0$, an invariant of the system is that all the devices reachable by the Skunkast protocol at this moment must be at least as fresh as $t_0$. This, however, does not imply that the complete invalidation log at the Skunk must be queryable: older portions of the invalidation log that are kept for currently disconnected devices, for example, need not be queryable because there is no danger of reading obsolete data from a disconnected device. When such a device later becomes reachable and can participate in Skunkast again, the system must restore the invariant by either making more of the older portion of the log queryable, or perhaps more sensibly, by playing this older portion of the log to the newly connected device to upgrade its freshness up to $t_0$. This invariant makes it possible to cache the queryable tail of the invalidation log entirely in memory, minimizing overhead paid on reads.

## 4.6 Peer-to-Peer Data Propagation

As explained earlier, a distinct advantage of the Skunk system compared to some existing epidemic exchange-based systems is that the propagation of the invalidation records and that of data can be decoupled: only the former is required for correctness while the latter is purely a performance optimization. When and what data to propagate is largely a policy decision.

There is, however, still an ordering constraint that one must follow for data propagation: data is only propagated from fresher devices to less fresh devices. This constraint ensures that if the propagated data is overwritten after its creation timestamp, the corresponding invalidation record is guaranteed to not have been played to the data receiver by the time of the data propagation event yet, so a future receipt of the invalidation record by the data receiver would properly invalidate the data. The timestamp of the data itself is copied as is into the receiver's block store. None of the other data structures on either machine is affected. Interestingly, there is no constraint on the relationship between the timestamp of the propagated data and the freshness of the receiver device: the former can be less than, equal to, or greater than the latter.

An opposite case of data propagation is data discard. Discard operations by themselves do not affect any data structures. One goal of the Skunk system design is to allow individual devices or subsets of devices to autonomously make data movement decision without relying on global state or global coordination. The system, however, needs to exercise care not to discard a last lone copy of the data. We adopt a simple solution: when data is initially created, a so called *golden copy* is established; and a device is not allowed to discard a golden copy without propagating a replacement golden copy to another device.

## 4.7 Snapshots

A *snapshot* represents a consistent state of an owner's Skunk storage system "frozen" at one point in time. Creating a snapshot is logically making a copy of the owner's entire Skunk system so that subsequent modifications to the storage system is reflected only in the new copy. The Skunk system uses snapshots to cope with device losses and to handle read sharing with other users. A snapshot is named by the timestamp at the time when the snapshot is created.

Physically, creating a snapshot is more like copy-on-write. When the owner decides to create a snapshot, all that is required is the appending of a snapshot creation record to the invalidation log. Since the system requires the invalidation log to be propagated to all devices in timestamp order, the local block store on each device would not inadvertently overwrite blocks of an older snapshot before it "sees" a snapshot creation record. (The local block store must, of course, support snapshot operations.) How snapshots are deleted depends on the purpose of the snapshots and is described below.

## 4.8 Backup and Restore

If a device that houses no golden copy is lost, no recovery action is necessary. If a device that stores some golden copies is lost, we must roll back the Skunk system to an earlier snapshot.

The Skunk device, which follows its owner as she travels, is likely to be the most vulnerable. Whenever possible, the system should attempt to migrate golden copies off the Skunk. At a point when the Skunk device houses no golden copy, the system can take a *Skunk-less* *snapshot*. In the event a Skunk device is lost with some golden copies stored on it, the system rolls back to the last Skunk-less snapshot and discards all data added to the system after that.

To recover from the loss of other devices that participate in an owner's Skunk system, we use a backup-restore scheme. Periodically, the owner instructs her Skunk system to take a *backup snapshot*. A backup snapshot is created so that it can be copied to backup devices such as tapes. Backup devices can be introduced into the Skunk 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 carried from site to site to backup all devices onto a single tape; an intermediate number of backup devices are of course possible as well; and golden copies of a backup snapshot can also be transferred from one site to another to be backed up via either the Skunk or a network. 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.

There are several possible optimizations. Incremental backup is natural and easy to support because the block store from which the backups are made uses copy-on-write already and an extension to the block store interface that identifies the incremental changes should not be hard to add. Another possible optimization is to leave data belonging to an old backup snapshot undeleted on the regular devices. During a restore, this data can be quickly resurrected without requiring one to copy from backup devices. So far, we have assumed that upon a restore, we discard all surviving blocks newer than the timestamp of a backup snapshot. Attempting to salvage some of this data by "rolling-forward" after a restore is a possible future research topic.

## 4.9 Read Sharing

The discussion on data reads in Section 4.5 assumes that the data reader is the owner of a Skunk system. This assumption simplifies the discussion as the reader coincides with the only possible writer. In general, however, a different user may attempt to read the data, and this reader may initially contact a device that is different from the one on which the owner is performing writes. We call this reader a *foreign reader*. Because the Skunk system is being developed as a storage system, the requests serviced by the Skunk system are at block-level. If we do not exercise sufficient care, the block requests belonging to multiple file system operations, some of which may be writes, may interleave, leading to undesirable behavior (including possibly even crashes).

The Skunk system uses snapshots to solve this problem. When a foreign reader A wishes to read from a different user B, A's requesting device starts a foreign read *session* and sends a message to the sole device that B can write to at the moment (which is quite possibly B's Skunk device itself). We will briefly discuss the device location mechanism later. In response, user B creates a consistent *read sharing snapshot*, starts a counter of the number of foreign readers of this snapshot, and sends the snapshot name back to the foreign reader. The foreign reader A builds a new (and different) Skunkast tree of B's devices. Upon receiving the snapshot name from B, A sends Skunkast queries to B's devices to read data only from this consistent snapshot. Note that within this single session, A can perform all manners of file system read operations such as listing a directory, changing working directories, in addition to, of course, reading one or more files. In the mean time, B can continue to modify her Skunk system without interfering with foreign readers. When A is finished reading, it ends the foreign read session and sends a foreign read session termination message to inform B and flushes its cache of all B's data. Upon receiving this message from A, if the counter on this snapshot becomes zero, B can delete this snapshot.

The protocol described above assumes that a foreign reader must always first reach the current device that the owner may write to to acquire a snapshot name. It is possible to loosen this restriction so that it is up to a foreign reader to query any of the other devices and choose a snapshot to read from. Unlike the rest of the Skunk system, which works at the storage level, the protocol that handles foreign reads requires a small modification at the file system level to start and end foreign read sessions.

## 4.10 Crash Recovery

One design goal of the Skunk system is that the system has virtually no distributed state. This goal aids crash recovery so that recovering individual crashed devices is sufficient. When the owner is writing to a device, some of the data and the tail of the invalidation log is buffered in memory and may be lost upon a crash. Upon rebooting the device, the system reads the tail of the invalidation log from the disk and finds the last timestamp; instructs the block store to return the block IDs and their timestamps of all the writes made stable in the block store after this timestamp; adds them to the invalidation log; and records the timestamp of the very tail of the invalidation log as the freshness of the device. (The block store itself employs a log-structured design so that finding the updates after a particular timestamp is easy, although other alternative implementations of the block store are possible as well.) The device provides only crash-consistent semantics so the file system that runs on top of the device may need to run its own recovery code (such as `fsck`).

During any peer-to-peer propagation of either invalidation records or data, we ensure that each communication event is atomic in that the entire communicated content must safely land on disk before we declare the communication complete. Playing invalidation records to the block store and adding data into it are idempotent operations so repeating these activities after a reboot is acceptable.

### 4.11 Other Issues

**Adding or removing devices.** The Skunk device stores the most complete invalidation log and truncates the head of the log only when all of a user's devices' freshness values have progressed beyond the timestamp of the log head. Peer-to-peer exchanges of freshness values allow the Skunk device to discover the freshness values of all devices. When a new device is added to the system, this event needs to be registered with the Skunk device so that it does not discard invalidation log entries until all devices have received them. When a device is to be removed from the system, the system must "walk" the block store of this device to identify all golden copies and migrate them off this device. The device removal event also needs to be registered with the Skunk device so it does not wait for invalidation log entries to be propagated to the removed device.

**Locating devices of a user.** When user A desires to read data of user B, A needs to find the name of at least one of B's devices. We expect a simple hierarchical scheme to be sufficient. First, each of a single user's devices "knows" all other devices of this user. In particular, each device "knows" the name of the current Skunk device. If a user B wants her data to be read by a foreign reader, she joins a service-wide registry by submitting a user name and a small number of the names of her devices. The foreign reader A contacts the registry to locate at least the name of one device of B's. From this device, A can uncover the names of all of B's devices, which A uses to construct a Skunkast tree for reads. This is similar to how DNS works; and the user name-based name space is similar to how email works.

**Access control.** When a user sends a request to a device, she must present proof of her identity to the device. Each device keeps an access control list and checks it for each operation. Known authentication and encryption techniques can be used.

## 5 Implementation

While Section 4 describes some of the more fundamental design issues, in this section, we describe some implementation choices and details that are somewhat less fundamental. Alternative choices could very well have been made without changing the basic philosophy of the Skunk system. For example, while the current implementation works at the storage level, a file system or user library could have worked too.

### 5.1 Volumes

We have developed the system on Linux. The Skunk system appears to the rest of the operating system as a regular disk: the owner initially makes an "ext2" file system on the Skunk system and mounts it on a computer. When the user travels, she unmounts the file system, takes the Skunk device with her, and mounts it on some other computer. A foreign reader mounts a separate read-only file system for each user whom she desires to read from. We call each of these file systems a *volume*.

We use a pseudo block device driver that exports the interface of a disk and redirects the requests sent to it to a user-level process via upcalls. All the rest of the Skunk system components are implemented in this user-level process. The skunk system provides 64-bit block IDs and each block is 4 KB. The timestamps are also 64-bit. The Linux 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 the Skunk system.

Although there are many possible policies that one may employ for data placement and migration, we have so far implemented only a few. 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 the Skunk system attempts to propagate the data in the volume to as many devices as possible given enough resources. If the volume is defined shared, the owner hints to the system that others may need the data, and the Skunk system attempts to propagate the data to a well-connected device and to cache a copy on the Skunk device if possible. If the volume is defined stationary, the user hints to the system that the data is most likely to be needed exclusively on this creator device, although accessing it from "elsewhere" is still possible. Clearly, other policies are possible; and these "attributes" do not have to be attached to an entire "volume" and can be defined for individual files instead (which would require a file system-level implementation).

### 5.2 The Block Store and the Invalidation Log

The block store is a log-structured logical disk, similar to the ones used in some earlier systems [4, 20]. One difference is that because the Skunk system uses 64-bit block IDs, it cannot use a simple in-memory table for mapping logical addresses to physical addresses, as these earlier systems did; instead, the block store employs an in-memory B-tree for the mapping. The block store accesses the physical disk using raw I/O via the Linux `/dev/raw/raw*` interface, bypassing the buffer cache. Due to time pressure, we have not been able to finish the snapshot feature of the block store. A more primitive temporary substitute currently remaps block IDs of new writes into a specially designated subspace which uses IDs not visible to the file system. These blocks are later made visible when the snapshot is deleted. Some additional small modifications are made to the in-kernel file system to cause a foreign reader file system to flush its client cache at the start of a foreign read session and to cause all in-memory dirty data structures of the block store to be flushed to disk upon an `fsync`.

The invalidation log is itself stored in the block store in a designated subspace of the 64-bit space. Each log record is a tuple consisting of the block ID, a timestamp, and a 32-bit attributes field. The attributes field is largely unused currently. Each log block contains a checksum of the block, the number of records in

| | Dell Inspiron Laptop | Compaq iPAQ 3870 |
|---|---|---|
| Processor | Intel Pentium III, 650 MHz | Intel StrongARM, 206 MHz |
| OS | Linux 2.4.7-10 | Linux 2.4.18-rmk3 |
| Memory | 256M | 64M |
| Disk model | IBM Travelstar 20GN | IBM Microdrive |
| Capacity | 10GB | 1GB |
| RPM | 4200 | 3600 |
| Average latency | 7.1 ms (measured) | 20.3ms (measured) |
| Bandwidth | 16.9 MB/s (measured) | 1.5 MB/s (measured) |
| Wireless Card | Cisco Aironet 350, 11Mbps | Cisco Aironet 350, 11Mbps |
| Modem Card | Viking PC Card Modem, 56Kbps | Viking PC Card Modem, 56Kbps |

*Table 1: Platform characteristics.*

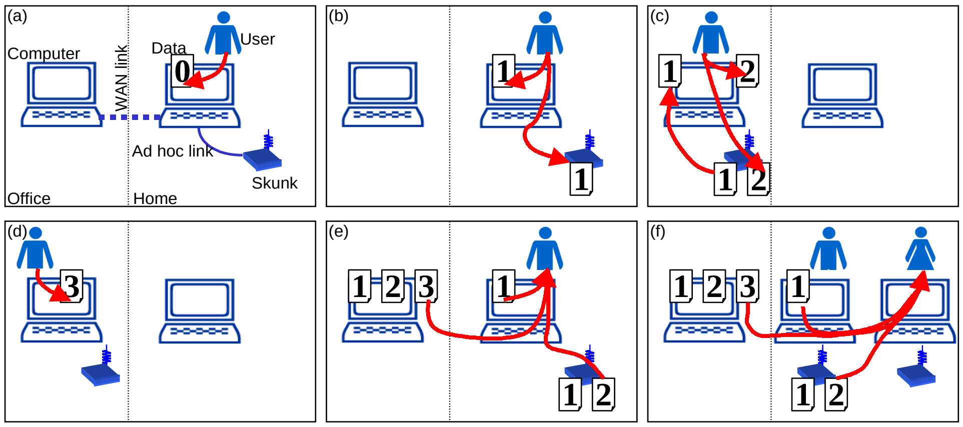

*Figure 2: The experiments. (a) The setup. (b) Phase 1. (c) Phase 2. (d) Phase 3. (e) Phase 4. (f) Phase 5.*

the block and then the records themselves in increasing timestamp order. As writes occur, new log records are buffered in memory. This allows us to eliminate records for recent overwrites. Once a log block is written to the block store it is not further compressed or altered. Since we have not done any optimizations in the encoding of the log records, a 4 KB log block stores only up to 204 records in the current implementation. A better encoding scheme could have significantly improved the already low log propagation cost presented in the later results section.

## 5.3 The iPAQ-Based Skunk Prototype

We use a Compaq iPAQ-based Skunk device in our prototype. Ideally, we would like to equip an iPAQ with at least three accessory devices: an IBM 1GB Microdrive, an 802.11 wireless card operating in ad hoc mode, and a cellular modem. With a double-slot PCMCIA sleeve, unfortunately, we are only able to equip the iPAQ with two out of three accessory devices at any one time. There are few occasions, however, when a Skunk user must have all three devices operating in the iPAQ at once, so occasional unplugging and plugging of PC cards in the iPAQ sleeve usually suffices. The iPAQ also runs Linux. In fact, the Skunk system programs that run on all devices share the same source.

## 5.4 Communication

The communication in the system occurs at three layers of abstraction. The task of the lowest connectivity layer is to route to a device regardless where it is and what physical connectivity interface it uses. The middle layer is Skunkast. The top layer is the storage system.

The Skunk device follows its owner and can be highly mobile. From a foreign reader A's point of view, a different user B's Skunk can be reached via: (1) an ad hoc wireless network encompassing both users' Skunks, (2) the Internet which connects to a remote ad hoc network within which B's Skunk is currently located, or (3) B's Skunk's wireless modem interface. In our implementation, the first two cases share a fixed IP address that is always valid (as long as the corresponding physical connectivity exists) and case (3) has its own IP address.

To reach a Skunk in the first two cases, we need a combination of an ad hoc routing component 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. 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 Skunkasts. We have used an existing Mobile IP implementation: Dynamics HUT Mobile IP version 0.8.1.

The above mechanisms handle routing and provide reachability between any two devices in the Skunk system. The Skunkast layer is built on top. It consists of two components: construction of a Skunkast tree and using the tree to perform the actual Skunkasts. The prototype implementation builds the Skunkast tree by measuring the latency from the current host to each of the other reachable devices owned by a user and uses a simple heuristic to order the devices into "groups"—

| Phase | Operation | Time (s) |
|---|---|---|
| 1 | Make Ext2 FS | 2.1 |
| | Create 7800 | 15.4 |
| | Create f1 (10 MB) | 1.4 |
| 2 | Create f2 (10 MB) | 1.3 |
| | Overwrite 2500 | 34.4 |
| 3 | Create f3 (1 MB) | 0.6 |
| | Overwrite 250 | 8.9 |
| 4 | Read f1 (10 MB) | 1.1 |
| | Read f2 (10 MB) | 32.6 |
| | Read f3 (1 MB) | 138.5 |
| | Read 3900 | 250.4 |
| 5 | Read f1 (10 MB) | 53.0 |
| | Read f2 (10 MB) | 77.8 |
| | Read f3 (1 MB) | 222.8 |
| | Read 3900 | 594.1 |

*Table 2: Times experienced by the user as he performs I/Os on a laptop while with an iPAQ Skunk.*

| Phase | Direction | Size (KB) | Time (s) |
|---|---|---|---|
| 1 | H → S | 456 | 2.8 |
| 2 | S → O | 456 | 1.6 |
| | O → S | 140 | 1.4 |
| 3 | O → S | 20 | 0.2 |
| 4 | S → H | 160 | 1.1 |

*Table 3: Statistics of log propagation: the size of the log and the propagation times. "H", "S", and "O" represent the home computer, the Skunk, and the office computer respectively.*

devices within a group are queried in parallel but different groups are queried sequentially during a Skunkast. This process is performed whenever the current device changes location and can be periodically rerun. We are considering ways of exploiting routing information other than latency for building Skunkast trees.

For point-to-point communication channels during actual Skunkasts, we have experimented with the socket interface (on both UDP and TCP) and the SUN-style RPC interface (running on top of both UDP and TCP transport). The socket version is more asynchronous in overlapping messages but, unfortunately, we observed some pathological performance on the ad hoc wireless network that is not seen on wired ethernet. The results that we report are based on the UDP/RPC version.

For the purpose of obtaining the list of devices belonging to any particular user, the simple DNS-like mechanism described in Section 4.11 has only a simple configuration file-based substitute at the moment.

### 5.5 Status

All aspects of the design described in Section 4 are implemented with the exception of full support for snapshots, backup and restore, and access control. The C prototype implementation consists of about 18K lines. The design of the Skunk system has sought to minimize the amount of distributed state and global coordination. The result has been a system that is relatively simple to understand and implement.

| Phase | Direction | Size (MB) | Time (s) |
|---|---|---|---|
| 1 | H → S | 90.2 | 239.3 |
| 2 | S → O | 90.2 | 187.5 |
| | O → S | 27.8 | 88.4 |

*Table 4: Statistics of data propagation: the amount of data and the propagation times.*

## 6 Experimental Results

In this section, we present some experimental results to show that: (1) the use of portable storage and ad hoc connectivity in the Skunk system compensates for a weak wide-area connectivity; (2) the propagation of invalidation log introduces little overhead; and (3) Skunkast can effectively locate close replicas without adding significant overhead.

Table 1 gives the platform characteristics. The role of the Skunk device is played by a Compaq iPAQ. The rest of the devices are played by Dell laptops. Figure 2(a) shows the setup. Two sites, home (H) and office (O), are connected by a weak WAN (56 Kbps modem). Each device is equipped with an ad hoc network interface. The Skunk device (S) follows the user. The experiment consists of five phases, each illustrated by a pane in the rest of Figure 2. The first three phases create data at different sites and the data is propagated in different ways. The last two phases read data that is distributed throughout the system.

Figure (b) illustrates the first phase of the experiment. User Bob is at home. He creates an Ext2 file system on an initially empty Skunk system. Then he creates a directory tree, which is four levels deep. Each directory has five subdirectories and ten files in it. There are a total of 7800 files, each of which is 8 KB in size. Then he creates a 10 MB file called "f1". The rows labeled as "Phase 1" in Table 2 give the latency experienced by Bob as he performs these activities. All this new data initially resides on the home laptop disk; so the latencies are determined by the efficiency of our per-device block store.

The invalidation log created by these activities, totaling 456 KB, as shown by the first row of Table 3, is propagated to the Skunk device over an ad hoc wireless network. As more idle time is available, the new data is propagated to the Skunk as well over the same ad hoc network. The first row of Table 4 gives the statistics of data propagation. Although the size of the invalidation log can be drastically reduced with more careful encoding, we see that it is already quite small compared to the amount of data. The bandwidth that we achieve on the ad hoc network can be improved further if we overlap messages more aggressively. At the end of this phase, the Skunk not only contains the complete invalidation log, it also contains all the new data.

Figure (c) illustrates the second phase. At the beginning of this phase, Bob arrives at his office with his Skunk device, and since idle time is available, both the invalidation log and the data carried in the Skunk are transfered to the office computer over the ad hoc network. The costs of these transfers are shown in the "S

$\rightarrow$ O" rows of Tables 3 and 4. Unlike most existing solutions, the propagation of the invalidation log and the data are decoupled in the Skunk system; so it is not mandatory to propagate the data immediately. This allows the user to use the storage system immediately without having to wait for lengthy data propagation. As the top two rows of Table 4 show, the "S $\rightarrow$ O" data propagation is faster than the "H $\rightarrow$ S" data propagation. This is due to the fact that the read performance of the block store on the Microdrive in the Skunk device is significantly better than its write performance.

Next in this phase, Bob performs more I/O: he creates another 10 MB file "f2" and overwrites 2500 randomly selected existing files. The latencies are reported in the "Phase 2" rows of Table 2. Overwriting existing files is slower than creating the directory structure due to extra disk reads incurred by lookup operations, whose latency cannot be overlapped. Both the invalidation log and the newly created data is propagated to the Skunk at the end of this phase. The costs of these propagations are given by the "Phase 2 / O $\rightarrow$ S" rows in Table 3 and 4.

Figure (d) illustrates the third phase. Bob creates more new data as he creates a 1 MB file "f3" and overwrites 250 randomly selected files. The performance of these activities are reported in the "Phase 3" rows of Table 2. Since Bob must leave office in a hurry, the newly created data resides only on the office computer. The system still, however, needs to propagate the invalidation log to the Skunk before he takes it home, and the cost of doing so is reported by the "Phase 3/ O $\rightarrow$ S" row in Table 3.

Figure (e) illustrate the fourth phase. Bob returns home with his Skunk. The home computer is quickly brought up-to-date as the invalidation log is propagated to it. This time is reported in the "Phase 4" row of Table 3. Since he needs to perform more I/Os, however, the data propagation does not commence. As he reads various files in this phase, some of which have been overwritten in the office, the Skunk system always directs his requests to copies that are both fresh and close. Bob first reads the three large files: f1, f2, and f3. The system performs Skunkasts and retrieves the three files from the local disk of the home computer, the Skunk over the ad hoc network, and the office computer over a WAN respectively. Bob then randomly chooses 50% of his small files and reads them. Again, since the data is smeared across the three devices, for each data request, Skunkast probes the three devices in the order: H, S, O. The performance of these operations is shown in the "Phase 4" rows of Table 2. An analysis of these numbers shows that querying the nearer devices by Skunkast adds little to the cost of retrieving data over the modem, and when reading from nearer devices, Skunkast is able to obtain all the bandwidth that UDP/RPC over the wireless ad hoc network or the local block store is able to deliver.

Finally, Figure (f) illustrates the fifth phase, which is an instance of read sharing of data in the Skunk system. Bob's colleague Alice shows up at Bob's home and wishes to read some of Bob's data. Alice's laptop is able to access data at H and S using the ad hoc interface, and the data at O via the modem link. In this benchmark, Alice reads the same data as read in phase four above. The data is smeared across H, O and S as at the end of phase 3. To satisfy reads generated by Alice's laptop, Skunkast probes Bob's device in the order: H, S, O. The read latencies perceived by Alice on her laptop (see "Phase 5" rows of Table 2) are worse than the corresponding latencies seen by Bob in Phase 4. The reason for this is that Skunkast incurs an extra hop in satisfying Alice's requests. [4]

| | 1<br>Coda | 2<br>Bayou | 3<br>F.R. | 4<br>PR0 | 5<br>Skunk |
|---|---|---|---|---|---|
| 1. Transparency | √ | × | √ | √ | √ |
| 2. P2P | × | √ | — | — | √ |
| 3. Infra.-less ops. | × | √ | × | √ | √ |
| 4. Partial repli. | — | × | — | × | √ |
| 5. Read sharing | √ | √ | √ | × | √ |
| 6. Write sharing | √ | √ | √ | × | × |
| 7. Conflict resol. | — | √ | — | × | × |
| 8. Metadata prop. | × | × | √ | √ | √ |
| 9. Redundancy | × | — | × | √ | × |

*Table 5: Comparison of five mobile storage systems: "√" indicates good (although not necessarily perfect) support of this feature; "×" indicates little or no support; and "—" indicates some limited support.*

## 7 Related Work

### 7.1 Mobile Storage Systems

Table 5 is a feature matrix of five mobile storage systems: (1) Coda [9, 13] is a client/server file system that allows disconnected or weakly connected mobile clients to operate out of their local "hoard". (2) Bayou [16, 23] is a peer-to-peer application construction framework that allows its users to craft application-specific mergers and conflict resolvers for dealing with concurrent writes to the same objects. (3) Fluid Replication (labeled "F.R." in the table) [8], 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. (4) PersonalRAID0 (labeled "PR0" in the table) [20] allows a mobile user to use a small portable device to transport modifications from one host to another to keep their images consistent without penalizing the user with these propagation delays. (5) The Skunk system. (The PR0 and Skunk systems are part of the umbrella PersonalRAID project and the former can be viewed as a primitive predecessor of the latter.) We examine nine features, each of which corresponds to a row in the table and a paragraph below.

**Transparency.** Although it is not difficult to construct cases that will expose non-transparent behavior

[4]In phase four, it makes sense to cache the data read by Bob at H, but we do not do so in our experiment. This is done to illustrate the fact that Skunkast incurs no significant overhead even in the case of read sharing when the data is distributed across multiple devices accessible via different levels of connectivity.

on each of these systems, all systems with the exception of Bayou are low-level solutions that support most existing application transparently. Bayou is the least transparent: new applications must be developed from scratch to take advantage of the Bayou framework.

**Peer-to-peer support.** We define peer-to-peer support to mean that all devices in the system are equals and any two devices can communicate directly with each other to create the illusion of a single system. (In one of the simplest cases, for example, two users who meet on the road should be able to spontaneously collaborate.) Coda lacks this support in that the system by design differentiates "clients" from "servers" and 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. While one intent of this differentiation is to allow better managed servers to provide a higher level of reliability guarantee, requiring a large number of wide-spread devices to communicate only with a server becomes too strict a constraint when peer-to-peer interactions could have done well. Fluid Replication allows some limited peer-to-peer support in that the WayStations are basically equals to each other but mobile clients still do not communicate with each other directly. PR0 allows some limited support as well in that all of a user's computers are peers but the only way for them to communicate with each other is via a portable storage device. Bayou and Skunk provide true peer-to-peer support.

**Infrastructure-less operations.** "Infrastructure support" refers to well-managed servers that are crucial for enabling user-to-user and device-to-device interactions and are not likely to be found at locations such as moving trains. We classify Coda servers and WayStations as infrastructures, without which two users cannot meaningfully collaborate. Bayou, PR0, and the Skunk system rely on no special infrastructure support.

**Partial replicas.** Bayou and PR0 attempt to maintain complete replicas at all devices. This simplifies some aspects of the system but is not always natural for all types of data and all types of devices, some of which, for example, may have capacity limitations. Coda and Fluid Replication allow a mobile client to contain only a subset of the data while the servers and WayStations should contain full replicas. In the Skunk system, any piece of data can reside on any device and no device is required to contain a complete replica. This provides a large degree of placement flexibility.

**Read sharing.** PR0 is designed to maintain the consistency of a *single* person's multiple computers with minimum resource requirements and minimum inconvenience to the user. It is the only system among the five that does not support any inter-personal sharing.

**Write sharing.** "Write sharing" refers to the system's ability of handling writes to the same data by different users. Coda, Bayou, and Fluid Replication all support this feature while PR0 and the Skunk system do not. (The PR0 paper [20] describes how writes on multiple devices belonging to the same user can be supported but this idea is not implemented in either PR0 or Skunk.) Section 3.5 contains a more detailed analysis. We believe that this choice simplifies the Skunk system, is appropriate for a user-centric name space and device ownership model, and still enables meaningful collaboration among different users.

**Conflict resolution.** Coda, Bayou, and Fluid Replication support varying degrees of conflict resolution while the PR0 and Skunk systems do not: as storage systems, these latter systems possess little information for handling conflicts. As discussed in Section 3.5, we believe that the coexistence of a general peer-to-peer storage system and specialized peer-to-peer shared read-write stores provides the best of both worlds.

**Decoupled metadata and data propagation.** In Coda and Bayou, data and metadata are intertwined in the update log and the process of bringing another machine up-to-date requires the propagation of both data and metadata. New data may reside at at least two places: both in the update log and in the "normal" data store. During this propagation process, which may involve a large amount of data movement, user access to the system needs to be suspended to avoid exposure to inconsistent state. Fluid Replication relieves this inconvenience as only metadata, namely, consistency information, is exchanged among the WayStations across a potentially weak link most of the time. In PR0, only the location of the latest updates needs to be propagated when two disconnected devices are reconnected so data propagation can happen in the background. In the Skunk system, only the data-less invalidation log needs to be propagated to keep devices consistent and data can be propagated in the background as well.

**Built-in data redundancy.** PR0 is the only system that guarantees a high level of data reliability: it tolerates any single device loss. Bayou replicas provide some safety in the case of a device loss but newly created data may be lost if the loss of the device occurs before the data is propagated elsewhere. All other systems rely on a backup-restore scheme. The problem is easier to solve in Coda where backing up a single server's data is sufficient, but it is more complex for Fluid Replication which may have its state distributed across numerous WayStations. The Skunk system uses a distributed snapshot scheme for backing up data.

## 7.2 Other Systems

In cluster file systems [1, 11, 24], like in the Skunk system, data may reside anywhere. The Skunk system is more aware of a potentially more diverse topology as it employs the topology-aware Skunkast to locate data. An alternative of providing storage support to mobile devices is to rely on an omnipresent file system "utility" [10] that the devices are always connected to. The utility itself may be made of peer-to-peer components. In the Skunk system, our goal is to exploit the storage elements embedded in end devices themselves to lessen the demand on a wide-area network. Other recent wide-area peer-to-peer file systems employ distributed hash table-based placement algorithms [3, 19]. The Skunk system is more sensitive to the physical topology and must control its own data placement in a flexible manner. Distributed databases [6, 14], like Bayou, use update logs to keep replicas consistent. The invalidation log in the Skunk system contains only invalidation records and the system does not need to propagate data to quickly bring other devices up-to-date. The use of a consistent storage snapshot for backup-restore has been used by many systems [15, 17]. The Skunk system leverages the propagation of the invalidation log to create a consistent snapshot across a number of distributed devices, which are then used to create a backup, potentially across multiple backup devices. Finally, Skunkast is similar to user-level multicast systems [2, 5, 7] except not all hosts in the target list of a Skunkast need to be reached.

## 8 Conclusion

We have designed and implemented a peer-to-peer storage system that caters to the needs of mobile users. The system coordinates a plethora of a user's existing devices to form a coherent whole so that it may ease the user's increasingly difficult data management chores. By judiciously introducing and enlisting the aid of ad hoc peer-to-peer connectivity and portable storage elements, the system can overcome the inadequacy of wide-area connectivity. We believe that a combination of the system's features, including the Skunkast data location mechanism, the lazy invalidation log propagation, and the distributed snapshot scheme, are uniquely suited for the decentralized, autonomous, and personalized environments it targets.

## 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] Chu, Y., Rao, S. G., and Zhang, H. A Case for End System Multicast. In *Proc. of the 2000 SIGMETRICS* (June 2000).

[3] 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.

[4] 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.

[5] 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.

[6] Gorelik, A., Wang, Y., and Deppe, M. Sybase Replication Server. In *Proc. ACM SIGMOD Conference* (May 1994), p. 468.

[7] 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).

[8] 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).

[9] Kistler, J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[10] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2000)* (November 2000).

[11] 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.

[12] Morris, R. From Atoms to People. Keynote Speech, First Conference on File and Storage Technologies, January 2002.

[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] Oracle Corporation. *Oracle7 Server Distributed Systems: Replicated Data*, 1994.

[15] Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., and Owara, S. SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[16] 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.

[17] Quinlan, S., and Dorward, S. Venti: a new approach to archival storage. In *Proc. First Conference on File and Storage Technologies* (January 2002).

[18] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (November 2001), pp. 329–350.

[19] 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).

[20] 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).

[21] Sony hangs its memory on a key chain. http://www.cnn.com/2001/TECH/ptech/11/13/-sony.keychain.idg/, November 2001.

[22] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM 2001* (August 2001).

[23] 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.

[24] 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).

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