Data Staging on Untrusted Surrogates

Jason Flinn[⋆‡], Shafeeq Sinnamohideen[†‡], Niraj Tolia[†‡], and M. Satyanarayanan[†‡]
[‡]Intel Research Pittsburgh, [⋆]University of Michigan, and [†]Carnegie Mellon University

Abstract

We show how untrusted computers can be used to facilitate secure mobile data access. We discuss a novel architecture, data staging*, that improves the performance of distributed file systems running on small, storage-limited pervasive computing devices. Data staging opportunistically prefetches files and caches them on nearby surrogate machines. Surrogates are untrusted and unmanaged: we use end-to-end encryption and secure hashes to provide privacy and authenticity of data and have designed our system so that surrogates are as reliable and easy to manage as possible. Our results show that data staging reduces average file operation latency for interactive applications running on the Compaq iPAQ handheld by up to 54%.*

1 Introduction

Can an untrusted and unmanaged computer facilitate secure mobile file access? Surprisingly, the answer is "yes." In this paper, we show how such a computer can be used as a data staging node to improve the performance of cache miss handling in an Internet-wide distributed file system. The untrusted computer, called a *surrogate,* plays the role of a second-level file cache for a mobile client. By prefetching files and staging them on the surrogate, cache misses from a nearby mobile client can be serviced at low latency (typically one wireless hop) instead of full Internet latency.

The use of surrogates for data staging can bridge a growing mismatch between the desires and expectations of users. On the one hand, users want the lightest and smallest wearable or handheld computer—for example, a wristwatch running Linux is no longer a fantasy [23]. On the other hand, users expect productivity improvements from mobile computing; ubiquitous access to personal and project data is a key part of this expectation. A distributed file system can provide such ubiquitous access, but requires crisp handling of cache misses to achieve good interactive performance. For a small file, network latency to a distant file server on the Internet is typically the dominant component of cache miss service time. This can be reduced by redirecting cache misses to data staged on a nearby surrogate while still maintaining the consistency guarantees of the underlying file system. The alternative of totally avoiding cache misses through hoarding [15, 18] is not viable because of limited cache space and the need to view recent updates by other users. Further, it is usually not possible to perfectly predict the set of files that will be accessed when mobile; the working set of files may change unexpectedly in response to real-world events such as phone calls. Consequently, the set of data that may *possibly* be accessed is much larger than the set of data that is *actually* accessed.

What is the likelihood of a mobile computer finding a nearby surrogate? Although the chances are low today, we predict that continuing decline in mass-market hardware prices will improve these chances in the future. Desktop computers at discount stores already sell for a few hundred dollars, with prices continuing to drop. In the foreseeable future, we envision public spaces such as airport lounges and coffee shops being equipped with surrogates for the benefit of customers, much as comfortable chairs and table lamps are provided today. These will be connected to the wired Internet through high-bandwidth networks, and to mobile clients in their neighborhood through wireless technologies such as 802.11 [13] or Bluetooth [12].

Since hardware cost is only a small part of the total cost of ownership of a system, it is essential that surrogates require virtually no maintenance or system administration. Like a chair or table lamp, they should require negligible attention after initial setup. Only then will they be cheap enough for widespread deployment. This leads to two important assumptions about surrogates in our work: they are *unmanaged* and *untrusted.* In particular, we make surrogates as reliable and easy to manage as possible by maintaining no hard-state on surrogates, building as much as possible on commodity software, and pushing functionality from surrogates to client and server machines.

We rely on the concept of *caching trust* to guard against malicious surrogates [26]. This end-to-end approach ensures privacy through encryption, and integrity through

verification of secure hashes. Even the most resource-challenged mobile client typically has enough disk or flash storage to cache hashes and encryption keys of all files of potential interest to the user. Hence, data never has to be stored in the clear on a surrogate—it is encrypted before transmission to the surrogate and remains encrypted there. When servicing a cache miss, the client decrypts file data received from the surrogate, calculates the hash, and verifies the computed hash against its cached copy. A compromised surrogate could, of course, still cause mischief through denial of service. The client's only recourse is to contact servers directly. Even in this case, performance is no worse than in the absence of data staging, except for an initial disruption while the client detects that a surrogate is misbehaving and abandons it.

We report on the feasibility of data staging on untrusted surrogates. Our prototype implementation is based on the Coda file system [28], but is structured for easy use with other distributed file systems. Measurements from this prototype confirm the performance benefits of data staging. For bursty, short-term workloads, data staging reduces the cumulative delay due to file operations by up to 54%. We have confirmed these results by replaying long-term traces of file-system activity—these experiments show reductions in file operation latency of up to 49%.

We focus on the file system aspects of data staging. Topics such as surrogate discovery (possibly through mechanisms such as Jini [36] or UPnP [20]) and load balancing across multiple surrogates are part of our plans for future work. We begin, in Section 2, by describing the design and implementation of data staging. The following section describes additional scenarios under which data staging can be profitably employed. We evaluate the benefit of data staging for storage-limited clients in Section 4. The final three sections discuss related work, describe our plans for future work, and summarize our results.

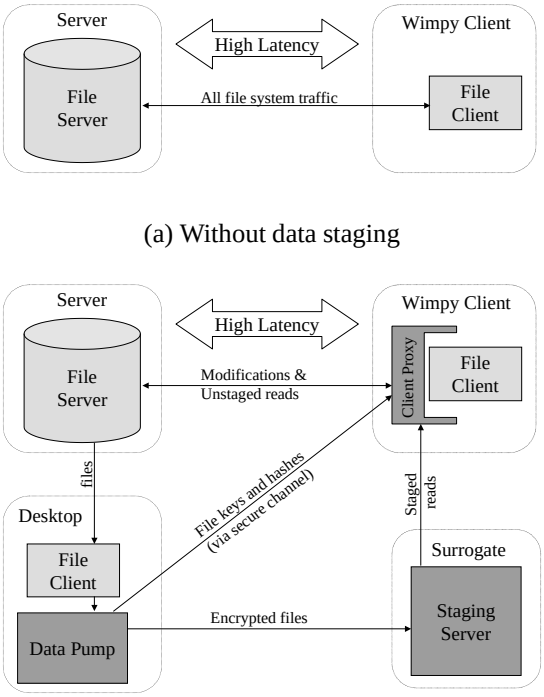


Figure 1: Data staging architecture

2 Design and implementation

2.1 Overview

Figure 1(a) shows a typical scenario that motivates the need for data staging. An interactive application running on a storage-limited client accesses files stored in a distributed file system. The file system attempts to reduce access time by caching files on the client machine, but limited space and imperfect prediction prevent it from caching all but a portion of the files that the user might potentially read. Consequently, many files needed by the application are not cached and must be fetched from the distant file server. The user experiences many frustrating delays because the application reads multiple files sequentially and reading each file incurs multiple network round-trips.

Figure 1(b) shows how data staging improves this scenario. On the client machine, we interpose a proxy that intercepts file system traffic. When the proxy observes that remote file accesses are incurring high latency, it finds a surrogate in the nearby network environment that is willing to provide extra storage capacity (currently, this process is manual). The proxy registers with the surrogate and stages the set of files that the user is most likely to access in the future. Since the surrogate has much greater storage capacity than the mobile device, it can store many more files.

Staging is expedited by a *data pump* that typically executes on a user's idle desktop computer located near the file server. When the client proxy wishes to stage a file, it sends a message to the pump through a secure channel. The pump authenticates the message, reads the file from the file system, encrypts the file, and generates a cryptographic hash of the data. The pump transmits the encrypted file to the surrogate and sends the file key and hash to the client through the secure channel. When a

staged file is read by the application, the proxy fetches the file from the nearby surrogate, decrypts it, and uses the hash to verify that the file has not been modified. Prefetching files to the surrogate decreases the number of high-latency, blocking file accesses and dramatically reduces the number of long delays experienced by the user.

Reads of unstaged data are serviced using the base file system protocol. The prefetching of files to the surrogate proceeds concurrently with file system traffic. After each file is staged on the surrogate, it becomes immediately available for client use. Thus, as the number of staged files grows, the percentage of cache misses that need to be serviced by the distant file server decreases, leading to significant improvement in interactive application performance. Additionally, the proxy sends all update traffic directly to the file server (although trickle reintegration in Coda may delay updates for a short period of time to improve performance [21]). The client proxy maintains consistency by marking modified files invalid. For workloads with a mixture of read and update traffic, this design improves read performance without compromising security, relaxing consistency guarantees, or significantly delaying updates.

The proxy-based architecture allows us to achieve a great deal of independence from the underlying distributed file system. Data staging requires no modification to file system source code; we use gray-box techniques [3] where necessary to infer file system state. Further, almost all file-system specific code is encapsulated within the client proxy. Thus, while our current system uses Coda, the changes needed to support additional file systems such as NFS and AFS would be minimal.

We first describe the threat model for data staging. In the subsequent three subsections, we describe the design and implementation of the surrogate, client proxy, and data pump in more detail.

2.2 Trust and threat model

Data staging defends against attacks that involve malicious or faulty surrogates. Since we propose that clients opportunistically discover and use third-party surrogates, we place no trust in any surrogate computer. Data staging must defend against attacks that attempt to read private data stored on a surrogate, as well as attacks that corrupt staged data or provide stale data through replay attacks. We also must defend against attacks that attempt to eavesdrop or modify network communication.

Data staging does not explicitly defend against denial-of-service attacks that render surrogates unavailable. Also, a malicious surrogate may periodically refuse to provide requested files to a client. However, if a surrogate performs significantly worse than expected, the client proxy may abandon use of the surrogate without further cost.

Data staging does not defend against attacks that compromise a user's client or desktop machine, or attacks that compromise a file server. We assume that these machines belong to a common administrative domain that enables secure distribution of public keys. Further, we assume that the network communication of the underlying distributed file system is secure. Finally, our protocol assumes that the cryptographic algorithms we employ are sufficiently strong to withstand brute-force attacks.

2.3 Surrogate

2.3.1 Design principles

We are convinced that widespread deployment of surrogates hinges on ease of management. As mentioned previously, surrogates should be as easy to manage as table lamps—they should not need a system administrator or a complex user manual. We have identified three design principles that improve ease of management:

- *Exploit commodity software.* We build as much as possible upon widespread commodity software, so as to leverage the improved reliability that comes through the extensive testing provided by a large user community. To this end, we use the Apache Web server as the base system for our surrogates. We have identified the minimum set of additional functionality that must be located on the surrogate, and provide this functionality with CGI scripts. All other functionality is pushed to the client proxy and data pump in order to keep the custom code base on the surrogate as simple and reliable as possible.
- *Avoid long-term state.* We maintain only soft state on the surrogate so that no critical information is lost if the surrogate is disrupted by power failure or a system crash. For example, we do not buffer client modifications to file data on surrogates. Thus, clients need not guard against malicious or faulty surrogates that might lose modifications. Further, surrogates do not need to run potentially complex reconciliation protocols.
- *Allow file system diversity.* Our surrogate implementation is completely independent of the underlying file system employed by the user. This means that surrogates need not be updated to reflect new file system versions. Further, a single surrogate can simultaneously service users who are employing different underlying file systems.

```
SurrogateRegister    (IN surrogate, IN pubkey, OUT clientid, OUT quota,
                     OUT sesskey, OUT expire);
SurrogateRenew       (IN surrogate, IN clientid, OUT expire);
SurrogateDeregister  (IN surrogate, IN clientid);
SurrogateStage       (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen);
SurrogateUnstage     (IN surrogate, IN clientid, IN fileid);
SurrogateGet         (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen,
                     IN key, IN hash);
```

The surrogate API is implemented as Perl CGI scripts. In total, these scripts consist of 643 lines of source code.

Figure 2: Surrogate API

2.3.2 Surrogate API

Figure 2 shows the surrogate API. Wrapper libraries on the client machine and data pump implement these functions as HTTP/1.1 operations. When `SurrogateGet` is called, the wrapper library issues a HTTP GET request; the remaining functions are implemented as POST operations that invoke CGI scripts on the surrogate.

A client proxy calls `SurrogateRegister` to start using a surrogate. The proxy provides its public key, which is used for authentication. If the surrogate is willing to provide storage space, it assigns the proxy a unique identifier and specifies a storage quota. The surrogate also generates a shared session key, encrypts it with the proxy's public key, and returns it to the proxy. The session key is used to authenticate all subsequent messages that modify surrogate state. `SurrogateRenew`, `SurrogateDeregister`, `SurrogateStage`, and `SurrogateUnstage` each send the surrogate a token encrypted with the session key that represents the command being executed; nonces are used to guard against replay attacks. When a proxy successfully registers, it is granted a lease to use the surrogate. The time duration of the lease is returned by `SurrogateRegister`. Before the lease expires, the proxy may renew it using `SurrogateRenew`.

The `SurrogateStage` function places files on the surrogate. The surrogate treats each file as a binary chunk of data with an identifier unique to the proxy. The CGI script for `SurrogateStage` stores the file data and updates the amount of storage currently used by the proxy. However, if storing the file would cause the proxy to exceed its quota, an error is returned. If a file with the same identifier already exists on the surrogate, the previous data is replaced and the quota updated appropriately.

Once a file is staged, it may be retrieved by the proxy using the `SurrogateGet` function. Alternatively, the proxy may delete the staged file to free up storage capacity with the `SurrogateUnstage` function. The final function, `SurrogateDeregister`, explicitly releases surrogate resources. After a proxy deregisters or its lease expires, the surrogate deletes all files stored on behalf of the proxy.

2.4 Client proxy

The client proxy performs three primary tasks, described further in the next three sections. These tasks are:

- Redirecting file requests to surrogates
- Controlling which files are staged
- Preserving consistency

2.4.1 Redirecting file requests

The client proxy intercepts all traffic bound for a specified set of servers. It masquerades as a local file server; thus, the file system client believes it is connected to a file server running on the local machine. When the proxy receives a request from the file system client, it either transparently forwards it to the distant file server for which it is masquerading, or it retrieves staged data from a nearby surrogate and responds to the request itself. In the presence of multiple file servers, our design allows us to interpose proxies only for high-latency servers—traffic to nearby servers need not incur the additional latency of passing through the proxy.

The proxy maintains a hash table of all files stored on the surrogate. When it intercepts a request to read data from a Coda server, it checks whether a valid copy of the file is currently staged on the surrogate. If the file is staged and valid, it calls `SurrogateGet` to retrieve the data, decrypts the file, computes a secure hash, and verifies that the hash matches its cached value. The file retrieval, decryption, and hash computation are pipelined to reduce access latency. If a valid copy of the file is not staged, the proxy forwards the request to the distant file server.

All modifications to file data on the client machine are sent directly to the file server. If a copy of the modi-

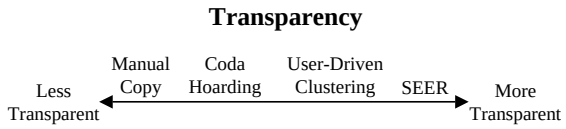


Figure 3: File prediction strategies

fied file is staged, the proxy marks the copy invalid—this process is described further in Section 2.4.3.

2.4.2 Controlling which files are staged

The client proxy predicts which files are most likely to be accessed in the future and arranges for them to be staged on nearby surrogates. In order to accurately predict future file accesses, it observes all file system traffic. Currently, the proxy takes advantage of Coda's `codacon` interface [27], which reports the name of each file upon invocation of the `open` system call. If the underlying file system did not provide this type of interface, other mechanisms such as the the Sysinternals Filemon tool [32] are available.

Prediction is implemented through a modular interface; whenever a file is opened, the client proxy passes the prediction module the file name, size, and access time. By avoiding a tight integration of the prediction algorithm and the staging implementation, we are able to more easily explore alternative prediction strategies. Figure 3 lists several possibilities, characterizing them by how transparent they are to the user. At one end are completely manual methods, such as explicit copying of files. These methods generate the most distraction; users must specify which files will be needed and must perform the prefetching themselves. The advantage of such methods is better user control of staging. At the other end of the spectrum are completely automated algorithms such as those proposed by Amer [2], Griffioen [11], Kroeger [16], and Kuenning [18]. For example, Kuenning's SEER observes file access patterns and creates clusters of files that are often accessed together. Once a file system determines that some files in a cluster will be accessed, it can prefetch all related files in that cluster.

To date, we have explored two prediction algorithms that lie between manual copy and fully-automated algorithms on the scale of user transparency. Both algorithms use the concept of *roles*; a role is an explicit grouping of files that is associated with a high-level task commonly performed by a user. For example, a graduate student may create roles such as `thesis`, `coursework`, and `personal`. Conceptually, roles are quite similar to SEER's clusters, except that they are externally visible to the user. Roles could potentially be integrated with

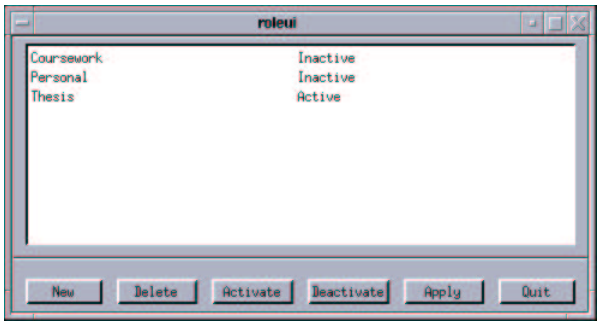


Figure 4: User interface for roles specification

higher-level abstractions such as Aura tasks [10, 29] or Lifestreams [9].

The first algorithm is based upon *Coda file system hoarding* [15]. For each role, a user explicitly specifies the set of file system subtrees that are most likely to be accessed. The user also orders these subtrees by expressing a relative priority for prefetching; higher-priority subtrees are more likely to be accessed and will therefore be staged before lower-priority subtrees. Using the interface shown in Figure 4, users specify the set of roles they are currently performing (these are referred to as *active* roles). When the client proxy discovers a surrogate, it creates a prefetch list that is the union of the files specified for all active roles. It stages files in order of priority until its storage quota on the surrogate is exceeded.

The interface in Figure 4 also allows users to cope with unexpected changes in their working set. For example, consider an engineer waiting for a flight in an airport lounge with a nearby staging server. The engineer receives e-mail from a colleague pertaining to a previous joint project. Since this project has unexpectedly become important, the engineer does not have the associated design documents and technical specifications cached. The engineer uses the interface in Figure 4 to activate the role associated with the project. The client proxy then recalculates the priority of files to stage and prefetches related files to the nearby surrogate. As the engineer works on the project, application performance continues to improve as more files are staged.

We call the second algorithm that we explored *user-driven clustering*. This algorithm automatically generates associations between files and roles; the relative priority of files to prefetch is determined through a simple LRU strategy. The prediction algorithm maintains separate LRU lists for each role. Whenever a file is opened, that file is placed at the head of the LRU list for all roles that are currently active. This is a conservative strategy: all files that are part of a role will be in the LRU list, but not all files in the LRU list will be part of a role.

When the client proxy prefetches files to a surrogate, it merges the LRU lists of all active roles, then prefetches the most recently accessed files until its storage quota is exceeded. This strategy avoids the need for users to explicitly specify which files to prefetch, but may potentially be less accurate. Similar to the previous algorithm, users can use the interface in Figure 4 to specify when their working set changes.

Our implementation is based upon two important observations. First, we expect available space on nearby surrogates to change by several orders of magnitude as users move: from zero when no surrogate is available, to gigabytes of storage when a high-capacity surrogate is nearby. Therefore, the amount of prediction information maintained should be independent of the current surrogate quota. We maintain LRU information sufficient to populate the allocated quota of any surrogate we may encounter in the future. While the storage requirements for the LRU data are not large (< 1 MB in our experience), this may still represent a significant portion of the storage capacity of a small handheld. Therefore, we store LRU data in the distributed file system, allowing it to be flushed from the client file system cache when additional storage is needed.

Our second observation is that it is now common for one person to access the same data on multiple machines; for example, a single user may own a home computer, a desktop at work, a laptop, and a handheld device. If the user has recently accessed a file on one machine, it is more likely that the user will soon access the file on other machines. We account for this behavior by storing per-machine LRU data in the distributed file system. When the proxy starts, it combines the LRU data from all machines that the user has recently accessed to generate a global LRU ordering. This ordering is then used to select which files are staged. To minimize update traffic, the client proxy reads and writes LRU information periodically (currently every hour). While this strategy has the potential to lose some data in the event of a system crash, the only effect of such a loss is a decrease in prediction accuracy.

2.4.3 Preserving consistency

The client proxy is the final arbiter of whether data staged on a surrogate is valid. It associates a valid bit with each file in its hash table of staged files. After successfully staging a file, this bit is set to valid. When the proxy intercepts a request that modifies file data, it searches for the file in the hash table and invalidates the entry if found. When another Coda client modifies a staged file, Coda provides a `callback` notification to all Coda clients that have accessed the file. The data pump receives this callback and forwards it to the client proxy. If the modified file is currently staged, the proxy invalidates it.

The proxy periodically rescans the LRU list and recalculates which files should be staged. It stages any file at the head of the LRU list that is marked invalid or is not currently staged. Files further toward the tail of the LRU list are unstaged to make room within the allocated quota. We have chosen to make the set of files staged on the surrogate inclusive with the client's Coda file cache, approximating the stack property. This means that files evicted from the Coda cache are immediately available from the surrogate. The performance penalty of inclusive caching is small, since the surrogate will typically have storage capacity several orders of magnitude greater than that of the client machine.

An alternate approach would be to stage files immediately after they are invalidated or newly created. However, when file modifications are bursty, this alternate approach would lead to many successive stages and invalidations, wasting client bandwidth and energy.

2.5 Data pump

The data pump fetches and stages files on behalf of the client proxy. Although a single data pump could run on the file server, we favor running a data pump on the desktop computer of each user instead. The latter alternative has the benefit of reducing load on file servers; since the desktops have large file caches, most requests to stage data can be serviced without contacting the file server.

Client proxies contact the data pump and establish a secure tunnel for communication. The two parties use public key cryptography to establish a symmetric session key. We use the session key to encrypt all further traffic because symmetric key encryption is less computationally demanding than public key encryption. Public key distribution is simplified since a single user will typically operate both machines (if the pump is located on a desktop), or both machines will lie within the same administrative domain (if the pump is located on a file server).

When the client proxy needs to stage a file, it sends the data pump the file pathname and surrogate IP address. The data pump retrieves the file from the underlying distributed file system, generates a random symmetric key, encrypts the file, and generates a cryptographic hash of the file data. The pump calls `SurrogateStage` to place the encrypted file on the surrogate. If successful, the pump sends the key and hash to the client proxy, which stores them for later reference. Our current implementation uses 64 bit DES encryption and generates 128 bit MD5 digests of file data. The storage requirement of 24

bytes per file is significantly less then the average file size reported in file system studies [7, 35].

The client proxy is multi-threaded, allowing it to overlap computation and network transmission, and to service multiple concurrent requests. As a performance enhancement, the proxy may batch multiple `SurrogateStage` requests into a single HTTP/1.1 POST request; this decreases the time needed to stage a large number of small files.

3 Further benefits of data staging

To date, our work has focused on exploring the benefits of data staging for storage-limited clients in pervasive computing environments. Yet, we believe that data staging will prove desirable in several other important scenarios.

The use of Infostations [37] or Data Blasters [19] has been suggested as a solution for overcoming the bandwidth limitations of wide-area wireless networks. Clients periodically pass through short-range, high-bandwidth zones located within the pervasive infrastructure. For example, such zones may be located near airport gates or at highway tollbooths. Data updates such as file modifications can be burst transmitted to the client during the short period of high-bandwidth connectivity. These data staging scenarios are particularly attractive given advances in ultrawideband (UWB) wireless technology that promise to deliver up to 500 Mb/s within a 5-10 meter radius with minimal energy cost [19]. To utilize this effective bandwidth fully, data must be staged in preparation for burst transmission.

Data staging also has large potential benefits for battery-powered clients. Studies of energy usage show that an 802.11 network interface represents a very large portion of the total energy budget of small handheld devices [8, 31]. Under periods of high network usage, such devices quickly run out of battery power.

Our data staging architecture can significantly extend the battery lifetime of mobile clients in two ways. First, fetching small files from surrogates is significantly faster than fetching the equivalent data from distant file servers. Since the network is active for shorter periods with data staging, the network interface can be put into longer and deeper power saving modes. A second, potentially greater, benefit is that the speculative prefetching of data to clients can be dramatically reduced. With data staging, the latency of a cache miss is much lower. Therefore, the client file system can afford to be less aggressive in keeping the cache up-to-date with the files the user is most likely to fetch. Instead, the client can stage such files on a nearby surrogate, knowing that the penalty of a cache miss will be small.

4 Evaluation

How much does data staging improve the performance of interactive applications running on storage-limited clients?

We answer this question by measuring the performance impact of data staging in two different usage scenarios. In the first scenario, we mirror short-term, bursty activity. We model a user browsing images from a large image library stored in the Coda file system and examine the potential performance benefits of data staging. In the second scenario, we examine the benefits of data staging over a longer time period. We use recorded traces of client file system workloads to represent the activities of a user on a multi-day visit to a distant work location and examine how data staging reduces file operation latency.

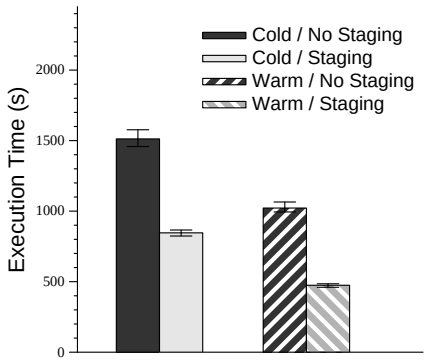
4.1 Experimental setup

We ran a Coda server on a powerful desktop computer with a 2 GHz Pentium 4 processor. We also chose to run the data pump on the same machine since scalability was not a focus of our evaluation. The surrogate ran on an identical desktop computer—we used NISTnet [5] to emulate a 30 ms. delay (60 ms. round-trip time) between the Coda server and surrogate. The 30 ms. delay is typical of current coast-to-coast delays in the United States. The client was a Compaq iPAQ 3850 handheld computer with a 206 MHz StrongArm processor. The iPAQ used a 11 Mb/s 802.11 wireless network card for communication. The wireless hub was on the same network segment as the surrogate; we also emulated a 30 ms. delay between the client and Coda server. All computers ran the Linux 2.4 operating system.

4.2 Image browsing

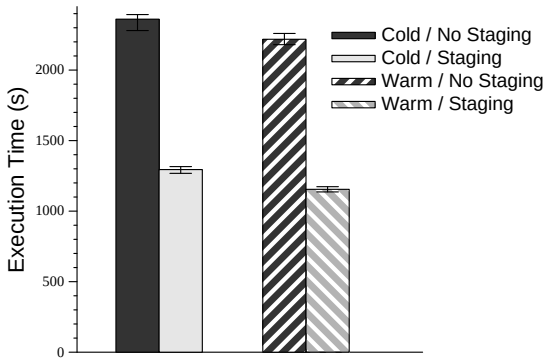
4.2.1 Methodology

Over a roughly one month period, we recorded accesses to a library of digital photographs stored in the Coda file system. The typical client activity captured in this trace first opens a large number of small, thumbnail images in a directory, then opens a smaller number of large, full-sized images in the same directory. From the log, we selected the first 10,148 file operations—these operations read 150 MB of unique file data. However, since many images are read more than once, the total amount of data accessed is 401 MB.



This figure shows the benefits of data staging for the image benchmark with a 64 MB client Coda cache. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 5: Image trace with 64 MB cache



This figure shows the benefits of data staging for the image benchmark with a 16 MB client Coda cache. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 6: Image trace with 16 MB cache

We replayed the trace as fast as possible using the DFSTrace tool [21], which performs file operations identical to those recorded in the trace. The figure of merit is the time needed to execute the complete trace—this corresponds to the total delay that the user experiences while loading images. This is distinct from the total amount of time the user will take to view the images, which will include a variable amount of think time. The benefits of staging would increase with think time since the client will continue to stage files during such pauses. Without data staging, think time has no noticeable effect upon total delay.

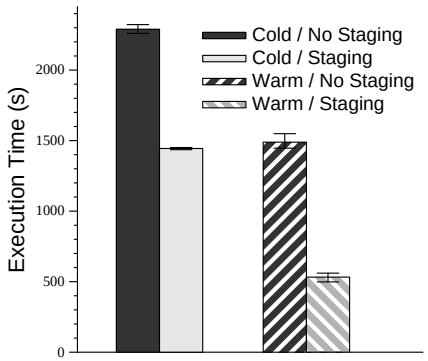
Since the initial state of the client Coda cache will have a large effect on benchmark execution time, we examine the two extreme ends of the spectrum. In the `cold` scenario, no data is contained in the Coda cache when we begin playing a trace; such a scenario will often occur when there is an unexpected change in a user's working set such as the one described in Section 2.4.2. In the `warm` scenario, we fill the entire Coda cache with the set of files initially accessed by the trace; this is the best possible initial cache state.

For both scenarios, we compare total file access time with and without data staging. In the `cold` scenario, the surrogate initially has no data staged. As the trace begins, the client proxy is notified of the change in working set; this emulates the activation of a new role through the interface in Figure 4. Replay of the trace and staging of the data proceed in parallel. Initially, no benefit is derived from data staging; however, as more files are staged, a greater percentage of file accesses are serviced by the surrogate, and average file access latency decreases. Files are staged in random order using a hoard file that lists all files accessed by the trace—the random ordering is a pessimistic assumption, in a production system, LRU or user-specified ordering would cause those files most likely to be accessed to be staged first. For the `warm` scenario, in addition to warming the client Coda cache, we also stage all files referenced by a trace on the surrogate before executing the trace. This represents a scenario where the user has given advance notice of the change in working set sufficient to prefetch all files.

The effectiveness of staging depends upon how closely the set of files staged on the surrogate matches the actual set of files accessed by the client. Inaccuracy is exhibited in two ways. First, the client proxy may stage files that are never accessed—we refer to this as wastage. Specifically, we define the *wastage ratio* to be the ratio of data staged but never accessed to the total amount of data staged. Second, the client proxy may decide not to stage files that it later accesses—we quantify this with a *staging miss ratio*. We calculate the staging miss ratio by dividing the amount of file data accessed by a trace but never staged by the total amount of data accessed by the trace. Our approach to handling this variability is to first choose initial baseline values, specifically a 33% wastage ratio and a 0% staging miss ratio, and then perform sensitivity analysis on each variable.

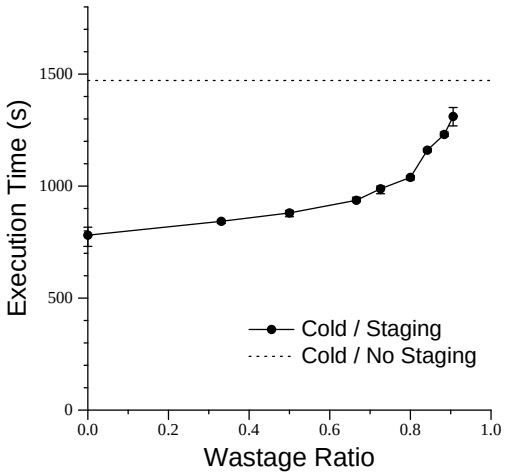
4.2.2 Results

Figure 5 shows the results of running the image benchmark with a 64 MB Coda cache size. In the `cold` scenario, data staging reduces the total time to execute the trace by 44% (11:06 minutes). The `warm` scenario executes in less time because the images initially viewed by the user are already in the Coda cache. In this scenario, the use of data staging reduces execution time by 54% (9:07 minutes).



This figure shows the benefits of data staging for the image benchmark with a 64 MB client Coda cache and a wide-area bandwidth cap of 1 Mb/s. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 7: Image trace with 1 Mb/s bandwidth cap



This figure shows the effect of different wastage ratios on the image trace with a 64 MB client Coda cache. Each data point is the mean of three trials; the error bars show minimum and maximum values.

Figure 8: Effect of wastage on image trace

Figure 6 shows results for a smaller, 16 MB cache size. Somewhat surprisingly, the relative benefits of staging are mostly unaffected by the reduction in cache size. For this trace, much of the short-term file reference locality that is captured by the 64 MB cache size is also captured by the 16 MB cache size. In the `cold` scenario, staging reduces execution time by 45% and, in the `warm` scenario, staging reduces execution time by 48%. However, the absolute magnitude of the benefit increases to 17:46 minutes in the `cold` scenario and 17:44 minutes in the `warm` scenario.

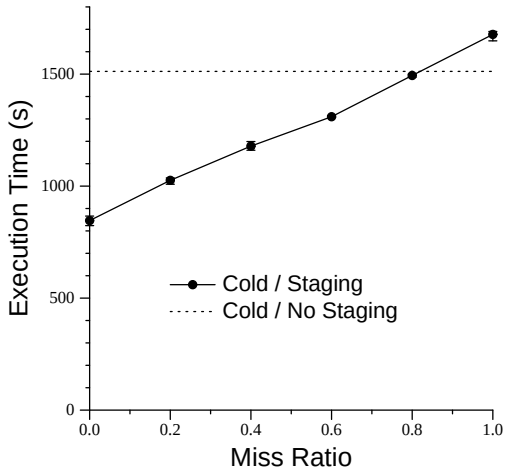
While our work assumes that network bandwidth is not a significant limitation, it is interesting to consider how data staging might perform if wide-area network throughput is limited. Using NISTnet, we capped the maximum throughput of the network link between the pump and surrogate at 1 Mb/s (in addition to imposing a 30 ms. latency). We imposed the same limitation on the link between the client and file server.

Figure 7 shows results for a 64 MB Coda cache size. Since many of the digital photographs are quite large, the bandwidth cap has a significant performance impact. Without data staging, the cap increases trace execution time by 51% in the `cold` scenario and by 46% in the `warm` scenario. In the `cold` scenario, data staging reduces trace execution time by 26%. The bandwidth cap reduces the relative benefit of staging because it takes longer to stage files at the surrogate; a greater percentage of files are accessed directly from the file server. In the `warm` scenario, trace execution time is essentially unaffected by wide-area bandwidth limitations, since all cache misses are serviced by the surrogate. Therefore, data staging is more effective; it reduces trace execution time by 64%. Note that these results reflect wide-area bandwidth limitations. If the bottleneck link is a 1 Mb/s Internet connection shared by client and surrogate, then prefetching traffic may significantly degrade file system performance. Methods that limit prefetching traffic [33] may prove effective in such scenarios.

We next executed the image benchmark with a 64 MB client Coda cache and performed a sensitivity analysis on the wastage ratio. In the `cold` scenario, wastage causes the staging of files accessed in the trace to be delayed. With enough wastage, a file may not be staged until after it is accessed during trace replay. After this point, staging provides no benefit for the file. Wastage does not affect the `warm` scenario since all files are staged before the trace is executed.

As Figure 8 shows, staging reduces the execution time of the image benchmark by 47% in the optimal case where there is no wastage. As wastage increases, the benefits of staging are gradually reduced. With a 91% wastage ratio, staging reduces execution time by only 11%. The effect of wastage may be overstated due to our assumption that files are staged in random order; we expect that if files more likely to be accessed were staged first, wastage would have less effect.

We also examined the effect of different staging miss ratios. We executed the image benchmark with a 64 MB client Coda cache and varied the percentage of files that are contained in the image trace but not staged. We held the wastage ratio constant at 33%. As Figure 9 shows, the benefits of staging decrease as the staging miss ratio grows. With a staging miss ratio of 80%, staging



This figure shows the effect of different staging miss ratios on the image trace with a 64 MB client Coda cache. Each data point is the mean of three trials; the error bars show minimum and maximum values.

Figure 9: Effect of staging miss ratio

achieves only a minimal 1% benefit. With a staging miss ratio of 100%, no data is staged. The resulting 11% overhead is the cost of our proxy implementation. Detailed analysis reveals that almost all of this overhead is caused by local remote procedure calls between the client proxy and Coda file system client.

Comparing Figures 8 and 9, it is interesting to note that a higher staging miss ratio reduces the benefit of staging more than the same wastage ratio. Thus, we conclude that prediction strategies should be liberal—if one is uncertain whether or not a file will be needed, it is best to stage it anyway.

4.3 Long-duration file traces

4.3.1 Methodology

To emulate the activity of a user on a multi-day visit to a distant work location, we replayed four traces of client file system activity. We selected these traces, which are summarized in Figure 10, from the set gathered by Mummert et al. [21] at Carnegie Mellon University. Each trace was gathered on a different single-user desktop computer between 1991 and 1993; their durations range from 15 to 55 hours of activity.

We used DFSTrace to replay each file trace. We repeated the methodology of section 4.2.1 by examining performance with and without data staging in both the `cold` and `warm` scenarios. In both scenarios, we stage the set of files that were captured in the trace using a manually-created hoard file. The order of staging is random in the `cold` scenario. The figure of merit is the total time

Trace	Number of Operations	Length (Hours)	Write Ops.	Working Set (MB)
purcell	87739	55.32	6%	254
messiaen	44027	42.54	2%	227
robin	37504	30.92	7%	85
berlioz	17917	15.70	8%	57

This figure shows the file system traces used for our evaluation. Since data staging currently uses Coda as the base file system and Coda uses the `open-close` semantics of AFS, individual `read` and `write` operations are not included. Hence, "write ops." refers to activities such as `close` after `write` and `mkdir`. The working set is the total size of the files accessed during a trace.

Figure 10: File traces used in evaluation

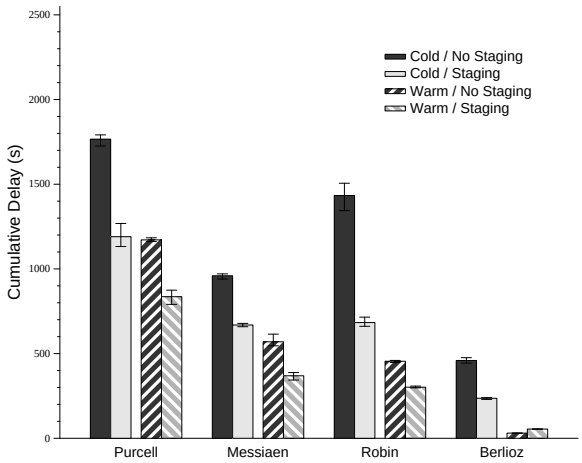
needed to perform all file operations in the trace; this is equivalent to the amount of delay the user would experience during the trace period while waiting for file operations to complete.

The traces record inter-request delays for file operations. We replay these delays for the first 15 minutes of each trace for the `cold` scenario—during this time, the client proxy stages data on the surrogate while it concurrently proxies file operations. After 15 minutes, staging completes for all traces. From this point on, we eliminate delays and replay the remainder of the trace as fast as possible. This allows us to complete the experiments in a reasonable amount of time. Elimination of delays does not affect the time to service file operations without data staging. With data staging the results are somewhat pessimistic since the client proxy does not restage data that has been invalidated due to modifications.

Although we assume here that all file activity is the result of foreground activity, it is likely that some of the trace activity was generated by background processes; however, this information is very difficult to distinguish from the trace data.

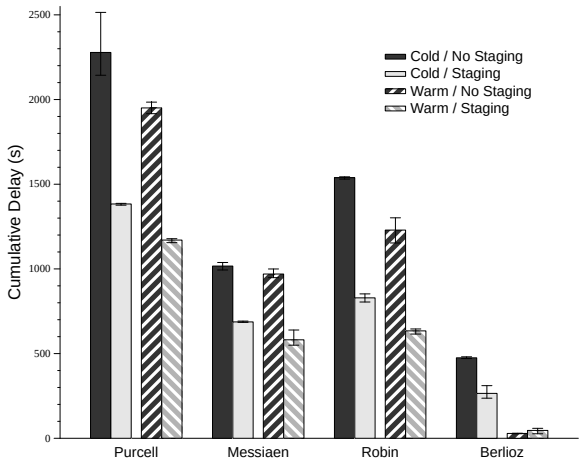
4.3.2 Results

We used an experimental setup identical to the one used for the image benchmark. We first replayed the traces assuming a 64 MB client Coda cache—Figure 11 shows the results. In the `cold` scenario, data staging provides significant benefit for all traces. Staging reduces file operation latency a minimum of 30% for the `messiaen` trace and a maximum of 49% for the `berlioz` trace. In the `warm` scenario data staging causes the `berlioz` trace to take slightly longer to complete. Since the trace's entire working set fits entirely in the Coda cache, there is no need to fetch data from the server—hence, staging provides no benefit. However, staging induces a minimal amount of overhead on each access in order to maintain



This figure shows the benefits of data staging for a 64 MB client Coda cache. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 11: File trace results with 64 MB cache



This figure shows the benefits of data staging for a 16 MB client Coda cache. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 12: File trace results with 16 MB cache

LRU prediction information; this results in the performance degradation shown in Figure 11. For the other traces, the relative benefit of staging ranges from 29% for the `purcell` trace to 35% for the `messiaen` trace.

Figure 12 shows results when we decrease the Coda cache size to 16 MB. Although the total amount of cumulative delay is larger with a smaller cache size, the relative benefits of staging do not significantly change. In the `cold` scenario, staging reduces cumulative delay from 32% to 46%. In the `warm` scenario, the `berlioz` trace again shows a slight degradation in performance. Even though its working set is larger than the cache size, much of this data is generated during trace replay—16 MB is sufficient to hold almost all data read by the trace. For the remaining traces, staging reduces cumulative delay from 40% to 48%.

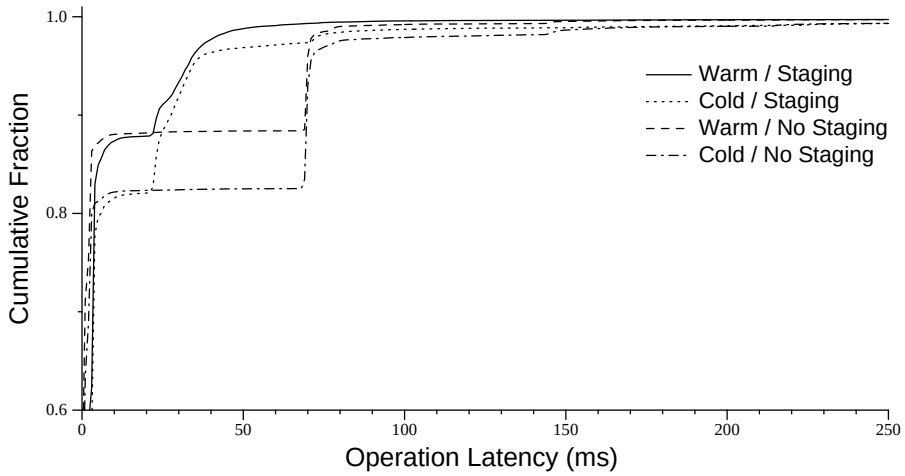
Figure 13 shows more detailed results for a representative trace—`messiaen` with a 64 MB Coda cache. Each line represents the cumulative fraction of file operations that complete within a given time period. For this trace, over 80% of the file operations are reads that hit in the Coda cache or writes that are buffered on the client for later reintegration—these incur negligible delay.

The shape of the `Cold / No Staging` line confirms one of our assumptions: latency, not bandwidth, is the real killer in distributed file system performance. Of the operations that do not complete immediately, the vast majority take slightly more than the 60 ms. round-trip delay. Data staging significantly shortens these high latency operations. For interactive applications that incur the cost of several sequential file operations, this dramatically reduces the frustrating delays experienced by the user.

Finally, we examined the effect of network latency between the client and file server. We expected the benefits of staging to decrease as network round-trip time was reduced. Figure 14 confirms our expectation. When the round-trip delay between client and file server is reduced to 30 ms., data staging decreases cumulative file delay between 5% and 26% in the `cold` scenario. In the `warm scenario`, data staging reduces cumulative file delay by less than 10% for all traces.

5 Future Work

Our current implementation of data staging provides a solid basis for future research. We plan to investigate methods that allow clients to discover servers dynamically. For this purpose, we hope to leverage existing service discovery protocols such as Jini [36] and UPnP [20]. Architectures such as VERSUDS [4] that allow clients to access multiple service discovery protocols through a common interface are especially promising for the heterogeneous environments we support. In addition, the distance-based discovery mechanism proposed by Noble et al. [24] for Fluid Replication may also



This figure shows how data staging reduces file operation latency for the `messiaen` trace with a 64 MB cache. Each line shows the cumulative fraction of file accesses that finish on or before the indicated time.

Figure 13: Reduction of file operation latency for `messiaen` trace

prove to be well-suited for data staging.

We hope to investigate how location prediction can increase the effectiveness of data staging. If the client proxy determines that it will pass near a surrogate in the future, it could proactively stage data there. This functionality is especially useful in constructing Infostation-like environments such as the one described in Section 3.

We plan to support additional file systems; a NFS implementation is in progress. This process is expedited by our encapsulation of file system dependent code in the client proxy and data pump. However, some new issues arise; for instance, without the whole-file caching of Coda, reduction of first-byte latency becomes important. We also plan to investigate the effectiveness of prefetching for different file system parameters (whole file caching vs. block caching, callbacks vs. leases, etc.) Finally, we plan to explore other prediction strategies, especially fully-automated ones such as SEER [18].

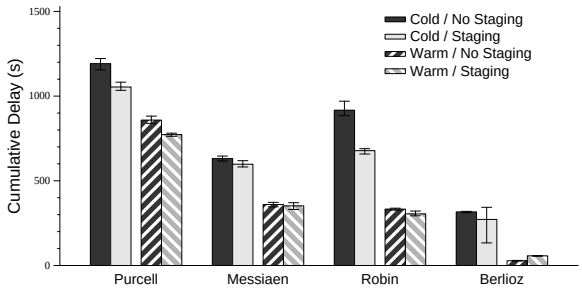
6 Related Work

This work is one of the first to focus on how untrusted and unmanaged hardware can improve distributed file system performance for small, storage-limited clients without compromising security or consistency. In effect, data staging applies the well-understood concept of prefetching [6, 25] to pervasive computing environments. Instead of prefetching file blocks from the disk, data staging prefetches whole files from a distant server.

At a conceptual level, data staging shares several goals with edge computing initiatives such as distributed Web caching. Companies such as Akamai [1] have developed content distribution networks that push data toward end nodes to reduce access latency. However, our focus on file data creates important differences. Consistency is a first-class concern for us: data staging preserves the consistency guarantees of the underlying distributed file system. This is necessary since file system clients are far more likely to modify data than Web clients. Additionally, we provide a mechanism for end-to-end encryption that avoids the need to trust unknown third parties. These issues also differentiate surrogates from caching Web proxies such as Squid [30].

WayStations in Fluid Replication [14] perform a role similar to that played by our surrogates. Yet, there are key differences. First, replicas on WayStations accept file modifications from clients. By transmitting modifications directly to file servers, data staging simplifies the trust model for surrogates. A second difference is that WayStations do not speculatively prefetch data, and thus may underperform in cold-cache scenarios.

OceanStore [17] provides floating replicas of data that can migrate to nearby servers. OceanStore partitions servers into those trusted to perform replication protocols and those not trusted to do so. While the untrusted servers may optimistically accept updates, the client must directly contact the trusted servers in order to ensure permanence and correctness. Thus, replicas on untrusted servers perform similar services to our surrogates. One of the primary differences is our focus on ease of management. We have taken care to place the absolute minimum of functionality on surrogates and have built on commodity software as much as possible—we believe that such steps are vital to ubiquitous surrogate deployment. Another difference is that surrogates are



This figure shows the benefits of data staging for a 64 MB client Coda cache when the network round-trip time is reduced to 30 ms. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 14: File trace results with 30 ms. round-trip time

file-system agnostic. Since all code specific to the underlying file system is isolated in the client proxy and data pump, a single surrogate may simultaneously service clients that employ diverse file systems.

We hope to build on two bodies of related work in the future. First, we hope to incorporate some of the automated prefetching algorithms that have been proposed for file systems [2, 11, 16, 18] and Web distributed cache placement [34]. Second, we plan to use surrogates to implement Infostations [37] that provide high-bandwidth access to data in mobile environments as described in Section 3.

Muntz and Honeyman [22] evaluated the use of intermediate caching for the AFS file system and found that it achieved little benefit. However, their evaluation environment is quite different from the target environment for data staging. Wide-area network latency imposes substantially greater penalties for cache misses.

7 Conclusion

Untrusted and unmanaged machines can facilitate mobile data access. Data staging uses nearby surrogates located in the pervasive computing environment to improve distributed file system performance for storage-limited clients. Clients borrow storage capacity from surrogates and use it as a second-level file cache to hide the latency of file operations.

Two important assumptions in our work are that surrogates are untrusted and unmanaged. Because surrogates are untrusted, we use end-to-end encryption to provide privacy, and secure hashes to ensure authenticity. We make surrogates as reliable and easy to manage as possible by maintaining no hard state on them, using commodity software, and pushing functionality from surrogates to client and server machines. We believe these design considerations will prove vital in ensuring the widespread deployment of a surrogate infrastructure.

8 Acknowledgments

We thank Jan Harkes for his insights into Coda file system behavior and Casey Helfrich for his assistance with setting up our experimental environment. We also thank our shepherd, Peter Honeyman, and the anonymous reviewers for their suggestions.

This research was partly supported by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center (SPAWAR) / U.S. Navy (USN) under contract N66001-99-28918. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either express or implied, of DARPA, SPAWAR, USN, Intel, the University of Michigan, Carnegie Mellon University, or the U.S. Government.

References

[1] AKAMAI CORPORATION. *http://www.akamai.com*.

[2] AMER, A., LONG, D. D., AND BURNS, R. C. Group-based management of distributed file caches. In *Proccedings of the 22nd International Conference on Distributed Computing Systems* (Vienna, Austria, July 2002), pp. 525–534.

[3] ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Information and control in gray-box systems. In *Proccedings of the 16th ACM Symp. on Operating Systems Principles* (Banff, Canada, October 2001), pp. 43–56.

[4] BALAN, R., FLINN, J., SATYANARAYANAN, M., SINNAMOHIDEEN, S., AND YANG, H.-I. The Case For Cyber Foraging. In the 10th ACM SIGOPS European Workshop, September 2002.

[5] CARSON, M. *Adaptation and Protocol Testing thorugh Network Emulation*. Internetworking Technologies Group, NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[6] CHANG, F., AND GIBSON, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symp. on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[7] DOUCEUR, J., AND BOLOSKY, W. A large-scale study of file-system contents. In *Proceedings of ACM SIGMETRICS* (Atlanta, GA, May 1999), pp. 59–70.

[8] FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of ACM SIGMETRICS* (Santa Clara, CA, June 2000).

[9] FREEMAN, E., AND GELERNTER, D. Lifestreams: A storage model for personal data. *SIGMOD Record 25*, 1 (1996).

[10] GARLAN, D., SIEWIOREK, D., SMAILAGIC, A., STEENKISTE, P. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing 1*, 2 (April-June 2002).

[11] GRIFFIOEN, J., AND APPLETON, R. Performance measurements of automatic prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems* (Sept. 1995).

[12] HAARSTEN, J. C. The Bluetooth radio system. *IEEE Personal Communications 7*, 1 (February 2000), 28–36.

[13] IEEE LOCAL AND METROPOLITAN AREA NETWORK STANDARDS COMMITTEE. *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications*. New York, New York, 1997.

[14] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (January 2002).

[15] KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[16] KROEGER, T., AND LONG, D. The case for efficient file access pattern modeling. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)* (Rio Rico, AZ, March 1999).

[17] 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 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), pp. 190–201.

[18] KUENNING, G., AND POPEK, G. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997).

[19] LEEPER, DAVID G. Wireless Data Blaster. *Scientific American* (May 2002).

[20] MICROSOFT CORPORATION. *Universal Plug and Play Forum*, June 1999. http://www.upnp.org.

[21] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995).

[22] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX* (January 1992), pp. 305–313.

[23] NARAYANASWAMI, C., KAMIJOH, N., RAGHUNATH, M., INOUE, T., CIPOLLA, T., SANFORD, J., SCHLIG, E., VENKITESWARAN, S., GUNIGUNTALA, D., KULKARNI, V., YAMAZAKI, K. IBM's Linux Watch: The Challenge of Miniaturization. *IEEE Computer 35*, 1 (January 2002).

[24] NOBLE, B., FLEIS, B., AND KIM, M. A case for fluid replication. In *Network Storage Symposium* (October 2000).

[25] PATTERSON, R., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995).

[26] SATYANARAYANAN, M. Caching trust rather than content. *Operating System Review 34*, 4 (October 2000).

[27] SATYANARAYANAN, M., EBLING, M. R., RAIFF, J., BRAAM, P. J., AND HARKES, J. *Coda File System User and System Administrators Manual*. Carnegie Mellon University, http://coda.cs.cmu.edu, 1995.

[28] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems 20*, 2 (May 2002).

[29] SOUSA, J., AND GARLAN, D. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'02)* (Karlsruhe, Germany, April 2002), pp. 7–20.

[30] *Squid Web Proxy Cache*. http://www.squid-cache.org/.

[31] STEMM, M., AND KATZ, R. H. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science 80*, 8 (August 1997), 1125–1131.

[32] SYSINTERNALS. *http://www.sysinternals.com*.

[33] VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. TCPNice: a mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 329–343.

[34] VENKATARAMANI, A., WEIDMANN, P., AND DAHLIN., M. Bandwidth constrained placement in a WAN. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing* (Aug. 2001).

[35] VOGELS, W. File system usage in Windows NT 4.0. In *Proccedings of the 15th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 93–109.

[36] WALDO, J. The Jini architecture for network-centric computing. *Communications of the ACM 42*, 7 (1999).

[37] WU, G., CHU, C.-W., WINE, K., EVANS, J., AND FRENKIEL, R. WINMAC: A novel transmission protocol for Infostations. In *Proceedings of IEEE VTC* (Houston, TX, 1999).