Using Content Addressable Techniques to Optimize Client-Server Systems

Niraj Tolia

October 2007

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
M. Satyanarayanan, Chair
Gregory R. Ganger
Peter Steenkiste
Antony Rowstron, Microsoft Research Cambridge

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the National Science Foundation (NSF) under grant numbers CCR-0205266 and CNS-0509004.

Keywords: Content Addressable Techniques, Content Addressable Storage, Network Optimizations, Bandwidth Optimizations, Portable Storage, Transfer Service, Distributed File Systems, Database Systems, Mobility

To my parents.

Abstract

Efficient access to bulk data over the Internet has become a critical problem in today's world. Even while bandwidth, both in the core and the edge of the network, is improving, the simultaneous growth in the use of digital media and large personal data sets is placing increasing demands on it. Further, with increasing trends towards mobility, an increasing amount of data access is over cellular and wireless networks. These trends are placing pressure on applications and the Wide-Area Network (WAN) to deliver better performance to the end user.

This dissertation puts forward the claim that external resources can be used in an opportunistic manner to optimize bulk data transfer in WAN-based client-server systems. In particular, it advocates the use of content-addressable techniques, a system-independent method for naming objects, to cleanly enable these optimizations. By detecting similarity between different data sources, these techniques allow external sources to be used without weakening any attributes, including consistency, of legacy systems and with no or minor changes to the original system.

This dissertation validates this claim empirically through the use of five case studies that encompass the two traditional forms of data storage, file systems and database systems, and then generalizes the claim in the form of a generic transfer service that can be shared by different applications. In each of these case studies, I focus on three questions. First, even with the addition of content-based optimizations, how does the resulting system still maintain the attributes and semantics of the original system? Second, how do these systems efficiently find similarity in data? Third, does the use of these content-addressable techniques provide a substantial performance improvement when compared to the original system? These questions are answered with a detailed description of the system design and implementation and a careful evaluation of the prototypes with both synthetic and real benchmarks.

Acknowledgments

This dissertation would not have been possible without the efforts of a large number of people who have supported me in my endeavors.

I am indebted to Satya, my advisor, for his help at every step of this journey. His guidance in how to approach research, his insight into the problems that were faced while working on this dissertation, and his constant optimism have been of invaluable help. His help in improving my writing and editing skills will stand me in good stead through the rest of my career. With an open door policy, Satya made sure that he was always available when I needed him. I consider it an honor to have worked with him. I am also thankful to the rest of my committee, Greg Ganger, Peter Steenkiste, and Ant Rowstron, for their help. Even with their busy schedules, they have always had time to listen, provide feedback, and give advice on this work.

I have been very fortunate to have worked with a number of brilliant people during my stay at Carnegie Mellon. In particular, I would like to thank Dave Andersen for his support, advice, and encouragement. I am also thankful for the opportunity to have collaborated with Michael Kozuch, Jason Flinn, Andrés Lagar-Cavilla, Jan Harkes, Michael Kaminsky, David O'Hallaron, and Eyal de Lara. Tracy Farbacher, Karen Lindenfelser, and Joan Digney have worked behind the scenes to make sure that I had all the support I needed to get my work done.

On a more personal note, special thanks go out to my family. My parents have worked hard to give me this opportunity and have always shown their belief in me. They, along with my uncles and aunts, have always been supportive of my decision to go to graduate school and have given me the chance to be where I am today.

Finally, I have to thank all the friends who have provided moral support during my graduate career at Carnegie Mellon. In particular, I am grateful for the company of Citlali Martinez, Rob Zlot, Aditya Ganjam, Sandra Martinez, Irina Shklovski, Ritesh Batra, Alice Zheng, Anubhav Gupta, Divyasheel Sharma and, of course, Eno Thereska, my roommate and colleague for over five years. They have been there for me in a world far away from home.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, users have an increasing number of options available for accessing the Internet. These choices include wired networks such as DSL, Fiber, and Cable, wireless networks such as 802.11 variants and WiMax, and cellular networks such EVDO, GPRS/EDGE, and UMTS. This increased availability has led users to expect access to their data anywhere and at any given point in time. However, even though the capacity of these networks has improved over the years, network usage has also grown in proportion. There are also large differences in the available bandwidth of these networks. For example, users will see a stark performance difference when they transition from a wired network to a cellular one. In addition, while the amount of per-capita manually generated content has remained relatively constant over the years, the growth in machine-generated data has exploded [2, 90]. Examples of such data include digital audio and video, health-related imaging, and increasing amounts of automatically generated data residing in relational databases.

These trends have placed increased pressure on applications and the Wide-Area Network (WAN) to deliver better performance to the end user. Even though the wired cores of most networks tend to be well-provisioned, the edges of the WAN or the "last-mile" access links frequently become a bottleneck. However, in a number of situations, there are alternatives to using the bottleneck link. For example, if sharing exists between different systems, users, or groups, the required data might be found on another user's machine. If the user has downloaded similar data in the past, a significant fraction of the required data might be found on the user's portable storage devices or the computer's hard drive and only the non-duplicated portions would need to be fetched over the WAN. External data sources can also prove to be very useful if the original sender is overloaded.

This dissertation describes how such external resources can be used in an opportunistic manner to optimize bulk data transfer in WAN-based client-server systems. In particular, it advocates the use of content-addressable techniques, a system-independent method of naming objects, to cleanly enable these optimizations. In this chapter, Section 1.1 first looks at why applications face difficulties in exploiting external data sources. Section 1.2 describes the thesis statement. Section 1.3 follows with a primer on content-addressable techniques. Section 1.4 provides a road map for the rest of the dissertation.

1.1 Identity in Data Transfer

Today, data objects are named by the system that stores them and these names are only guaranteed to be unique within that particular system. For example, a file within a file system is identified by a pathname. There is value in user-defined naming as the names can have mnemonic significance. However, files stored in different file systems can have identical pathnames but completely different contents. Thus, system or user-defined names are not guaranteed to be globally unique. Further, even these names are lost once files are copied out of the file system by applications. Consider the case when a user emails a file stored on her local file system to a colleague. The recipient might make minor edits, renames the file to reflect the edits, and send it back to the sender. While a large fraction of the content between the original and edited file might be similar, the sending and receiving machines have no obvious method to detect the link between the two files and detect similarity. Thus the file will be need to be transferred over the Internet in its entirety.

Securely assigning global names to objects usually requires centralization or complex distributed coordination mechanisms. Implementing either of these solutions would create new scalability, security, and performance challenges. The absence of global names prevents out-of-band optimizations of bulk data transfers. For example, during actual transfers, objects might be defined only as a part of the protocol's control flow. As a result, data transfers become very tightly coupled to the application's control channel and sometimes to the particular TCP/IP connection the control channel is operating on. This prevents optimizations such as the use of parallel TCP over high bandwidth-delay networks, using multi-path when multiple network interfaces are available, or even the use of portable storage.

In theory, these techniques as well as the various data transfer optimizations shown in Figure 1.1 could be individually incorporated within individual applications. However, this requires significant developer effort, access to the application's source code, and the ability to modify the standards that define the application's protocol. While application-specific edge caches can ameliorate this problem to a certain extent, they are not generic solutions and require a deep understanding of every application protocol being optimized. Further, even application-specific optimizations will be unable to make opportunistic use of all resources available in the environment. For example, if two users in the same office receive the same file via separate transfer mechanisms such as email (over the SMTP protocol) and the web (over the HTTP protocol), they will fetch two copies over the WAN. Exploiting secondary data sources such as portable storage is even harder.

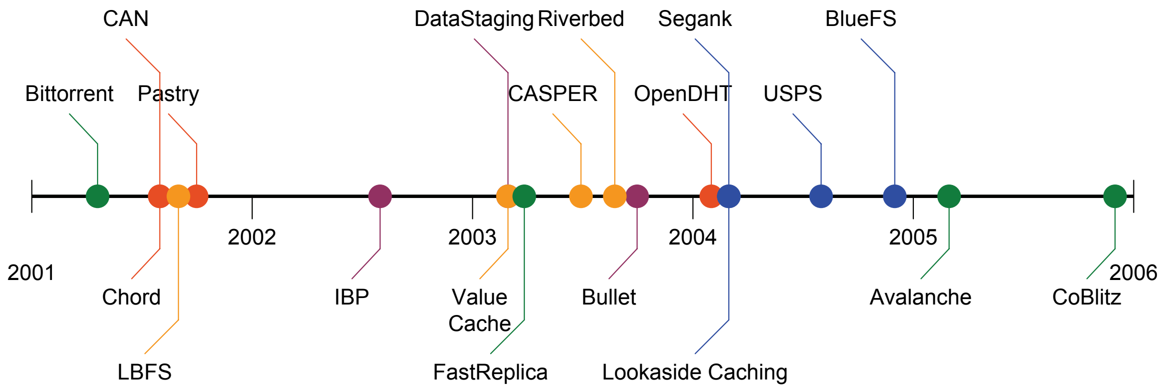
1.2 Thesis Statement

This thesis states that:

Using Content-Addressable Techniques to identify bulk data can significantly improve performance for both legacy and future client-server systems. These techniques allow external data sources to be used without weakening any attributes, including consistency, of legacy systems and with no or minor changes to the original system.

1.3 Using Content-Addressable Techniques

In order to uniquely identify data, this dissertation advocates the use of content-addressable techniques. Unlike conventional naming methods, these techniques use cryptographic hashes to name data in a unique and self-identifying manner without requiring a central authority or global communication. As these names are self-verifying,



The technologies illustrated above include P2P systems [43, 44, 61, 101], Distributed Hash Tables (DHTs) [112, 115, 120, 140], CAS-based systems [91, 114, 117, 149], portable storage based file systems [96, 136, 150, 160], and optimizations based on in-network storage [19, 58, 75].

Figure 1.1: Examples of Different Techniques for Improving Bulk Data Transfer

they allow external data sources and a variety of different transfer techniques to be used in an opportunistic manner to improve transfer speeds.

The use of content-addressable techniques allows systems to leverage the client's increasingly powerful CPUs and increasingly large storage capacities to compensate for weak connectivity. In comparison to the cost of network transfers, modern CPUs reduce the cost of generating cryptographic hashes to a negligible amount. Large local storage capacities allow systems to cache previous transfers and store substantial amounts of data for opportunistic use without impacting normal usage.

The cryptographic hash functions used by these techniques map any arbitrary finite length input to a fixed length output (the hash value). Further, for any given input, the hash value is easy to compute. While more details are provided by Menezes et al. [85], these functions generally have the following properties:

- **One-Way:** Given an output, it is computationally infeasible to find any input that hashes to it.
- **Weak Collision Resistance:** Given an input, it is computationally infeasible to find a second input that has the same hash output.
- **Strong Collision Resistance:** It is computationally infeasible to find any two distinct inputs that hash to the same output.

The work described in the following chapters assumes that the original source of data is trusted. The source is depended upon to provide the hash values that were generated from the object to be transferred. The "weak" collision resistance property allows this work to assume that the given hash value can serve as a unique identifier for that input. Therefore, if two hash values are equal, so are the source objects they were generated from. This property has the advantage that it breaks a client's dependence on the original data source. Further, these content-based hashes are also self-verifying. Any data received from an untrusted source can simply be rehashed and the new hash value compared to the one fetched from the trusted source. As it is computationally infeasible to find a second input that has the same hash output, if the original and new hash values match, data authenticity is assured.

Although concerns about the collision-resistance assumption of CAS have been expressed [67], this dissertation believes that they are not an issue because of the reasons outlined in the rebuttal by Black [22]. Collisions for cryptographic hash functions have an extremely low probability. For a 160-bit hash function, the probability of hash collision is 2^{-160}. As the trusted source generates the hash values, this work is immune to birthday attacks. Even with a birthday attack, in terms of computational effort, the number of operations required to generate a hash collision would be 2^{-80}. Further, this dissertation does not depend critically on any specific hash function. Replacing a given hash function used with a different one would be simple. No re-hashing of permanent storage would be required since hash functions are only used on volatile data and never for addressing persistent data. While a much stronger function such as SHA-256 [129] would increase computational effort at the client and server, the increased overhead is very modest and the length of the hash values would still be much smaller than the objects they represent.

1.4 Dissertation Road Map

I believe that the use of content-addressable techniques is a simple yet powerful optimization that can significantly improve performance. To validate the thesis, this dissertation will empirically demonstrate how these techniques apply to a broad category of systems. In particular, it will examine the two predominant and widely-used forms of high-level storage: distributed file systems and databases. Building upon this work with legacy storage systems, I will then generalize the lessons learned and propose a new architecture for future systems and applications that perform Internet data transfers.

In all of these systems, content-addressable techniques will be used in an opportunistic manner and solely as a performance improvement. No assumptions are made as to the existence of similarity and, for the targeted environments, its absence should provide performance that is equivalent to the original system.

When describing these systems, the focus will be on the following design aspects:

- **Achieving Transparency:** Even with the addition of content-based optimizations, how does the resulting system still maintain the attributes and semantics of the original system? Two forms of transparency are explored. The first allows the introduction of these optimizations without any modifications required to either the legacy system or applications. The second allows minor modifications to the legacy system but requires that no application modifications be made.

- **Detecting Similarity:** How do the new systems efficiently find similarity in data? Where does this similarity arise from and how is it used? Does detecting similarity involve the use of domain-specific information or can it be generic?

- **Performance Improvement:** What are the potential benefits from using content-addressable techniques and what workloads might benefit from these systems? Further, what are the potential overheads of using these systems when no similarity is detected by the system or when operating in environments they are not targeted towards?

First, in Chapter 2, I demonstrate the ease with which external networked data repositories can be opportunistically used to optimize the performance of the CASPER distributed file system [149]. These external repositories can be set up completely independently of the original distributed file system. Further, no explicit cache-coherence protocol is required to maintain consistency as the repositories are addressed using content identifiers fetched from the authoritative file server. If updates occur on the server, any stale data in the repository will simply no longer be referenced.

I then extend these ideas in Chapter 3 to leverage non-networked data sources, such as portable storage, through the use of *Lookaside Caching* [150]. Lookaside caching, a powerful and versatile technique, uses CAS to combine the strengths of distributed file systems and portable storage devices, while negating their weaknesses. By unifying distributed storage and portable storage into a single abstraction, it allows users to treat devices they carry as merely performance and availability assists for distant file servers. Careless use of portable storage has no catastrophic consequences.

Both the CASPER and lookaside caching systems optimize the performance of distributed file systems. When objects change, it is usually due to in-place updates, deletions, and insertions. These systems are also characterized by relatively large opaque objects. Further, as consistency is usually maintained at a per-object level, injecting CAS-based optimizations was also relatively straightforward. The question arises as to whether the thesis statement holds for systems that have different update patterns, smaller objects, and stricter consistency requirements. In answer, this dissertation next discusses the thesis in the context of database systems.

Chapter 4 examines how CAS can be useful in multi-tiered architectures where dynamic web content is generated from relational databases. I describe a middleware layer called Ganesh [148] that reduces the volume of data transmitted without semantic interpretation of queries or results. It achieves this reduction through the use of cryptographic hashing to detect similarities with previous results. These benefits do not require any compromise of the strict consistency semantics provided by the back-end database. Further, Ganesh does not require modifications to applications, web servers, or database servers, and works with closed-source applications and databases.

Chapter 5 then examines how CAS can be useful in exploiting stale data replicas. It presents Cedar [152], a system that enables mobile database access with good performance over low-bandwidth networks. This is accomplished without degrading consistency. Cedar exploits the disk storage and processing power of a mobile client to compensate for weak connectivity. Its central organizing principle is that even a stale client replica can be used to reduce data transmission volume from a database server. The reduction is achieved by using content addressable storage to discover and elide commonality between client and server results. This organizing principle allows Cedar to use an optimistic approach to solving the difficult problem of database replica control.

Finally, I generalize the lessons learned from creating the above systems in the creation of DOT, a generic *transfer service* [151]. DOT, described in Chapter 6, specifies a flexible architecture for Internet data transfer. Through the use of CAS, this architecture allows a clean separation of content negotiation from the data transfer. Applications determine *what* data they need to send and then use a transfer service to send it. This transfer service acts as a common interface between applications and the lower-level network layers, facilitating innovation both above and below. The transfer service frees developers from re-inventing transfer mechanisms in each new application. New transfer mechanisms, in turn, can be easily deployed without modifying existing applications.

Related Work is discussed in Chapter 7. Chapter 8 concludes with a summary of the contributions of this dissertation and outlines possible areas for continuing this work.

Chapter 2

Opportunistic Use of Networked Storage for Distributed File Systems

2.1 Introduction

This chapter shows how networked data sources can be opportunistically used to improve the performance of legacy distributed systems. In particular, it examines the use of POSIX-style distributed file systems such as NFS, AFS, Coda, and CIFS. However, there has always been a significant gap between the user experience on a WAN versus a much faster Local Area Network (LAN). As WAN links are usually characterized by low bandwidth and high latency, one of the main performance bottlenecks is the cost of fetching large amounts of data over a WAN link. While caching can alleviate this problem to some extent, these techniques do not help in retrieving updates or new data. In particular, this chapter examines how the performance of legacy distributed file systems can be improved by using Content Addressable Storage (CAS) but without weakening any attributes of the original system.

The use of CAS in this scenario is totally opportunistic. The file system does not depend on CAS providers for correct working behavior and does not change the naming, consistency, and write-sharing semantics associated with the file system. CAS providers are only used to accelerate read performance and the central file server is the sole authority with respect to file content, access control, naming, etc. In fact, the use of CAS is completely transparent to both users and applications. The failure or absence of CAS providers only affects performance with files having to be fetched across the WAN.

Central to the integration of distributed file systems and CAS is the concept of a *recipe*. A recipe for a file is a synopsis of the data in terms of discrete blocks with each block being identified by a cryptographic hash of the contents in the block. Thus, given the data blocks from a CAS provider, one can reconstruct the file corresponding to the recipe description. In this way, recipes would be very useful for low-bandwidth networks where a client, only able to fetch the recipe across the WAN, could then use available CAS providers on a LAN to reconstruct the file. As a recipe's content-addressable description is of finer granularity than a whole-file hash, if only parts of files are found on a CAS provider close to the client, the rest can be fetched across the WAN. The concept of using a recipe will also be used in the other systems describe in this dissertation.

There are a number of different entities that can serve as CAS providers. CAS providers can be organized peer-to-peer networks, such as Chord [140] and Pastry [120]. In some environments, a system administrator may prefer to dedicate one or more machines as CAS providers. Such a dedicated CAS provider, such as EMC's Centera [55], is referred to in this work as a *jukebox*. CAS providers may also be impromptu systems. For example, each desktop on a LAN could be enhanced with a daemon that provides a CAS interface to its local disk. With such an arrangement, system administrators can offer CAS access without being forced to abide by any peer-to-peer protocol or provide additional storage space. The CAS interface used by jukeboxes can be extremely simple; just two calls, explained in more detail in Section 2.3, `(Multi)Query` and `(Multi)Fetch` are used. In particular, CAS providers need not make any guarantees regarding content persistence or availability.

The prototype distributed file system that has been implemented using the recipe paradigm is called CASPER. The novel aspect of CASPER is that clients make opportunistic use of nearby CAS providers to improve performance and scalability. The functionality of the system has been tested by evaluating it on a set of standard benchmarks at different bandwidths and latencies. The results from these experiments show that substantial performance improvement is possible by using recipes in low-bandwidth and high latency scenarios.

A detailed explanation of recipes and their utility in detecting similarity is presented in Section 2.2. The CASPER approach to preserving transparency to the legacy distributed file system, applications, and users is then described in Section 2.3. This is followed by CASPER's performance evaluation in Section 2.4. Section 2.5 concludes with a summary of this chapter's findings.

2.2 Detecting Similarity

This section first describes recipes, a critical component in CASPER's approach to detecting similarity. Next, it briefly examines the amount of similarity the CASPER file system might find among different workloads.

2.2.1 Recipes

Recipes form the central core of the CASPER file system. A recipe for a file is a description of each constituent content addressable block belonging to the file. Examples of these descriptions include breaking up a file into fixed size 4 KB blocks or into variable sized chunks based on some deterministic algorithm. Each block of the file is identified in the recipe by a cryptographic hash such as SHA-1 [128], SHA-256 [129], etc. Thus, given a file's recipe, the file can be reconstructed by fetching the data with the same hash signature and combining the individual blocks in the order specified by the recipe.

Recipes allow multiple descriptions of a file to be present in the same recipe. This allows the users of the recipe to pick whatever representation is best suited or choose a combination of them to get the best possible performance. For portability reasons, recipes are described in XML (Extensible Markup Language) [26]. An example of a recipe is given in Figure 2.1. The recipe is representative of a file named `casper.pl` that is 125,637 bytes long. Apart from the hashes, additional metadata such as the last modified time and other file system specific information has been inserted at the beginning of the recipe. This inserted metadata is specific to CASPER and allows it to maintain recipe consistency and detect errors. Naturally, this section can be tailored to the needs of other applications that might choose to use recipes.

The metadata section is followed by three different descriptions, or *recipe choices*,

```
<?xml version="1.0"?>
<recipe type="file">
  <metadata>
    <length>125637</length>
    <last_modified>11/12/2002 16:24:37</last_modified>
    <file_system type="Coda">
      <name>/coda/projects/shared/casper.pl</name>
      <fid>312567 0 678</fid>
      <version>6 7 3 4 9</version>
    </file_system>
  </metadata>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="fixed" fixed_size="4096"
               number="31">
      <hash>09d2af8dd22...</hash>
      <hash>e5fa44f2b31...</hash>
      .
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="SHA-1" block_type="variable" number="36">
      <hash size="3576">7448d8798a4...</hash>
      <hash size="1278">a3db5c13ff9...</hash>
      .
      .
    </hash_list>
  </recipe_choice>

  <recipe_choice>
    <hash_list hash_type="SHA-256" block_type="fixed" fixed_size="125637"
               number="1">
      <hash>9c6b057a2b9...</hash>
    </hash_list>
  </recipe_choice>
</recipe>
```

Figure 2.1: Example of a File Recipe

that represent the contents of the file. The first choice (lines 12-20) displays a list of SHA-1 hashes describing 4 KB fixed size blocks. In fact, as the 31 4 KB blocks describe an object slightly greater than the length defined in the metadata section, the reconstructed object will have to be truncated to the correct length. The next description (lines 22-29) is a list of SHA-1 hashes of variable-length blocks. Such variable sized blocks could be generated, for example, by employing Rabin fingerprints [108, 137] to find block boundaries. This description again describes the entire file. The third and final example (lines 31-36) of a description is a single SHA-256 hash of the entire file. As this is a special case of variable sized blocks, the hash could be used to request the entire file as a single block of data. If any of the previous recipe choices are used to reconstruct the file, the single hash can also be used to ensure integrity.

The different choices described above allow the user some flexibility in selecting between CAS providers. For example, if the only available CAS provider exports fixed-sized 4 KB blocks, the user could select the first choice. On the other hand, if different CAS providers are available that provide both fixed as well as variable sized blocks, a user could potentially use one of them for the majority of the requests and use the others to satisfy requests for data that wasn't found in the first.

CASPER currently supports recipes containing variable-sized blocks discovered using Rabin fingerprinting, fixed-sized blocks, and whole file-hashes. However, it predominantly uses Rabin fingerprinting because of its effectiveness in finding similarity in opaque objects.

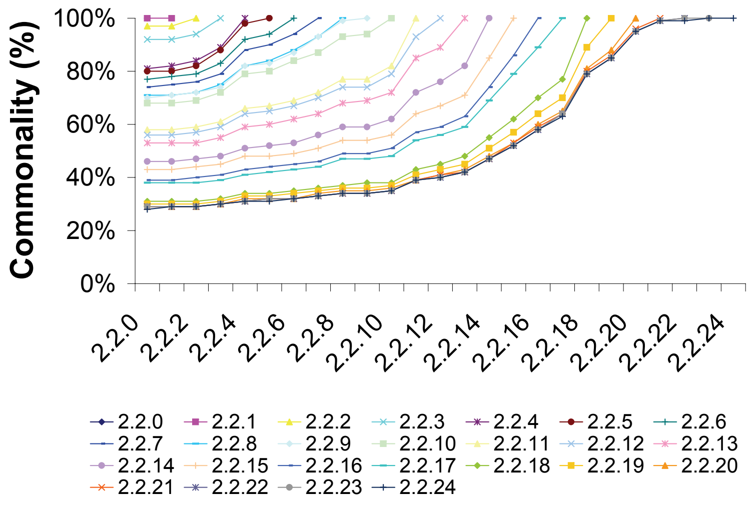
2.2.2 Commonality Measurements

The performance improvement seen from using CASPER is highly dependent on the commonality in data between the file server and the CAS providers. Further, the degree of commonality is expected to depend on the applications that are trying to exploit it. Some applications, like the Virtual Machine (VM) migration benchmark used in the evaluation, would be well-suited to a CAS-based approach. This benchmark reconstructs the data, including common operating system and application code, found on typical disk drives. A detailed examination of usage data from a VM-based client management system [94] along with prior work [24, 46] showed that a high degree of commonality can be expected for this application.

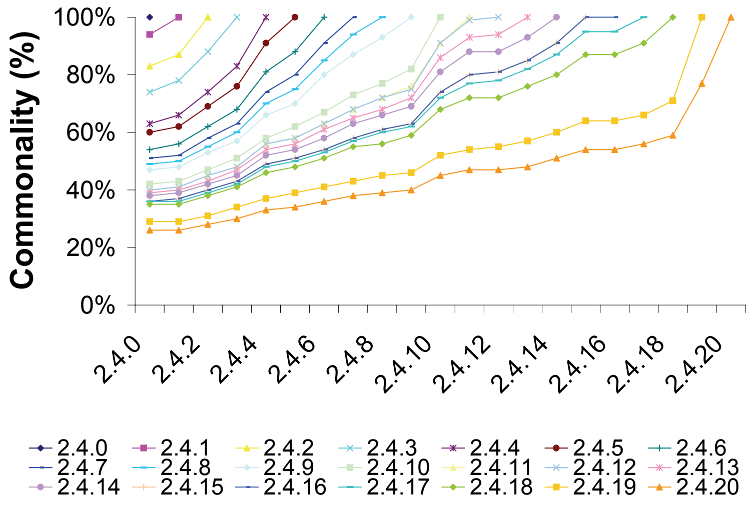
Commonality may also be expected when retrieving common binary or source-code distributions. For a software developer, different versions of a source tree seem to exhibit this trend. For example, the commonality between various releases of the Linux kernel source code was compared. Figure 2.2 shows the measured commonality between versions of the Linux 2.2 and 2.4 kernels. For this study, the definition of commonality is the fraction of unique variable-sized blocks in common between pairs of releases. The block-extraction algorithm used in CASPER was applied to each kernel's source tree and the unique hashes of all blocks in the tree were collected. The set of hashes derived from each release was then compared to the set of hashes derived from each previous release in its series. The results show that commonality of 60% is not uncommon, even when the compared source trees are several versions apart. The minimal commonality observed is approximately 30%. As shown later in Section 2.4.4 even this degree of commonality can lead to significant performance benefits from CAS in low-bandwidth conditions. The commonality differences shown between different releases can be attributed to both in-situ changes in code as well as the addition of new code.

As small changes accumulate in source code over time, the corresponding compiled binary will diverge progressively more in content from earlier binaries. To assess how commonality in binaries changes as small source-level changes accumulate, the nightly Mozilla binary releases from March 16th, 2003 to March 25th, 2003 were examined. The measurements from this analysis are presented in Figure 2.3. I believe that these changes are representative of the effect of applying security or performance patches to an application. Nightly binaries, on average, share 61% of their content in common with the binary of the preceding revision. In the worst case, where the CAS jukebox only has the binary from March 16th, but the client desires the binary from the 25th, a commonality of 42% is observed. The same analysis on major Mozilla releases observed significantly less commonality for those binaries because of the significant changes in code and functionality between releases. One interesting exception concerned the 1.2.1 release, a security fix; this release had 91% commonality with the previous one.

While these measurements of cross-revision commonality for both source code and binaries are promising, they are somewhat domain-specific. The degree of commonality in users' files bears future investigation. Highly personalized or sensitive data is unlikely to be shared between multiple users. But there may be cases where users



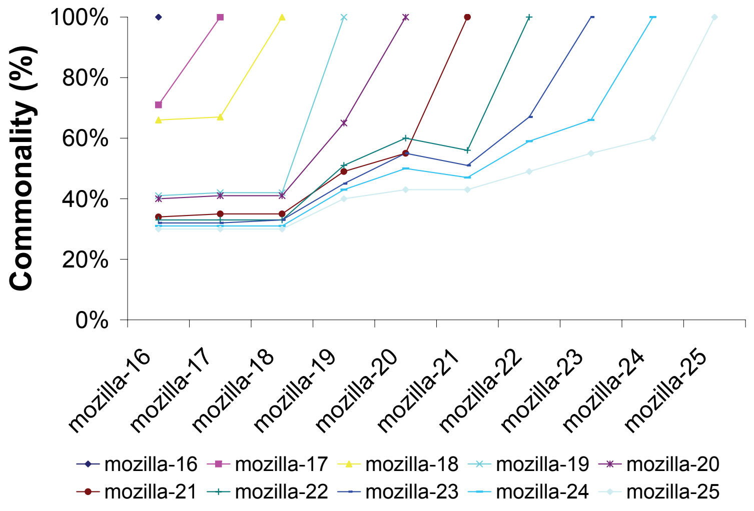
(a) Linux 2.2 Kernel - Commonality



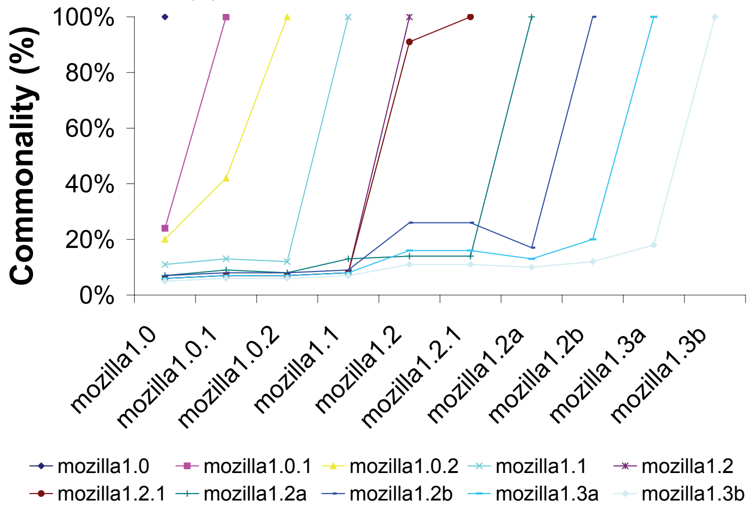
(b) Linux 2.4 Kernel - Commonality

For the above graphs, each data series represents the measured commonality between a reference version of the software package and all previous releases of that package. Each point in the data series represents the percentage of data blocks from the reference version that occur in the previous release. The horizontal axis shows the set of possible previous releases, and the vertical axis relates the percentage of blocks in common. Each data series peaks at 100% when compared with itself.

Figure 2.2: Linux Kernel Commonality



(a) Mozilla Nightly Binaries



(b) Mozilla Release Binaries

For the above graphs, each data series represents the measured commonality between a reference version of the software package and all previous releases of that package. Each point in the data series represents the percentage of data blocks from the reference version that occur in the previous release. The horizontal axis shows the set of possible previous releases, and the vertical axis relates the percentage of blocks in common. Each data series peaks at 100% when compared with itself.

Figure 2.3: Mozilla Commonality

share data, such as email attachments sent to many other users in the same organization. While a detailed study of cross-user commonality is beyond the scope of this work, its existence has been quantified by others [23, 77, 104]. Further, as shown in Section 2.4, CASPER offers performance improvements over low-bandwidth connections even in the presence of relatively little commonality. Moreover, on WANs, the overhead of not finding the required data on CAS providers is small.

2.3 Achieving Transparency

The CASPER prototype is derived from the Coda distributed file system [73, 124] and therefore caches data at a whole-file granularity to provide Coda's file-session oriented, open-close consistency model. To maintain transparency to the legacy Coda file system, applications, and users, CASPER has adopted a modular, proxy-based layering approach. As CASPER is solely a performance optimization, it relies on the Coda file servers to guarantee data persistence and file consistency.

Figure 2.4 depicts the organization of the CASPER system. The Coda client and Coda server modules are unmodified releases of the Coda client and server, respectively. The Proxy module is responsible for intercepting communication between the client and server and determining whether or not to activate CASPER's CAS functionality. The Coda client and proxy together act as a CASPER client. Likewise, the Coda file server and Recipe Server together act as a CASPER server. The recipe server is the component responsible for generating replies to recipe requests.

The proxy-based design enabled the prototyping of new file-system features such as the CAS-based performance enhancements without modifying Coda. In this design, the proxy provides the client with an interface identical to the Coda server interface. The proxy can monitor network conditions and determine when to use an available CAS provider. Under good networking conditions, the CASPER system behaves identically to an unmodified Coda installation without the proxy. After detecting low bandwidth network conditions, however, the proxy registers with the recipe server and a CAS provider. The provider in this example is a jukebox. As explained earlier, a jukebox is a dedicated server that acts as a CAS provider.

While low-bandwidth conditions persist, the proxy intercepts all file requests from the client and asks for the corresponding recipe from the recipe server. The recipe

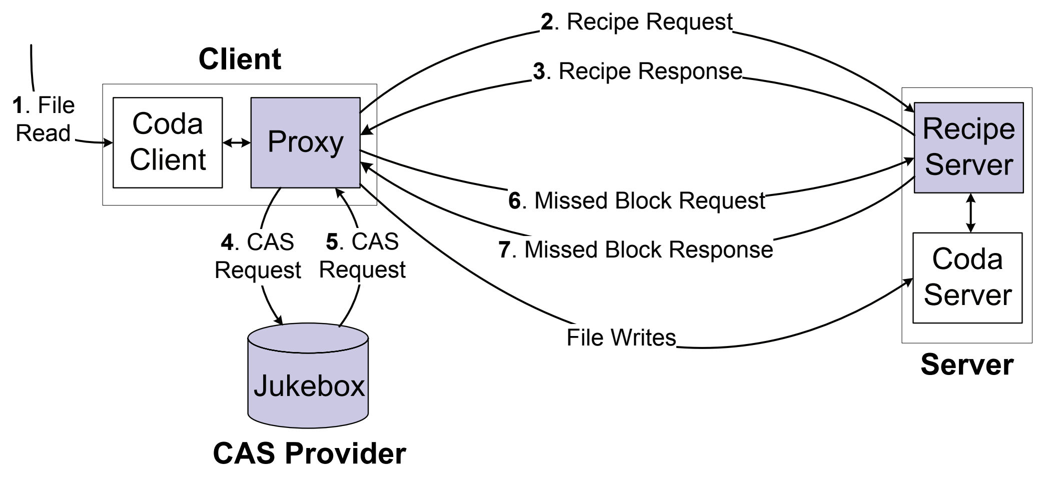


Figure 2.4: CASPER System Diagram

server is responsible for generating a recipe for the current version of the file and delivering it to the proxy. Using the hashes contained in the recipe, the proxy then attempts to retrieve the data blocks named in the file from nearby CAS providers (including the client's own file cache). The proxy will then request any blocks not found on the CAS providers from the recipe server. Once reconstruction is complete, the file is passed back to the Coda client, which places the file in its file cache. No other traffic, such as writes, is currently intercepted by the proxy; it is passed directly to the file server.

The CASPER file system is primarily concerned with client reads. However, the current implementation could easily be extended to accommodate client write operations by adopting a similar local cache lookup mechanism on the server-side. To leverage the recipe-based mechanism, client writes can be viewed as server reads of the modified file. When sending file modifications to the server, a client would send a recipe of the modified file rather than the file contents. The server would then peruse its own data for instances of the blocks, then request blocks that are not found locally from nearby CAS providers, and finally retrieve any remaining blocks from the client directly.

The next three sections describe the design and implementation of the recipe server, proxy, and jukebox in more detail.

Recipe Server

As the name indicates, the recipe server generates the recipe representations of files. It is a specialized user process that accesses file-system information through the standard file-system interface. The implementation maintains generated recipes as files in the CASPER file system. For user files in a directory, `zeta`, the recipe server stores the corresponding recipes in a sub-directory of `zeta` (e.g. `zeta/.shadow/`). As recipes in CASPER store version information, the consistency between files and their recipes can be managed in a lazy fashion.

In Figure 2.4, the recipe server is shown to be co-located with the Coda server. However, any machine that is well-connected to the server may act as the recipe server. In fact, if a user has a LAN-connected workstation, that workstation may make an excellent choice for the location of the user's primary recipe server, since it is the most likely machine to have a warm client cache.

When a recipe request arrives at the recipe server, the recipe server first determines if a recipe file corresponding to the request exists in the file system. If so, the recipe is read, either from the recipe server's cache or from the Coda file server, and checked for consistency. That is, the version information in the recipe is compared to the current version of the file. If the recipe is stale or does not exist, the recipe server will generate a new one, send it to the proxy, and store it in the shadow directory for potential reuse. Because CASPER is targeted towards the WAN, the XML recipes are compressed using the `bzip2` algorithm (based on the Burrows-Wheeler transform [30]) before being sent to the proxy.

Small files often will not benefit from CAS because of the recipe metadata and network overhead costs. Consequently, CASPER does not employ the recipe mechanism when transferring small files. Instead, the recipe server responds to requests for recipes of small files with the file's contents rather than the file's recipe. The threshold for small files in the current implementation is 4 KB. The primary reason for this special handling is that each recipe has a fixed overhead due to the metadata information stored in it. The penalty for transferring a recipe almost as large as the file and then encountering a miss on CAS providers can be very high on low bandwidth networks. This, combined with the overhead of consistency checks, compression, and extra network transmissions can make the cost of sending recipes for small files prohibitive.

Data blocks that are not found on any of the queried CAS providers are referred to as *missed blocks*. In the current implementation, the proxy fetches missed blocks from the recipe server. Even if the recipe server does not have the file cached, it can request it from the Coda server, which is guaranteed to contain the authoritative copy of the file. To service missed blocks, the recipe server supports requests for byte extents. That is, a missed block request is of the form `fetch(file, offset, length)`. To reduce the number of transmitted requests, missed blocks that are adjacent in the file are combined into a single extent request by the proxy, and multiple fetch requests are combined into a single multi-fetch message. In alternative implementations, missed block requests could be serviced by the file server. The recipe server was chosen instead to reduce the file server load in installations where the recipe server is not co-located with the file server.

The recipe server also helps to maintain consistency by forwarding callbacks from the file server to the proxy for files that the proxy has reconstructed via recipes. That is, if the recipe server receives an invalidation callback for a file after sending a recipe for the file to a client, the callback is forwarded to the requesting client via the proxy. Because the recipe server is a user process and not a Coda client, the recipe server does not receive the consistency callbacks directly. Instead, the recipe server communicates with the Coda client module through *codacon* [127], Coda's standard socket-based interface, which exports callback messages. Consequently, implementation of the callback-forwarding mechanism does not require modification of the client or server. Mechanisms that gather similar information, such as the SysInternals' Filemon tool [141], can potentially serve the same function should CASPER be ported to network file systems that do not provide similar interfaces.

Proxy

The proxy is the entity that transparently adds CAS-based file reconstruction to the Coda distributed file systems without modifying the file system code. This arrangement is not necessary for the operation of CASPER but eases the implementation of the prototype. The two main tasks that it performs are fetching recipes and reconstructing files.

All client cache misses induce file fetch requests. Because the proxy acts as the server for the client, the proxy intercepts these operations and sends a request for the

```
Retval Query(in Hash h, out Bool b);
Retval MultiQuery(in Hash harray[], out Bool barray[]);
Retval Fetch(in/out Hash h, out Bool b, out Block blk);
Retval MultiFetch(in/out Hash harray[], out Bool barray[],
                        out Block blks[]);
```

The `Retval` is an enumerated type indicating whether or not the operation succeeded.

Table 2.1: Summary of the CAS API

file's recipe to the recipe server; other file system operations are forwarded directly to the Coda server. Assuming that a recipe fetch request is successful, the proxy compares the version in the received recipe to the expected version of the file. If the recipe version is more up-to-date than the proxy's expected version number, the proxy contacts the Coda server to confirm the file's version number. If the recipe version is older, there might be a problem with the recipe server (e.g. the recipe server may have become disconnected from the Coda server), and therefore, the file request is redirected to the file server. The request is also redirected to the file server if any unexpected recipe server errors occur.

If the version numbers match, the proxy selects one of the available recipe choices in the recipe file, and then proceeds to reconstruct the file. After interpreting the XML data it received, the proxy queries both the client's local cache as well as any nearby jukeboxes for the hashes present in the recipe. Matching blocks, if any, are fetched from the jukeboxes, and the remaining blocks are fetched from the recipe server. Finally, the proxy assembles all constituent pieces and returns the resulting file to the client.

Content Addressable Storage API

The CAS API, summarized in Table 2.1, defines the interface between CAS consumers and CAS providers. The core operation is the retrieval of a file block identified by its hash value through the `Fetch` call. The consumer specifies the `hash` of the object to be retrieved (the hash data structure includes the hash type, block size, and hash value). `Fetch` returns a boolean value indicating if the block was found and, if so, the block itself. A consumer may also make a single request for multiple blocks through the `MultiFetch` function, specifying an array of hashes. This function returns an array of booleans values and a set of data blocks. The CAS API also provides associated `Query` and `MultiQuery` operations for inquiring after the presence of a block without allocating the buffer space or network bandwidth to have the block returned.

Jukebox

To evaluate the CASPER prototype, a jukebox CAS provider was implemented that conforms to the API described in Table 2.1. The consumer-provider (proxy-jukebox) communication that supports the `Query` and `Fetch` functions is currently implemented by using a lightweight RPC protocol. The jukebox also creates an in-memory index of the data, keyed by hash, at startup to provide an efficient lookup mechanism.

The jukebox, a Linux-based workstation, uses the native `ext3` file system to store the CAS data blocks. Currently, the system administrator determines which set of files are to be exported as CAS objects. The initial version of the jukebox had the file name of each hashed data block as its SHA-1 hash rendered in ASCII. As each block had an average size of 4 KB, this involved splitting most files into its constituent blocks. However, given the the uniform randomness of the hashes and the organization of the blocks on the file system as a B-tree, the data corresponding to each file was scattered all over the disk. This destroyed locality and required a large number of disk seeks to reconstruct a file. However, as recipes already index files in a content addressable way, recipes for the files were used instead of splitting them. This solution allows the system to get sequential read performance for block requests from the same file. The use of recipes also allows multiple content addressable representations of a file while keeping around only one copy of the actual data.

2.4 Performance Improvement

Three different benchmarks were used to evaluate the CASPER file system: a Mozilla software installation, an execution trace of a virtual machine application, and a modified Andrew Benchmark. The descriptions of these benchmarks together with experimental results is presented in this section.

2.4.1 Experimental Methodology

The performance improvement seen due to the use of CASPER depends on the sensitivity of the system performance to bandwidth on the WAN link as well as the amount of CAS data it found had to be examined. Towards this goal, three different benchmarks were used to measure the performance benefit of CAS-based file reconstruction. Sensitivity analysis was used to evaluate each benchmark under several combinations of network bandwidth, network latency, and jukebox hit-ratio.

The experimental infrastructure, shown in Figure 2.5 consisted of a number of single-processor machines. The jukebox and client were workstations with 2.0 GHz Pentium 4 processors. While the client had 512 MB of SDRAM, the jukebox had 1 GB. The file server contained a 1.2 GHz Pentium III Xeon processor and 1 GB of SDRAM.

The jukebox and client ran the Red Hat 7.3 Linux distribution with the 2.4.18-3 kernel, and the file server ran the Red Hat 7.2 Linux distribution with the 2.4.18 kernel. All machines ran version 5.3.19 of Coda with a large enough disk cache on the client to prevent eviction during the experiments. The recipe server process was co-located with the file server. To discount the effect of cold I/O buffer caches, one trial of each experiment was executed before taking any measurements. However, it was ensured that the Coda client cache was cold for all experiments.

To model different bandwidths and latencies, the NISTNet [36] network emulator (version 2.0.12) was used. The client was connected to the jukebox via 100 Mbit/s Ethernet, but, as shown in Figure 2.5, the client's connection to the server was controlled via NISTNet. All benchmarks ran at three different client-server bandwidths: 10 Mbit/s, 1 Mbit/s, and 100 Kbit/s. No results are reported for a 100 Mbit/s client-server connection, because CASPER clients should fetch data from the server directly when the client-server bandwidth is equal to, or better than, the client-jukebox bandwidth. Extra latencies of 10 ms and 100 ms were introduced for the 1 Mbit/s and 100 Kbit/s cases respectively. No extra latency was introduced for the 10 Mbit/s case.

The effectiveness of CAS-based reconstruction in CASPER is highly dependent on the fraction of data blocks the client can locate on nearby CAS providers. Sensitivity analysis was used to explore the effect of the CAS provider hit rate on CASPER performance. For each of the experiment and bandwidth cases, the jukebox was pop-

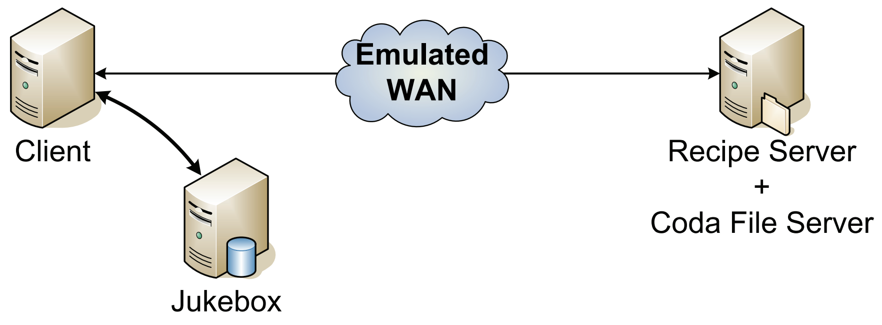


Figure 2.5: Experimental Setup

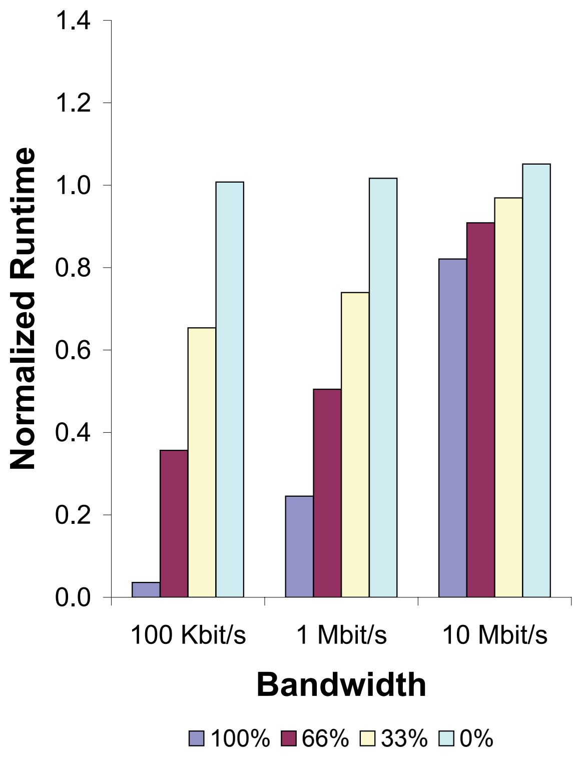
ulated with various fractions of the requested data: 100%, 66%, 33%, and 0%. That is, in the 66% case, the jukebox is explicitly populated by randomly selecting 66% of the data that would be requested during the execution of the benchmark. While the requisite percentage in the above cases was met by selecting the required count of hashes, this also equated to a similar percentage of data because each individual hash, on average, represented 4 KB of data. The two extreme hit-ratios of 100% and 0% give an indication of best-case performance and the system overhead. The baseline for comparison is the execution of the benchmarks with an unmodified Coda client and server.

The recipes employed by CASPER for these experiments included a single type of recipe choice: namely, SHA-1 hashes of variable-size blocks with an average block size of 4 KB. During the experiments, none of the recipes were cached on the client. Every client cache miss generated a recipe fetch for the file. Further, because the client cache was big enough to prevent eviction of any reconstructed files, the client never requested a file or a recipe more than once.

Also, to accurately measure the performance gain from using CAS on low-bandwidth connections, write traffic was prevented from competing with read traffic. To achieve this, all experiments were run with the client in write-disconnected mode.

2.4.2 Mozilla Install

Software distribution is one application for which CASPER was proven effective. Often when installing or upgrading a software package, a previous version of the package is found on the target machine or another machine near the target; the previous version may provide a wealth of matching blocks. To evaluate software



Mozilla install times with different Jukebox hit-ratios at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s with 100 ms, 10ms, and no added latency respectively. Each bar is the mean of three trials with with the maximum standard deviation observed as 1%.

Figure 2.6: Mozilla Install Results

distribution using CASPER, the time required to install the Mozilla 1.1 browser was measured. The installation was packaged as 6 different RPM Package Manager [121] (RPM) files ranging from 105 KB to 10.2 MB with a total size of 13.5 MB. All the files were kept in a directory on the Coda File System and were installed by using the `rpm` command with the `-Uhv` options.

Figure 2.6 reports the time taken for the Mozilla install to complete at various network bandwidths. The times shown are the mean of three trials with a maximum standard deviation of 1%. To compare benefits at these different settings, the time shown is normalized with respect to the time taken for the baseline install with just Coda. The observed average baseline install times at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s are 1238 seconds, 150 seconds, and 44 seconds, respectively. Note that apart from the network transfer time, the total time also includes the time taken for the `rpm` tool to install the packages.

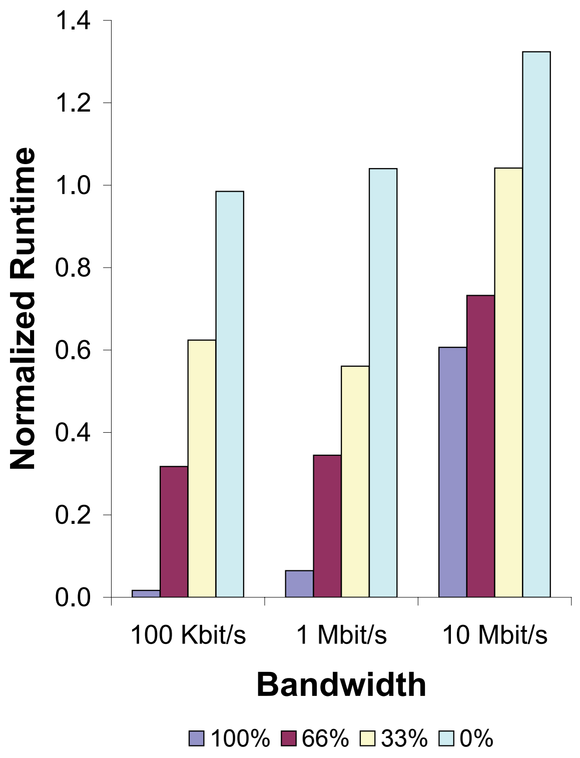
As expected, the gain from using CASPER is most pronounced at low bandwidths where even relatively modest hit-rates can dramatically improve the performance of the system. For example, the reduction in benchmark execution time is 26% at 1 Mbit/s with a 33% jukebox population. Further, even on networks with high bandwidth and low latency characteristics as in the 10 Mbit/s case, the graph shows that the worst-case overhead is low (5%) and a noticeable reduction in benchmark time is possible (approximately 20% in the 100% population case).

2.4.3 Internet Suspend/Resume

The original motivation for pursuing opportunistic use of Content Addressable Storage arose from the work done on Internet Suspend/Resume (ISR) [76]. ISR enables the migration of a user's entire computing environment by layering a virtual machine system on top of a distributed file system. The key challenge in ISR is the transfer of the user's virtual machine state, which may be very large. However, the virtual machine state usually includes installations of popular operating systems and software packages – data which may be expected to be found on CAS providers.

To investigate the potential performance gains obtained by using CASPER for ISR, a trace-based approach was employed. For the workload, the virtual disk drive accesses were traced during the execution of an ISR system that was running a desktop application benchmark. The CDA (Common Desktop Application) benchmark [123] uses Visual Basic scripts to automate the execution of common desktop applications in the Microsoft Office suite executing in a Windows XP environment. By replaying the trace, the file-access pattern of the ISR system was regenerated. The ISR data of interest is stored as 256 KB files in CASPER. During the trace replay, which does not include think time, the files are fetched through CASPER on the client machine. During the benchmark, approximately 1000 such files are accessed.

Figure 2.7 shows the results from the Internet Suspend/Resume benchmark at various network bandwidths. The time shown is again the mean of three trials with a maximum standard deviation of 9.2%. The times shown are also normalized with respect to the baseline time taken using unmodified Coda. The actual average baseline times at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s are 5 hours and 59 minutes, 2046 seconds, and 203 seconds, respectively.



Internet Suspend/Resume benchmark times with different Jukebox hit-ratios at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s with 100 ms, 10ms, and no added latency respectively. Each bar is the mean of three trials with the maximum standard deviation observed as 9.2%.

Figure 2.7: Internet Suspend/Resume Benchmark Results

By comparing with the previous Mozilla experiment, it can be seen that the benefit of using CASPER becomes even more pronounced as the quantity of data transferred grows. Here, unlike in the Mozilla experiment, even the 10 Mbit/s case shows significant gains from exploiting commonality for hit-ratios of 66% and 100%.

Note that in the ISR experiment that used a 1 Mbit/s link and a 33% hit rate, CASPER reduced execution time by 44%. The interaction of application reads and writes explains this anomaly. Every time an application writes a disk block after the virtual machine resumes, the write triggers a read of that block, if that block has not yet been fetched from the virtualized disk. Writes are asynchronous, and are flushed to disk periodically. If CASPER speeds execution, fewer writes are flushed to disk during the shorter run time, and fewer write-triggered reads occur during that execution. Conversely, if CASPER slows execution, more writes are flushed to

Jukebox	Network Bandwidth		
hit-ratio	100 Kbit/s	1 Mbit/s	10 Mbit/s
100%	261.3 (0.5)	40.3 (0.9)	17.3 (1.2)
66%	520.7 (0.5)	64.0 (0.8)	20.0 (1.6)
33%	762.7 (0.5)	85.0 (0.8)	21.3 (0.5)
0%	1069.3 (1.7)	108.7 (0.5)	23.7 (0.5)
Baseline	1150.7 (0.5)	103.3 (0.5)	13.3 (0.5)

Results from the Copy Phase of the modified Andrew Benchmark run at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s with 100 ms, 10ms, and no added latency respectively. Each result is the mean of three trials with the standard deviation given in parentheses

Table 2.2: Modified Andrew Benchmark: Copy Phase Results

disk during the longer run time, and more write-triggered reads occur during that execution. This interaction between writes and reads also explains the higher-than-expected overhead in the 33% and 0% hit-ratio cases on the 10 Mbit/s link.

2.4.4 Modified Andrew Benchmark

The CASPER system was also evaluated by using a modified Andrew Benchmark [68]. The benchmark is very similar to the original but, like Pastiche [46], uses a much larger source tree. The source tree, Apache 1.3.27, consists of 977 files that have a total size of 8.62 MB. The script-driven benchmark contains five phases. Beginning with the original source tree, the scripts recreate the original directory structure, execute a bulk file copy of all source files to the new structure, stat every file in the new tree, scan through all files, and finally build the application.

As mentioned earlier, all experiments were executed in write disconnected mode. Thus, once the Copy phase of the modified Andrew benchmark was complete, a local copy of the files allowed all remaining phases to be unaffected by network conditions; only the Copy phase showed benefit from the CASPER system.

The modified Andrew benchmark differs from the Mozilla and ISR benchmarks in that the average file size is small and more than half of the files are less than 4 KB in size. As described in Section 2.3, the current system is configured such that all recipe requests for files less than 4 KB are rejected by the recipe server in favor of sending the file instead. Thus, while the jukebox might have the data for that file, CASPER fetches it across the slow network link.

Note that, as all operations happened in write-disconnected state, MakeDir, ScanDir, ReadAll, and Make, being local operations, were not affected by bandwidth limitations. As Table 2.3 shows, these phases are not affected by network conditions.

Table 2.2, therefore, only presents results for the Copy phase for all experiments, as Copy is the only phase for which the execution time depends on bandwidth. The results in the table show the relative benefits of using opportunistic CAS at different bandwidths. The time shown is the mean of three trials with standard deviations included within parentheses. Interestingly, the total time taken for the 100 Kbit/s experiment with a hit-ratio of 0% is actually less than the baseline. This behavior can be attributed to the fact that some of the data in the Apache source tree is replicated and consequently matches blocks in the client cache.

The results for the 10 Mbit/s case show a high overhead in all cases. This benchmark illustrates a weakness of the current CASPER implementation. When many small files are transferred over the network, the performance of CASPER is dominated by sources of overhead. For example, the proxy-based approach induces several additional inter-process transfers per file relative to the baseline Coda installation. Further, the jukebox inspection introduces multiple additional network round-trip time latencies associated with the jukebox query messages and subsequent missed-block requests.

Normally, these sources of overhead would be compensated for by the improved performance of converting WAN accesses to LAN accesses. However, for this particular benchmark, the difference in peak bandwidth between the LAN connection and simulated WAN connection was not sufficient to compensate for the additional overhead. In practice, when CASPER detects such a high-bandwidth situation, the system should revert to baseline Coda behavior. Such an adjustment would avoid the system slowdown illustrated by this example.

100 Kbit/s

AB phase	100%	66%	33%	0%	Baseline
MakeDir	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
Copy	261.3 (0.5)	520.7 (0.5)	762.7 (0.5)	1069.3 (1.7)	1150.7 (0.5)
ScanDir	2.0 (0.0)	2.0 (0.0)	2.3 (0.5)	2.3 (0.5)	2.0 (0.0)
ReadAll	4.0 (0.0)	3.7 (0.5)	3.7 (0.5)	3.7 (0.5)	4.3 (0.5)
Make	15.3 (0.5)	16.7 (0.9)	15.7 (0.5)	16.0 (0.0)	15.7 (0.5)
Total	282.7 (0.9)	543.0 (1.4)	784.7 (0.9)	1091.3 (1.7)	1172.7 (0.9)

1 Mbit/s

AB phase	100%	66%	33%	0%	Baseline
MakeDir	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
Copy	40.3 (0.9)	64.0 (0.8)	85.0 (0.8)	108.7 (0.5)	103.3 (0.5)
ScanDir	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.3 (0.5)	2.0 (0.0)
ReadAll	3.7 (0.5)	3.7 (0.5)	4.0 (0.0)	3.7 (0.5)	3.7 (0.5)
Make	15.3 (0.5)	16.0 (0.0)	15.7 (0.5)	16.0 (0.0)	15.7 (0.5)
Total	61.3 (0.9)	85.7 (1.2)	106.7 (0.5)	130.7 (0.5)	124.7 (0.5)

10 Mbit/s

AB phase	100%	66%	33%	0%	Baseline
MakeDir	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.7 (0.5)	0.0 (0.0)
Copy	17.3 (1.2)	20.0 (1.6)	21.3 (0.5)	23.7 (0.5)	13.3 (0.5)
ScanDir	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.0 (0.0)	2.7 (0.5)
ReadAll	3.7 (0.5)	4.0 (0.0)	3.7 (0.5)	3.7 (0.5)	3.7 (0.5)
Make	16.0 (0.0)	16.0 (0.0)	16.0 (0.0)	16.0 (0.0)	15.7 (0.5)
Total	39.0 (0.8)	42.0 (1.6)	43.0 (0.9)	46.0 (0.8)	35.3 (0.5)

Modified Andrew Benchmark run at 100 Kbit/s, 1 Mbit/s, and 10 Mbit/s. Each column represents the hit-ratio on the jukebox. Times are reported in seconds with standard deviations given in parentheses

Table 2.3: Complete Results from the Modified Andrew Benchmark

2.5 Summary

This chapter has shown the design and implementation of CASPER, a distributed file system that opportunistically uses content-addressable techniques to improve the performance of legacy distributed systems. This is achieved without weakening any of the system attributes of the legacy system or requiring modifications to either the system or applications. The evaluation of the prototype demonstrates that CASPER has low overhead in WAN conditions and its use can, even with moderate amounts of commonality, lead to a significant gain in low bandwidth scenarios. Finally, even in the absence of CAS providers, CASPER would be no worse off than the legacy file system as it can fetch files directly from the centralized file server.

Chapter 3

Integrating Portable and Distributed Storage

3.1 Introduction

This chapter examines how non-networked storage resources, such as portable hard drives and the increasingly popular USB flash drives, can be exploited via the use of content-addressable techniques to improve the performance of legacy distributed systems. However, unlike CASPER, this integration is complicated by the fact that non-networked storage exhibits very different performance characteristics, usage models, and failure modes when compared to networked CAS resources.

This chapter focuses on a technique called *lookaside caching* [150] that uses content-addressable techniques to unify "storage in the cloud" (distributed storage) and "storage in the hand" (portable storage) into a single abstraction. By combining the strengths of distributed file systems and portable storage devices, while negating their weaknesses, lookaside caching allows users to treat the devices they carry as merely performance and availability assists for distant file servers.

As distributed file systems are well-entrenched today, it can be assumed that these are successful because they offer genuine value to their users. Hence, the goal of lookaside caching is to integrate portable storage devices into such a system in a transparent manner such that it is minimally disruptive of its existing usage model. In addition, no changes are made to the native file system format of a portable storage device; all that is required is that the device be mountable as a local file system at any client of the distributed file system. In spite of its simplicity, this technique can prove to be powerful and versatile. Further, careless use or the absence of portable storage has no catastrophic consequences.

Section 3.2 begins by examining the strengths and weaknesses of portable storage and distributed file systems. The design and implementation of lookaside caching is then described in Section 3.3. Using three different benchmarks, Section 3.4 quantifies the performance benefit of lookaside caching. Section 3.5 concludes with a brief summary of this chapter.

3.2 Background on Portable Storage

Floppy disks were the sole means of sharing data across users and computers in the early days of personal computing. Although they were trivial to use, considerable discipline and foresight was required of users to ensure data consistency and availability, and to avoid data loss — if the user did not have the right floppy at the right place and time, they were in trouble! These limitations were overcome by the emergence of distributed file systems such as NFS [122] and AFS [68]. In such a system, responsibility for data management is delegated to the distributed file system and its operational staff.

Personal storage has come full circle in the recent past. There has been explosive growth in the availability of USB- and Firewire-connected storage devices such as flash memory keychains and portable disk drives. Although very different from floppy disks in capacity, data transfer rate, form factor, and longevity, their usage model is no different. In other words, they suffer from the same limitations mentioned above. To understand the continuing popularity of portable storage, it is useful to review the strengths and weaknesses of portable storage and distributed file systems. While there is considerable variation in the designs of distributed file systems, there is also a substantial degree of commonality across them. The discussion below focuses on these common themes.

Performance: A portable storage device offers uniform performance at all locations, independent of factors such as network connectivity, initial cache state, and temporal locality of references. Except for a few devices such as floppy disks, the access times and bandwidths of portable devices are comparable to those of local disks. In contrast, the performance of a distributed file system is highly variable. With a warm client cache and good locality, performance can match local storage. With a cold cache, poor connectivity and low locality, performance can be intolerably slow.

Availability: If a user has a portable storage device in hand, she can access its data. Short of device failure, which is very rare, no other common failures prevent data access. In contrast, distributed file systems are susceptible to network failure, server failure, and a wide range of operator errors.

Robustness: A portable storage device can easily be lost, stolen or damaged. Data on the device becomes permanently inaccessible after such an event. In contrast, data in a distributed file system continues to be accessible even if a particular client that uses it is lost, stolen or damaged. For added robustness, the operational staff of a distributed file system perform regular backups and typically keep some of the backups off site to allow recovery after catastrophic site failure. Backups also help recovery from user error: if a user accidentally deletes a critical file, she can recover a backed-up version of it. In principle, a highly disciplined user could implement a careful regimen of backup of portable storage to improve robustness. In practice, few users are sufficiently disciplined and well-organized to do this. It is much simpler for professional staff to regularly back up a few file servers, thus benefiting all users.

Sharing/Collaboration: The existence of a common name space simplifies sharing of data and collaboration between the users of a distributed file system. This is much harder if done by physical transfers of devices.

Consistency: Without explicit user effort, a distributed file system presents the latest version of a file when it is accessed. In contrast, a portable device has to be explicitly kept up to date. When multiple users can update a file, it is easy to get into situations where a portable device has stale data without its owner being aware of this fact.

Capacity: Any portable storage device has finite capacity. In contrast, the client of a distributed file system can access virtually unlimited amounts of data spread across multiple file servers. Since local storage on the client is merely a cache of server data, its size only limits working set size rather than total data size.

Security: The privacy and integrity of data on portable storage devices relies primarily on physical security. A further level of safety can be provided by encrypting the data on the device, and by requiring a password to mount it. These can be valuable as a second layer of defense in case physical security fails. Denial of service is impossible if a user has a portable storage device in hand. In contrast, the security of data in a distributed file system is based on more fragile assumptions. Denial of service may be possible through network attacks. Privacy depends on encryption of network traffic. Fine-grained protection of data through mechanisms such as access control lists is possible, but relies on secure authentication using a mechanism such as Kerberos [139].

Ubiquity: A distributed file system requires operating system support. In addition, it may require environmental support such as Kerberos authentication and specific firewall configuration. Unless a user is at a client that meets all of these requirements, she cannot access her data in a distributed file system. In contrast, portable storage only depends on widely-supported low-level hardware and software interfaces. If a user sits down at a random machine, she can be much more confident of accessing data from portable storage in her possession than from a remote file server.

3.3 Achieving Transparency

The goal of lookaside caching is to exploit the performance and availability advantages of portable storage to improve these same attributes in a distributed file system. The resulting design should be transparent, i.e, it should preserve all other characteristics of the underlying distributed file system and not require any changes to applications or user behavior. In particular, there should be no compromise of robustness, consistency or security. There should also be no added complexity in sharing and collaboration. Finally, the design should be tolerant of human error: improper use of the portable storage device (such as using the wrong device or forgetting to copy the latest version of a file to it) should not hurt correctness.

Lookaside caching is an extension of AFS2-style whole-file caching [68] that meets the above goals. It is based on the observation that virtually all distributed file system protocols provide separate remote procedure calls (RPCs) for access of metadata and access of data content. Lookaside caching extends the definition of metadata to include a cryptographic hash of data content. This extension only increases the size of metadata by a modest amount: just 20 bytes if SHA-1 [128] is used as the hash.

Since hash size does not depend on file length, it costs very little to obtain and cache hash information even for many large files. Using POSIX terminology, caching the results of "`ls -lR`" of a large tree is feasible on a resource-limited client. This continues to be true even if one augments `stat` information for each file or directory in the tree with its SHA-1 hash.

Since lookaside caching treats the hash as part of the metadata, there is no compromise in consistency. The underlying cache coherence protocol of the distributed file system determines how closely client state tracks server state. There is no degradation in the accuracy of this tracking if the hash is used to redirect access of data content. To ensure no compromise in security, the file server should return a null hash for any object on which the client only has permission to read the metadata but not the data itself.

It should be noted that lookaside caching can be viewed as a degenerate case of the use of recipes as used by CASPER in Chapter 2. In CASPER, a recipe is an XML description of file content that enables block-level reassembly of the file from content-addressable storage. One can view the hash of a file as the smallest possible recipe for it. The implementation using recipes is considerably more complex than the support for lookaside caching. In return for this complexity, synthesis from recipes may succeed in many situations where lookaside fails.

Further, the fact that lookaside caching system does not modify the portable device's storage layout allows it to use any device that exports a generic file system interface. This allows files to be stored on the device in any manner chosen by the user, including the same tree structure as the distributed file system. For example, in the Kernel Compile benchmark described in Section 3.4.1, the portable device was populated by simply unarchiving a normal kernel source tree. The advantage of this is that user can still have access to the files in the absence of a network or even a distributed file system client. However, this also allows the user to edit files without the knowledge of the lookaside caching system. While recomputation of the file's hash at the time of use can expose the update, it is up to the user to manually copy the changes back into the distributed file system.

3.3.1 Prototype Implementation

Lookaside caching has been implemented in the Coda file system on Linux. The user-level implementation of both the Coda client cache manager and server code greatly simplified development effort since no kernel changes were needed. The implementation consists of four parts: a small change to the client-server protocol; a quick index check (the "lookaside") in the code path for handling cache misses; a tool for generating lookaside indexes; and a set of user commands to include or exclude specific lookaside devices.

The protocol change replaces two RPCs, `ViceGetAttr()` and `ViceValidateAttrs()` with the extended calls `ViceGetAttrPlusSHA()` and `ViceValidateAttrsPlusSHA()` that have an extra parameter for the SHA-1 hash of the file. `ViceGetAttr()` is used to obtain metadata for a file or directory, while `ViceValidateAttrs()` is used to revalidate cached metadata for a collection of files or directories when connectivity is restored to a server. The implementation preserves compatibility with legacy servers. If a client connects to a server that has not been upgraded to support lookaside caching, it falls back to using the original RPCs mentioned above. Due to the simplicity of this scheme, the changes required for lookaside caching, unlike CASPER, were made directly to the Coda file system. Lookaside caching only added ∼900 Lines of Code (LoC) to a codebase of ∼126,000 LoC.

Once a client possesses valid metadata for an object, it can use the hash to redirect the fetch of data content. If a mounted portable storage device has a file with a matching hash value, the client can obtain the contents of the file from the device rather than from the file server. Whether it is beneficial to do this depends on factors such as file size, network bandwidth, and device transfer rate. The important point is that possession of the hash gives a degree of freedom that clients of a distributed file system do not possess today.

The lookaside occurs just before the execution of the `ViceFetch()` RPC to fetch file contents. Before network communication is attempted, the client consults one or more lookaside indexes to see if a local file with identical SHA-1 value exists. Trusting in the "weak" collision resistance property, as defined in Chapter 1, of SHA-1 [85], a copy operation on the local file can then be a substitute for the RPC. To detect version skew between the local file and its index, the SHA-1 hash of the local file is re-computed. In case of a mismatch, the local file substitution is suppressed and

Command	Description
`cfs lka --clear`	*exclude all indexes*
`cfs lka +db1`	*include index db1*
`cfs lka -db1`	*exclude index db1*
`cfs lka --list`	*print lookaside statistics*

Table 3.1: Lookaside Commands on Client

the cache miss is serviced by contacting the file server. Coda's consistency model is not compromised, although some small amount amount of work is wasted on the lookaside path.

The index generation tool traverses the file tree rooted at a specified pathname. It computes the SHA-1 hash of each file and enters the filename-hash pair into the index file, which is similar to a Berkeley DB database [99]. The tool is flexible regarding the location of the tree being indexed: it can be local, on a mounted storage device, or even on a nearby NFS or Samba server. For a removable device such as a USB storage keychain or a DVD, the index is typically located right on the device. This yields a self-describing storage device that can be used anywhere. Note that an index captures the values in a tree at one point in time. No attempt is made to track updates made to the tree after the index is created. The tool must be re-run to reflect those updates. Thus, a lookaside index is best viewed as a collection of *hints* [143].

Dynamic inclusion or exclusion of lookaside devices is done through user-level commands. Table 3.1 lists the relevant commands on a client. Note that multiple lookaside devices can be in use at the same time. The devices are searched in order of inclusion.

3.4 Performance Improvement

How much of a performance win can lookaside caching provide? The answer clearly depends on the workload, on network quality, and on the overlap between data on the lookaside device and data accessed from the distributed file system. To obtain a quantitative understanding of this relationship, controlled experiments were conducted using three different benchmarks: a kernel compile benchmark, a virtual

Kernel	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.4.18	118.0	100%	100%	02/25/02	0
2.4.17	116.2	90%	79%	12/21/01	66
2.4.13	112.6	74%	52%	10/23/01	125
2.4.9	108.0	53%	30%	08/16/01	193
2.4.0	95.3	28%	13%	01/04/01	417

In this experiment, the kernel being compiled was always version 2.4.18. The kernel on the lookaside device varied across the versions listed above. The second column gives the size of the source tree of a version. The third column shows what fraction of the files in that version remain the same as in 2.4.18. For the given percentage of similar files, the number of bytes, relative to total release size, is given in the fourth column. The last column gives the difference between the release date of a version and the release date of 2.4.18.

Table 3.2: Characteristics of Linux Kernel Source Trees

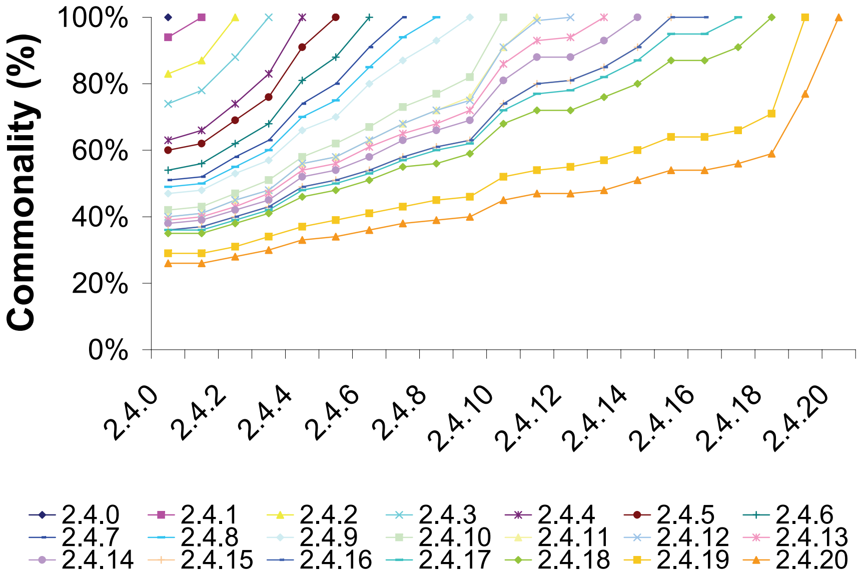
machine migration benchmark, and a single-user trace replay benchmark. The rest of this section presents the benchmarks, experimental setups, and results.

3.4.1 Kernel Compile

Benchmark Description

The first benchmark models a nomadic software developer who does work at different locations such as her home, her office, and a satellite office. Network connection quality to her file server may vary across these locations. The developer carries a version of her source code on a lookaside device. This version may have some stale files because of server updates by other members of the development team.

Version 2.4 of the Linux kernel was used as the source tree in this benchmark. Table 3.2 shows the measured degree of commonality across five different minor versions of the 2.4 kernel, obtained from the FTP site `ftp.kernel.org`. This data shows that there is a substantial degree of commonality even across releases that are many weeks apart. The experiments only use five versions of Linux, but Figure 3.1 confirms that commonality across minor versions exists for all versions of the Linux 2.4 kernel series.



Each curve above corresponds to one minor version of the Linux 2.4 kernel. That curve represents the measured commonality between the minor version and all previous minor versions. The horizontal axis shows the set of possible minor versions. The vertical axis shows the percentage of data in common. The rightmost point on each curve corresponds to 100% because each minor version overlaps 100% with itself.

Figure 3.1: Commonality across Linux 2.4 Versions

Experimental Setup

Figure 3.2 shows the experimental setup used for the evaluation. The client contained a 3.0 GHz Pentium 4 processor (with Hyper-Threading) with 2 GB of SDRAM. The file server contained a 2.0 GHz Pentium 4 processor (without Hyper-Threading) with 1 GB of SDRAM. Both machines ran Red Hat 9.0 Linux and Coda 6.0.2. The client file cache size was large enough to prevent eviction during the experiments, and the client operated in write-disconnected mode. The client file cache was always cold at the start of an experiment. To discount the effect of a cold I/O buffer cache on the server, a warming run was done prior to each set of experiments.

All experiments were run at four different bandwidth settings: 100 Mbit/s, 10 Mbit/s, 1 Mbit/s, and 100 Kbit/s. The NISTNet [36] network emulator (version 2.0.12) was used to control bandwidth and latency. No extra latency was added at

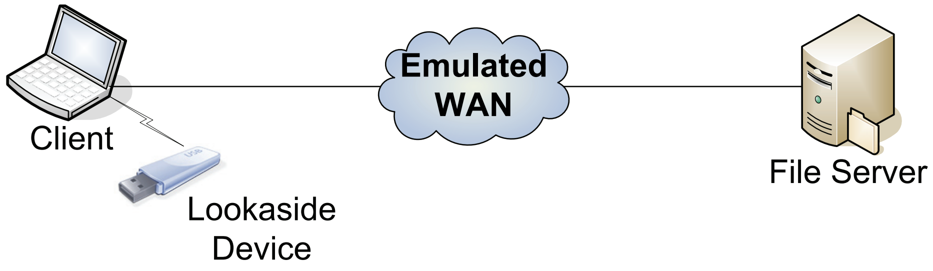


Figure 3.2: Experimental Setup

	Measured Data Rate	
File Size	Read (Mbit/s)	Write (Mbit/s)
4 KB	6.3	7.4
16 KB	6.3	12.5
64 KB	16.7	25.0
256 KB	25.0	22.2
1 MB	28.6	25.8
10 MB	29.3	26.4
100 MB	29.4	26.5

This tables displays the measured read and write bandwidths for different file sizes on the portable storage device used in the experiments. To discount caching effects, the device was unmounted and remounted before each trial. For the same reason, all writes were performed in synchronous mode. Every data point is the mean of three trials; the standard deviation observed was negligible.

Table 3.3: Portable Storage Device Performance

100 Mbit/s and 10 Mbit/s. For 1 Mbit/s and 100 Kbit/s, NISTNet was configured to add round-trip latencies of 10 ms and 100 ms respectively.

The lookaside device used in the experiments was a 512 MB Hi-Speed USB flash memory keychain. The manufacturer of this device quotes a nominal read bandwidth of 48 Mbit/s, and a nominal write bandwidth of 36 Mbit/s. A set of tests were conducted on the client to verify these figures. As the results in Table 3.3 show, for all file sizes ranging from 4 KB to 100 MB, the measured read and write bandwidths were much lower than the manufacturer's figures.

Results

The performance metric in this benchmark is the elapsed time to compile the 2.4.18 kernel. This directly corresponds to the performance perceived by a hypothetical software developer. Although the kernel being compiled was always version 2.4.18 in the experiment, the contents of the portable storage device were varied to explore the effects of using stale lookaside data. The portable storage device was unmounted between each experiment run to discount the effect of the buffer cache.

Table 3.4 presents the results. For each portable device state shown in that figure, the corresponding "Files Same" and "Bytes Same" columns of Table 3.2 bound the usefulness of lookaside caching. The "Days Stale" column indicates the staleness of device state relative to the kernel being compiled.

At the lowest bandwidth (100 Kbit/s), the win due to lookaside caching is impressive: over 90% with an up-to-date device (improving from 9348.8 seconds to 884.9 seconds), and a non-trivial 10.6% (from 9348.8 seconds to 8356.7 seconds) with data that is over a year old (version 2.4.0)! Data that is over two months old (version 2.4.17) is still able to give a win of 67.8% (from 9348.8 seconds to 3011.2 seconds).

At a bandwidth of 1 Mbit/s, the wins are still impressive. They range from 63% (from 1148.3 seconds to 424.8 seconds) with an up-to-date portable device, down to 4.7% (1148.3 seconds to 1094.3 seconds) with the oldest device state. A device that is stale by one version (2.4.17) still gives a win of 52.7% (1148.3 seconds to 543.6 seconds).

On a slow LAN (10 Mbit/s), lookaside caching continues to give a strong win if the portable device has current data: 27.1% (388.4 seconds to 282.9 seconds). The win drops to 6.1% (388.4 seconds to 364.8 seconds) when the portable device is one version old (2.4.17). When the version is older than 2.4.17, the cost of failed lookasides exceeds the benefits of successful ones. This yields an overall loss rather than a win (represented as a negative win in Table 3.4). The worst loss at 10 Mbit/s is 8.4% (388.4 seconds to 421.1 seconds).

Only on a fast LAN (100 Mbit/s) does the overhead of lookaside caching exceed its benefit for all device states. The loss ranges from a trivial 1.7% (287.7 seconds to 292.7 seconds) with current device state to a substantial 24.5% (287.7 seconds to 358.1 seconds) with the oldest device state. Since the client cache manager already monitors

	Lookaside Device State					
Bandwidth	No Device	2.4.18	2.4.17	2.4.13	2.4.9	2.4.0
100 Mbit/s	287.7 (5.6)	292.7 (6.4) [-1.7%]	324.7 (16.4) [-12.9%]	346.4 (6.9) [-20.4%]	362.7 (3.4) [-26.1%]	358.1 (7.7) [-24.5%]
10 Mbit/s	388.4 (12.9)	282.9 (8.3) [27.1%]	364.8 (12.4) [6.1%]	402.7 (2.3) [-3.7%]	410.9 (2.1) [-5.8%]	421.1 (12.8) [-8.4%]
1 Mbit/s	1148.3 (6.9)	424.8 (3.1) [63.0%]	543.6 (11.5) [52.7%]	835.8 (3.7) [27.2%]	1012.4 (12.0) [11.8%]	1094.3 (5.4) [4.7%]
100 Kbit/s	9348.8 (84.3)	884.9 (12.0) [90.5%]	3011.2 (167.6) [67.8%]	5824.0 (221.6) [37.7%]	7616.0 (130.0) [18.5%]	8356.7 (226.9) [10.6%]

These results show the time (in seconds) taken to compile the Linux 2.4.18 kernel. The column labeled "No Device" shows the time taken for the compile when no portable device was present and all data had to be fetched over the network. The column labeled "2.4.18" shows the results when all of the required data was present on the storage device and only metadata (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had versions of the Linux kernel older than 2.4.18. Each data point is the mean of three trials; standard deviations are in parentheses. The numbers in square brackets give the "win" for each case: that is, the percentage improvement over the "no device" case.

Table 3.4: Time for Compiling Linux Kernel 2.4.18

bandwidth to servers, it would be simple to suppress lookaside at high bandwidths. Although this simple change has not been implemented yet, I am confident that it can result in a system that almost never loses due to lookaside caching.

3.4.2 Internet Suspend/Resume

Benchmark Description

The second benchmark is based on Internet Suspend/Resume (ISR) [76], a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there. The ISR-2 prototype [123] was used for this benchmark. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed file system. The ISR prototype layers VMware on Coda, and represents VM state as a tree of 256 KB files.

A key ISR challenge is large VM state, typically many tens of GB. When a user resumes on a machine with a cold file cache, misses on the 256 KB files can result in significant performance degradation. This overhead can be substantial at resume sites with poor connectivity to the file server that holds VM state. If a user is willing to carry a portable storage device with him, part of the VM state can be copied to the device at suspend. Lookaside caching can then reduce the performance overhead of cache misses at the resume site. It might not always be possible to carry the entire VM state as writing it to the portable device may take too long for a user in a hurry to leave. In contrast, propagating updates to a file server can continue after the user leaves.

A different use of lookaside caching for ISR is based on the observation that there is often substantial commonality in VM state across users [23, 94]. For example, the installed code for applications such as Microsoft Office is likely to be the same for all users running the identical software release of those applications. Since this code does not change until a software upgrade (typically many months apart), it would be simple to distribute copies of the relevant data on DVD or CD-ROM media at likely resume sites.

Notice that lookaside caching is tolerant of human error in both of the above contexts. If the user inserts the wrong USB storage keychain into her machine at resume, stale data on it will be ignored. Similarly, use of the wrong DVD or CD-ROM does not hurt correctness. In both cases, the user sees slower performance but is otherwise unaffected.

Since ISR is intended for interactive workloads typical of laptop environments, the experiment uses the Common Desktop Application (CDA) benchmark [123] that models an interactive Windows user. CDA uses Visual Basic scripting to drive Microsoft Office applications such as Word, Excel, Powerpoint, Access, and Internet Explorer. It consists of a total of 113 independently-timed operations such as `find-and-replace`, `open-document`, and `save-as-html`. Note that each of these macro-operations may result in more than one file system call within the VM and, consequently, multiple requests to Coda. Minor user-level actions such as keystrokes, object selection, or mouse-clicks are not timed.

Experimental Setup

The experimental infrastructure consists of clients with 2.0 GHz Pentium 4 processors connected to a server with a 1.2 GHz Pentium III Xeon processor through 100 Mbit/s Ethernet. All machines have 1 GB of RAM, and run Red Hat 7.3 Linux. Unless indicated otherwise, a Hi-Speed USB flash memory keychain was used. Clients use VMware Workstation 3.1 and have an 8 GB Coda file cache. The VM is configured to have 256 MB of RAM and 4 GB of disk, and runs Windows XP as the guest OS. As in the previous benchmark, the NISTNet network emulator is used to control bandwidth.

Results

From a user's perspective, the key performance metrics of ISR can be characterized by two questions:

- *How slow is the resume step?*
  This speed is determined by the time to fetch and decompress the physical memory image of the VM that was saved at suspend. This is the smallest part of total VM state that must be present to begin execution. The rest of the state can be demand-fetched after execution resumes. The delay between the resume

	No Lookaside	With Lookaside	Win
100 Mbit/s	14 (0.5)	13 (2.2)	7.1%
10 Mbit/s	39 (0.4)	12 (0.5)	69.2%
1 Mbit/s	317 (0.3)	12 (0.3)	96.2%
100 Kbit/s	4301 (0.6)	12 (0.1)	99.7%

This table shows the resume latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a USB flash memory keychain. Each data point is the mean of three trials; standard deviations are in parentheses.

Table 3.5: Resume Latency

	No Lookaside	With Lookaside	Win
100 Mbit/s	173 (9)	161 (28)	6.9%
10 Mbit/s	370 (14)	212 (12)	42.7%
1 Mbit/s	2688 (39)	1032 (31)	61.6%
100 Kbit/s	30531 (1490)	9530 (141)	68.8%

This table gives the total operation latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a DVD. Each data point is the mean of three trials, with standard deviation in parentheses. Approximately 50% of the client cache misses were satisfied by lookaside on the DVD. The files on the DVD correspond to the image of a freshly-installed virtual machine, prior to user customization.

Table 3.6: Total Operation Latency

command and the earliest possible user interaction is referred to as *Resume Latency.*

- *After resume, how much is work slowed?*
  The user may suffer performance delays after resume due to file cache misses. The metric used to reflect the user's experience is the total time to perform all the operations in the CDA benchmark (this excludes user think time). This metric is referred to as *Total Operation Latency.*

Portable storage can improve both resume latency and total operation latency. Table 3.5 presents the results for the case where a USB flash memory keychain is updated at suspend with the minimal state needed for resume. This is a single 41 MB file corresponding to the compressed physical memory image of the suspended virtual machine. Comparing the second and third columns of this figure, it can be seen that the effect of lookaside caching is noticeable below 100 Mbit/s, and is dramatic at 100 Kbit/s. A resume time of just 12 seconds rather than 317 seconds (at 1 Mbit/s) or 4301 seconds (at 100 Kbit/s) can make a world of a difference to a user with a few minutes of time in a coffee shop or a waiting room. Even at 10 Mbit/s, resume latency is a factor of 3 faster (12 seconds rather than 39 seconds). The user only pays a small price for these substantial gains: she has to carry a portable storage device, and has to wait for the device to be updated at suspend. With a Hi-Speed USB device, this wait is just a few seconds.

To explore the impact of lookaside caching on total operation latency, a DVD was constructed with the VM state captured after installation of Windows XP and the Microsoft Office suite, but before any user-specific or benchmark-specific customizations. This DVD was used as a lookaside device after resume. In a real-life deployment, an entity such as the computing services organization of a company, university, or ISP would create a set of VM installation images and matching DVDs for its clientele. Distributing DVDs to each ISR site does not compromise ease of management because misplaced or missing DVDs do not hurt correctness. A concerned user could, of course, carry her own DVD.

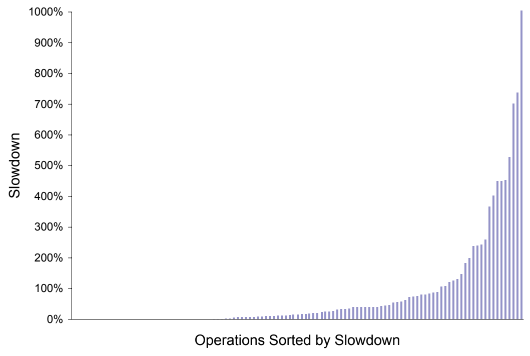
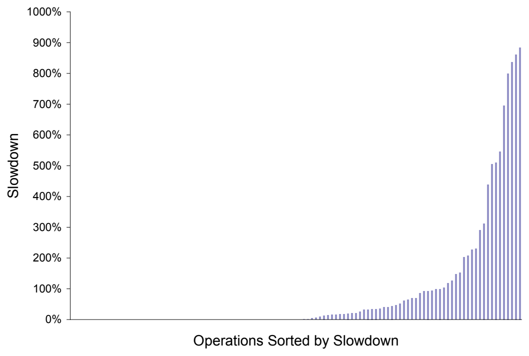
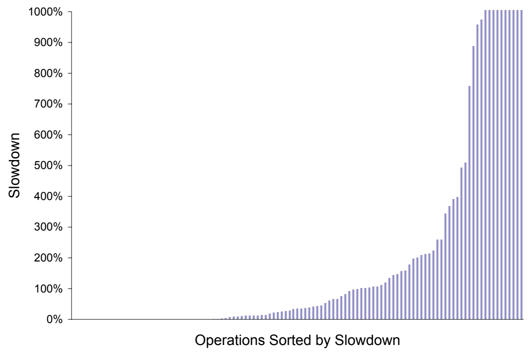
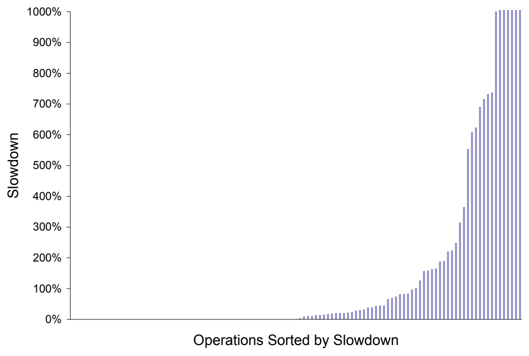
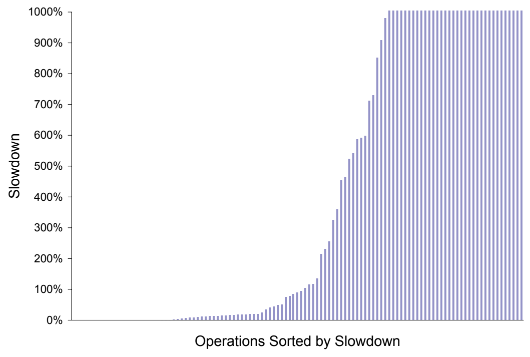
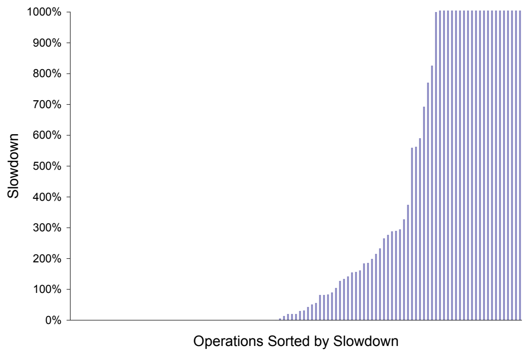
Table 3.6 shows that lookaside caching reduces total operation latency at all bandwidths, with the reduction being most noticeable at low bandwidths. Table 3.7 shows the distribution of *slowdown* for individual operations in the benchmark. Slowdown is defined as $(T_{\mathrm{BW}} - T_{\mathrm{NoISR}})/T_{\mathrm{NoISR}}$, with T_{BW} being the benchmark running time at the given bandwidth and T_{NoISR} its running time in VMware without ISR. Table 3.7, sorted by the degree of slowdown, confirms that lookaside caching reduces the number of operations with very large slowdowns.

3.4.3 Trace Replay

Benchmark Description

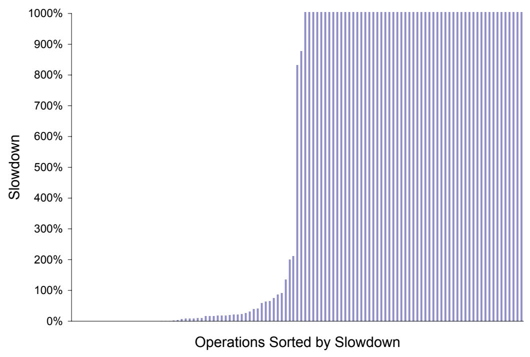
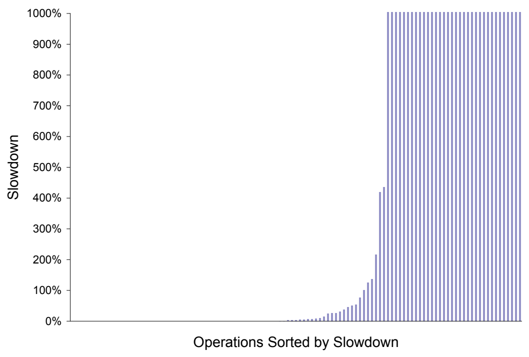
Finally, the trace replay benchmark described by Flinn et al. [58] in their evaluation of data staging was used. This benchmark consists of four traces that were obtained

Table 3.7: Slowdown Improvements with Lookaside Caching

No Lookaside	With Lookaside
100 Mbit/s	
	
10 Mbit/s	
	
1 Mbit/s	
	

Continued on next page

Table 3.7 – continued from previous page

No Lookaside	With Lookaside
100 Kbit/s	
	

Trace	Number of Operations	Length (Hours)	Update Ops.	Working Set (MB)
purcell	87739	27.66	6%	252
messiaen	44027	21.27	2%	227
robin	37504	15.46	7%	85
berlioz	17917	7.85	8%	57

This table summarizes the file system traces used for the benchmark described in Section 3.4.3. "Update Ops." only refers to the percentage of operations that change the file system state such as `mkdir`, `close-after-write`, etc. but not individual reads and writes. The working set is the size of the data accessed during trace execution.

Table 3.8: Trace Statistics

from single-user workstations and that range in collection duration from slightly less than 8 hours to slightly more than a day. As the original traces only capture file sizes and not content, the trace replay tool, at startup, fills the files with random data. Table 3.8 summarizes the attributes of these traces. To ensure a heavy workload, these traces were replayed as fast as possible, without any filtering or think delays.

Experimental Setup

The experimental setup used was the same as that described in Section 3.4.1.

Results

The performance metric in this benchmark is the time taken for trace replay completion. Although no think time is included, trace replay time is still a good indicator of performance seen by the user [154].

To evaluate the performance in relation to the portable device state, the amount of data found on the device was varied. This was done by examining the pre-trace snapshots of the traced file systems and then selecting a subset of the trace's working set. For each trace, initially 33% of the files were randomly selected from the pre-trace snapshot as the initial portable device state. Files were again randomly added to raise the percentage to 66% and then finally to 100%. However, these percentages do not necessarily mean that the data from every file present on the portable storage device was used during the benchmark. The snapshot creation tool also creates files that might be overwritten, unlinked, or simply `stat`-ed. Therefore, while these files might be present on the portable device, they would not be read from it during trace replay.

Figure 3.9 presents the results. The baseline for comparison, shown in column 3 of the figure, was the time taken for trace replay when no lookaside device was present. At the lowest bandwidth (100 Kbit/s), the win due to lookaside caching with an up-to-date device was impressive: ranging from 83% for the Berlioz trace (improving from 1281.2 seconds to 216.8 seconds) to 53% for the Purcell trace (improving from 2828.7 seconds to 1343.0 seconds). Even with devices that only had 33% of the data, lookaside caching was still able to get wins ranging from 25% for the Robin trace to 15% for the Berlioz and Purcell traces.

At a bandwidth of 1 Mbit/s, the wins still remain substantial. For an up-to-date device, they range from 68% for the Berlioz trace (improving from 94.0 seconds to 30.2 seconds) to 39% for the Purcell trace (improving from 292.8 seconds to 178.4 seconds). Even when the device contain less useful data, the wins still range from 24% to 46% when the device has 66% of the snapshot and from 13% to 24% when the device has 33% of the snapshot.

On a slow LAN (10 Mbit/s) the wins can be strong for an up-to-date device: ranging from 28% for the Berlioz trace (improving from 12.9 seconds to 9.3 seconds) to 6% for Messiaen (improving from 36.3 seconds to 34.1 seconds). Wins tend to tail off beyond this point as the device contains lesser fractions of the working set, but it is important to note that performance is never significantly below that of the baseline.

Only on a fast LAN (100 Mbit/s) does the overhead of lookaside caching begin to dominate. For an up-to-date device, the traces show a loss ranging from 6% for Purcell (changing from 50.1 seconds to 53.1 seconds) to a loss of 20% for Messiaen (changing from 26.4 seconds to 31.8 seconds). While the percentages might be high, the absolute difference in number of seconds is not and might be imperceptible to the user. It is also interesting to note that the loss decreases when there are fewer files on the portable storage device. For example, the loss for the Robin trace drops from 14% when the device is up-to-date (a difference of 4.3 seconds) to 2% when the device has

	Lookaside Device State			
Bandwidth	No Device	100%	66%	33%
Purcell				
100 Mbit/s	50.1 (2.6)	53.1 (2.4)	50.5 (3.1)	48.8 (1.9)
10 Mbit/s	61.2 (2.0)	55.0 (6.5)	56.5 (2.9)	56.6 (4.6)
1 Mbit/s	292.8 (4.1)	178.4 (3.1)	223.5 (1.8)	254.2 (2.0)
100 Kbit/s	2828.7 (28.0)	1343.0 (0.7)	2072.1 (30.8)	2404.6 (16.3)
Messiaen				
100 Mbit/s	26.4 (1.6)	31.8 (0.9)	29.8 (0.9)	27.9 (0.8)
10 Mbit/s	36.3 (0.5)	34.1 (0.7)	36.7 (1.5)	37.8 (0.5)
1 Mbit/s	218.9 (1.2)	117.8 (0.9)	157.0 (0.6)	184.8 (1.3)
100 Kbit/s	2327.3 (14.8)	903.8 (1.4)	1439.8 (6.3)	1856.6 (89.2)
Robin				
100 Mbit/s	30.0 (1.6)	34.3 (3.1)	33.1 (1.2)	30.6 (2.1)
10 Mbit/s	37.3 (2.6)	33.3 (3.8)	33.8 (2.5)	37.7 (4.5)
1 Mbit/s	229.1 (3.4)	104.1 (1.3)	143.2 (3.3)	186.7 (2.5)
100 Kbit/s	2713.3 (1.5)	750.4 (5.4)	1347.6 (29.6)	2033.4 (124.6)
Berlioz				
100 Mbit/s	8.2 (0.3)	8.9 (0.2)	9.0 (0.3)	8.8 (0.2)
10 Mbit/s	12.9 (0.8)	9.3 (0.3)	9.9 (0.4)	12.0 (1.6)
1 Mbit/s	94.0 (0.3)	30.2 (0.6)	50.8 (0.3)	71.6 (0.5)
100 Kbit/s	1281.2 (54.6)	216.8 (0.5)	524.4 (0.4)	1090.5 (52.6)

The above results show how long it took for each trace to complete at different portable device states as well as at different bandwidth settings. The column labeled "No Device" shows the time taken for trace execution when no portable device was present and all data had to be fetched over the network. The column labeled 100% shows the results when all of the required data was present on the storage device and only metadata (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had varying fractions of the working set. Each data point is the mean of three trials; standard deviations are in parentheses.

Table 3.9: Time for Trace Replay

33% of the files present in the snapshot (a difference of 0.6 seconds). As mentioned earlier in Section 3.4.1, the system should suppress lookaside in such scenarios.

Even with 100% success in lookaside caching, the 100 Kbit/s numbers for all of the traces are substantially greater than the corresponding 100 Mbit/s numbers. This is due to the large number of metadata accesses, each incurring RPC latency.

3.5 Summary

"Sneakernet," the informal term for manual transport of data, is alive and well today in spite of advances in networking and distributed file systems. Carrying data on a portable storage device gives a user full confidence that she will be able to access that data anywhere, regardless of network quality, network or server outages, and machine configuration. Unfortunately, this confidence comes at a high price. Remembering to carry the right device, ensuring that data on it is current, tracking updates by collaborators, and guarding against loss, theft and damage are all burdens borne by the user. Most harried mobile users would gladly delegate these chores if only they could be confident that they would have easy access to their critical data at all times and places.

Lookaside caching suggests a way of achieving this goal. By assigning the true home of a user's data to be in a distributed file system, it allows them to make a copy of their critical data on a portable storage device. If users find themselves needing to access the data in a desperate situation, they can just use the device directly. In all other situations, the device can be used for lookaside caching. On a slow network or with a heavily loaded server, users will benefit from improved performance. With network or server outages, they will benefit from improved availability if the distributed file system supports disconnected operation and if they had hoarded all their metadata.

Users make the decision to use the device directly or via lookaside caching at the point of use, not *a priori*. This preserves maximum flexibility up front, when there may be uncertainty about the exact future locations where the user will need to access the data. Lookaside caching thus integrates portable storage devices and distributed file systems in a manner that combines their strengths. It preserves the intrinsic advantages of performance, availability and ubiquity possessed by portable devices, while simultaneously preserving the consistency, robustness and ease of sharing/collaboration provided by distributed file systems.

One can envision many extensions to lookaside caching. For example, the client cache manager could track portable device state and update stale files automatically. This would require a binding between the name space on the device and the name space of the distributed file system. With this change, a portable device effectively becomes an extension of the client's cache. Another extension would be to support lookaside caching on individual blocks of a file rather than on a whole-file basis. While this is conceptually more general, it is not clear how useful it would be in practice because parts of files would be missing if the portable device were to be used directly rather than via lookaside.

Overall, the current design of lookaside caching represents a sweet spot in the space of design tradeoffs. It is conceptually simple, easy to implement, and tolerant of human error. It provides good performance and availability benefits without compromising the strengths of portable storage devices or distributed file systems.

Chapter 4

Improving Database Performance in Multi-Tiered Architectures

4.1 Introduction

As the World Wide Web has grown, many web sites have decentralized their data and functionality by pushing them to the edges of the Internet. Today, eBusiness systems often use a three-tiered architecture consisting of a front-end web server, an application server, and a back-end database server. Figure 4.1 illustrates this architecture. The first two tiers can be replicated close to a concentration of clients at the edge of the Internet. This improves user experience by lowering end-to-end latency and reducing exposure to backbone traffic congestion. It can also increase the availability and scalability of web services.

However, an increasing fraction of web content is dynamically generated from back-end relational databases. Even when database content remains unchanged, without application-specific knowledge, dynamic content is not cacheable by web browsers or by intermediate caching servers such as Akamai [3]. In a multi-tiered architecture, each web request can therefore stress the WAN link between the web server and the database and can cause user experience to be highly variable. Replication of the database is not feasible either, because of the difficult task of simultaneously providing strong consistency, availability, and tolerance to network partitions in a distributed database system [27]. Due to this difficulty, previous attempts in caching dynamic

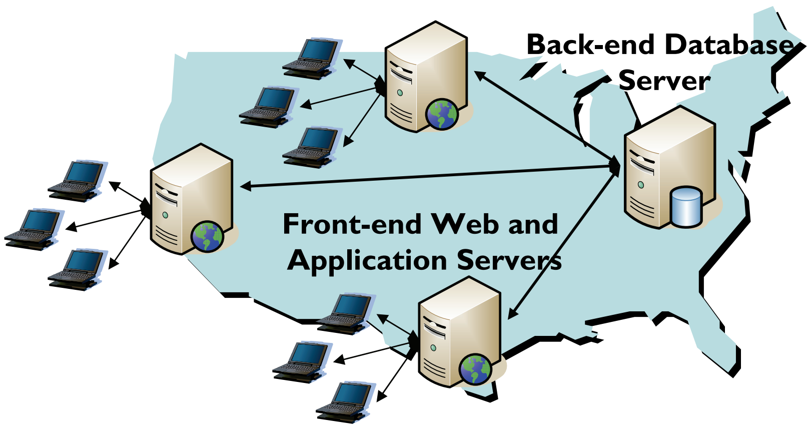


Figure 4.1: Multi-Tier Architecture

database content have generally weakened transactional semantics [6, 9] or required application modifications [59, 133]. As a result, databases tend to be centralized to meet the strong consistency requirements of many eBusiness applications such as banking, finance, and online retailing [165]. In the absence of both caching and replication, WAN bandwidth can easily become a limiting factor in the performance and scalability of data-intensive applications.

This chapter reports on a new solution that takes the form of a database-agnostic middleware layer called *Ganesh*. Ganesh makes no effort to semantically interpret the contents of queries or their results. Instead, it relies exclusively on content-addressable techniques to detect similarities with previous results. While content-addressable techniques were used in the CASPER and Lookaside Caching systems described in Chapters 2 and 3, optimizing these systems was relatively straightforward as they maintained consistency at a per-file level. These systems are also characterized by relatively large objects where updates, deletes, and insertions usually occur *in-place* and therefore domain-independent methods could be used to detect similarity in opaque objects. However, applying these techniques to relational databases is more complex as the data is stored at much finer levels of granularity, has rich internal structure, and updates are less predictable.

At least three challenges are faced in applying hash-based similarity detection to databases query results. First, previous work in this space has traditionally viewed storage content as uninterpreted bags of bits with no internal structure. This allows hash-based techniques to operate on long, contiguous runs of data for maximum effectiveness. In contrast, relational databases have rich internal structure that may not be as amenable to hash-based similarity detection. These difficulties preclude straightforward application of previous results from distributed file systems to the domain of relational databases. Second, relational databases have very tight integrity and consistency constraints that must not be compromised by the use of hash-based techniques. Third, the source code of commercial databases is typically not available. This is in contrast to previous work which presumed availability of source code.

The experimental results show that Ganesh, while conceptually simple, can improve performance significantly at bandwidths representative of today's commercial Internet. On benchmarks modeling multi-tiered web applications, the throughput improvement was as high as tenfold for data-intensive workloads. For workloads that were not data-intensive, throughput improvements of up to twofold were observed. Even when bandwidth was not a constraint, Ganesh had low overhead and did not hurt performance. The experiments also confirmed that exploiting the structure present in database results is crucial to performance improvement.

First, Ganesh's approach to detecting similarity is described in Section 4.2. The design and implementation behind Ganesh's transparent proxy-based solution is presented in Section 4.3. The evaluation of Ganesh is then reported in Sections 4.4. Finally, this chapter concludes with a brief summary of the findings in Section 4.5.

4.2 Detecting Similarity

Ganesh's focus is on efficient transmission of results by discovering similarities with the results of previous queries. As SQL queries can generate large results, content-addressable techniques lend themselves well to the problem of efficiently transferring these large results across bandwidth constrained links. By exploiting redundancy in the result stream, Ganesh can avoid transmitting result fragments that are already present at the query site. Redundancy can arise naturally in many different ways. For example, a query repeated after a certain interval may return a different result because of updates to the database; however, there may be significant commonality in the two results. As another example, a user who is refining a search may generate a sequence of queries with overlapping results. When Ganesh detects redundancy, it suppresses transmission of the corresponding result fragments. Instead, it transmits a much smaller digest of those fragments and lets the query site reconstruct the result through hash lookups in a cache of previous results. In effect, Ganesh uses computation at the edges to reduce Internet communication.

One of the key design decisions in Ganesh was how similarity would be detected. There are many potential ways to decompose a result into fragments. The optimal way is, of course, the one that results in the smallest possible object for transmission for a given query's results. Finding this optimal decomposition was a difficult problem because of the large space of possibilities and because the optimal choice depends on many factors such as the contents of the query's result, the history of recent results, and the cache management algorithm used.

When an object is opaque, the use of Rabin fingerprints [29, 108] to detect common data between two objects has been successfully shown in the past by systems such as LBFS [91], CASPER [149], and Lookaside Caching [150]. Rabin fingerprinting uses a sliding window over the data to compute a rolling hash. Assuming that the hash function is uniformly distributed, a chunk boundary is defined whenever the lower order bits of the hash value equal some predetermined value. The number of lower order bits used defines the average chunk size. These sub-divided chunks of the object become the unit of comparison for detecting similarity between different objects.

As the locations of boundaries found by using Rabin fingerprints is stochastically determined, they usually fail to align with any structural properties of the underlying data. This is usually not a problem for storage systems that store unstructured data. However, while the algorithm deals well with *in-place* updates, insertions and deletions, it performs poorly in the presence of any reordering of data.

Figure 4.2 shows an example where two results, `A` and `B`, consisting of three rows, have the same data but have different sort attributes. In the extreme case, Rabin fingerprinting might be unable to find any similar data due to the way it detects chunk boundaries. Fortunately, Ganesh can use domain specific knowledge for more precise boundary detection. The information Ganesh exploits is that a query's result reflects the structure of a relational database where all data is organized as tables and rows. It is therefore simple to check for similarity with previous results at two granularities: first over the entire result, and then at individual rows. The end of a row in a result serves as a natural chunk boundary. It is important to note that using the tabular structure in results only involves a shallow interpretation of the data.

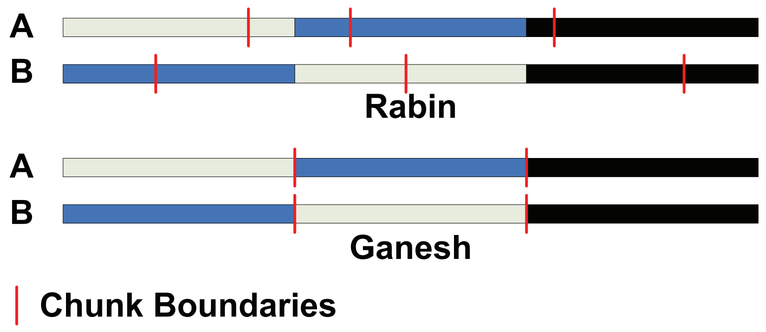


Figure 4.2: Rabin Fingerprinting vs. Ganesh's Chunking

Ganesh does not perform any deeper semantic interpretation such as understanding data types, result schema, or integrity constraints.

Tuning Rabin fingerprinting for a workload can also be difficult. If the average chunk size is too large, chunks can span multiple result rows. However, selecting a smaller average chunk size increases the amount of metadata required to the describe the results. This, in turn, would decrease the savings obtained via its use. Rabin fingerprinting also needs two computationally-expensive passes over the data: once to determine chunk boundaries and once again to generate cryptographic hashes for the chunks. Ganesh only needs a single pass for hash generation as the chunk boundaries are provided by the data's natural structure.

The performance comparison in Section 4.4.4 shows that Ganesh's row-based algorithm outperforms Rabin fingerprinting. Given that previous work has already shown that Rabin fingerprinting performs better than gzip [91], this chapter does not compare Ganesh to compression algorithms.

4.3 Achieving Transparency

The key factor influencing Ganesh's design was the need for it to be completely transparent to all components of a typical eBusiness system: web servers, application servers, and database servers. Without this, Ganesh stands little chance of having a significant real-world impact. Requiring modifications to any of the above components would raise the barrier for entry of Ganesh into an existing system, and thus reduce its chances of adoption. Preserving transparency is simplified as Ganesh is purely a performance enhancement, not a functionality or usability enhancement.

This section first discusses how the Ganesh architecture makes it completely invisible to all components of a multi-tier system. Then, it explains Ganesh's proxy-based caching approach. Finally, the data flow for detecting similarity is described.

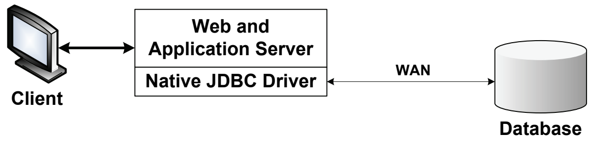
4.3.1 Proxy-Based Interpositioning

Agent interposition [71] was chosen as the architectural approach to realizing Ganesh's goal. This approach relies on the existence of a compact programming interface that is already widely used by target software. It also relies on a mechanism to easily add new code without disrupting existing module structure.

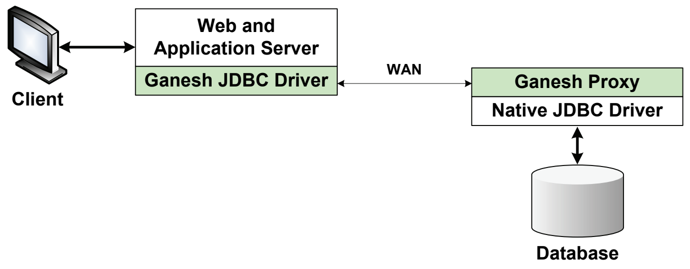
These conditions are easily met in Ganesh's context because of the popularity of standardized interfaces. For example, the widely used Java Database Connectivity (JDBC) API [113] allows Java applications to access a wide variety of databases and even tabular data repositories such as flat files. Access to data is provided by JDBC drivers that translate between the JDBC API and the database communication mechanism. Figure 4.3(a) shows how JDBC is typically used in an application.

As the JDBC interface is standardized, one JDBC driver can be substituted for another without application modifications. The JDBC driver thus becomes the natural module to exploit for code interposition. Java's support for Remote Method Invocation (RMI) completes the picture by simplifying code placement.

Figure 4.3(b) shows how Ganesh is interposed. At the web and application server end of the WAN, the JDBC driver is replaced with a Ganesh JDBC driver that presents the same interface to the application. The Ganesh driver maintains an in-memory cache of result fragments from previous queries and performs reassembly of results. At the database end of the WAN, a new process called the Ganesh proxy is added. This proxy, which can be shared by multiple front-end nodes, consists of two parts: code to detect similarity in result fragments and the original native JDBC driver that communicates with the database. The use of a proxy at the database end makes Ganesh database-agnostic and simplifies prototyping and experimentation. Ganesh is thus able to work with a wide range of databases and applications, requiring no modifications to either. While not currently implemented, the same concepts apply to other standardized interfaces such as ODBC [60].



(a) Native Architecture



(b) Ganesh's Interposition-based Architecture

Figure 4.3: Native vs. Ganesh Architecture

4.3.2 Proxy-Based Caching

The native JDBC driver shown in Figure 4.3(a) is a lightweight code component supplied by the database vendor. Its main function is to mediate communication between the application and the remote database. It forwards queries, buffers entire results, and responds to application requests to view parts of results.

The Ganesh JDBC driver shown in Figure 4.3(b) presents the application with an interface identical to that provided by the native driver. It provides the ability to reconstruct results from compact hash-based descriptions sent by the proxy. To perform this reconstruction, the driver maintains an in-memory cache of recently-received results. This cache is only used as a source of result fragments in reconstructing results. No attempt is made by the Ganesh driver or proxy to track database updates. The lack of cache consistency does not hurt correctness as a description of the results is always fetched from the proxy — at worst, there will be no performance benefit from using Ganesh. Stale data is simply paged out of the cache over time.

The Ganesh proxy interacts with the database via the native JDBC driver, which remains unchanged between Figures 4.3(a) and (b). The database is thus completely unaware of the existence of the proxy. The proxy does not examine any queries received from the Ganesh driver but passes them to the native driver. Instead, the

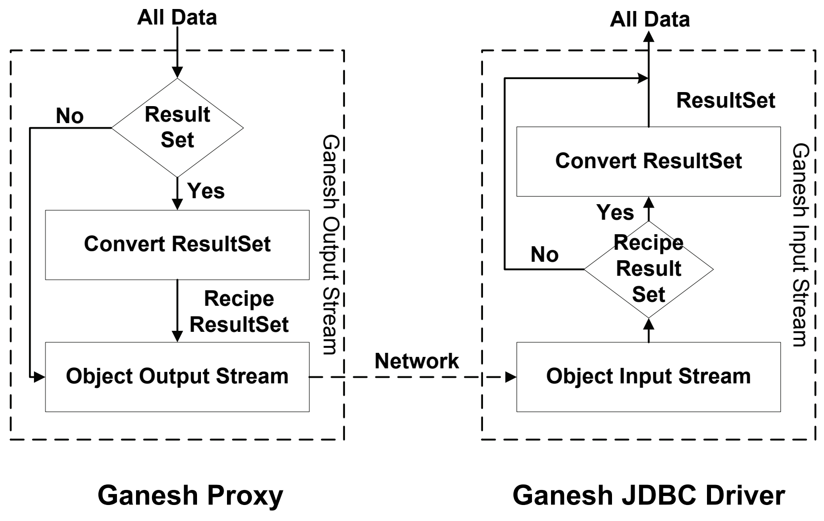


Figure 4.4: Dataflow for Result Handling

proxy is responsible for inspecting database output received from the native driver, detecting similar results, and generating hash-based encodings of these results whenever enough similarity is found.

To generate a hash-based encoding, the proxy must be aware of what result fragments are available in the Ganesh driver's cache. One approach is to be optimistic, and to assume that all result fragments are available. This will result in the smallest possible initial transmission of a result. However, in cases where there is little overlap with previous results, the Ganesh driver will have to make many calls to the proxy during reconstruction to fetch missing result fragments. To avoid this situation, the proxy loosely tracks the state of the Ganesh driver's cache. Since both components are under Ganesh's control, it is relatively simple to do this without resorting to gray-box techniques [13] or explicit communication for maintaining cache coherence. Instead, the proxy simulates the Ganesh driver's cache management algorithm and uses this to maintain a list of hashes for which the Ganesh driver is likely to possess the result fragments. In case of mistracking, there will be no loss of correctness but there will be extra round-trip delays to fetch the missing fragments. If the client detects loss of synchronization with the proxy, it can ask the proxy to reset the state shared between them. Also note that the proxy does not need to keep the result fragments themselves, only their hashes. This allows the proxy to remain scalable even when it is shared by many front-end nodes.

4.3.3 Encoding and Decoding Results

The Ganesh proxy receives database output as Java objects from the native JDBC driver. It examines all objects to see if `ResultSet` Java objects, a data type used to store results from database queries, are present. If a `ResultSet` object is found, it is shrunk as discussed below. All other Java objects are passed through unmodified.

The proxy converts all `ResultSet` objects it sees into objects of a new type called `RecipeResultSet`. The term "recipe" is used for this compact description of a database result because of its similarity to the *file recipes* used by the CASPER file system described in Chapter 2. The conversion replaces each result fragment that is likely to be present in the Ganesh driver's cache by a SHA-1 hash of that fragment. Previously unseen result fragments are retained verbatim. The proxy also retains hashes for the new result fragments as they will be present in the driver's cache in the future. It should be noted that the proxy only caches hashes for result fragments and does not cache recipes.

Based on the discussion in Section 4.2, the proxy first checks for entire result similarity and then at the row level before constructing the `RecipeResultSet`. If the entire result is predicted to be present in the Ganesh driver's cache, the `RecipeResultSet` is simply a single hash of the entire result. Otherwise, it contains hashes for those rows predicted to be present in that cache; all other rows are retained verbatim. If the proxy estimates an overall space savings, it will perform a serialization operation, the Java equivalent of marshaling, and transmit the `RecipeResultSet`. Otherwise the original `ResultSet` is transmitted.

A reverse transformation of `RecipeResultSet` objects into `ResultSet` objects is performed by the Ganesh driver. Figure 4.4 illustrates `ResultSet` handling at both ends. Each SHA-1 hash found in a `RecipeResultSet` is looked up in the local cache of result fragments. On a hit, the hash is replaced by the corresponding fragment. On a miss, the driver contacts the Ganesh proxy to fetch the fragment. All previously unseen result fragments that were retained verbatim by the proxy are hashed and added to the result cache.

There should be very few misses if the proxy has accurately tracked the Ganesh driver's cache state. A future optimization would be to batch the fetch of missing fragments. This would be valuable when there are many small missing fragments in

Benchmark	Dataset	Details
BBOARD	2.0 GB	500,000 Users, 12,000 Stories 3,298,000 Comments
AUCTION	1.3 GB	1,000,000 Users 34,000 Items

Table 4.1: Benchmark Dataset Details

a high-latency WAN. Once the transformation is complete, the fully reconstructed `ResultSet` object is passed up to the application.

4.4 Performance Improvement

Three questions follow naturally from the goals and design of Ganesh:

- First, can performance be improved significantly by exploiting similarity across database results?
- Second, how important is Ganesh's structural similarity detection relative to Rabin fingerprinting-based similarity detection?
- Third, is the overhead of Ganesh's proxy-based design acceptable?

The section answers these question through controlled experiments with the Ganesh prototype. The evaluation is based on two benchmarks [10] that have been widely used by other researchers to evaluate various aspects of multi-tier and eBusiness architectures. Prior work has used these benchmarks to explore the performance of different architectures for serving dynamic content [37], the evaluation of method-caching in multi-tier systems [102], and the development of recovery techniques for Internet services [33]. The first benchmark, `BBOARD`, is modeled after *Slashdot*, a technology-oriented news site. The second benchmark, `AUCTION`, is modeled after *eBay*, an online auction site. Both benchmarks are maintained by the Java Middleware Open Benchmarking (JMOB) group of the ObjectWeb Consortium [97], an international open source community hosted by INRIA. In both benchmarks, most content is dynamically generated from information stored in a database. Details of the datasets used can be found in Table 4.1.

For these benchmarks, the two metrics used to quantify the performance improvement obtainable through the use of Ganesh were throughput, from the perspective of the web server, and average response time from the perspective of the client. Throughput is measured in terms of the number of client requests that can be serviced per second.

The next section describes the experimental procedure used. The results from the two benchmarks, used to answer the first question posed above, are then presented in Sections 4.4.2 and 4.4.3. The importance of using structure-based similarity detection, the answer to the second question, is presented in Section 4.4.4. Finally, to address the third question, the overhead of the proxy-based design in discussed in Section 4.4.5.

4.4.1 Experimental Methodology

Both benchmarks involve a synthetic workload of clients accessing a web server. The number of clients emulated is an experimental parameter. Each emulated client runs an instance of the benchmark in its own thread, using a matrix to transition between different benchmark states. The matrix defines a stochastic model with probabilities of transitioning between the different states that represent typical user actions. An example transition is a user logging into the `AUCTION` system and then deciding on whether to post an item for sale or bid on active auctions. Each client also models user think time between requests. The think time is modeled as an exponential distribution with a mean of 7 seconds.

Ganesh is evaluated along two axes: number of clients and WAN bandwidth. Higher loads are especially useful in understanding Ganesh's performance when the CPU or disk of the database server or proxy is the limiting factor. A previous study has shown that approximately 50% of the wide-area Internet bottlenecks observed had an available bandwidth under 10 Mbit/s [4]. Based on this work, the evaluation focuses on the WAN bandwidth of 5 Mbit/s with 66 ms of round-trip latency, representative of severely constrained network paths, and of 20 Mbit/s with 33 ms of round-trip latency, representative of a moderately constrained network path. Ganesh's performance at 100 Mbit/s with no added round-trip latency is also reported. This bandwidth, representative of an unconstrained network, is especially useful in revealing any potential overhead of Ganesh in situations where WAN band-

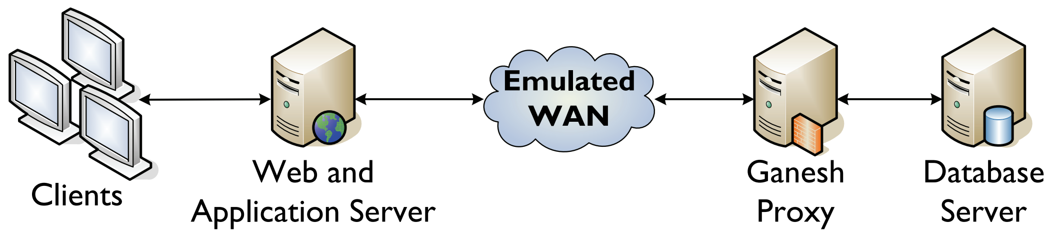


Figure 4.5: Experimental Setup

width is not the limiting factor. For each combination of number of clients and WAN bandwidth, results from the two configurations listed below were measured:

- *Native:* This configuration corresponds to Figure 4.3(a). Native avoids Ganesh's overhead in using a proxy and performing Java object serialization.
- *Ganesh:* This configuration corresponds to Figure 4.3(b). For a given number of clients and WAN bandwidth, comparing these results to the corresponding *Native* results gives the performance benefit due to Ganesh.

The metric used to quantify the improvement in throughput is the number of client requests that can be serviced per second. The metric used to quantify Ganesh's overhead is the average response time for a client request. For all of the experiments, the Ganesh driver used by the application server used a cache size of 100,000 items[1]. The proxy was effective in tracking the Ganesh driver's cache state; for all of the experiments, the miss rate on the driver never exceeded 0.7%.

Experimental Configuration

The experimental setup used for the benchmarks can be seen in Figure 4.5. All machines were 3.2 GHz Pentium 4s (with Hyper-Threading enabled.) With the exception of the database server, all machines had 2 GB of SDRAM and ran the Fedora Core Linux distribution. The database server had 4 GB of SDRAM.

Apache's Tomcat was used as both the application server that hosted the Java Servlets and the web server. Both benchmarks used Java Servlets to generate the

[1]As Java does not provide a `sizeof()` operator to determine memory usage, Java caches limit their size based on the number of objects. The maximum memory footprint of cache dumps taken at the end of the experiments never exceeded 212 MB.

dynamic content. The database server used the open source MySQL database. For the native JDBC drivers, the Connector/J driver provided by MySQL was used. The application server used Sun's Java Virtual Machine as the runtime environment for the Java Servlets. The `sysstat` tool [142] was used to monitor the CPU, network, disk, and memory utilization on all machines.

The machines were connected by a switched gigabit Ethernet network. As shown in Figure 4.5, the front-end web and application server was separated from the proxy and database server by a NetEm router [66]. This router controls the bandwidth and latency settings on the network. The NetEm router was a standard PC with two network cards running the Linux Traffic Control and Network Emulation software. The bandwidth and latency constraints were only applied to the link between the application server and the database for the native case and between the application server and the proxy for the Ganesh case. There was no communication between the application server and the database with Ganesh as all data flowed through the proxy. As the focus of the evaluation was on the WAN link between the application server and the database, there were no constraints on the link between the simulated test clients and the web server.

4.4.2 The `BBOARD` Benchmark

The `BBOARD` benchmark, also known as RUBBoS (Rice University Bulletin Board System) [10], models Slashdot, a popular technology-oriented web site. Slashdot aggregates links to news stories and other topics of interest found elsewhere on the web. In addition to being a news aggregation site, Slashdot also allows users to customize their view of the site. The site also serves as a bulletin board by allowing users to comment on the posted stories in a threaded conversation form. It is not uncommon for a story to gather hundreds of comments in a matter of hours. The `BBOARD` benchmark is similar to the site and models the activities of a user, including read-only operations such as browsing the stories of the day, browsing story categories, and viewing comments as well as write operations such as new user registration, adding and moderating comments, and story submission.

The benchmark consists of three different phases: a short warm-up phase, a runtime phase representing the main body of the workload, and a short cool-down phase. This chapter only reports results from the runtime phase. The warm-up phase is important in establishing dynamic system state, but measurements from that phase are not significant for the evaluation. The cool-down phase is solely for allowing the benchmark to shut down.

The warm-up, runtime, and cool-down phases took 2 minutes, 15 minutes, and 2 minutes respectively. Ganesh was tested with four client configurations where the number of test clients was set to 400, 800, 1200, and 1600. The benchmark is available in a Java Servlets and PHP version and has different datasets; Ganesh was evaluated using the Java Servlets version and the Expanded dataset.

The `BBOARD` benchmark defines two different workloads. The first, the *Authoring* mix, consists of 70% read-only operations and 30% read-write operations. The second, the *Browsing* mix contains only read-only operations.

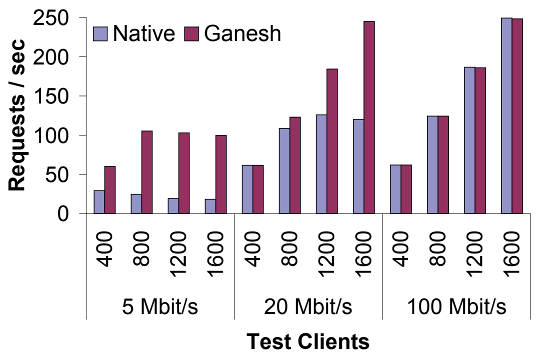
Authoring Mix

Figures 4.6 (a) and (b) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for `BBOARD`'s Authoring Mix.

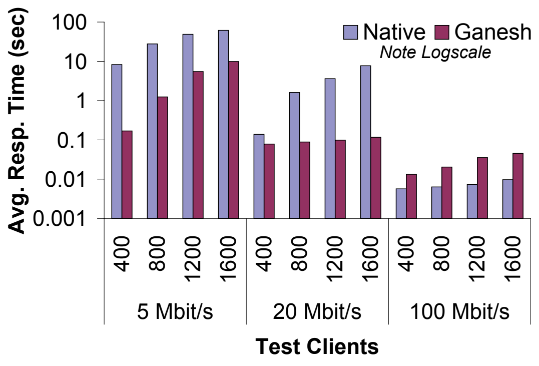
As Figure 4.6 (a) shows, Native easily saturates the 5 Mbit/s link. At 400 clients, the Native solution delivers 29 requests/sec with an average response time of 8.3 seconds. Native's throughput drops with an increase in the number of test clients as clients timeout due to congestion at the application server. Usability studies have shown that response times above 10 seconds cause the user to move on to other tasks [87, 169]. Based on these numbers, increasing the number of test clients makes the Native system unusable. Ganesh at 5 Mbit/s, however, delivers a twofold improvement with 400 test clients and a fivefold improvement at 1200 clients. Ganesh's performance drops slightly at 1200 and 1600 clients as the network becomes saturated. Compared to Native, Figure 4.6 (b) shows that Ganesh's response times are substantially lower with sub-second response times at 400 clients.

Figure 4.6 (a) also shows that for 400 and 800 test clients Ganesh at 5 Mbit/s has the same throughput and average response time as Native at 20 Mbit/s. Only at 1200 and 1600 clients does Native at 20 Mbit/s deliver higher throughput than Ganesh at 5 Mbit/s.

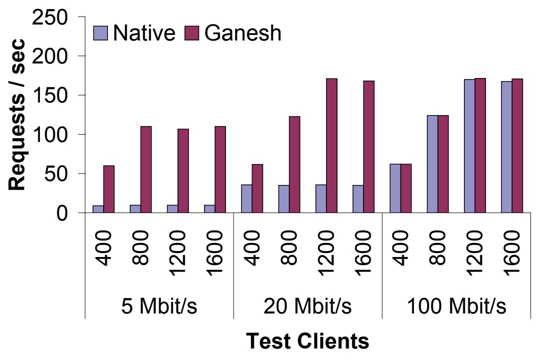
Comparing both Ganesh and Native at 20 Mbit/s, it can be seen that Ganesh



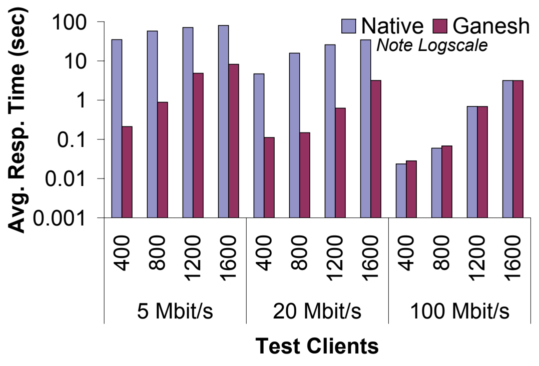
(a) Throughput: Authoring Mix



(b) Response Time: Authoring Mix



(c) Throughput: Browsing Mix



(d) Response Time: Browsing Mix

All results are the mean of three trials. The maximum standard deviation for throughput and response time was 9.8% and 11.9% respectively of the corresponding mean.

Figure 4.6: BBOARD Benchmark - Throughput and Average Response Time

is no longer bandwidth constrained and delivers up to a twofold improvement over Native at 1600 test clients. As Ganesh does not saturate the network with higher test client configurations, at 1600 test clients, its average response time is 0.1 seconds rather than Native's 7.7 seconds.

As expected, there are no visible gains from Ganesh at the high bandwidth case of 100 Mbit/s where the network is no longer the bottleneck. Ganesh, however, still tracks Native in terms of throughput. Due to the overhead of the proxy, the average response time is higher for Ganesh. The absolute difference, however, is in between 0.007 and 0.036 seconds and would be imperceptible to the end-user.

Browsing Mix

Figures 4.6 (c) and (d) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for BBOARD's Browsing Mix.

Regardless of the test client configuration, Figure 4.6 (c) shows that Native's throughput at 5 Mbit/s is limited to 10 reqs/sec. Ganesh at 5 Mbit/s with 400 test clients, delivers more than a sixfold increase in throughput. The improvement increases to over an elevenfold increase at 800 test clients before Ganesh saturates the network. Further, Figure 4.6 (d) shows that Native's average response time of 35 seconds at 400 test clients makes the system unusable. These high response times further increase with the addition of test clients. In comparison, even with the 1600 test client configuration, Ganesh delivers an average response time of 8.2 seconds.

Surprisingly, due to the data-intensive nature of the Browsing mix, Ganesh at 5 Mbit/s performs much better than Native at 20 Mbit/s. Further, as shown in Figure 4.6 (d), while the average response time for Native at 20 Mbit/s is acceptable at 400 test clients, it is unusable with 800 test clients and shows an average response time of 15.8 seconds. Like the 5 Mbit/s case, this response time increases with the addition of extra test clients.

Ganesh at 20 Mbit/s and both Native and Ganesh at 100 Mbit/s are not bandwidth limited. However, performance plateaus out after 1200 test clients due to the database CPU being saturated.

Filter Variant

Native performance for the BBOARD benchmark was surprising. At the bandwidth of 5 Mbit/s, Native performance was lower than what had been expected. It turned out the benchmark code that displays stories read all the comments associated with the particular story from the database and only then performed some post-processing to select the comments to be displayed. While this is exactly the behavior of SlashCode, the code base behind the Slashdot web site, the BBOARD benchmark was modified to perform some pre-filtering at the database. This modified benchmark, named the *Filter Variant*, models a developer who applies optimizations at the SQL level to transfer less data. The results from this benchmark are presented in Figure 4.7 and are briefly summarized below.

For the Authoring mix, at 800 test clients at 5 Mbit/s, Figure 4.7 (a) shows that Native's throughput increases by 85% when compared to the original benchmark while Ganesh's improvement is smaller at 15%. Native's performance drops above 800 clients as the test clients time out due to high response times. The most significant gain for Native is seen at 20 Mbit/s. At 1600 test clients, when compared to the original benchmark, Native sees a 73% improvement in throughput and a 77% reduction in average response time. While Ganesh sees no improvement when compared to the original, it still processes 19% more requests/sec than Native. Thus, while the optimizations were more helpful to Native, Ganesh can still improve performance.

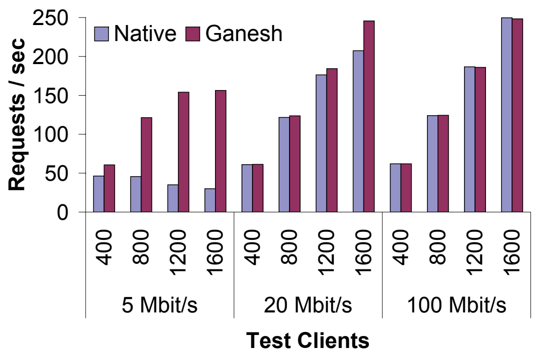
For the Browsing mix, Figures 4.7 (c) and (d) show that Native again benefits at the bandwidth of 20 Mbit/s. When compared to the original benchmark, at 400 test clients, Native sees a 53% throughput improvement and an 80% reduction in average response time. However, it still saturates the network and the addition of extra clients only increases the average response time. While the overall gain due to Ganesh at 20 Mbit/s decreases, Figure 4.7 (c) shows that it can still deliver more than a fourfold throughput improvement at 1600 test clients.

A side-effect of the benchmark optimization was that it decreased the load on the database CPU. Therefore, when comparing Figure 4.6 (c) and Figure 4.7 (c), it can be seen that Ganesh at 20 Mbit/s as well as Native and Ganesh at 100 Mbit/s, can deliver a higher throughput than the original benchmark.

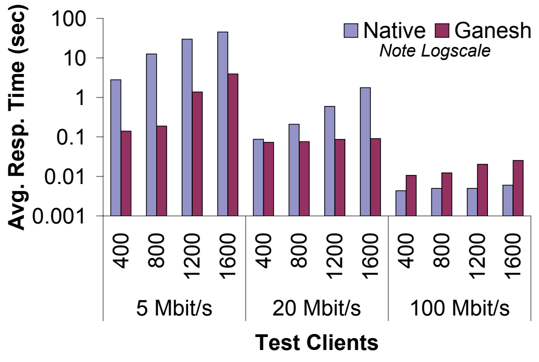
4.4.3 The `AUCTION` Benchmark

The `AUCTION` benchmark, also known as RUBiS (Rice University Bidding System) [10], models eBay, the online auction site. The eBay web site is used to buy and sell items via an auction format. The main activities of a user include browsing, selling, or bidding for items. Modeling the activities on this site, this benchmark includes read-only activities such as browsing items by category and by region, as well as read-write activities such as bidding for items, buying and selling items, and leaving feedback on completed transactions.

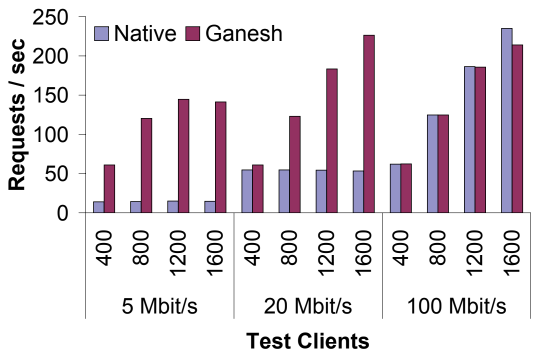
As with `BBOARD`, the benchmark consists of three different phases. The warm-up,



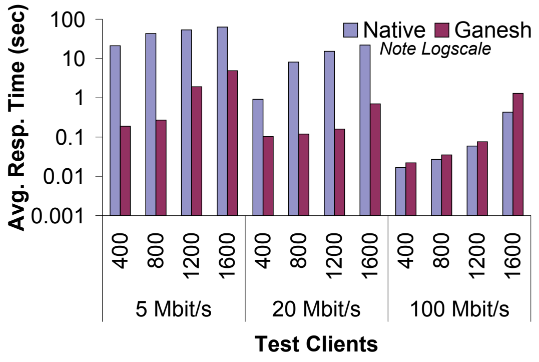
(a) Throughput: Authoring Mix



(b) Response Time: Authoring Mix



(c) Throughput: Browsing Mix



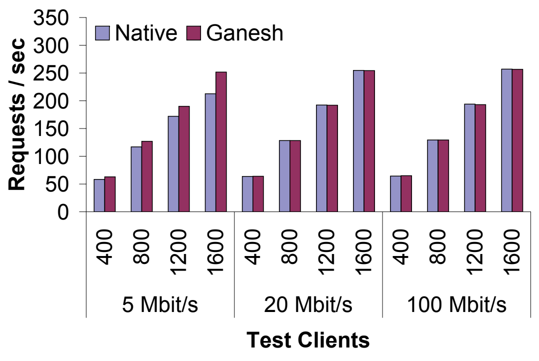
(d) Response Time: Browsing Mix

All results are the mean of three trials. The maximum standard deviation for throughput and response time was 7.2% and 11.5% respectively of the corresponding mean.

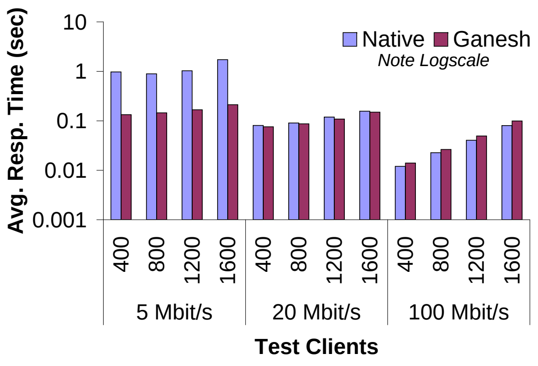
Figure 4.7: Filter `BBOARD` Benchmark - Throughput and Average Response Time

runtime, and cool-down phases for this experiment took 1 minute and 30 seconds, 15 minutes, and 1 minute respectively. Ganesh was tested with four client configurations where the number of test clients was set to 400, 800, 1200, and 1600. The benchmark is available in an Enterprise Java Bean (EJB), Java Servlets, and PHP version and has different datasets; Ganesh was evaluated with the Java Servlets version and the Expanded dataset.

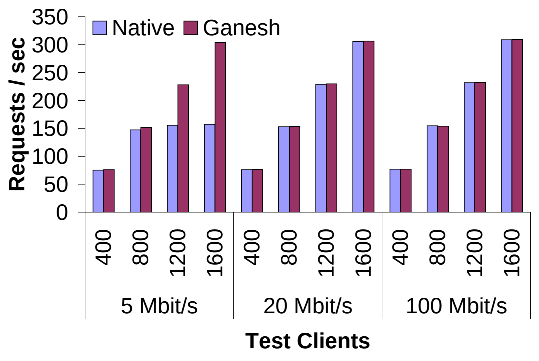
The `AUCTION` benchmark defines two different workloads. The first, the *Bidding* mix, consists of 70% read-only operations and 30% read-write operations. The second, the *Browsing* mix contains only read-only operations.



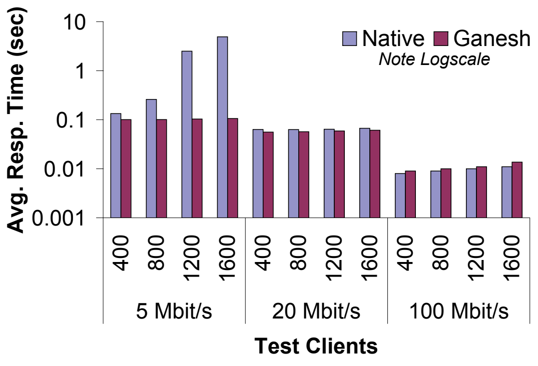
(a) Throughput: Bidding Mix



(b) Response Time: Bidding Mix



(c) Throughput: Browsing Mix



(d) Response Time: Browsing Mix

All results are the mean of three trials. The maximum standard deviation for throughput and response time was 2.2% and 11.8% respectively of the corresponding mean.

Figure 4.8: AUCTION Benchmark - Throughput and Average Response Time

Bidding Mix

Figures 4.8 (a) and (b) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients for AUCTION's Bidding Mix. As mentioned earlier, the Bidding mix consists of a mixture of read and write operations.

The AUCTION benchmark is not as data intensive as BBOARD. Therefore, most of the gains are observed at the lower bandwidth of 5 Mbit/s. Figure 4.8 (a) shows that the increase in throughput due to Ganesh ranges from 8% at 400 test clients to 18% with 1600 test clients. As seen in Figure 4.8 (b), the average response times for Ganesh are significantly lower than Native ranging from a decrease of 84% at 800 test clients to 88% at 1600 test clients.

Figure 4.8 (a) also shows that with a fourfold increase of bandwidth from 5 Mbit/s to 20 Mbit/s, Native is no longer bandwidth constrained and there was no performance difference between Ganesh and Native. However, with the higher test client configurations, the bandwidth used by Ganesh was lower than Native. Ganesh might still be useful in these non-constrained scenarios if bandwidth is purchased on a metered basis. Similar results were seen for the 100 Mbit/s scenario.

Browsing Mix

For `AUCTION`'s Browsing Mix, Figures 4.8 (c) and (d) present the average number of requests serviced per second and the average response time for these requests as perceived by the clients.

Again, most of the gains are observed at lower bandwidths. At 5 Mbit/s, Native and Ganesh deliver similar throughput and response times with 400 test clients. While the throughput for both remains the same at 800 test clients, Figure 4.8 (d) shows that Ganesh's average response time is 62% lower than Native. Native saturates the link at 800 clients and adding extra test clients only increases the average response time. Ganesh, regardless of the test client configuration, is not bandwidth constrained and maintains the same response time. At 1600 test clients, Figure 4.8 (c) shows that Ganesh's throughput is almost twice that of Native.

At the higher bandwidths of 20 and 100 Mbit/s, neither Ganesh nor Native are bandwidth limited and both deliver equivalent throughput and response times.

4.4.4 Structural vs. Rabin Similarity

In this section, the second question raised in Section 4.4 is addressed: How important is Ganesh's structural similarity detection relative to Rabin fingerprinting-based similarity detection? To answer this question, microbenchmarks and the `BBOARD` and `AUCTION` benchmarks were used. As Ganesh always performed better than Rabin fingerprinting, this section only presents the results from the `BBOARD` benchmark.

Benchmark	Orig. Size	Ganesh Size	Rabin Size
SelectSort_1	223.6 MB	5.4 MB	219.3 MB
SelectSort_2	223.6 MB	5.4 MB	223.6 MB

Table 4.2: Similarity Microbenchmarks

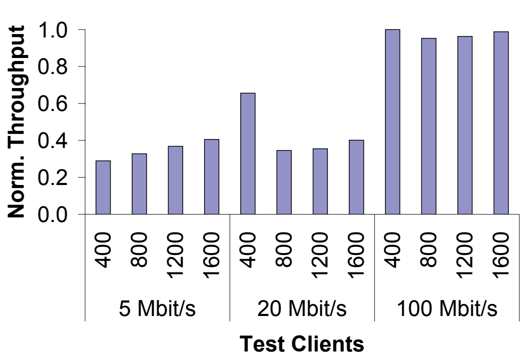
Microbenchmarks

Two microbenchmarks show an example of the effects of data reordering on Rabin fingerprinting algorithm. In the first micro-benchmark, *SelectSort*$_1$, a query with a specified sort order selects 223.6 MB of data spread over approximately 280 K rows. The query is then repeated with a different sort attribute. While the same number of rows and the same data is returned, the order of rows is different. In such a scenario, a large amount of similarity is expected between both results. As Table 4.2 shows, Ganesh's row-based algorithm achieves a 97.6% reduction while the Rabin fingerprinting algorithm, with the average chunk size parameter set to 4 KB, only achieves a 1% reduction. The reason, as shown earlier in Figure 4.2, is that with Rabin fingerprinting, the spans of data between two consecutive boundaries usually cross row boundaries. With the order of the rows changing in the second result and the Rabin fingerprints now spanning different rows, the algorithm is unable to detect significant similarity. The small gain seen is mostly for those single rows that are large enough to be broken into multiple chunks.

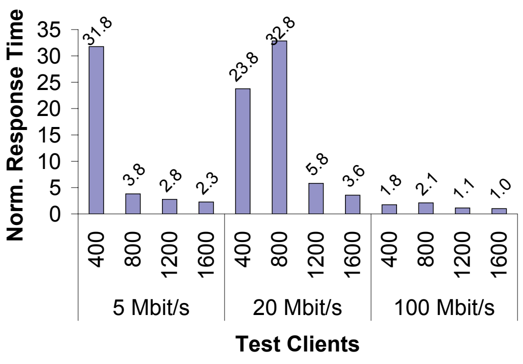
SelectSort$_2$, another micro-benchmark, executed the same queries but increased the minimum chunk size of the Rabin fingerprinting algorithm. As can be seen in Table 4.2, even the small gain from the previous microbenchmark disappears as the minimum chunk size becomes greater than the average row size. While these problems can be partially addressed by dynamically varying the parameters of the Rabin fingerprinting algorithm, this can be computationally expensive, especially in the presence of changing workloads.

Application Benchmarks

The `BBOARD` benchmark, described in Section 4.4.2, was executed on two versions of Ganesh: the first with Rabin fingerprinting used as the chunking algorithm and the



(a) Normalized Throughput
Higher is better



(b) Normalized Response Time
Higher is worse

For throughput, a normalized result greater than 1 implies that Rabin is better, For response time, a normalized result greater than 1 implies that Ganesh is better. All results are the mean of three trials. The maximum standard deviation for throughput and response time was 9.1% and 13.9% respectively of the corresponding mean.

Figure 4.9: Normalized Comparison of Ganesh vs. Rabin - BBOARD Browsing Mix

second with Ganesh's row-based algorithm. Rabin's results for the Browsing Mix are normalized to Ganesh's results and presented in Figure 4.9.

As Figure 4.9 (a) shows, at 5 Mbit/s, independent of the test client configuration, Rabin significantly underperforms Ganesh. This happens because of a combination of two reasons. First, as outlined in Section 4.2, Rabin finds less similarity as it does not exploit the result's structural information. Second, this benchmark contained some queries that generated large results. In this case, Rabin, with a small average chunk size, generated a large number of objects that evicted other useful data from the cache. In contrast, Ganesh was able to detect these large rows and correspondingly increase the size of the chunks. This was confirmed as cache statistics showed that Ganesh's hit ratio was roughly three times that of Rabin. Throughput measurements at 20 Mbit/s were similar with the exception of Rabin's performance with 400 test clients. In this case, Ganesh was not network limited and, in fact, the throughput was the same as 400 clients at 5 Mbit/s. Rabin, however, took advantage of the bandwidth increase from 5 to 20 Mbit/s to deliver a slightly better performance. At 100 Mbit/s, Rabin's throughput was almost similar to Ganesh's, as bandwidth was no longer a bottleneck.

The normalized response time, presented in Figure 4.9 (b), shows similar trends.

At 5 and 20 Mbit/s, the addition of test clients decreases the normalized response time as Ganesh's average response time increases faster than Rabin's. However, at no point does Rabin outperform Ganesh. Note that at 400 and 800 clients at 100 Mbit/s, Rabin does have a higher overhead even when it is not bandwidth constrained. As mentioned in Section 4.2, this is due to the fact that Rabin has to hash each `ResultSet` twice. The overhead disappears with 1200 and 1600 clients as the database CPU is saturated and limits the performance of both Ganesh and Rabin.

4.4.5 Proxy Overhead

In this section, the third question raised in Section 4.4 is addressed: Is the overhead of Ganesh's proxy-based design acceptable? To answer this question, one needs to concentrate on its performance at the higher bandwidths. The `BBOARD` and `AUCTION` benchmarks showed that Ganesh, when compared to Native, can deliver a substantial throughput improvement at lower bandwidths. It is only at higher bandwidths that latency, measured by the average response time for a client request, and throughput, measured by the number of client requests that can be serviced per second, overheads would be visible.

Looking at the Authoring mix of the original `BBOARD` benchmark, there are no visible gains from Ganesh at 100 Mbit/s. Ganesh, however, still matches Native in terms of throughput. While the average response time is higher for Ganesh, the absolute difference is between 0.01 and 0.04 seconds and would be imperceptible to the end-user. The Browsing mix shows an even smaller difference in average response times. The results from the filter variant of the `BBOARD` benchmarks are similar. Even for the `AUCTION` benchmark, the difference between Native's and Ganesh's response time at 100 Mbit/s was never greater than 0.02 seconds. The only exception to the above results was seen in the filter variant of the `BBOARD` benchmark where Ganesh at 1600 test clients added 0.85 seconds to the average response time. Thus, even for much faster networks where the WAN link is not the bottleneck, Ganesh always delivers throughput equivalent to Native. While some extra latency is added by the proxy-based design, it is usually imperceptible.

4.5 Summary

The growing use of dynamic web content generated from relational databases places increased demands on WAN bandwidth. Traditional caching solutions for bandwidth and latency reduction are often ineffective for such content. This chapter shows that the impact of WAN accesses to databases can be substantially reduced through the Ganesh architecture without any compromise of a database's strict consistency semantics. The essence of the Ganesh architecture is the use of computation at the edges to reduce communication through the Internet. Borrowing techniques from the storage world, Ganesh is able to use content-addressable techniques to detect similarity with previous results and send compact recipes of results rather than full results. The design uses interposition to achieve complete transparency: clients, application servers, and database servers are all unaware of Ganesh's presence and require no modification.

The experimental evaluation confirms that Ganesh, while conceptually simple, can be highly effective in improving throughput and response time. Using two benchmarks representative of dynamic web sites, the results show that Ganesh can increase end-to-end throughput as much as tenfold in low bandwidth environments. The experiments also confirm that exploiting the structure present in database results to detect similarity is crucial to this performance improvement.

Chapter 5

Improving Mobile Database Access Without Degrading Consistency

5.1 Introduction

Relational databases lie at the core of many business processes such as inventory control, order entry, customer relationship management, asset tracking, and resource scheduling. A key contributor to this success is a database's ability to provide a consistent view of shared data across many geographically dispersed users, even in the face of highly concurrent updates at fine granularity. Transactional semantics are part of this consistency model.

Preserving consistency with acceptable performance under conditions of weak connectivity is a difficult challenge. Indeed, it has been widely accepted since the early days of mobile computing that shared data access involves a fundamental tradeoff between consistency, good performance, and tolerance of poor network quality [27]. This has led to a variety of approaches [6, 9, 59, 133] that relax consistency. However, failing to preserve consistency undermines the very attribute that makes databases so attractive for many applications.

This chapter describes a new approach to mobile database access that avoids the need to compromise consistency. In other words, it shows how to achieve good client performance under conditions of weak connectivity without degrading consistency. The critical insight is that content-addressable techniques can use the disk storage and processing power of a mobile client to compensate for weak connectivity. The following sections report on the detailed design, implementation, and experimental validation of this approach in a system called *Cedar.* Unlike the Ganesh system, described in Chapter 4, which was designed to optimize multi-tiered architectures, Cedar is targeted towards a different category of applications. It is expected that Cedar will benefit a broad category of applications for mobile clients including those for mobile commerce [130, 166], traveling sales people, mobile customer relationship management [134], and disaster recovery [47].

Cedar uses a simple client-server design in which a central server holds the master copy of the database. Cedar's organizing principle is that *even a stale client replica can be used to reduce data transmission volume.* It accomplishes this through the use of content-addressable techniques. The volume reduction is greatest when the client replica and the master copy are identical. However, even a grossly divergent replica will not hurt consistency; at worst, it will hurt performance. This organizing principle allows Cedar to use an optimistic approach to solving the difficult problem of database replica control [20, 21]. At infrequent intervals when a client has excellent connectivity to the server (which may occur hours or days apart), its replica is refreshed from the master copy. Cedar primarily targets laptop-class machines as mobile clients, but this work also explore Cedar's applicability to PDA-class mobile clients. Since laptop disk sizes can be 100GB or more today, replicating the entire database is often feasible. For databases that are too large, Cedar provides support for creating a useful partial replica. Cedar is implemented in Java, and runs on Linux as well as Windows. It should also work on any other operating system that supports Java.

On a Cedar client, a `select` query is first executed on the local replica to obtain a tentative result. Using content-addressable techniques, the client constructs a compact summary of the tentative result and sends it to the server along with the query. The query is re-executed at the server to obtain the authoritative result. The server then constructs a compact summary of the difference between the tentative and authoritative results and sends it back to the client. If the client and server are in sync, there will be no difference between the results. The further out of sync they are, the larger the difference in results. However, the difference is never larger than the size of the result that would be generated without Cedar. By applying the difference to the tentative result, the client reconstructs the authoritative result and returns it as the response to the `select` query. Since applications never see tentative results, they perceive the same database behavior that they would without Cedar. `update` queries go directly to the server without special handling.

In order to achieve transparency, Cedar's design and implementation pay careful attention to practical considerations. First, Cedar is completely transparent to client applications and hence requires no changes to them. Second, databases from different vendors can be used at the server and client. This simplifies interoperability, and allows client and server software to be independently optimized for their respective hardware resources and workloads. Third, Cedar does not require source code access to any database.

5.2 Background on Wireless Wide-Area Networks

Millions of users today own personal computing devices that can access, query, and update data stores and databases over wireless networks. The increasing availability of wide-area connectivity options such as cellular GPRS/EDGE, EVDO, and WiMax has encouraged the notion of access to data anywhere and anytime [80]. However, many of these wireless technologies exhibit high variability in peak theoretical throughput, as shown in Table 5.1. End-to-end latency is also highly variable. To make matters worse, recent studies have shown that achievable throughput is often only 25–65% of the theoretical maximum [40, 42], and that large drops in throughput are seen during peak usage periods [98].

Wireless Wide-Area Network (WWAN) technologies remain important for the foreseeable future in spite of the growing popularity of WiFi (802.11) technology. First, WiFi coverage is limited to compact areas where dense base station infrastructure can be economically sustained. In contrast, WWAN technologies require much less dense infrastructure that can often be piggybacked on existing cell phone infrastructure. Hence, they are economically sustainable over much larger geographic areas. Second, organizational and business considerations may preclude use of WiFi. For example, a user may not subscribe to the wireless service provider of a hotspot. As another example, corporate security guidelines may prohibit a salesman from using the WiFi coverage available at a customer site; hence unplanned use of WiFi might not be possible. Third, existing WiFi infrastructure may be damaged or unusable in situations such as disaster recovery and military operations. Rapid setup of new

Wireless Technology	Peak Theoretical Throughput
GPRS	30 – 89 Kbit/s
GPRS/EDGE	56 – 384 Kbit/s
CDMA (1xRTT)	144 Kbit/s (downlink)
	64 Kbit/s (uplink)
EVDO (Rev. 0)	2,500 Kbit/s (downlink)
	154 Kbit/s (uplink)
EVDO (Rev. A)	3,100 Kbit/s (downlink)
	1,800 Kbit/s (uplink)
WiMax (802.16)	500 Kbit/s – 2,000 Kbit/s
	(per connection)

Table 5.1: Wireless WAN Technologies Prevalent in 2006–07

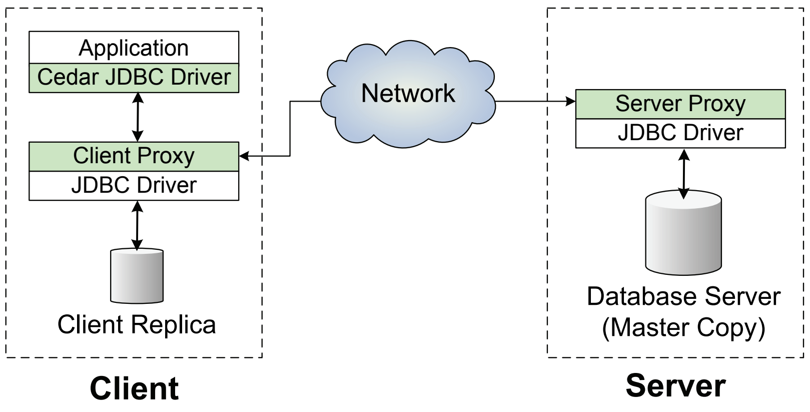
wireless infrastructure is feasible only if it is sparse. This typically implies reliance on WWAN technologies.

5.3 Achieving Transparency

As mentioned earlier, a key factor influencing Cedar's design was the need for it to be completely transparent to both applications and databases. This lowers the barrier for adoption of Cedar, and broadens its applicability. Cedar uses a proxy-based design to meet this goal. The task is simplified by the fact that Cedar is purely a performance enhancement, and not a functionality or usability enhancement.

5.3.1 Application Transparency

Cedar does not require access to application source code. Instead, it leverages the fact that most applications that access databases are written to a standardized API. This compact API (JDBC in the current implementation) is a convenient interposition point for new code. Figure 5.1 shows how Cedar is interposed. On the application end, the native database driver is replaced by Cedar's drop-in replacement driver that implements the same API. The driver forwards all API calls to a co-located proxy.



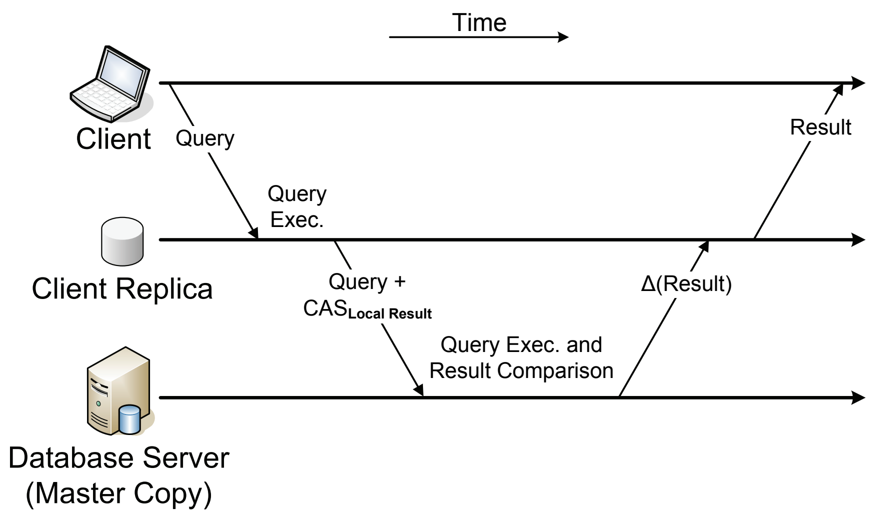
This figure shows how Cedar is transparently interposed into an existing client-server system that uses JDBC. The colored boxes represent Cedar components.

Figure 5.1: Proxy-Based Cedar Implementation

This architecture is very similar to the one used by Ganesh, with the only exception being the addition of the client-side proxy.

Figure 5.2 shows how these components interact when executing a `select` query. The client proxy first executes the query on the client replica and generates a *recipe*, a compact content-addressable description of the result. It then forwards the original query and the recipe to the database server. Note that while one could combine Cedar's database driver and proxy for performance reasons, separating them allows the proxy to be shared by different applications. While Cedar currently interposes on the JDBC interface, it can also support ODBC-based C, C++, and C# applications by using an ODBC-to-JDBC bridge.

Note that as Cedar's focus is on improving performance without compromising consistency, it assumes that at least weak or limited connectivity is available on the client. It is not targeted towards applications where database access is required while disconnected. Cedar's central organizing principle, as stated earlier in Section 5.1, is that even a stale client replica can be used to reduce data transmission volume. There is no expectation that a tentative result from a client replica will be correct, only that it will be "close" to the authoritative result at the server. Thus, the output of a client replica is never accepted without first checking with the server. Cedar uses a "check-on-use" or detection-based approach to replica control as in AFS-1 [126]. A "notify-on-change" or avoidance-based approach was rejected even though it has been shown


Time
Client
Query
Result
Query Exec.
Client Replica
Query + $\text{CAS}_{\text{Local Result}}$
Δ(Result)
Query Exec. and Result Comparison
Database Server (Master Copy)

This figure maps Cedar's architecture to the protocol executed for a `select` query.

Figure 5.2: Cedar Protocol Example

to have superior performance in file systems such as AFS-2 [68] and its successors. There were multiple reasons for this decision. First, it reduces wasteful invalidation traffic to a weakly-connected client in situations where most of the invalidations are due to irrelevant update activity. Second, performance issues may arise on a busy database server that has to actively track state on a large number of clients. Finally, it simplifies the implementation at both the client and the server

5.3.2 Database Transparency

Similar to Ganesh's architecture, Cedar also introduces a proxy co-located with the database server in order to be completely transparent to the database. As Figure 5.1 shows, the client's query and recipe is received by the server proxy. The server proxy could either be co-located with the server or placed on a separate machine. The only restriction is that it should be well connected to the server. This proxy re-executes the query on the server and generates a recipe of the result. It then compares the client's and server's recipe, eliminates commonality, and only sends back the differences.

In this way, the database server is totally unaware of the client replica — it is only the server proxy that is aware. Note that the differences received from the server cannot be used to update the client replica. A query might only select a subset of the columns of each accessed table and the result would therefore only contain a partial copy of the original rows. Using it to update the replica and tracking the partially updated rows would be infeasible without requiring extensive database modifications.

On both the client and the database server, Cedar's functionality was not directly incorporated into the database. Instead, Cedar itself uses the JDBC API to access both the server and client databases. This design choice to allow Cedar to be completely independent of the underlying database has a number of advantages. First, the increased diversity in choice can be very useful when a mobile client lacks sufficient resources to run a heavyweight database server. In such situations, a lightweight or embedded database could be used on the mobile client while a production database such as MySQL or Oracle could be used on the server. Cedar's interoperability has been tested with MySQL and SQLite [138], an embedded database with a small footprint. For queries that use non-standard SQL extensions only supported by the server, Cedar transparently ignores the failure of the replica to interpret such queries and forwards them verbatim to the database server. Second, it makes a database's physical layout completely transparent to Cedar. Thus, a centralized database in a data center can be easily replicated for scalability reasons without modifying Cedar.

5.3.3 Adaptive Interpositioning

Although a Cedar client is optimized for weakly-connected operation, it may sometimes experience good connectivity. In that situation, its low-bandwidth optimizations may be counter-productive. For example, as seen in Figure 5.2, the latency added by executing the query twice may exceed the savings from reduced data transmission volume. There may also be situations in which a mobile client is so resource-poor that Cedar's approach of using its disk storage and processing power to compensate for weak connectivity is infeasible.

Cedar therefore adapts its behavior to available bandwidth. When the client proxy detects a fast network, it stops handling new queries. The Cedar JDBC driver then transparently switches to direct use of the server proxy. This eliminates the computational overhead of using content-addressable techniques and the cost of an extra hop through the client proxy. Transactions that are currently in progress through the proxy are completed without disruption. If, at a later point in time, the client proxy detects unfavorable network conditions, it is once again transparently interposed into the query handling path. The client proxy uses the packet-pair based IGI/PTR tool [69] to measure bandwidth. Experience with the system indicates that relatively coarse bandwidth estimation is adequate for triggering proxy state changes.

While the server proxy could also have been bypassed, this would require dynamic substitution of the native JDBC driver for Cedar's JDBC driver. It would be difficult to implement this in a manner that does not disrupt queries in progress. To preserve transparency, this optimization was not implemented. Fortunately, the measurements shown in Section 5.6 indicate that the server proxy overhead is low.

Implementing adaptation with respect to the staleness of the client replica would also be easy to implement. A very stale replica is likely to produce tentative results that have little commonality with authoritative results. In that situation, it is better to stop using the client replica until it can be brought in sync with the server. The implementation approach is to have the client proxy track the savings from eliding commonality. When the savings are consistently below a threshold, the interposition of the client proxy is removed. Queries then go directly to the server proxy, as described above.

5.4 Detecting Similarity

The next three sections examine specific aspects of Cedar's approach to detecting similarity. Sections 5.4.1 and 5.4.2 describe how Cedar detects commonality across results and constructs compact recipes. Section 5.4.3 describes Cedar's support for populating the client replicas from where similarity is extracted.

5.4.1 Exploiting Structure in Data

Rabin fingerprinting is an excellent tool for detecting commonality across opaque objects [29, 108]. An attractive property of this tool is that it finds commonality even after in-place updates, insertions, and deletions are performed on an object. Unfortunately, the data boundaries found by Rabin fingerprinting rarely aligns with natural boundaries in structured data. This makes it less attractive for database output, which is typically organized by rows and columns. Simple reordering of rows,

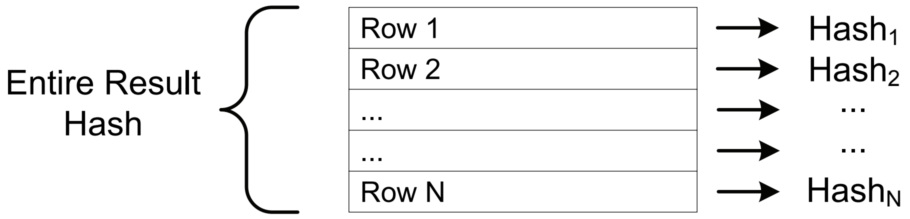


Figure 5.3: Hashing in Cedar

as might occur from a `SORT BY` clause in a SQL statement, can degrade the ability of Rabin fingerprinting to find commonality.

Cedar therefore uses the approach, also used by Ganesh in Chapter 4, which worked better than Rabin fingerprinting: the end of each row in a database result is used as a natural chunk boundary. It is important to note that Cedar's use of a tabular structure in results only involves shallow interpretation of Java's result set data type. There is no deeper semantic interpretation such as understanding data type values, result schema, or integrity constraints.

5.4.2 Generating Compact Recipes

As Figure 5.3 shows, Cedar finds commonality by hashing at two granularities: the entire result and each individual row. This makes Cedar's approach to detecting commonality more amenable for selections and joins. For operations such as aggregates, if Cedar detects that the size of the recipe is larger than the tentative result, it chooses to forgo any content-addressable optimizations and forwards only the query to the database server.

However, for large results, the hash per row can still add up to a sizable recipe. Cedar therefore uses a simple but effective approach to reducing the per-row contribution. This approach recognizes that the entire result hash is sufficient to ensure correctness. Since the sole purpose of a per-row hash is to support comparisons at fine granularity, a subset of the bits in that hash is adequate to serve as a strong hint of similarity. Hence, the recipe of a result only uses the lower n bits of each per-row hash. When two per-row hashes match, the server proxy assumes that the corresponding row already exists in the tentative result. If this is an erroneous assumption it will be detected in the final step of the comparison process.

After the comparison of tentative and authoritative results is complete, the server sends the client truncated hashes for the common rows found, data for the rows known to be missing on the client, and an *untruncated* hash of the authoritative result. The client reconstructs its version of the authoritative result and verifies that its entire hash matches that sent by the server. In the rare event that there is a mismatch, the authoritative result is fetched verbatim from the server.

A low value of n increases the probability of a collision, and hence many verbatim fetches. On the other hand, a large value of n renders a CAS description less compact. Given a hash of length n bits, the probability p of a collision amongst m other hashes is

$$\begin{aligned} p &= 1 - \Pr\{\text{No Collision}\} \\ &= 1 - \left(\frac{2^n - 1}{2^n}\right)^m \end{aligned}$$

Cedar currently uses a value of $n = 32$ bits. As only result hashes from the *same* query are compared, m is expected to be small. However, even with $m = 10,000$, the probability of seeing a collision and having to refetch verbatim results from the server would only be $p = 2.3 \times 10^{-6}$. It should be noted that a collision was never encountered during the development and evaluation of Cedar.

5.4.3 Populating Client Replicas

As Cedar extracts similarity from results generated by the client replica, correctly populating the replicas is critical to improving performance. Ideally, a client replica would contain a complete copy of the database. However, some mobile clients may be too resource-poor to support a full database replica. Even when resources are adequate, there may be privacy, security, or regulatory concerns that restrict copying an entire database to an easily-compromised mobile client. These considerations lead to the need for creating partial client replicas. Cedar's challenge is to help create a partial replica that is customized for a mobile user. Fortunately, there are many business and social practices that are likely to result in temporal locality in the query stream at a Cedar client. A partial replica that is tuned to this locality will help Cedar achieve good performance.

For example, consider a company that assigns a specific geographic territory or a specific set of customers to each of its salesmen. The query stream generated by a salesman's mobile client is likely to be confined to her unique territory or unique set of customers. Although the company's customer database may be enormous, only a small subset of it is likely to be relevant to a particular salesman. Similar reasoning applies to an insurance adjuster who is visiting the homes of customers affected by a natural disaster. There is typically a step in the workflow of claim handling that assigns a specific adjuster to each claim. These assignments are typically made at the granularity of many hours, possibly a whole day. For the entire duration of that period, an adjuster is likely to only generate queries that pertain to her assignments.

Cedar's solution is inspired by Coda's use of *hoarding* to support disconnected operation [73]. Cedar faces a simpler problem than Coda because its hoarding does not have to be perfect. If a partial replica cannot produce the correct response to a query, the only consequence in Cedar is poor performance. No disruptive error handling is needed relative to availability or consistency.

Hoard Granularity

Previous implementations of hoarding have typically operated at the granularity of individual objects, such as files or mail messages. In contrast, Cedar hoards data at the granularity of tables and table fragments. Cedar's approach is based on the long-established concepts of *horizontal fragmentation* [39] and *vertical fragmentation* [95], as shown in Figure 5.4. Horizontal partitioning preserves the database schema. It is likely to be most useful in situations that exploit temporal locality, such as in the sales and insurance examples mentioned earlier. Hoarding in Cedar is done through horizontal fragmentation. If a query references a table that is not hoarded on a mobile client, Cedar forwards that query directly to the database server.

Database Hoard Profiles

Hoarding at a Cedar client is controlled by a *database hoard profile* that expresses hoard intentions within the framework of a database schema. A database hoard profile

Original Table

ID	Name	Address	Zip	Email
1	John Doe	412 Avenue	15213	jd2@eg.com
2	Richard Roe	396 Road	15217	rr@eg.com
3	Mary Major	821 Lane	15232	mm@eg.com
4	John Stiles	701 Street	94105	js@eg.com
5	Jane Doe	212 Way	94112	jd@eg.com

Horizonal Partitioning

ID	Name	Address	Zip	Email
1	John Doe	412 Avenue	15213	jd2@eg.com
2	Richard Roe	396 Road	15217	rr@eg.com
3	Mary Major	821 Lane	15232	mm@eg.com

ID	Name	Address	Zip	Email
4	John Stiles	701 Street	94105	js@eg.com
5	Jane Doe	212 Way	94112	jd@eg.com

Vertical Partitioning

ID	Name	Address
1	John Doe	412 Avenue
2	Richard Roe	396 Road
3	Mary Major	821 Lane
4	John Stiles	701 Street
5	Jane Doe	212 Way

ID	Zip	Email
1	15213	jd2@eg.com
2	15217	rr@eg.com
3	15232	mm@eg.com
4	94105	js@eg.com
5	94112	jd@eg.com

Figure 5.4: Horizontal and Vertical Partitioning

is an XML file that contains a list of weighted *hoard attributes*. Each hoard attribute specifies a single database relation and a predicate. Cedar uses these predicates to horizontally partition a relation. A database hoard profile may be manually created by a user. However, it is much more likely to be created by a database administrator or by using the tool described below.

Figure 5.5 illustrates the syntax of hoard profiles. Each hoard attribute is also associated with a weight that indicates its importance. Cedar uses this weight to prioritize what should be hoarded on a mobile client that has limited storage. As tentative results are always verified with the server, supporting external referential constraints, such as foreign keys, is not required for correctness. However, if needed, hoard profiles can be modified to support referential cache constraints [7] by extending hoard predicates to support cross-table relationships.

Tools for Hoarding

To specify hoard attributes, a user needs to be aware of the SQL queries that she is likely to generate while weakly connected. It is often the case that a user does not

```
USERS(nickname, fullname, password, zip)
```

(a) Schema of table named `USERS`

```
<attr table="USERS" predicate="zip >= 15222 AND zip <= 15295"
weight="10"/>
```

(b) Hoard attribute

A hoard attribute is expressed in a notation that is almost identical to the predicate of a `WHERE` clause in a `SELECT` query. The hoard attribute in (b) would cache all rows of the table specified in (a) whose zip codes lie between 15222 and 15295.

Figure 5.5: Syntax of Hoard Attributes

```
INSERT INTO users VALUES ("jd_nick", "John Doe", "jd_password", 15213)
```

(a) Original query from log

```
INSERT INTO users VALUES ("string", "string", "string", number)
```

(b) Abstracted query produced

Figure 5.6: Query Abstraction by SLM

directly specify SQL queries, but indirectly generates them through an application. To help the user in these situations, Cedar is accompanied by a tool called *SQL Log Miner (SLM).* This tool analyzes query logs to aid in the creation of database hoard profiles. It first abstracts queries into templates by removing unique user input, as shown in Figure 5.6. It then analyzes the abstracted log to determine unique queries and outputs them in frequency sorted order. SLM is also able to use a database's `EXPLAIN` feature to display queries that generate the largest results.

SLM was used to analyze database logs from a number of applications traces including an auction benchmark [10], a bulletin board benchmark [10], and `datapository.net`, a web-enabled database used for network trace analysis by the academic community. As Table 5.2 shows, the number of unique queries was very small relative to the total number of queries. The data in Table 5.2 suggests that it may be tractable

Application	Total Queries	Unique Queries	Unique %
AUCTION	115,760	41	0.04 %
BBOARD	131,062	59	0.05 %
datapository.net	9,395,117	278	0.003 %

Table 5.2: Unique Queries within Workloads

to extend SLM to automate hoard profile creation. Contextual information [111] and data mining [164] may be additional sources of input for this automation task.

Refreshing Stale Client Replicas

As the performance improvement seen from using Cedar is dependent on finding similarity between the client replica and that database, it is advantageous to prevent large divergence in their stored state. For this purpose, Cedar offers two mechanisms for bringing a client replica in sync with the database server. If a client has excellent connectivity, a new replica can be created or a stale replica updated by simply restoring a database dump created using the client's hoard profile. Although this is bandwidth-intensive, it tends to be faster and is the preferred approach.

To handle extended periods of weak connectivity, Cedar also provides a log-based refresh mechanism. The database server continuously maintains a timestamped update log. This can be accomplished without requiring any database changes as all production database servers support logging today. Since logging is a lightweight operation, it typically has less than a 1% impact on performance [93, page 375]. When a client detects available bandwidth, it can obtain the log from the server and apply all the updates since the last refresh. Bandwidth detection mechanisms can ensure that the log fetch does not impact foreground workloads. The server allocates a finite amount of storage for its log, and recycles this storage as needed. It is the responsibility of clients to obtain log entries before they are recycled. Once a log entry is recycled, any client that needs it has to restore its entire replica using the method described in the previous paragraph. This refresh mechanism is not supported for partial replicas as update queries in the log might require data that is not cached on the client.

5.5 Experimental Methodology

How much of a performance win can Cedar provide? The answer clearly depends on the workload, the network quality, and on the commonality between the client replica and the server. For resource-poor clients, it can also depend on the computational power of the client. To obtain a quantitative understanding of this relationship, experiments were conducted using both micro- and macro-benchmarks. This section presents the benchmarks, the evaluation procedure, and the experimental setup. Results of the experiments are presented in Sections 5.6 and 5.7.

5.5.1 Benchmarks

Microbenchmarks

Sensitivity analysis was used to explore how Cedar performs with different hit rates on the client replica. A number of single-query microbenchmarks were executed where both the amount of data selected by the query and the fraction of the result data that was available on the mobile client was varied. The queries in each microbenchmark fetched approximately 0.1, 0.5, and 1 MB of data with the results being equally distributed over 100 rows. For each of the microbenchmarks, the fraction of "fresh" data present on the mobile client was varied between 0% and 100%. For example, in the 60% case, the client replica was populated with 60% of data that would match the authoritative result generated by the server. The remaining 40% was explicity modified such that there would be no match. As the client is unaware of the data that might be stale, it still has to hash the entire tentative result to generate the recipe. The two extreme hit-ratios of 100% and 0% give an indication of best-case performance and the system overhead.

The MobileSales Benchmark

The unavailability of any widely used benchmark for evaluating mobile access to databases led to the creation of MobileSales, a new benchmark based on the TPC-App [155] benchmark from the Transaction Processing Performance Council. While TPC-App is targeted towards an application server environment, I believe that a modified version of this benchmark is also applicable to the mobile scenario.

Interaction Type	Percentage
Create New Customer	1%
Change Payment Method	5%
Create Order	50%
Order Status	5%
View New Products	7%
View Product Detail	30%
Change Item	2%

Table 5.3: Default Mix of Client Interactions

The TPC-App benchmark consists of an online distributor that allows clients to interact with a sales system. The workload consists of a set of *interactions*: clients can add new customers, create new orders, change payment types, check on the status of previous orders, view new products the distributor might have recently added, look at detailed descriptions of products, and make changes to the product catalog.

Individual interactions can exhibit dependencies on each other. For example, the 'Create Order' interaction takes as input a customer's unique identifier that could have previously been created by a 'Create New Customer' interaction. Some interactions can also be composed of multiple queries. For example, the 'Create Order' interaction takes as input a customer's ID, shipping information, the items and quantities requested, and the payment information. The interaction will validate the customer's ID and then check if the shipping information already exists. If not, a new address is entered into the database. It will then verify whether the customer qualifies for discounts and if the order can be satisfied by the current inventory. Only then is the order placed. However, interactions such as 'View Product Detail' only use a single query to fetch the product descriptions for a number of items. More details on the database schema, interaction types, test workload, query types, and dataset used in MobileSales can be found in the benchmark specification [155].

Each client's test run is defined by a time period that specifies the length of benchmark execution and a *mix* that specifies the probability of each interaction type. The default mix is shown in Table 5.3. There is no think time between interactions. The benchmark dataset can be scaled with respect to the number of customers. The dataset size is defined by $(0.8 \times S + 2006)$ MB where S is the scale factor.

While the focus of TPC-App is to test application server performance, I believe that the model behind the benchmark's interactions is representative of a number of mobile use scenarios including traveling salesmen, insurance adjusters, and customer relation management (CRM) applications. MobileSales therefore retains the same workload (dataset, queries, and interaction types) but, unlike TPC-App, allows mobile clients to directly access the database instead of being mediated through the application server. This is not uncommon today and is recommended for enterprise applications on mobile clients [168].

The performance of an individual mobile client is measured in terms of throughput (total number of interactions) and latency (average interaction completion time). MobileSales can also execute clients on separate load generator machines. Apart from modeling concurrent database access, updates made by these load clients ensure that the client replica diverges from the server as the benchmark progresses.

5.5.2 Experimental Procedure

For both benchmarks, the experimental setup shown in Figure 5.7 was used. Both real and emulated networks were used to evaluate Cedar's performance. For the real Wireless WAN, a Novatel Wireless (Merlin S720) PCMCIA card was used to connect the laptop directly to the database server over a Sprint EVDO (Rev. A) network link. While the theoretical throughput for this network is 3,100 Kbit/s down and 1,800 Kbit/s up, Sprint advertises 600–1400 Kbit/s down and 350–500 Kbit/s up. Iperf [147] and *ping* were used to measure the bandwidth and round trip latency between the EVDO-enabled laptop and the database server. The network was probed every 10 minutes over an approximately 13 hour period. The throughput was averaged over a 20 second transfer and the round-trip latency over 25 pings. The measurements, presented in Figure 5.8, showed an average download and upload bandwidth of 535 and 94 Kbit/s and an average round-trip latency of 112 ms.

Based on these measurements and the theoretical throughput of other wireless technologies shown in Table 5.1, Cedar was evaluated with emulated networks of 100 Kbit/s (representative of CDMA 1xRTT), 500 Kbit/s and 1 Mbit/s (representative of EVDO), and 2 Mbit/s (representative of WiMax). The round-trip time on each emulated network was varied between 33, 66, and 100 ms. The emulated WAN was implemented as a NetEm bridge [66]. As the evaluation's focus was on the per-

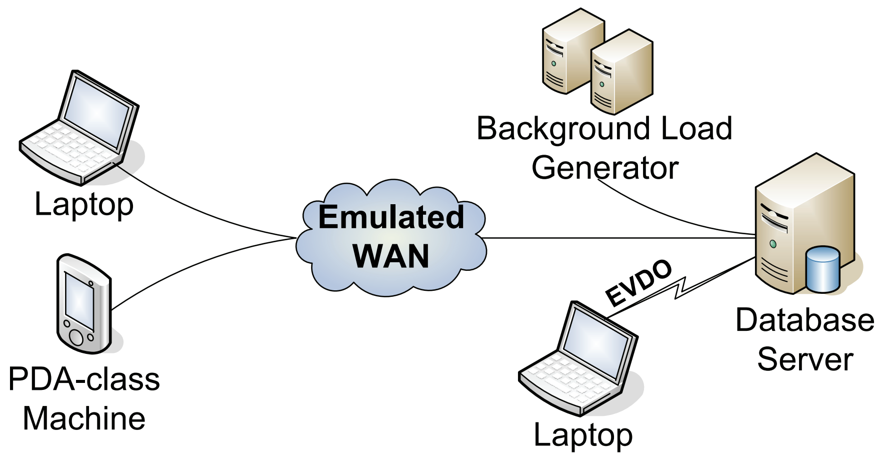


Figure 5.7: Experimental Setup

formance of the mobile client, there were no constraints on the network link between the database and the background load generator used by the MobileSales benchmark.

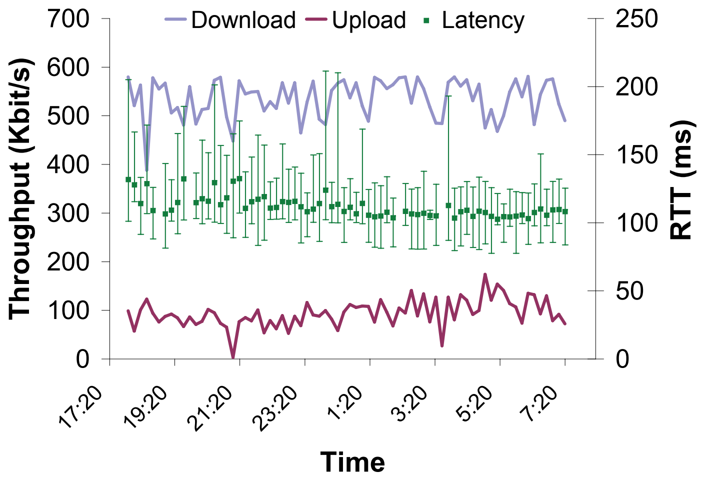
For each benchmark, results from the two configurations listed below were measured:

- *Native:* This configuration corresponds to the mobile client directly accessing the database server using its native JDBC driver. Native does not use Cedar's replica and avoids the overhead of using proxies.
- *Cedar:* This configuration corresponds to using a database replica present on the mobile client. Comparing these results to the corresponding *Native* results gives the performance benefit due to the Cedar system.

5.5.3 Experimental Configuration

In the experimental setup shown in Figure 5.7, the database server and the background load generator were 3.2 GHz Pentium 4s (with Hyper-Threading) with 4 and 2 GB of RAM respectively. The laptop was a Thinkpad T41p with a 1.7 GHz Pentium M processor and 1 GB of RAM. The PDA-class machine (henceforth referred to as the PDA) was a 1997-era Thinkpad 560X with a 233 MHz Pentium MMX processor and 64 MB of RAM.

All the machines ran the Fedora Core 5 Linux distribution with the 2.6.18-1 SMP kernel. Sun's Java 1.5 was used as Cedar's runtime environment. The database server



The error bars for the latency measurements indicate the maximum and minimum observed values. Four outliers (5X–30X of the mean) were removed from the ping data.

Figure 5.8: EVDO Network Throughput and Latency

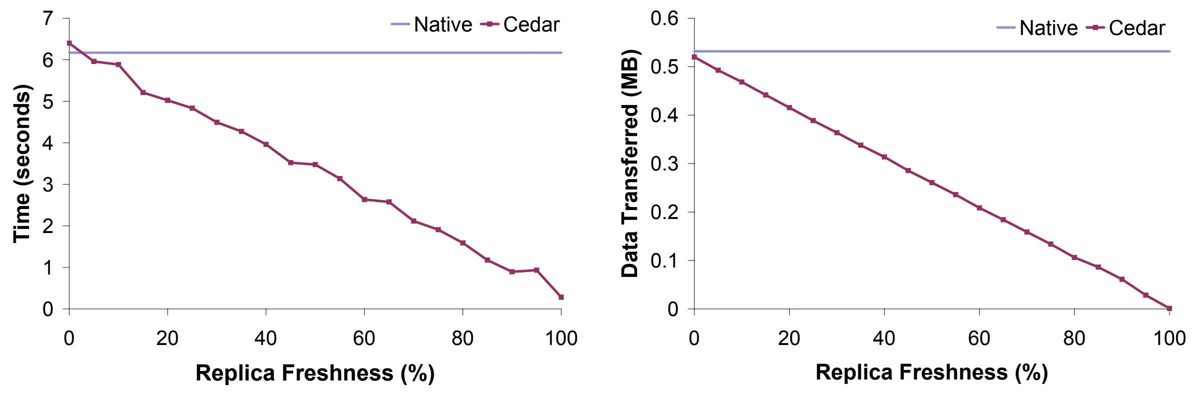
and the client replica used the open source MySQL database. The server proxy was located on the same machine as the database server.

To detect and elide commonality between results, the approach described in Section 5.4.1 was used. As content-addressable techniques have been shown to be better than compression [91], the latter was not considered in the evaluation.

5.6 Performance Improvement: Microbenchmarks

As all results displayed identical trends, only a subset of the microbenchmark results are presented here. For the microbenchmark that fetched approximately 0.5 MB of data, Figure 5.9 presents the query completion time and the amount of data downloaded by the mobile client. The results show that Cedar's performance is almost a linear function of the amount of data found on the client replica. Cedar's latency overhead is visible only when the amount of stale data found is greater than 95%. Irrespective of whether the laptop or PDA was used, the same trends were observed for all the other microbenchmarks and network configurations.

In Figure 5.9 (b), note that even when Cedar finds no useful results on the client replica (the 0% case), it still transfers slightly less data than the Native scenario that



(a) Query Latency (b) Data Transferred

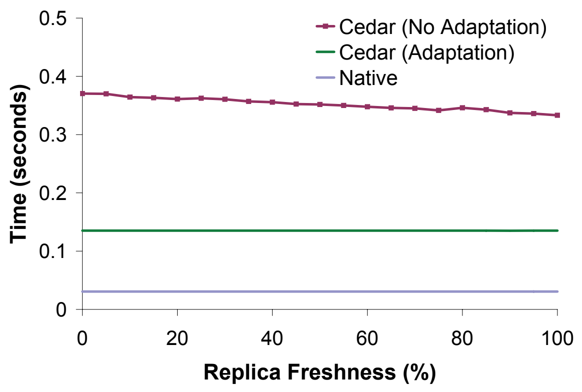
The microbenchmark was executed on the laptop. Results are the mean of five trials. The maximum standard deviation for data transferred was 2% of the corresponding mean. There was no significant standard deviation for the query execution time.

Figure 5.9: 0.5 MB Microbenchmark: 1 Mbit/s and 100ms Network

uses MySQL. This occurs because MySQL and Cedar use different encodings over the wire and Cedar's encoding was slightly more efficient for the microbenchmark's dataset. This result also shows that even though Cedar uses Java objects for its client-server communication, its wire protocol is as efficient as MySQL's native JDBC driver.

Apart from the networks described in Section 5.5.2, the microbenchmarks were also executed over a Gigabit Ethernet link. While Cedar is not geared towards extremely fast networks, this allows an evaluation of Cedar's adaptability in a high bandwidth setting. As shown in Figure 5.10, Cedar, without adaptation, is almost a factor of 10X slower than Native. The analysis showed that the extra software hops through Cedar's client and server proxy were the most significant cause for the this slowdown. A smaller, but nevertheless noticeable, fraction of the overhead was the computation cost of generating recipes. This can be seen in the slight increase in latency as the amount of fresh data on the replica decreases.

As described in Section 5.3.3, whenever the client detects a fast network, it will bypass the local proxy and switch to directly contacting the server-side proxy. This adaptation gets Cedar's overhead down to within 4.5X of the native performance.



The microbenchmark was executed on the laptop. Results are the mean of five trials and had no significant standard deviation.

Figure 5.10: 0.5 MB Microbenchmark Query Latency: Gigabit

While still relatively high, the absolute difference is within 0.1 seconds of Native. Tuning the unoptimized prototype should further improve Cedar's performance.

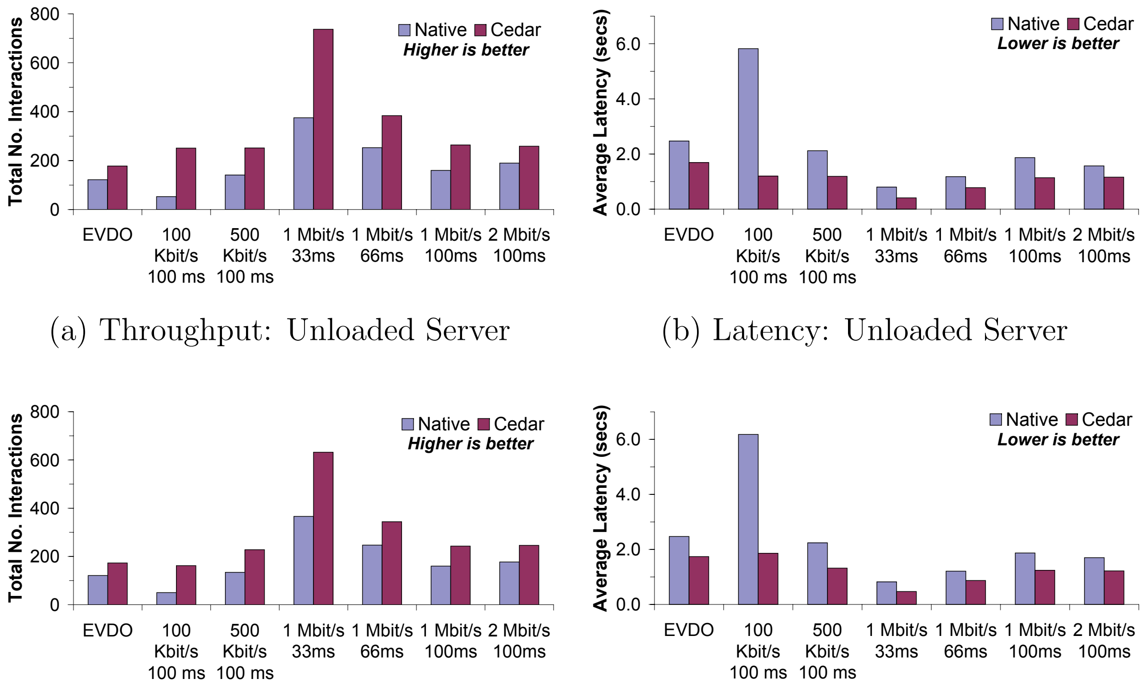
5.7 Performance Improvement: MobileSales

5.7.1 Benchmark Setup

For the results below, from the MobileSales benchmark, the scale factor S defined in Section 5.5.1 was set to 50. This generates a dataset large enough for simultaneous access by 50 clients. While the raw data size is 2 GB, it occupies 6.1 GB on disk due the addition of indexes and other MySQL metadata.

Each test run executed for five minutes using the default mix of client interactions shown in Table 5.3. This mix has a greater percentage of update interactions than reads. The high write ratio biases the results against Cedar as updates are not optimized. A higher read ratio would only increase Cedar's benefits.

During different benchmark runs, the number of load clients was set to either 0, 10, 30, or 50. As the results with 10 and 30 clients fall in between the 0 and 50 clients cases, this section only presents the results for the unloaded server (0 load clients) and 50 clients. The baseline for comparison is direct database access via MySQL's native JDBC driver. Relative to the baseline, improvement is defined as



(a) Throughput: Unloaded Server

(b) Latency: Unloaded Server

(c) Throughput: Server Load = 50

(d) Latency: Server Load = 50

All results are the mean of three trials. The maximum standard deviation for throughput and latency was 6% and 7% respectively of the corresponding mean.

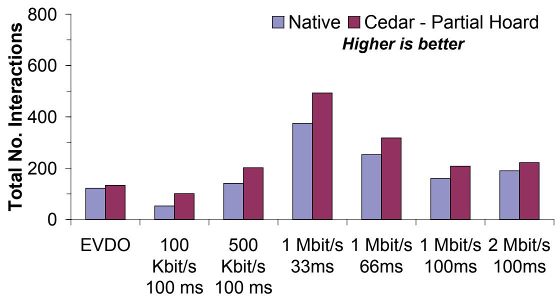
Figure 5.11: Mobile Sales - Laptop with Full Hoard Profile

$$Improvement = \frac{Result_{Cedar} - Result_{Native}}{Result_{Native}}$$

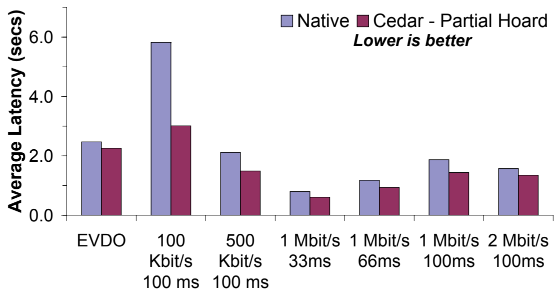
The performance of the mobile clients was evaluated with two different hoard profiles. The first profile, named Full Hoard, selected the entire database for replication. The second profile, named Partial Hoard, only selected half of the product catalog (a predominantly read-only portion of the database) and did not include customer or order information.

5.7.2 Laptop: Full Hoard Results

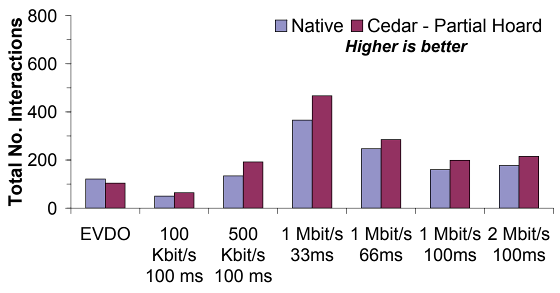
Figure 5.11 presents the results of the MobileSales benchmark with the Full hoard profile. Figures 5.11 (a) and (b) first display the results from the unloaded server case



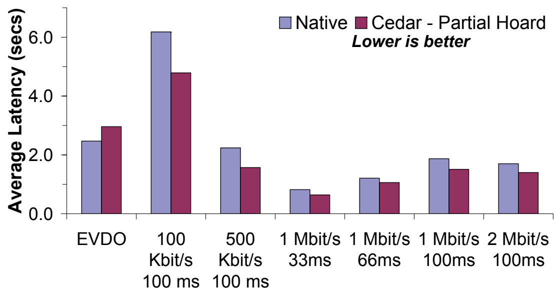
(a) Throughput: Unloaded Server



(b) Latency: Unloaded Server



(c) Throughput: Server Load = 50



(d) Latency: Server Load = 50

All results are the mean of three trials. The maximum standard deviation for both throughput and latency was 8% of the corresponding mean.

Figure 5.12: Mobile Sales - Laptop with Partial Hoard Profile

where the laptop is the only client accessing the database. The results show that the throughput improvements due to Cedar range from 36% in the high bandwidth case of 2 Mbit/s to as much as 374% in the low bandwidth case of 100 Kbit/s. The average interaction latency shows similar improvements, ranging from a 26% reduction at 2 Mbit/s to a 79% reduction at 100 Kbit/s. The results from the emulated networks match the real EVDO network where Cedar shows a 46% improvement in throughput and a 30% reduction in the average interaction latency. As latency on the 1 Mbit/s network increases, throughput drops for both Native and Cedar. This occurs because neither system can fill the bandwidth-delay product.

Figures 5.11 (c) and (d) show that even when the number of load clients is increased to 50, Cedar retains its performance advantage. Native's performance shows very little change when compared to an unloaded server. As the database is not the bottleneck, the impact on both Native's throughput and average interaction latency is less than 8%. Cedar, when compared to the unloaded server, does show a more noticeable drop. As updates made by the load clients increase the divergence between the client replica and the server, Cedar has to fetch more data over the low bandwidth network. Compared to the unloaded server, Cedar experiences a 3% to 35% drop in throughput and a 3% to 55% increase in latency. However, it still performs significantly better than Native in all experimental configurations. The throughput improvements range from 39% at 2 Mbit/s to 224% at 100 Kbit/s and latency reductions range from 28% at 2 Mbit/s to 70% at 100 Kbit/s.

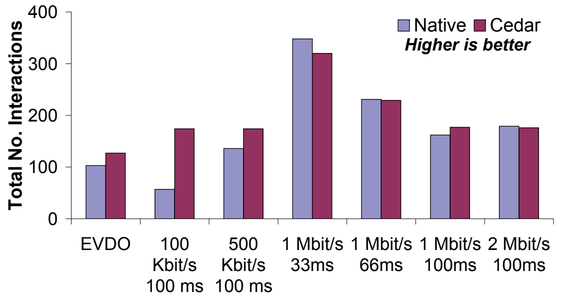
5.7.3 Laptop: Partial Hoard Results

Figure 5.12 presents the results of the MobileSales benchmark with the Partial Hoard profile. As the hoard profile has no impact on Native, its results are unchanged from Section 5.7.2. While the performance gains due to Cedar drop when compared to the Full hoard profile, they are still substantial. For an unloaded server, as seen in Figures 5.12 (a) and (b), Cedar can deliver a throughput improvement ranging from 9% at EVDO to 91% at 100 Kbit/s and an average interaction latency reduction ranging from 9% at EVDO to 48% at 100 Kbit/s. Even for the faster 1 and 2 Mbit/s networks, Cedar's throughput and latency improvements are in the range of 17–31% and 14–24%.

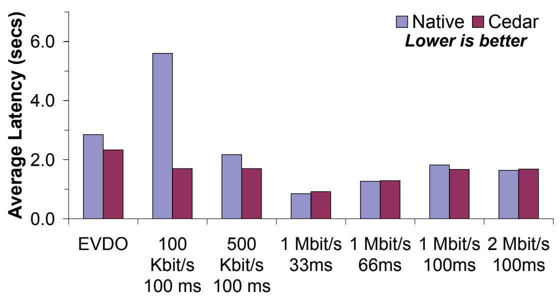
Figures 5.12 (c) and (d) show that as Cedar's throughput drops when compared to an unloaded server, the improvement relative to Native now ranges from a throughput increase of 15% at 1 Mbit/s with 33ms to 43% at 500 Kbit/s. The corresponding reduction in average interaction latency ranges from 12% at 1 Mbit/s with 66ms to 30% at 500 Kbit/s. The only exception is the EVDO network where Cedar's throughput is 14% lower than Native. This anomaly was attributed to temporary network phenomena as these EVDO results exhibited the greatest variance in the experiments.

5.7.4 PDA: Full Hoard Results

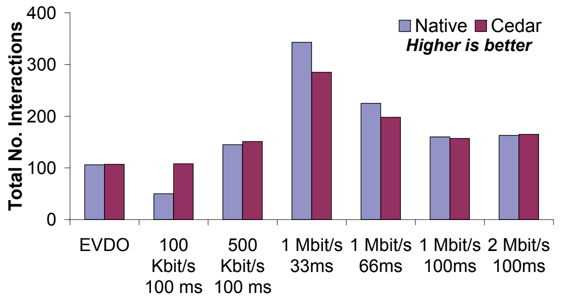
Figure 5.13 presents the results from running MobileSales on the PDA with the Full hoard profile. It can be seen that, even with an extremely resource-limited client, Cedar can still deliver a significant performance improvement. With the EVDO



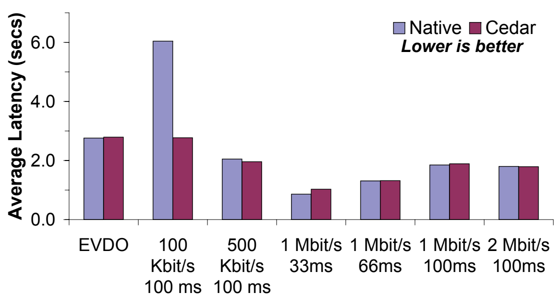
(a) Throughput: Unloaded Server



(b) Latency: Unloaded Server



(c) Throughput: Server Load = 50



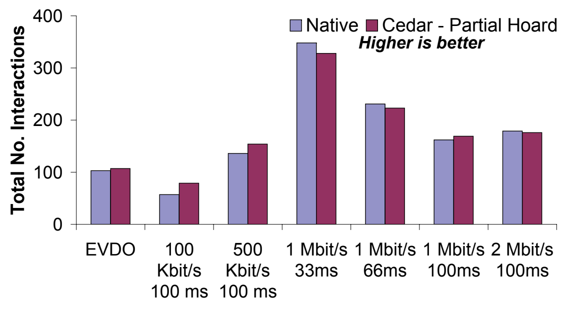
(d) Latency: Server Load = 50

All results are the mean of three trials. The maximum standard deviation for throughput and latency was 8% and 11% respectively of the corresponding mean.

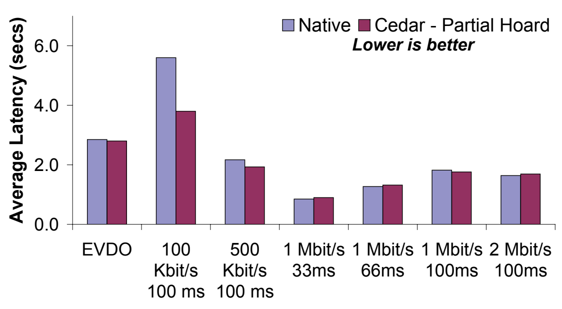
Figure 5.13: Mobile Sales - PDA with Full Hoard Profile

network and an unloaded server, a 23% improvement in throughput and an 18% reduction in latency can be seen. For the 100 Kbit/s network, the results show a 205% improvement in throughput and a 70% reduction in latency for an unloaded server and a 116% improvement in throughput and a 54% reduction in latency with 50 load clients.

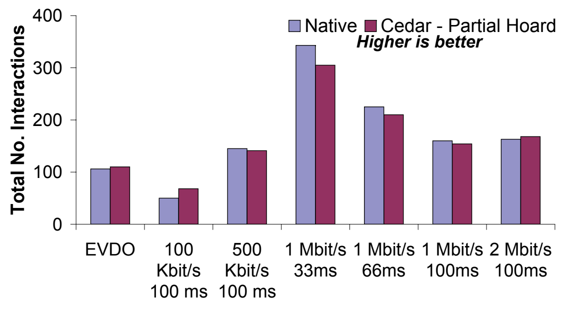
The gains from using Cedar tail off at the higher bandwidths with equivalent performance in most cases. However, Cedar actually performed slightly worse than Native on a 1 Mbit/s with 33ms network. Comparing it to Native's throughput, the results show a drop of 8% with an unloaded database server and a drop of 17% with 50 load clients. A similar drop is seen with 50 load clients on a 1 Mbit/s with 66ms network. This performance drop arose because the 233 MHz CPU became a bottleneck when large amounts of data needed to be hashed. This is exactly the



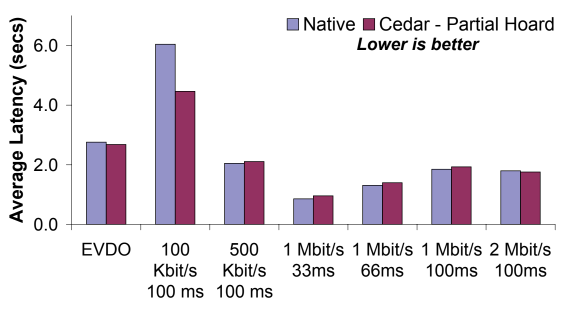
(a) Throughput: Unloaded Server



(b) Latency: Unloaded Server



(c) Throughput: Server Load = 50



(d) Latency: Server Load = 50

All results are the mean of three trials. The maximum standard deviation for both throughput and latency was 6% of the corresponding mean.

Figure 5.14: Mobile Sales - PDA with Half Hoard Profile

scenario where the adaptation technique proposed in Section 5.3.3 would be useful. Tuning Cedar for PDAs should further decrease this overhead.

5.7.5 PDA: Partial Hoard Results

Switching the PDA from the Full hoard profile to the Partial hoard profile showed the same behavior as above, and therefore, a detailed description of these results is omitted. Overall, as seen in Figure 5.14, Cedar delivered equivalent or better performance that Native in almost all of the network configurations. The only exceptions were the two 1 Mbit/s network configurations highlighted in Section 5.7.4. In these cases, relative to Native, Cedar's worst case overheads for throughput and latency were 11% and 12% respectively.

5.7.6 Summary of MobileSales Results

The results from the MobileSales benchmark demonstrate that Cedar can significantly improve performance in low-bandwidth conditions. These results are especially encouraging as they were achieved without any compromise in consistency and without any modifications to the application or database. Also, as described in Section 5.7.1, these results arise from a write-intensive workload that was biased against Cedar. Cedar would perform even better for workloads with a greater percentage of reads. Further, while Cedar is primarily targeted towards laptop-class clients, the results indicate that even resource-limited PDA-class machines can benefit.

5.8 Combining Ganesh with Cedar

As seen in this chapter, Cedar and Ganesh, while adopting different approaches to improving performance, are closely related. They both use content-addressable approaches to detecting similarity, use the same techniques to remain transparent to both applications and databases, and don't weaken any of the traditional transactional or consistency properties provided by the underlying database.

For real deployments, one would envisage using both techniques together as they are complementary. For example, Cedar's replicas are currently only lazily updated and the time period between updates can be large. The amount of useful data obtained on the client therefore decreases with time as the contents of the replica start diverging from the database server. This shortcoming can be easily addressed by Ganesh's approach of caching result rows. For the intervening period between replica updates, data not found on the replica could be cached by Ganesh for future use. Similarly, Ganesh can currently only improve performance for the queries it has seen in the past. Even though the result might be present in its cache, the data could be spread over multiple results from different queries and Ganesh would not be able to combine them in response to a previously unseen query. Adding a database engine, such as the replica used by Cedar, would help in dealing with this problem.

For a system that combines the principles used in Ganesh and Cedar, a careful investigation of resource allocation will be needed. For example, how should the memory and on-disk cache be divided between the two and should the partition be dynamic? Given the relatively inexpensive computational cost of the cryptographic hashes used by Ganesh and the more expensive I/O incurred when using an on-disk replica, how should these techniques be used together and what workloads would benefit from them? While a detailed study of these and similar questions is needed, it is my belief that a system that combined these two techniques would be demonstrably better than the use of either system in isolation.

5.9 Summary

In summary, this chapter presents the design and implementation of Cedar, a system that enables mobile database access with good performance while preserving consistency. Cedar leverages a mobile client's storage and computational resources to compensate for weak connectivity. The evaluation results demonstrate that Cedar has low overhead and can significantly improve performance for its intended usage environment.

Chapter 6

A Data Oriented Transfer Service

6.1 Introduction

The last few chapters have shown that content-addressable techniques can be a very powerful tool to improve the performance of distributed systems deployed over WANs. As can be seen in Table 6.1, these techniques are useful in a number of different scenarios. They have been used to improve performance by exploiting external networked data sources in CASPER, portable storage devices in Lookaside Caching, locality in result streams in Ganesh, and stale data replicas in Cedar. Similarity has been found at different levels of granularity (whole vs. partial objects), by using generic and application-specific algorithms, and without requiring application modifications.

These content-addressable techniques have been used without modifying or weakening any of the system or application semantics. These benefits arise from the fact that content-based naming allows the easy separation of two very different functions. The first is *content negotiation*, which is very application-specific and forms a part of its control flow. The second function is *data transfer*, in which the actual data bits are exchanged. Given the increasing prevalence of bulk data transfers over the Internet (more than 70% of Internet traffic according to Azzouna and Guillemin [14]), this separation of control versus data flow has become even more critical to performance.

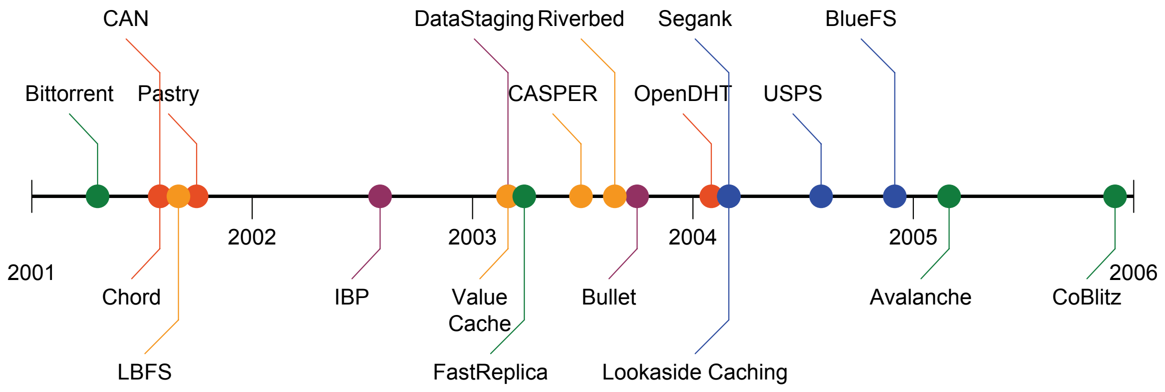
However, the previously described systems in this thesis have assumed the existence of standardized APIs such as the file system interface or the JDBC database interface in order to transparently interpose their optimizations. For applications and

Property	CASPER	Lookaside Caching	Ganesh	Cedar
Origin of Similarity	External Data Sources	Portable Storage	Intra-Result Similarity	Stale Database Replicas
Granularity of Similarity	Partial File	Whole File	Whole and Partial Results	Whole and Partial Results
Transparency Method	Proxy-based	File System Interface	Proxy-based	Proxy-based
Original System Modified?	No	Yes	No	No
Applications Modified?	No	No	No	No

Table 6.1: Property Summary for CASPER, Lookaside Caching, Ganesh, and Cedar

systems that lack such standardized interfaces, the ability to transparently optimize data transfers is significantly impacted. Even assuming the availability of source code, adding each optimization to an application can take significant effort. Further, even if different applications want to use the same optimization technique, there will be duplication of work as the technique will need to be individually added to each system. Any new techniques that might arise in the future will also be incompatible with legacy applications.

Optimizing data transfers in these legacy applications is difficult for a number of reasons. Even though the process of data transfer is generally independent of the application, applications and protocols have traditionally almost always bundled these functions together. The first reason for this is likely expediency: TCP and the socket API provide a mechanism that is "good enough" for application developers who wish to focus on the other, innovative parts of their programs. The second reason is the challenge of naming. In order to transfer a data object, an application must be able to name it. The different ways that applications define their namespaces and map names to objects is one of the key differences between many protocols. For example, FTP and HTTP both define object naming conventions, and may provide different names for the same objects. While the use of content-based naming can help in separating these two, the problems of adding different data transfer optimizations to individual applications still remain.



The technologies illustrated above include P2P systems [43, 44, 61, 101], Distributed Hash Tables (DHTs) [112, 115, 120, 140], CAS-based systems [91, 114, 117, 149], portable storage based file systems [96, 136, 150, 160], and optimizations based on in-network storage [19, 58, 75].

Figure 6.1: Timeline for Data Transfer Technologies

In order to address this problem, and as a generalization of the work described in Chapters 2 through 5, this chapter proposes the creation of a *transfer service*, a new way of structuring programs that do bulk data transfer, by separating their application-specific control logic from the generic function of data transfer. Applications still perform content negotiation using application-specific protocols, but they use the transfer service to perform bulk point-to-point data transfers. The applications pass the data object that they want to send to the transfer service. The transfer service is then responsible for ensuring that this object reaches the receiver. The simplest transfer service might accomplish this by sending the data via a TCP connection to the receiver. A more realistic transfer service could implement the data transfer techniques outlined in Table 6.1, making them available to SMTP, HTTP, and other applications.

Separating data transfer from the application provides several benefits. The first benefit is for application developers, who can re-use available transfer techniques instead of re-implementing them. The second benefit is for the inventors of innovative transfer techniques. As can be seen in Figure 6.1, first shown in Chapter 1, there have been a number of efforts that have examined ways to improve the efficiency and speed of bulk transfers. However, these innovations must be reimplemented for each application. Not surprisingly, the rate of adoption of innovative transfer mechanisms, particularly in existing systems, is slow. Applications that use the transfer service can

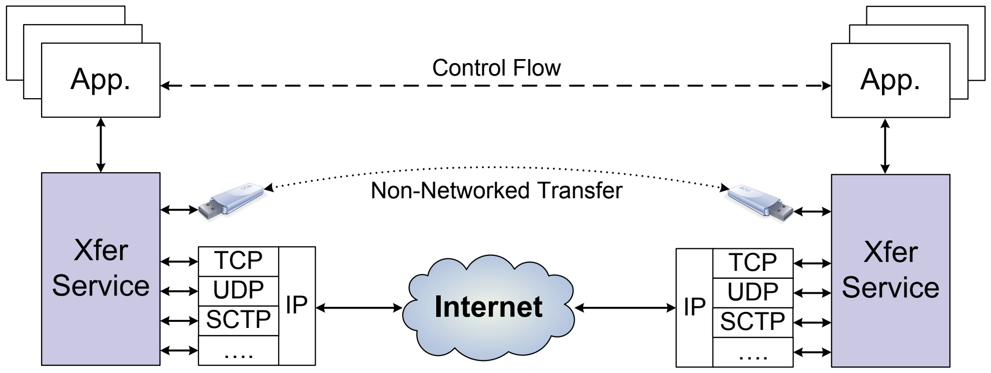


Figure 6.2: DOT Overview

immediately begin using the new transfer techniques without modification. Innovative ideas do not need to be hacked into existing protocols using application-specific tricks. Third, because the transfer service sees the whole data object that the application wants to transfer, it can also apply techniques such as coding, multi-pass compression, and caching, that are beyond the reach of the underlying transport layers. The transfer service itself is not bound to using particular transports, or even established transports – it could just as well attempt to transfer the data using a different network connection or a portable storage device.

Moving data transfer into a new service requires addressing three challenges. First, for both legacy and new applications, the service must provide a convenient and standard API for data transfer applications. However, as shown in Section 6.4, this does not preclude the creation of transparent proxy-based solutions when standardized protocols exist. Second, the architecture should allow easy development and deployment of new transfer mechanisms. Finally, the service must be able to support applications with diverse negotiation and naming conventions.

The following sections present the design and implementation of a Data-Oriented Transfer service, or DOT, shown in Figure 6.2. The design of DOT centers around a clean application interface to a modular, plugin based architecture that facilitates the adoption of new transfer mechanisms. DOT uses content-addressable techniques to name objects based upon their cryptographic hash, providing a uniform naming scheme across all applications.

This chapter begins in Section 6.2 with examples of how a transfer service can be used. Next, in Section 6.3, it describes an effective design for such a transfer service, its API and extension architecture, and its core transfer protocols. Section 6.4 demonstrates how the transfer service's API was used to create an application-specific proxy for the SMTP protocol used by email clients. Section 6.5 then evaluates the implementation of DOT with a number of micro- and macro-benchmarks, finding that it is easy to integrate with applications, and that by using DOT, applications can achieve significant bandwidth savings and easily take advantage of new network capabilities. The chapter finally concludes with a discussion on issues facing a transfer service deployment in Section 6.6, the relationship of DOT to the other systems described in this dissertation in Section 6.7, and a brief summary of the chapter's findings in Section 6.8.

6.2 Transfer Service Scenarios

The advantage of a generic interface for data transfer is that it enables new transfer techniques across several applications. While several transfer techniques have been implemented within the DOT prototype, it can be argued that its true power lies in its ability to accommodate a diverse set of scenarios beyond those in the initial prototype. This section examines several scenarios that a transfer service would enable, and it concludes with an examination of situations for which the transfer service might be inappropriate.

- A first benefit the transfer service could provide is cross-application caching. A DOT-based cache could benefit a user who receives the same (or similar) file through an Instant Messaging application as well as via an email attachment. The benefits increase with multi-user sharing. An organization could maintain a single cache that handled all inbound data, regardless of which application or protocol requested it.

- Content delivery networks such as Akamai [3] could extend their reach beyond just the Web. A "data delivery network" could accelerate the delivery of Web, Email, NNTP, and any other data-intensive protocol, without customizing the service for each application. DOT could provide transparent access to Internet Backplane Protocol storage depots [19], to storage infrastructures such as Open DHT [115], or to a collection of BitTorrent peers.

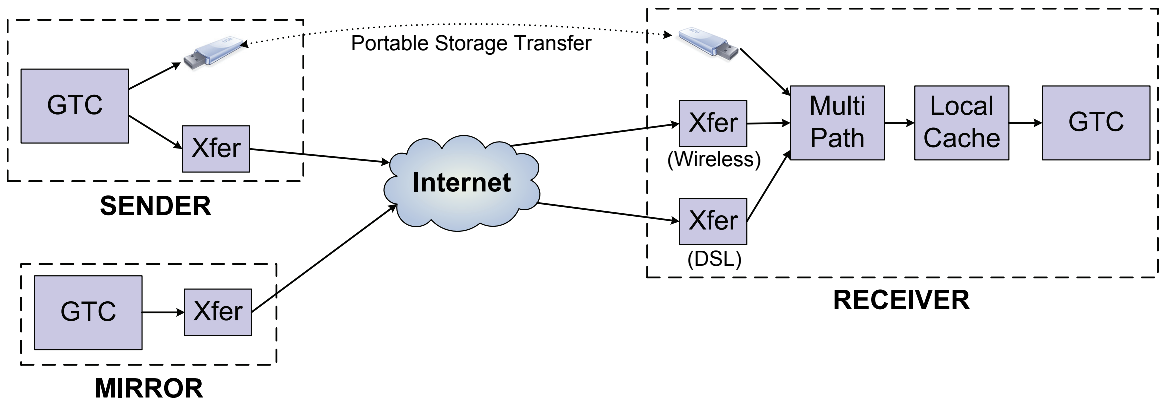


Figure 6.3: Example DOT Configuration

- The transfer service is not bound to a particular network, layer, or technology. It can use multi-path transfers to increase its performance. If future networks provide the capability to set up dedicated optically switched paths between hosts, the transfer service could use this facility to speed large transfers. The transfer need not even use the network: it could use portable storage to transfer data [56, 160].
- Finally, the benefits of the transfer service are not limited to simply exchanging bits. DOT creates the opportunity for making cross-application data processors that can interpose on all data transfers to and from a particular host. These proxies could provide services such as virus scanning or compression. Data processors combined with a delegation mechanism such as DOA [159] could also provide an architecturally clean way to perform many of the functions provided by today's network middleboxes.

The transfer service might be inappropriate for real-time communication such as `telnet` or teleconferencing. DOT's batch-based architecture would impose high latency upon such applications. Nor is the transfer service ideal for applications whose communication is primarily "control" data, such as instant messaging with small messages. The overhead of the transfer service may wipe out any benefits it would provide.

With these examples of the benefits that a transfer service can and cannot provide in mind, the next section examines a more concrete example of a DOT configuration.

6.2.1 Example DOT Configuration

Figure 6.3 shows an example DOT configuration supported by the current prototype that allows data transfers to proceed using multiple Internet connections and a portable storage device. The Generic Transfer Client (GTC) shown in the figure provides the core of the transfer service functionality. This configuration provides three properties:

Surviving transient disconnection and mobility: Coping with disconnection requires that a transfer persist despite the loss of a transport-layer connection. In the example configuration, the multi-path plugin uses both the wireless and DSL links simultaneously to provide redundancy and load balancing. Mobility compounds the problem of disconnection because the IP addresses of the two endpoints can change. DOT's data-centric, content-based naming offers a solution to this problem because it is not tied to transport-layer identifiers such as IP addresses.

Transferring via portable storage: Portable storage offers a very high-latency, high-bandwidth transfer path [56, 160]. DOT's modular architecture provides an easy way to make use of such unconventional resources. A portable storage plugin might, for example, copy some or all of the object onto an attached disk. When the disk is plugged into the receiver's machine, the corresponding transfer plugin can pull any remaining data chunks from the disk. An advanced implementation might also make these chunks available to other machines in the network.

Using caching and multiple sources: The receiver can cache chunks from prior transfers, making them available to subsequent requests. The configuration above also shows how the transfer service can fetch data from multiple sources. Here, the multi-path plugin requests chunks in parallel from both the portable storage plugin and a set of network transfer plugins. The transfer plugins pull data from two different network sources (the sender and a mirror site) over two network interfaces (wireless and DSL).

6.3 Design and Architecture

This section first presents an overview of the DOT system architecture in Section 6.3.1. It then examines the manner in which applications use the transfer service in Sections 6.3.2 and 6.3.3. Next, it describes how transfer plugins and the default transfer protocol operates in Section 6.3.4 and the way that the transfer service accesses storage resources in Section 6.3.5. Finally, Section 6.3.6 examines the way that the plugins can be chained together to extend the system.

6.3.1 Architectural Overview

Applications interact with the transfer service as shown in Figure 6.2. Applications still manage their control channel, which handles content negotiation, but they offload bulk transfers to the transfer service. The transfer service delivers the data to the receiver using lower layer system and network resources, such as TCP or portable storage.

DOT is a receiver-pull, point-to-point service. The receiver-pull model was chosen to ensure that DOT only begins transferring data after both the sending and receiving applications are ready. Receivers also have a better knowledge of available resources that the sender might be unaware of. For example, the receiver might detect similarity between the object to be transferred and a file on its local storage system. DOT focuses on point-to-point service for two reasons: First, such applications represent the most important applications on the Internet (e.g., HTTP, email, etc.). Second, point-to-multipoint applications that focus on bulk data transfer can be easily supported by DOT's plug-in model (e.g., transparent support for BitTorrent or CDNs) [106].

As DOT is based on a receiver-pull model, *hints* are used by the sender to inform the receiver's Generic Transfer Client (GTC) of possible data locations. Hints, associated with the object being transferred, are generic location specifiers. A hint has three fields: method, priority, and weight. The method is a URI-like string that identifies a DOT plugin and then provides plugin-specific data. Some examples might be `gtc://sender.example.com:9999/` or `dht://OpenDHT/`. As with DNS SRV records [64], the priority field specifies the order in which to try different sources; the weight specifies the probability with which a receiver chooses different sources with the same priority. By convention, the sender's GTC will include at least one hint – itself – because the sender's GTC is guaranteed to have the data. However, the receiver is free to ignore hints or combine them with other data sources if it desires.

DOT is implemented in C++ using the libasync [84] library. Libasync provides

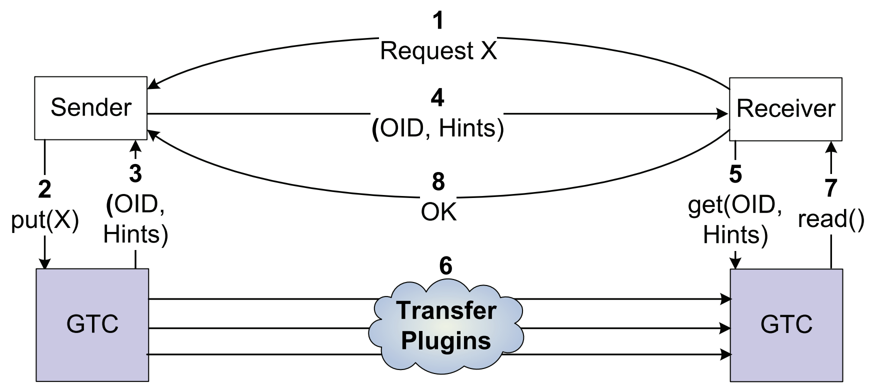


Figure 6.4: Data Exchange using DOT

a convenient callback-based model for creating event-driven services. DOT makes extensive use of libasync's RPC library, libarpc, for handling communication both on the local machine and over the network. For instance, the default protocol used to communicate between different transfer service daemons is implemented using RPC.

Figure 6.4 shows the basic components involved in a DOT-enabled data transfer: the sender, receiver, and the DOT transfer service. The core of the transfer service is provided by the DOT GTC. The GTC uses a set of plugins to access the network and local machine resources:

- **Transfer plugins** accomplish the data transfer between hosts. Transfer plugins include the default GTC-to-GTC transfer plugin and the portable storage transfer plugin.
- **Storage plugins** provide the GTC with access to local data, divide data into chunks, and compute the content hash of the data. Storage plugins include the disk-backed memory cache plugin used in the implementation, or a plugin that accesses locally indexed data on the disk.

Sending data using DOT involves communication among the sending and receiving application, and the GTC. Figure 6.4 enumerates the steps in this communication:

(1) The receiver initiates the data transfer, sending an application-level request to the sender for object X. Note that the sender could also have initiated the transfer.

(2) The sender contacts its local GTC and gives the GTC object X using the `put` operation.

(3) When the `put` is complete, the GTC returns a unique object identifier (OID) for X to the sender as well as a set of hints. These hints help the receiver know where it can obtain the data object. The OID and hints are treated as an opaque object by both the sending and receiving application.

(4) The sender passes the OID and the hints to the receiver over the application control channel.

(5) The receiver uses the `get` operation to instruct its GTC to fetch the object corresponding to the given OID.

(6) The receiver's GTC fetches the object using its transfer plugins, described in Section 6.3.4, and then

(7) returns the requested data to the receiver.

(8) After the transfer is complete, the receiver continues with its application protocol.

DOT names objects by OID; an OID contains a cryptographic hash of the object's data plus protocol information that identifies the version of DOT being used. The hash values in DOT include both the name of the hash algorithm and the hash value itself.

6.3.2 Transfer Service API for New and Legacy Applications

The application-to-GTC communication, shown in Table 6.2, is structured as an RPC-based interface that implements the **put** and **get** family of functions for senders and receivers respectively. A key design issue for this API is that it must support existing data-transfer applications while simultaneously enabling a new generation of applications designed from the ground-up to use a transfer service. The experience from implementing different systems based on content-addressable techniques, as shown in Table 6.1, has revealed several key design elements that are reflected in DOT's API:

Minimal changes to control logic: Using the data transfer service should impose minimal changes to the control logic in existing applications. Similar to the modifications needed for lookaside caching, the application protocol with DOT's API only needs to be modified to send the OID and hints instead of the actual data from the sender to the receiver. As shown later in Section 6.4, other minor changes might

Return Type / Command	Description
PUT Commands	
`t_id put_init()`	Initiates an object "put". Returns a transaction identifier.
`void put_data(t_id, data)`	Adds object data to the GTC
`(OID,Hints) put_commit(t_id)`	Marks the end of object data transfer. Returns an opaque structure consisting of the OID and Hints
`(OID,Hints) put_fd(fd)`	Optimized put operation that uses an open file descriptor
`void done(OID)`	Allows the GTC to release resources associated with OID.
GET Commands	
`t_id get_init(OID, mode, hints)`	Initiates an object fetch. Returns a transaction identifier. Mode can be Sequential or Out-of-Order.
`int get(t_id, buf, &offset, &size)`	Read data from the GTC. Returning zero indicates EOF.

All function names are preceded by GTC_ in the actual implementation

Table 6.2: The Application-to-GTC API

also be required in the application's handshake protocol if backward compatibility with non-DOT enabled systems is required. Apart from the above RPC-based API, a stub library, described in Section 6.3.3 was also created for easier integration with legacy applications.

Permit optimization: The second challenge is to ensure that the API does not impose an impossible performance barrier. In particular, the API should not mandate extra data copies, and should allow an optimized GTC implementation to avoid unnecessarily hashing data to compute OIDs. Some applications also use special OS features (e.g., the zero-copy `sendfile` system call) to get high performance; the API should allow the use of such services or provide equivalent alternatives. To address such performance concerns, the GTC provides a file-descriptor passing API that sends data to the GTC via RPC. Caching of OIDs, similar to systems such as EMC's Centera [55], is another possible optimization. DOT gives high-performance applications the choice to receive data out-of-order to reduce buffering and delays. DOT's API can also be used in a synchronous or asynchronous manner.

Data and application fate sharing: The third design issue is how long the sender's GTC must retain access to the data object or a copy of it. The GTC will retain the data provided by the application at least until either: (a) the application calls `GTC_done()`; or (b) the application disconnects its RPC connection from the GTC (e.g., the application has exited or crashed). Currently, there is no limit to how long the GTC may cache data provided by the application. The current implementation retains this data in cache until it is evicted by newer data. Adding support for one-time transfers and secure deletes from DOT's cache is a part of future work.

6.3.3 Transfer Service API for Legacy Applications

While the API described in Section 6.3.2 is both small and relatively easy to use, most applications are not written to send data using RPC calls to a local transfer service. Requiring the use of an RPC library might be a more intrusive change to the application than desired. Therefore, to make it easy to port existing applications to use DOT, a socket-like stub library was created that preserves the control semantics of most applications. This interface requires the GTC to place data in-order before sending it to the application. The library provides five functions:

```
dot_init_get_data(oid+hints)
dot_init_put_data()
dot_read_fn(fd, *buf, timeout)
dot_write_fn(fd, *buf, timeout)
dot_fds(*read, *write, *time)
```

`get` and `put` return a forged "file descriptor" that is used by the `read` and `write` functions. The `read` and `write` functions provide a blocking read and write interface. The `fds` function allows select-based applications to query the library to determine which real file descriptors the application should wait for, and for how long. Section 6.4 shows how this support for legacy applications made it easy to integrate DOT with a production mail server package. Note that with some more engineering effort, the stub library could be converted to export a pure socket interface with the addition of special functions to mark object boundaries.

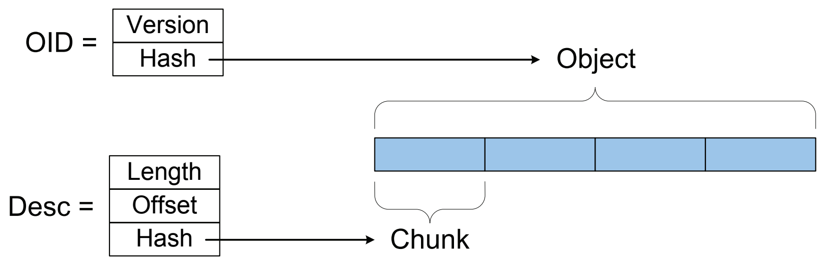


Figure 6.5: DOT Objects, Chunks, OIDs and Descriptors

6.3.4 DOT Transfer Plugins

The GTC transfers data through one or more transfer plugins. These plugins have a simple, standard interface that can implement diverse new transfer techniques. Internally, the GTC splits objects into a series of *chunks*. Each chunk is named with a *descriptor*, which contains a hash of the chunk, its length, and its offset into the object. Like CASPER, the list of descriptors that correspond to the OID is also known as a recipe. Figure 6.5 shows the relationship between objects, OIDs, chunks, and descriptors. The transfer plugin API has three functions:

```
get_descriptors(oid, hints[], cb)
get_chunks(descs[], hints[], cb)
cancel_chunks(descriptors[])
```

A transfer plugin must support the first two functions. Given a set of OID and hints, the first function is used to fetch the descriptor list, or recipe, that corresponds to the OID received over the application's control channel. After inspecting the recipe, the GTC will use the second function to only fetch the data corresponding to descriptors that cannot be found locally. The `cb` parameter in these two functions is an asynchronous callback function to which the plugin should pass the returned descriptors or chunks. The third function is used to cancel an in-progress transfer if the data was fetched via some other transfer plugin. Some plugins may not be able to cancel an in-progress request once it has gone over the network, and so may discard cancel requests if necessary.

To receive data, the GTC calls into a single transfer plugin with a list of the required chunks. That plugin can transfer data itself or it can invoke other plugins. Complex policies and structures, such as parallel or multi-path transfers, can be achieved by a cascade of transfer plugins that build upon each other. For example, the current DOT prototype provides a multi-path plugin which, when instantiated, takes a list of other transfer plugins to which it delegates chunk requests.

Default GTC-to-GTC Transfer Plugin

Every GTC implementation must include a default GTC-to-GTC plugin that is available on all hosts. This plugin transfers data between two GTCs via a separate TCP connection, using an RPC-based protocol. The receiver's GTC requests a recipe from the sender's GTC that correspond to the desired OID. Once it starts receiving descriptors, the receiver's GTC can begin sending pipelined requests over the same plugins to transfer the individual chunks.

Portable Storage Transfer Plugin

On the sender side, the portable storage plugin registers to receive notifications about new OIDs generated by the GTC. When a new OID is discovered, the plugin copies the blocks onto the storage device. The implementation is naïve, but effective: Each chunk is stored as a separate file named by its hash, and there is no index.

On the receiver, the portable storage plugin acts as a transfer plugin accessed by the multi-path plugin that receives a request for all descriptors. Polling the device, while less efficient than receiving notification from the OS, was selected for detecting updates as it is more portable. Adding support for OS-specific mechanisms like `inotify` [81] would be an easy change. The plugin currently polls the portable storage device every 5 seconds to determine if new data is available. If the device has changed, the plugin scans the list of descriptors stored on the flash device and compares them to its list of requested descriptors, returning any that are available.

Multi-path Transfer Plugin

The DOT multi-path plugin acts as a load balancer between multiple transfer plugins, some of which could be bound to different network interfaces on a multi-homed machine. The plugin is configured with a list of transfer plugins to which it should send requests. The sub-plugins can be configured in one of two ways. Some plugins can receive requests in small batches, e.g., network plugins that synchronously request data from a remote machine. Other plugins instead receive a request for all chunks at once, and will return them opportunistically. The latter mode is useful for plugins such as the portable storage plugin that opportunistically discovers available chunks.

The multi-path plugin balances load among its sub-plugins in three ways. First, it parcels requests to sub-plugins so that each always has ten requests outstanding. Second, to deal with slow or failed links, it supports request borrowing where already-issued requests are shifted from the sub-plugin with the longest queue to one with an empty queue. Third, it cancels chunk requests as they are satisfied by other sources.

6.3.5 Storage Plugins

The main purpose of the GTC is to transfer data, but the GTC must sometimes store data locally. The sender's GTC must hold onto data from the application until the receiver is able to retrieve it. The receiver's GTC, upon retrieving data might need to reassemble out-of-order chunks before handing the data to the receiver, or may wish to cache the data to speed subsequent transfers.

The GTC supports multiple storage plugins, to provide users and developers with flexible storage options. Examples of potential back-ends to the storage plugins include in-memory data structures, disk files, or a SQL database. The current DOT prototype contains a single storage plugin that uses in-memory hash tables backed by disk. The storage plugin is asynchronous, calling back to the GTC once it stores the data. All DOT storage plugins provide a uniform interface to the GTC. To add data to the storage plugin, the GTC uses an API that mirrors the application PUT API in Table 6.2.

In addition to whole object insertion, the storage plugins export an API for single chunk storage. This API allows the GTC or other plugins to directly cache or buffer chunk data. The lookup functions are nearly identical to those supported by the transfer plugins, except that they do not take a hints parameter:

```
put_chunk(descriptor, data)
release_chunk(descriptor)
get_descriptors(oid, cb)
get_chunks(descriptors[], cb)
```

Chunking

The storage plugin also divides input data into chunks. To do so, it calls into a "chunker" library. The chunker is instantiated for each data object sent to the storage plugin. It provides a single method, `chunk_data`, that receives a read-only pointer to additional data. It returns a vector of offsets within that data where the storage plugin should insert a chunk boundary. The current implementation supports two chunkers, one that divides data into fixed-length segments, and one that uses Rabin fingerprinting [83, 108] to select data-dependent chunk boundaries. Rabin fingerprinting involves computing a function over a fixed-size sliding window of data. When the value of the function is zero, the algorithm signals a boundary. The result is that the boundaries are determined by the value of the data; they are usually the same despite small insertions or deletions of data.

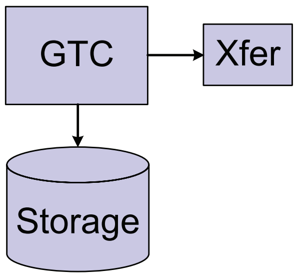
6.3.6 Configuring Plugins

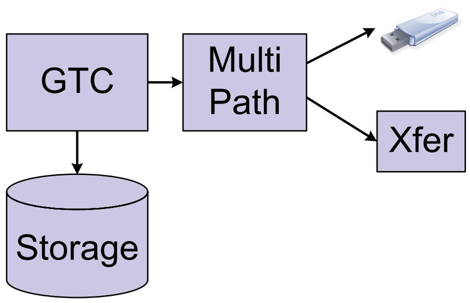
DOT transfer plugins are configured as a data pipeline, passing `get_descriptors` and `get_chunks` requests on to subsequent transfer plugins. Two example plugin configurations along with their instantiation code are shown in Figure 6.6. The first consists of the GTC, a local storage plugin, and a default GTC-to-GTC transfer plugin. The second adds a multi-path plugin and portable storage plugin to the configuration.

Plugins that can push data, such as the portable storage plugin, need to be notified when new data becomes available. These plugins register with the GTC to receive notice when a new data object is inserted into the GTC for transfer, by calling the `register_push` function.

6.4 Application Integration Case Study: Postfix

To evaluate the ease with which DOT integrates with existing, real networked systems, the Postfix mail program [158] was modified to use DOT, when possible, to send email messages between servers. Postfix is a popular, full-featured mail server that is in wide use on the Internet, and represented a realistic integration challenge for DOT.



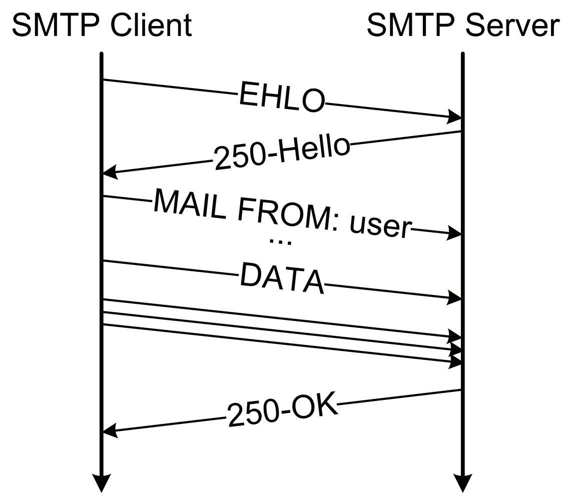


```
gtc = new gtcdMain();                gtc = new gtcdMain();

sp = new storagePlugin(gtc);         sp = new storagePlugin(gtc);
xp = new xferPlugin(gtc);            xp = new xferPlugin(gtc);
                                     psp = new portablePlugin(gtc);
                                     mp = new multiPathPlugin(gtc);

gtc->set_storagePlugin(sp);          gtc->set_storagePlugin(sp);
                                     mp->set_xferPlugin(psp)
                                     mp->set_xferPlugin(xp)
gtc->set_xferPlugin(xp);             gtc->set_xferPlugin(mp);
```

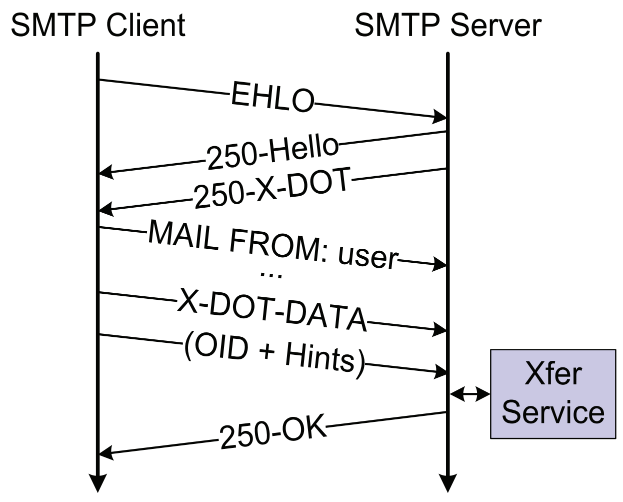
Figure 6.6: Example Plugin Configurations

Examining the benefits of running DOT on a mail server was chosen for a number of reasons. First, mail is a borderline case for a mechanism designed to facilitate large data transfers. Unlike peer-to-peer or FTP downloads, most mail messages are small. Mail provides an extremely practical scenario: the servers are complex, the protocol has been around for years and was not designed with DOT in mind, and any benefits DOT provides would be beneficial to a wide array of users.

Postfix is a modular email server, with mail sending and receiving decomposed into a number of separate programs with relatively narrow functions. Outbound mail transmission is handled by the *smtp* client program, and mail receiving is handled by the *smtpd* server program. Postfix uses a "process per connection" architecture in which the receiving demon forks a new process to handle each inbound email message. These processes use a series of blocking calls, with timeouts, to perform



(a) Normal SMTP



(b) DOT-enabled SMTP

Figure 6.7: SMTP Protocol Modifications

network operations. Error handling is controlled via a setjmp/longjmp exception mechanism.

As the modified Postfix server can also act as a proxy, it is also useful for an incremental deployment of DOT. In most settings today, individual clients communicate with a local mail server to send email. In turn, this local server communicates with the corresponding server on the recipient's end that is responsible for receiving email. Finally, the recipient's server will deliver that mail to the recipient's client. In such scenarios, even if the individual clients are not DOT-enabled, the benefits from DOT's optimized transfer techniques can be realized for the WAN-based data transfers between the two email servers. Further, assuming that data transfers over the WAN are more expensive than in a local setting, the WAN transfers are exactly what would benefit the most from DOT.

DOT was integrated into this architecture as an SMTP protocol extension. All DOT-enabled SMTP servers, in addition to the other supported features, reply to the EHLO greeting from the client with a "X-DOT-DATA" response. Any SMTP client compliant with the RFC [74] can safely ignore unrecognized SMTP options beginning with "X". This allows non-DOT enable clients to use the standard method of data transfer and allows the server to be backward compatible.

As shown in Figure 6.7, on the presentation of X-DOT by the server, any DOT-

Program	**Original LoC**	**New LoC**	**%**
GTC Library	-	421	-
Postfix	70,824	184	0.3%
smtpd	6,413	107	1.7%
smtp	3,378	71	2.1%

Table 6.3: Lines of Code Added or Modified in Postfix

enabled client can use the X-DOT-DATA command as a replacement for the "DATA" command. Clients, instead of sending the data directly to the server, only send the OID and hints to the server as the body of the X-DOT-DATA command. Upon receipt of these, the server opens a connection to its local GTC and requests the specified data object. The server sends a positive completion reply only after successfully fetching the object. In the event of a GTC error, the server assumes that the error is temporary and a transient negative completion reply is sent to the client. This will make sure that the client either retries the DOT transfer or, after a certain number of retries, falls back to normal DATA transmission. Also note that even though the actual bulk data transfer within DOT is receiver driven, this design choice does not place any restrictions on the application's control protocol. As shown in Figure 6.7, this allows the SMTP protocol to remain sender driven.

6.4.1 Integration Effort

Without having any knowledge of Postfix, integrating DOT took me less than a week. This time includes the time required to create the adapter library for C applications that do not use libasync, discussed in Section 6.3.3. Table 6.3 presents the number of lines of code (LoC) needed to enable DOT within Postfix. The modifications touched two Postfix subsystems, the `smtpd` mail server program, and the `smtp` mail client program. While only two applications have thus far been used with DOT (Postfix and `gcp`, described below), the ease with which DOT integrated with each is encouraging.

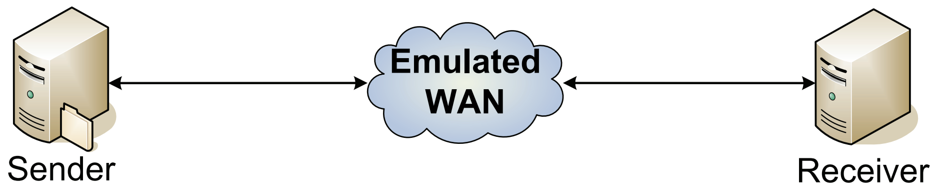


Figure 6.8: Experimental Setup for Microbenchmarks

6.5 Performance Improvement

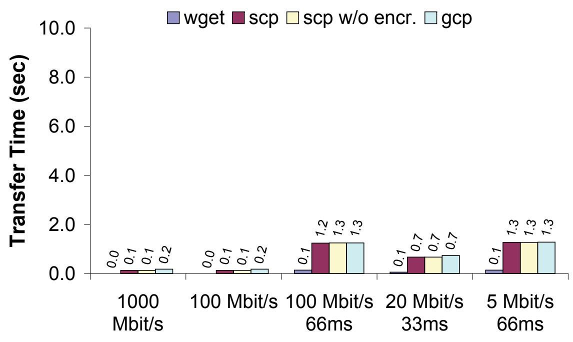
The primary goal of the DOT architecture is to facilitate innovation without impeding performance. This section first examines a number of microbenchmarks of basic DOT transfers to understand if there are bottlenecks in the current system, and to understand whether they are intrinsic or could be circumvented with further optimization. It then examines the performance of plugins that use portable storage and multiple network paths to examine the benefits that can arise from using a more flexible framework for data transfer. This section concludes by examining the integration of DOT into Postfix.

6.5.1 Microbenchmarks

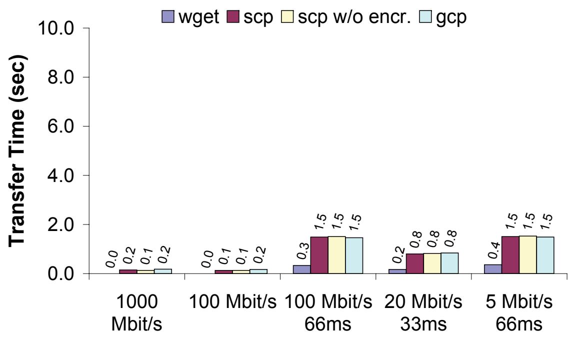
To demonstrate DOT's effectiveness in transferring bulk data, a simple file transfer application, `gcp`, was created. `gcp` is similar to the secure copy (`scp`) program provided with the SSH suite. `gcp`, like `scp`, uses ssh to establish a remote authenticated connection and negotiate the control part of the transfer such as destination file path, file properties, etc. However, as the bulk data transfer occurs via GTC-to-GTC communication, the actual data is not encrypted during the transfer.

Figure 6.8 shows the experiment setup for these experiments. All experiments were performed on a Gigabit Ethernet network at Emulab [162] with a dummynet [118] middlebox controlling network bandwidth and latency. The machines were Emulab's "pc3000" nodes with 3 GHz 64-bit Intel Xeon processors and 2 GB of DRAM.

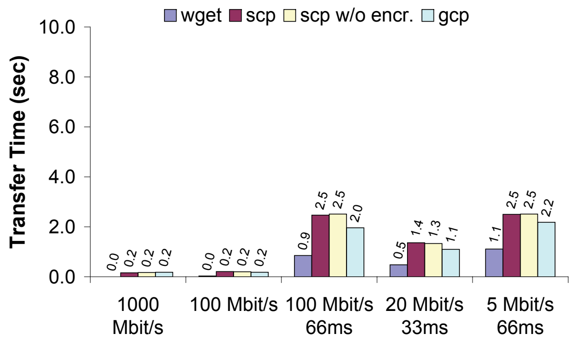
This program was used to transfer files of sizes ranging from 4 KB to 40 MB under varying network conditions. These results, shown in Figure 6.9, compare the performance of `gcp` to `wget`, a popular utility for HTTP downloads, `scp`, and a modified version of `scp` that, like `gcp`, does not have the overhead of encryption for



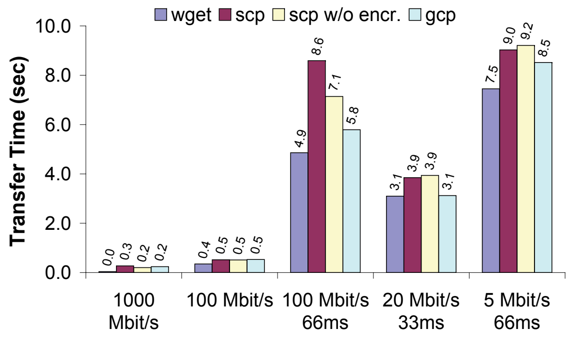
(a) 4 KB File Transfer



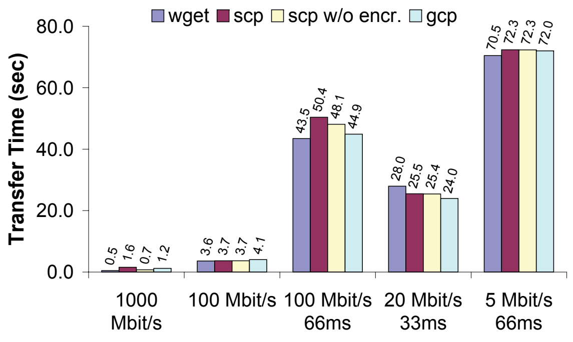
(b) 40 KB File Transfer



(c) 400 KB File Transfer



(d) 4 MB File Transfer



(e) 40 MB File Transfer

Figure 6.9: `gcp` vs. Other Standard File Transfer Tools

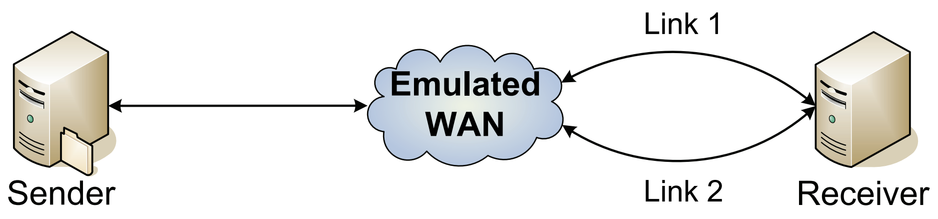
the bulk data transfer. All tools used the system's default values for TCP's send and receive buffer size. `gcp` uses the fixed-size chunker for these experiments. DOT was started with a cold cache for all experiments and there was no intra-file similarity that DOT could extract commonality from. All displayed results are the mean of 10 trials and displayed no noticeable standard deviation. This section concentrates on the results from the 40 MB file transfer as it is easier to observe the potential overheads of DOT.

As shown in Figure 6.9 (e), for WAN conditions, `gcp` exhibits very little or no overhead when compared to the other file transfer tools and its performance is equivalent to `scp` both with and without encryption. On the local Gigabit network, `gcp` begins to show overhead. In this case, `wget` is the fastest. Unlike both `scp` and `gcp`, the Apache HTTP server is always running and `wget` does not pay the overhead of spawning a remote process to accept the data transfer. `gcp` is only slightly slower than `scp`. This overhead arises primarily because the GTC on the sender side must hash the data twice: once to compute the OID for the entire object and once for the descriptors that describe parts of the object. The hashing has two effects. First, the computation of the whole data hash must occur before any data can be sent to the receiver, stalling the network temporarily. Second, the hashes are relatively expensive to compute.

This overhead can be reduced, and the network stall eliminated, by caching OIDs (as noted earlier, some systems such as EMC's Centera [55] already provide this capability). While the computational overhead of computing the chunk hashes remains, it can be overlapped with communication. The computational overhead could also be reduced by generating a cheaper version of the OID, perhaps by hashing the descriptors for the object. However, the latter approach sacrifices the uniqueness of the OID and removes its usefulness as an end-to-end data validity check.

6.5.2 Multi-Path Plugin

The multi-path plugin, described in Section 6.3.4, load balances between multiple GTC-to-GTC transfer plugins. Its performance was evaluated using the same Emulab nodes as above. The receiver, as shown in Figure 6.10, is configured with two network interfaces, and the sender with a 1 Gbit/s link to the dummynet middlebox that controls network bandwidth and latency.



The capacities of Links 1 and 2 are varied.

Figure 6.10: Experimental Setup for the Multi-Path Plugin Benchmark

The performance of the multi-path plugin was examined while transferring 40 MB, 4 MB, and 400 KB files via `gcp`. DOT was started with a cold cache for all experiments and there was no intra-file similarity from which DOT could extract commonality. As the same trends were shown in all of the results, only the 40 MB results are presented here, note that the performance on 400 KB files was somewhat lower because they were not sufficiently large to allow TCP to consume the available capacity. All transfers were conducted using FreeBSD 5.4 with the TCP socket buffers increased to 128 KB and the initial slow-start flight size set to 4 packets.

Table 6.4 presents several combinations of link bandwidths and latencies, showing that the multi-path plugin can substantially decrease transfer time. For example, when load balancing between two directly connected 100 Mbit/s Ethernet links, the multi-path plugin reduced the transfer time from 3.59 seconds on a single link to 1.90 seconds. The best possible time to transfer a 40 MB file over 100 Mbit/s Ethernet is 3.36 seconds, so this represents a substantial improvement over what a single link could accomplish. Note that the multi-path plugin was created and deployed with no modifications to `gcp` or the higher layer DOT functions.

In high bandwidth×delay networks, such as the 100 Mbit/s link with 66 ms latency (representing a high-speed cross-country link), TCP does not saturate the link with a relatively small 40 MB transfer. In this case, the benefits provided by the multi-path plugin are similar to those achieved by multiple stream support in GridFTP and other programs. Hence, using the multi-path plugin to bond a 100 Mbit/s and 10 Mbit/s link produces greater improvements than one might otherwise expect.

Finally, in cases with saturated links with high asymmetry between the link capacities (the second to last line in the table, a 10 Mbit/s link combined with a 0.1 Mbit/s link), the multi-path plugin provides some benefits by adding a second TCP stream,

Link 1	Link 2	single	multipath	savings
100/0	100/0	3.59	1.90	47.08%
	10/0	3.59	3.54	1.39%
100/33	100/33	21.46	11.15	48.04%
	10/33	21.46	13.58	36.72%
	1/33	21.46	20.44	4.75%
100/66	100/66	43.33	23.20	46.46%
	10/66	43.33	22.97	46.99%
	1/66	43.33	38.25	11.72%
10/66	10/66	48.39	23.42	51.60%
	1/66	48.39	39.20	18.99%
	0.1/66	48.39	44.14	8.78%
1/66	0.1/66	367.39	313.42	14.69%

Links indicate bandwidth in Mbit/s and latency in milliseconds. The **single** column shows the time for a gcp transfer using only the fastest link of the pair.

Table 6.4: Multi-Path Performance

reducing the transfer time by roughly 9%. To understand the impact of request borrowing, the same experiment was re-ru after disabling it. Without request borrowing, the multi-path plugin slowed the transfer down by almost a factor of three, requiring 128 seconds to complete a transfer that the single link gcp completed in 48 seconds. Without request borrowing, the queue for the fast link empties, and the transfer is blocked at the speed of the slowest link until its outstanding requests complete.

6.5.3 Portable Storage Plugin

DOT's use of portable storage was evaluated using a workload drawn from Internet Suspend/Resume (ISR) [123]. ISR is a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed storage system.

A key ISR challenge is in dealing with the large VM state, typically many tens of

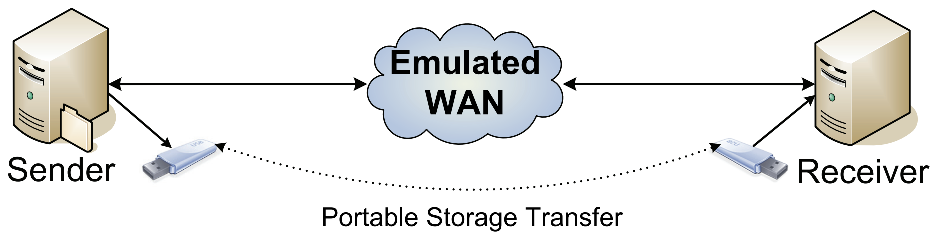
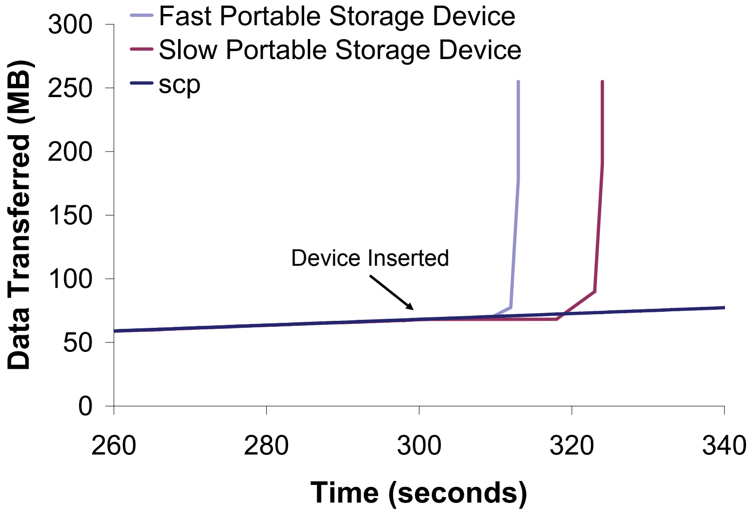


Figure 6.11: Experimental Setup for the Portable Storage Plugin Benchmark



The receiving machine completed the transfer from the USB flash device once it was inserted 300 seconds into the experiment. The `scp` transfer took 1126 seconds to complete.

Figure 6.12: Portable Storage Performance

GB. When a user suspends an ISR machine, the size of new data generated during the session is in the order of hundreds of MBs. This includes both the changes made to the virtual disk as well as the serialized copy of the virtual memory image. Given the prevalence of asymmetric links in residential areas, the time taken to transfer this data to the server that holds VM state can be substantial. However, if a user is willing to carry a portable storage device, part of the VM state can be copied to the device at suspend time.

To evaluate the benefit of DOT in such scenarios, a fast residential cable Internet link with a maximum download and upload speed of 4 Mbit/s and 2 Mbit/s respectively was emulated. Figure 6.11 shows the experiment setup for this experiment.

All experiments were performed on a Gigabit Ethernet network with a NetEm [66] middlebox controlling network bandwidth and latency. All machines had 3.2 GHz Intel Pentium 4 CPUs with 2 GB of SDRAM and ran the 2.6.10-1.770-SMP Linux kernel.

An analysis of data collected from the ISR test deployment on Carnegie Mellon's campus [94] revealed that the average size of the data transferred to the server after compression is 255 MB. To model this scenario, DOT was used to transfer a single 255 MB file representing the combined checkin state. DOT was started with a cold cache for all experiments and there was no intra-file similarity that DOT could extract commonality from. A USB flash device was inserted into the target machine approximately five minutes after the transfer was initiated. The test used two USB devices, a "fast" device (approx. 20 MB/sec. read times) and a "slow" device (approx. 8 MB/sec).

The speed of the transfer and total time for competition is presented in Figure 6.12. While the experiment was run five times for each device, there was no noticeable difference between runs. Immediately after the portable storage was inserted and detected at the end of the portable storage plugin's poll interval, the system experienced a small reduction in throughput as the event-driven DOT prototype blocked as it scanned the USB device for data. This slowdown could be avoided by spawning a helper process to perform the disk I/O but was small enough to be negligible. As it started reading the data from the device, the transfer rate increased substantially. The reason for the sharp spike in the amount of data transferred was that the portable storage plugin read the data from the portable device in random order but the `gcp` application needed it in sequential order. Thus, data was cached in the storage plugin by the GTC for in-order delivery. However, once all the data was read and cached, the transfer completed almost instantly.

6.5.4 Postfix SMTP Server

In order to quantify the benefits from using a DOT-enabled email server, an analysis of email workloads was required. A medium-volume research group mail server was therefore modified to trace all messages received by it. The analysis of the gathered trace served two purposes. First, it examined the benefits of DOT's content-hash-based chunked encoding and caching. While chunked encoding is well known to benefit bulk peer-to-peer downloads [44] and Web transfers [114], the analysis demonstrated that these benefits extend to an even wider range of applications. The second purpose was to generate a trace of email messages with which to evaluate the modified Postfix server.

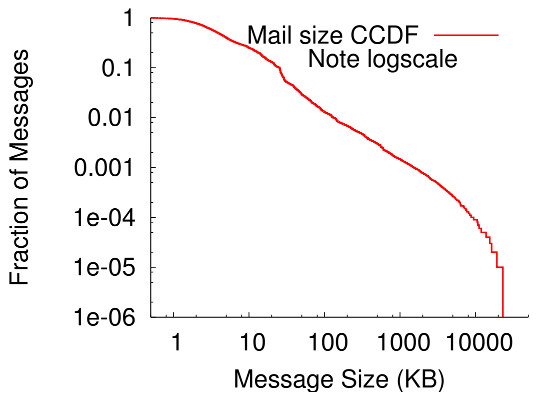
This section first presents the analytical results from the analysis of the email trace and then the evaluation of the DOT-enabled Postfix server.

Mail Server Trace Analysis

In the captured trace, each message records an actual SMTP conversation with the mail server, recorded by a Sendmail mail filter. The traces are anonymized, reporting the keyed hash (HMAC) of the sender, receiver, headers, and message body. The anonymization also chunks the message body using a static sized chunk and Rabin fingerprints and records the hashes of each chunk, corresponding to the ways in which DOT could chunk the message for transmission. The analysis below examines how many of these chunks DOT would transmit, but does not include the overhead of DOT's protocol messages. These overheads are approximately 100 bytes per 20 KB of data, or 0.5%, and are much smaller than the bandwidth savings or the variance between different days in the trace.

The email workload was an academic research group, with no notable email features (such as large mailing lists). The mail server handles approximately 3,400 messages per day. The traces cover 1,010,905 messages over 301 days. The distribution of email sizes, shown in Figure 6.13, appears heavy-tailed and consistent with other findings about email distribution. The sharp drop at 10-20 MB represents common cut-off values for allowed message sizes. Two message that were 224 MB and 86 MB were eliminated as they were believed to have been test messages sent by the server administrator.

This section examines four different scenarios. **SMTP default** examines the number of bytes sent when sending entire, unmodified messages. With **DOT body**, the mail server sends the headers of the message directly, and uses DOT to transfer the message body. Only whole-file caching is performed: either the OID has been received, or not. With **Rabin body**, the mail server still sends the headers separately, but uses Rabin fingerprinting to chunk the DOT transfer. Finally, **Rabin whole** sends the entire message, headers included, using DOT. Because the headers change



The distribution follows a heavy-tailed distribution until message size limits are reached.

Figure 6.13: Mail Message Size Distribution

for each message, sending a statically-chunked OID for the entire message is unlikely to result in significant savings. The Rabin whole method avoids this problem, at the cost of some redundancy in the first content block. Allowing the GTC to chunk the entire message also allows the simplest integration with Postfix by avoiding the need for the mail program to parse the message content when sending. The analysis assumes that DOT is enabled on both SMTP clients and servers. Table 6.5 shows the number of bytes sent by DOT in these scenarios.

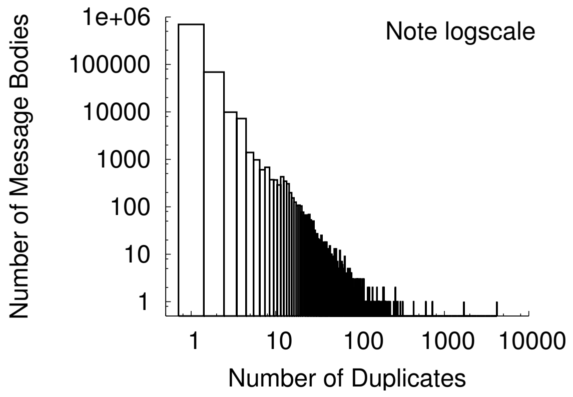
In all, DOT saves approximately 20% of the total message bytes transferred by the mail server. These benefits arise in a few ways. As can be seen in Figure 6.14, a moderate number of messages are duplicated exactly once, and a small number of messages are duplicated many times – nearly 100 copies of one message arrived at the mail server. Second, as Table 6.5 showed, there is considerable partial redundancy between email messages that can be exploited by the Rabin chunking.

While the study did not include the number of large email attachments that might be more common in corporate environments, it did observe a few such examples. 1.5% of the bytes in the trace came from a 10 MB email that was delivered eleven times to local users. The administrator of the machine revealed that a non-local mailing list to which several users were subscribed had received a message with a maximum-sized email attachment. Such incidents occur with moderate frequency, and are a common bane for email administrators. By alleviating the bandwidth, processing,

Method	Total Bytes	Percent Bytes
SMTP default	16,228 MB	-
DOT body	13,946 MB	85.94 %
Rabin body	12,013 MB	74.03 %
Rabin whole	12,907 MB	79.54 %

Savings using different DOT integration methods. The results does not include DOT overhead bytes.

Table 6.5: Savings in Email Traffic using DOT



There were 696,757 messages with unique bodies, 69,052 with a body duplicated once, and so on. One message was duplicated 4,180 times.

Figure 6.14: Histogram for Repetitions of Message Bodies

and storage pain from such incidents, DOT can help allow users to communicate in the way most convenient to them, instead of following arbitrary decrees about application suitability.

System Throughput

To evaluate DOT's overhead, 10,000 email messages were generated from the beginning of the mail trace. Each message is the same size as a message from the trace, and the message's content is generated from Rabin-fingerprint generated hash blocks in the trace message. Each unique hash value is deterministically assigned a unique content chunk by seeding a pseudo-random number generator with the hash value.

Program	Seconds	Bytes sent
Postfix	468	172 MB
Postfix-DOT	468	117 MB

Table 6.6: Postfix Throughput for 10,000 Email Messages

The generated emails preserve some of the similarity between email messages, but because it cannot regenerate the original content (or other content that contains the same Rabin chunk boundaries), the generated emails are slightly less redundant than the originals.

The 10,000 generated messages were replayed through Postfix running on the same machines used for the portable storage evaluation, connected with 100 Mbit/s Ethernet. Table 6.6 shows that the DOT-enabled Postfix required only 68% of the total bandwidth, including all protocol overhead, but both systems had identical transfer times.

The Postfix workload is extremely disk-seek bound. Neither the considerable bandwidth savings or the slight CPU overhead provided by DOT was sufficient to change this. A widespread adoption of DOT in mail servers would therefore have a positive effect on bandwidth use without imposing noticeable overhead.

6.6 Discussion

The previous sections presented the benefits observed using the initial DOT implementation and discussed some of the benefits DOT can realize from a transfer service in other scenarios. The experience with DOT thus far has revealed several lessons and remaining challenges.

Efficiency. The design of the DOT default transfer protocol assumes that its extra round trips can be amortized across file transfers of sufficient length. While this design is effective for DOT's intended target of large transfers, ideally the transfer service should be efficient for as many applications as possible, even if their transfers are small. For example, highly interactive Web services such as Google go to considerable effort to reduce the number of round-trips experienced by clients, and providers such as Akamai tout such reduction as a major benefit of their service. As the email study in Section 6.5.4 noted, email and many other protocols have a heavy-tailed distribution with numerous small files being transferred.

There are two likely approaches that can reduce DOT's overhead for small transfers: either allow the application to send small objects directly, or implement a transfer plugin that passes small objects in-line as a hint over its control channel to bypass the need for additional communication. For reasons discussed below, the former may be a better approach.

Application resilience. The DOT-enabled Postfix falls back to normal SMTP communication if either the remote host does not support DOT or if it cannot contact the GTC running on the local machine. This fallback to direct transfer makes the resulting Postfix robust enough to run as the primary mail program on a user's personal machine. As Cappos and Hartman noted on Planetlab, applications that depend on cutting-edge services are wise to fall back to simpler and more reliable mechanisms [35]. Until a transfer service like DOT becomes a part of the Internet's core infrastructure, this is sound advice for any application making use of DOT.

Security. In a system like DOT, applications have several choices that trade privacy for efficiency. The simplest way for an application to ensure privacy is to encrypt its data before sending it to the transfer service. Unfortunately, this places restrictions on the transfer service's ability to cache data and obtain data from independent sources. Another simple (and common) choice is to not encrypt the data at all, and trust in whatever host or network security mechanisms are available. This approach allows the transfer service the most opportunities to exploit features of the data to improve transfers, but offers little security to the user. A promising middle ground is to support convergent encryption [51], in which data blocks are encrypted using their own hash values as keys. Only recipients that know the hashes of the desired data can decrypt the data blocks.

DOT must interact with many different applications, each of which may have its own mechanisms and policies for providing security and privacy. A transfer service must support a wide variety of application requirements, and should not impose a particular security model or flavor of cryptography or key management. However, an efficient implementation of convergent encryption requires some support from the transfer service: The algorithm that splits data into chunks must be consistent across implementations, or the resulting encrypted blocks cannot be effectively cached. Exploring how to support convergent encryption in the near future will give a better insight into the tradeoffs.

Application preferences and negotiation. The current implementation of DOT performs data transfers completely independently of the application. While this suffices for a large and important class of applications, DOT should also be able to benefit applications that desire more control over how their transfers are effected. Examples of such applications include those that have specific encryption requirements or that wish to take advantage of different sending rates or quality of service categories.

The design of capability discovery and negotiation is an important part of future work. As a first cut at supporting plugins that must negotiate specific capabilities, two types of metadata are being added to DOT: per-object metadata that allows the receiving side to construct the plugin-chain required to process the data; and per-chunk metadata that allows the plugins to send information (such as encryption keys) needed to process the chunks when they are received.

Selecting Plugins and Data Sources. A DOT receiver may be able to obtain its desired data from multiple sources using several different plugins. The DOT architecture uses a hierarchy of plugins, some of which implement selection policies, to arbitrate between sources. If a variety of plugins become available, DOT's plugin configuration interface may need to evolve to support more sophisticated configurations that allow plugins to determine the capabilities of their upstream and downstream plugins. For example, it may be advantageous for a multi-path plugin to know more about the capacities of attached links or storage devices. While such an interface could be useful, its design would be purely speculative in the absence of a wide variety of plugins from which to choose.

Several of the plugins represent promising starts more than finished products. The multi-path plugin now supports fetching data from mirror sites and multi-homed servers [106]. The portable storage plugin needs to be extended to support a "rendezvous" service that allows devices to be plugged in to a third party machine, instead of directly to the receiver.

Supporting dynamically generated objects. The design calls for DOT to handle dynamically generated objects that cannot be fully hashed before transmission by assigning them a random OID. This change requires a small modification to the

API to return the OID before the `put` call has completed, and requires that the remote `get_descriptors` call be able to return an indicator that the receiver should continue checking for more descriptors. A drawback is that random OIDs sever the tie between the OID and the data contents, which prevents per-object (but not per-chunk) caching.

Exploiting structure in data. A final issue in the design of DOT is the division of labor between the application and the transfer service in dividing data into chunks. In the current design, the transfer service is solely responsible for chunking. As the evaluation of email traces showed, the use of Rabin fingerprinting can remove the need for application-specific chunking decisions in some cases, but there may remain other cases in which application knowledge is helpful. For instance, as shown in Ganesh and Cedar, many applications transfer structured data. Databases, for certain workloads, may return data that has repetitions at the row level, which may be much smaller than DOT's default chunk size. While the current techniques work well for email and Web objects, the use of application or domain-specific chunking algorithms merits further exploration.

6.7 Retrospective

DOT's design draws heavily from the other systems described in this dissertation. It uses techniques similar to those found in CASPER, Lookaside Caching, Ganesh, and Cedar, and can be viewed as a generalization of these systems. However, it is still useful to examine the tradeoffs made by switching from the application-specific solutions described earlier to the generic data transfer architecture presented in this chapter. Further, what would the above systems have looked like if a data transfer service like DOT was available to begin with?

By comparing and contrasting the previous systems with DOT, one can quickly see that some systems would easily work with a transfer service while others would require more effort. Both CASPER and Lookaside Caching would work seamlessly with DOT without any significant performance difference or the need to re-architect the original system. For example, CASPER's method of fetching a recipe from the file server and then redirecting data fetches is identical to DOT's approach of first fetching a recipe from the sending application's transfer service and then obtaining data from nearby sources such as edge caches and local mirrors. Similarly, DOT's OID would be an exact substitute for Lookaside Caching's need for a whole-file hash. Further, DOT's support for content-based transfers as well as non-networked portable storage would have dramatically reduced the implementation effort.

However, in contrasting DOT with Ganesh and Cedar, it is easier to see the tradeoffs faced by system developers. Once data has been transferred to DOT, the transfer service no longer has application or domain-specific knowledge about any inherent structure in the data. If used together with Ganesh, a generic transfer service would not be able to precisely determine chunk boundaries because it would not know where each row of the database result ends. This problem was not faced by systems like CASPER as the chunking algorithms only dealt with opaque data. To address this, it is possible to build in heuristics within the transfer service to detect popular object types and exploit knowledge of the internal formats of these objects. While this requires an additional implementation effort, it should be possible to optimize a large fraction of transfers.

Examples of other tradeoffs include the fact that application-specific solutions can be optimized for specific workloads. For example, given the relatively small size of database results and the cost of adding additional WAN round trips, both Ganesh and Cedar try to aggressively minimize the number of required round trips for each transfer. Ganesh's server-side proxy can efficiently predict the contents of the client's cache while Cedar's client first estimates what its local replica might contain before sending a request to the database server. In contrast, DOT would first fetch the OID for the result, then a recipe for the OID, and finally any missing data.

Due to a transfer service's extensible nature, features such as Ganesh's cache prediction algorithm can be easily incorporated within DOT. The stale database replica used by Cedar can be implemented as a network or even a storage plugin that can be addressed via a hint, or URI, that would include the SQL query being optimized. However, reducing the number of round trips, while possible, is a fundamental design decision. DOT's design is geared towards delivering data transfer functionality that is beneficial for a large number of applications. While a generic transfer service can improve performance in most cases, the performance gains seen, for certain categories of applications, might be lower than those of application-specific solutions. Implementing application-specific optimizations would introduce extra complexity in both the transfer service's implementation and its application API. Thus, I believe that the current design is the correct choice as it would allow for a faster adoption and deployment of a transfer service.

6.8 Summary

This chapter presented the design and implementation of an extensible data-oriented transfer service, DOT. DOT decouples application-specific content negotiation from the more general process of transferring data across the network. Using such a transfer service reduces re-implementation at the application layer and facilitates the adoption of new technologies to improve data transfer. Through a set of microbenchmarks and an examination of a production mail server modified to use DOT, this chapter has shown that the DOT architecture imposes little overhead. DOT provides significant benefits, reducing bandwidth use and making new functionality such as multi-path or portable storage-based transfers readily available.

While the design still faces several challenges, introducing data transfer as a system *service* is a worthwhile goal. While a number of different systems in this dissertation have shown the advantage of using content-addressable techniques, a widely deployed transfer service might be the simplest method for a large category of applications to take advantage of content-based optimizations. If universally deployed, a content-addressable service like DOT could provide significant benefits to both legacy and future applications, networks, and ultimately, to users.

Chapter 7

Related Work

This dissertation has made a number of contributions in showing the value of using content-addressable techniques for legacy and future client-server systems over WANs. This dissertation shows how performance can be improved even though the content-addressable data sources used are untrusted, might contain data that is incorrectly named or even no useful data, and make no guarantees about consistency.

As the systems covered in this dissertation are diverse and there is no single related system, this chapter discusses related work that intersects different areas of it. First, Section 7.1 describes other systems that use similar content-addressable techniques and, in particular, content-addressable storage systems. Then, in Section 7.2, other approaches to using portable storage are discussed. Next, Section 7.3 discusses other database approaches to improving performance over the WAN. Finally, Section 7.4 describes other approaches to improving bulk data transfers.

7.1 Content-Addressable Systems

The use of content-addressable techniques to identify similar data in storage systems has been explored previously. Single Instance Storage [23] uses content hashes for compression, to coalesce duplicate files, and Venti [107] employs a similar approach at the block level. Unlike delta encoding [52, 77, 88, 146, 156], these techniques do not need a previous version of an object to compare against or any fixed file references.

While some systems, such as Bittorrent [44], use fixed size blocks to divide objects, the Rabin Fingerprinting technique [83, 108] has been used by a large number of systems to identify intra and inter-object similarity for objects when no application or domain-specific knowledge is available. Spring and Wetherall [137] used this technique to identify and remove redundant network traffic, the Low Bandwidth File System [91] for distributed file systems, the Pastiche backup system [46] for P2P backups, and by Layer-2 bandwidth optimizers such as Riverbed to discover spans of commonality in data. This approach, also used by CASPER and DOT, is especially useful when it can be assumed that data items are modified *in-place* through insertions, deletions, and updates. However, as shown in Ganesh, when this assumption fails, the performance of this technique can show a dramatic drop in performance. In such situations, as shown by both Ganesh and Cedar, using application-specific algorithms for detecting similarity can be very valuable.

Example of other P2P networks that use cryptographic hashes to address data include Distributed Hash Tables (DHTs) such as Chord [140], Pastry [120], and CAN [112] and the storage and file systems, including CFS [48], PAST [53], and Ivy [92], built on top of them. These systems are completely decentralized and share the model that participants that join the overlay offer write access to their local storage to other participants, either directly or indirectly. Adopting this architecture requires a very dramatic shift in system design and usage. On the other hand, this dissertation has shown how content-addressable storage can be used with legacy systems, in a transparent manner, and without depending on the storage providers always being available. In fact, one can envisage a hybrid model where the above DHTs and P2P networks can act as data providers for systems like CASPER and DOT. Clients can choose to use the widely dispersed providers when beneficial but always have the benefit of being able to fall back on the centralized file servers.

Almost all of the above systems only use cryptographic hashes internally and do not expose this metadata. In contrast, this dissertation has shown the value of making the hash value a first class object and using it to query external data sources. This increases the degrees of freedom available to a client as even potentially untrusted sources can be accessed. Given the collision-resistant feature of cryptographic hash functions that make the data self-verifying, it is easy for clients to check for correctness. More recently, systems such as Shark [12] have used this approach to allow distributed file system clients to securely share their local caches.

Further, a large number of the above systems use cryptographic hashes for addressing persistent data. This represents a deeper level of faith in their collision-resistance than that assumed by this dissertation. If time reveals shortcomings or weaknesses in the hash algorithm, the effort involved in correcting the flaw for such systems is much greater. As all the systems used in this dissertation only use cryptographic hash functions for data in-flight, addressing such an event for them is merely a matter of replacing the hash algorithm with a stronger one.

Note that using cryptographic hashes to query untrusted data sources might leak information. For example, requesting files corresponding to a particular hash value could allow the provider to determine what data the client is interested in. Of course, this would assume that the provider had the data, and therefore the corresponding hash value, to begin with. Using anonymous routing layers such as onion routers [50] could mitigate this problem. Further, in systems such as Lookaside Caching, Ganesh, and Cedar, the data providers are all trusted as they are either under the control of the system (Ganesh) or the user (Lookaside Caching and Cedar).

Alternatively, if a client exposes its local cache, external clients might be able to determine the user's working set or look for well known objects such as pirated digital media, sensitive or potentially embarrassing documents, etc. However, as shown by Annapureddy et al. [12], this kind of sharing can be done in a secure manner. Scenarios where a malicious server can inspect a client's cache [89] does not significantly impact this work as servers form a trusted part of the studied client-server systems.

7.2 Portable Storage

The lookaside caching technique described in this dissertation has very different goals and design philosophy from systems such as PersonalRAID [135], Segank [136], and Footloose [100]. The starting point for lookaside caching is the well-entrenched base of distributed file systems in existence today. It is assumed that these are successful because they offer genuine value to their users. Hence, the goal of lookaside caching is to integrate portable storage devices into such a system in a manner that is minimally disruptive of its existing usage model. In addition, this technique makes no changes to the native file system format of a portable storage device; all it requires is that the device be mountable as a local file system at any client of the distributed file system.

In contrast, all the above systems take a much richer view of the role of portable storage devices. They view them as first-class citizens rather than as adjuncts to a distributed file system. They also use customized storage layouts on the devices. Therefore, while lookaside caching's design and implementation is much simpler, it is also more limited in functionality. Closer to the lookaside caching work is the Blue file system [96] that combines the use of portable storage with distributed file systems, but with an emphasis on energy efficiency.

Other projects with overlapping goals include the Personal Server [161] and Soulpad [31] efforts. The Personal Server system tries to integrate computation, communication, and storage to provide ubiquitous access to personal information and applications. However, being a mobile computer, it is more heavyweight in terms of the hardware requirements. The Soulpad system uses a small standalone portable device to carry a user's data by encapsulating the operating system along with a suspended virtual machine on the device. There are also a number of commercial solutions providing mobility solutions through the use of portable storage devices. Migo [86], one of these products, has combined a USB portable storage device with synchronization software for personal files, email, and other settings. U3 [157], an industry effort, allows applications to be executed directly from a specially formatted USB flash drive. However, these systems focus exclusively on the use of the portable device and do not integrate network storage.

7.3 Improving Database Performance

To the best of my knowledge, Ganesh and Cedar are the first systems that combine the use of hash-based techniques with caching of database results to improve throughput and response times for applications with dynamic content. I also believe that they are also the first systems to demonstrate the benefits of using structural information found in database results for detecting similarity. This section discusses alternative approaches to caching database content and improving performance for mobile clients.

7.3.1 Caching Database Content

At the database layer, a number of systems have advocated middle-tier caching where parts of the database are replicated at the edge or server. Examples of such systems include DBCache [6, 25, 82], MTCache [79], DBProxy [8, 9], and ABR-cache [49]. These systems either cache entire tables in what is essentially a replicated database or use materialized views from previous query replies [78]. They require tight integration, and therefore source code access, with the back-end database to ensure a time bound on the propagation of updates. These systems are also usually targeted towards workloads that do not require strict consistency and can tolerate stale data. Further, unlike Ganesh or Cedar, some of these mid-tier caching solutions [6, 7], suffer from the complexity of having to participate in query planning and distributed query processing.

Gao et al. [59] propose using a distributed object replication architecture where the data store's consistency requirements are adapted on a per-application basis. These solutions require substantial developer resources and detailed understanding of the application being modified. While systems that attempt to automate the partitioning and replication of an application's database exist [133], they do not provide full transaction semantics. In comparison, both Ganesh and Cedar are completely transparent to both applications and databases and do not weaken any of the semantics provided by the underlying database.

Recent work in the evaluation of edge caching options for dynamic web sites [165] has suggested that, without careful planning, employing complex offloading strategies can hurt performance. Instead, the work advocates for an architecture in which all tiers except the database should be offloaded to the edge. For centralized databases, to improve database scalability, C-JDBC [38], Ganymed [103], and GlobeTP [63] also advocate the use of an interposition-based architecture to transparently cluster and replicate databases at the middleware level. The approaches of these architectures and this dissertation are complementary and they would benefit each other.

Moving up to the presentation layer, there has been widespread adoption of fragment-based caching [54, 110], which improves cache utilization by separately caching different parts of generated web pages. While fragment-based caching works at the edge, a recent proposal has proposed moving web page assembly to the clients to optimize content delivery [109]. Delta encoding [16, 88] of web content has also been advocated as an approach to reducing data transfer. While Ganesh and Cedar are not used at the presentation layer, the same principles have been applied in Duplicate Transfer Detection [89] to increase web cache efficiency as well as for web access across bandwidth limited links [114].

7.3.2 Mobile Database Systems

While the benefits of caching data in mobile systems has long been known [41], most systems weaken traditional consistency semantics for performance reasons. Weakly-Consistent replication in Bayou [144] was proposed for collaborative systems. Alonso et al. [5] proposed *quasi-copies* for database systems to improve performance by relaxing consistency when clients participate in caching data. The Mariposa [131] distributed data manager showed how consistent, although potentially stale, views can be seen across a network of machines. Gray et al. [62] proposed a two-tier replication scheme that allows tentative update transactions on the mobile client that are later committed to the master copy.

An obvious disadvantage of these systems is that they provide a different consistency or transactional model than what developers and users expect. Tentative transactions increase the probability of conflicts and require additional application complexity or user interaction for conflict resolution. In contrast, this dissertation's focus is on maintaining the transactional and consistency semantics provided by the underlying database. While this design decision prevents disconnected operation, this is an acceptable tradeoff for an important class of real-world usage scenarios.

Barbará and Imielinski [18] suggest informing mobile databases clients of changes made to a database server by broadcasting Invalidation Reports (IRs) at periodic intervals. However, being a push-based system, IRs are only effective if a large number of clients are interested in the same data. While hybrid push-pull systems have been proposed [1], in the absence of locality, they degenerate into a pull-based behavior. Further, if clients are disconnected for longer than the time window covered by the IRs, they need to discard their entire cache. While there is related work to alleviate this problem [34, 70, 163], these techniques are are only effective in a broadcast medium when a large number of clients are interested in the same data. IRs also add significant latency to queries as each mobile client has to wait for the periodic IR broadcast before it can verify data freshness. In contrast, systems like Cedar use a purely opportunistic approach to finding the required results. Barbará [17] provides a more complete survey of previous work on IRs and mobile databases.

Some systems [17, 167] allow clients to obtain an exclusive copy of the section of the database it is accessing. This can significantly degrade performance for other clients when mobile clients are weakly connected. By recognizing that the local database replica could be stale, Cedar instead ensures that there is no strong dependency between the main database server and the mobile client.

Cedar's hoard profiles also bear some similarity to clustering attributes [15] and client profiles [28]. While these methods are also used to express preferences for database content, they have very different functions. Clustering attributes define a database server's storage layout for improved access by mobile clients. Client profiles are used to indicate freshness and latency requirements for systems that relax consistency.

7.4 Improving WAN-Based Bulk Data Transfers

There are a number of efforts that have explored better ways to accomplish data transfers and architectures that insert a protocol between the application and the transport layers. DOT differs from prior work in choosing an architectural split (running as a service and primarily supporting point-to-point object transfers) that is both powerful enough to support a diverse set of underlying mechanisms, and generic enough to apply to a wide variety of applications.

The design for DOT borrows from content-addressable systems such as CASPER, BitTorrent, and DHTs. Like DHTs, DOT uses content-based naming to provide an application-independent handle on data. DOT also bears similarity to the Internet Backplane Protocol (IBP) [19], which aims to unify storage and transfer, particularly in Grid applications. Unlike IBP, DOT does not specify a particular underlying method for data transfer; rather, DOT separates transfer methods from applications, so that future protocols like IBP could be implemented and deployed more rapidly.

At the protocol level, BEEP, the Blocks Extensible Exchange Protocol [119], is close in spirit to DOT. BEEP aims to save application designers from re-inventing an application protocol each time they create a new application, by providing features such as subchannel multiplexing and capability negotiation on top of underlying transport layers. BEEP is a protocol framework, available as a library against which applications can link and then extend to suit their own needs. BEEP's scope covers the application's content negotiation and data transfer. In contrast, DOT is a service that is shared by all applications; thus, a single new DOT plug-in can provide new transfer mechanisms or interpose on data to all applications.

Protocols such as FTP [105], GridFTP [145], ISO FTAM (ISO 8571), and even HTTP [57] can be used by applications to access data objects, either by invoking a client for that protocol or by implementing it within the application protocol. Many of the transfer techniques that distinguish these protocols (e.g., GridFTP's use of parallel data streams or negotiation of transfer buffer sizes) could be implemented as a DOT transfer plugin. By doing so, an unmodified "DOT-based" FTP client would then be able to take advantage of the new functionality, reducing the effort required to adopt the protocol enhancements.

Proxies are commonly used to process legacy application traffic in new ways. While DOT aims to be more general than application-specific examples such as Web proxies, it bears resemblance to generic proxies such as the DNS-based OCALA [72] or the packet capture approaches used by RON [11] and the X-bone [153] to re-route traffic to an overlay. The drawback of these more generic approaches is that they lack knowledge of what the application is attempting to do (e.g., transfer a certain block of data) and so become limited in the tools they can apply. However, some of the advantages of DOT can be realized through the use of protocol-specific proxies. For example, the DOT-enabled Postfix email server can be used as a mail relay or mail proxy when co-located with unmodified mail servers.

The initial DOT plugins borrow techniques from several systems described in this dissertation. From CASPER, DOT borrows the idea of using recipes to reduce bandwidth usage. The portable storage plugin was strongly influenced by the work on lookaside caching.

Finally, distributed object and file systems attempt to provide a common layer for implementing distributed systems. These systems range from AFS [68] and NFS [32] to research systems too numerous to cover in detail. Prominent among these are the storage systems that use content-addressable techniques for routing, abstracting identity, and saving bandwidth and storage [46, 48, 53, 91, 107, 149]. Like these distributed storage systems, DOT aims to mask the underlying mechanics of network data transfer from applications. Unlike these systems, DOT does not provide a single mechanism for performing the transfers. Its extensible architecture does not assume a “one size fits all” model for the ways applications retrieve their data. The user of a DOT-based application should not be forced to depend on an underlying distributed system if they only wish to perform point-to-point transfers. DOT complements many of these distributed systems by providing a single location where service developers can hook in, allowing applications to take advantage of their services.

Chapter 8

Conclusion

Efficient access to bulk data over the Internet has become a critical problem in today's world. Even while bandwidth, both in the core and at the edge of the network, is improving, the simultaneous growth in the use of digital media and large personal data sets is placing increasing demands on it [2, 90]. Further, with increasing trends towards mobility and data access over cellular and wireless networks, optimizing bulk data transfers for client-server systems will be valuable in the years to come.

This dissertation has presented a detailed exploration of how the use of content-addressable techniques can be a powerful tool in optimizing these data transfers over the WAN. It has put forward the thesis that these techniques can be useful for both legacy and future client-server systems. They allow external data sources to be used without weakening any attributes, including consistency, of legacy systems. This thesis has been empirically validated by a number of case studies. First, the use of content-addressable techniques was examined in the context of four distinct systems that covered the two traditional forms of data storage: file systems and database systems. Next, the underlying principles behind these systems were generalized to present a unified architecture for improving bulk data transfer across different systems, applications, and protocols.

In this chapter, Section 8.1 begins with a review of the contributions of this dissertation. Next, Section 8.2 discusses future directions for the research outlined in this work. Finally, Section 8.3 concludes with a review of the major lessons learned from this dissertation.

8.1 Contributions

The contributions of this dissertation can be divided into three broad areas. First, the conceptual contributions consist of the novel ideas generated by this research. Second, there exist a number of software artifacts that were built as a part of this dissertation. The final contribution is the experimental results that validate the thesis. This section discusses each of these contributions in turn.

8.1.1 Conceptual Contributions

While other systems have identified content-addressable techniques as a possible solution to the bulk data transfer problem, the proposed solutions have usually required significant changes to system design. For example, using DHTs [120, 140] involves a paradigm shift from traditional client-server based architectures to a peer-to-peer system. Other solutions have required significant system modification or redesign [100], introduced new standalone applications [44], relaxed traditional consistency semantics [136], or required changes in user behavior.

In contrast, this work has shown how, for traditional client-server systems, content-addressable techniques can be used in a transparent manner without weakening any of the traditional properties or semantics of the original system. For legacy systems with standardized interfaces, these techniques can be used without modification to the original systems or applications. Even for systems that lack such standardized interfaces, this dissertation has shown that the changes required are minor and confined to small areas of the codebase. Further, applications that use these systems require no modifications at all.

8.1.2 Software Artifacts

In the course of this dissertation, I developed a number of major software artifacts that were used to validate the thesis. They include the CASPER file system which has been used by other researchers and the Lookaside Caching system that is now a part of the production Coda file system [124].

Next, I designed and created the Ganesh and Cedar systems that were used to evaluate the effectiveness of content-addressable techniques with database systems.

These two artifacts have been used in class projects and, once open sourced, could serve as the basis for further research in the area. My research into database systems also involved significant work on benchmarks. I made significant improvements to the `AUCTION` and `BBOARD` benchmarks used to evaluate Ganesh. I also generated new datasets for these benchmarks that more accurately reflect current systems. These two benchmarks have since been used for the evaluation of other database systems. The MobileSales benchmark, created for the evaluation of mobile database clients, has also been open sourced.

Finally, the DOT system that was created as a generalization of my work has also been released as open source software (OSS). It is predicted that DOT will serve as an infrastructure for inventors of other innovative transfer techniques. Apart from being used as the codebase in the SET [106] system, DOT is also being used by other projects outside of Carnegie Mellon University.

8.1.3 Evaluation Results

The evaluation results in this dissertation contain important insights for future systems that use content-addressable techniques. They have shown how performance gains in WANs are possible by exploiting commonality in data streams. As the use of content-addressable techniques have low overhead, a system's performance can still be improved even when the amount of commonality is low.

The evaluation has also shown in which cases the use of content-addressable techniques should be avoided. Such scenarios include very high bandwidth and low latency networks. In these scenarios, the cost of extra round trips to fetch recipes and the overhead of computing cryptographic hashes over the data might outweigh any savings in the actual data transfer. While the cost of hashing will be reduced in the future with the advent of increasingly powerful CPUs, the extra round trips needed to fetch a recipe might make the use of these techniques unsuitable for applications that predominantly initiate small data transfers. Examples of applications include Instant Messaging and the `telnet` remote console tool. Other unsuitable applications would include those that have strict real-time constraints, such as video conferencing.

Finally, the results have also shown that a broad category of applications and workloads can benefit from similarity. These workloads include email, users of a distributed file system, users of virtual machines, ecommerce systems, and mobile database access. It is predicted that the general trends will also be visible in other workloads that exhibit data similarity or sharing.

8.2 Future Work

This dissertation has opened up several avenues for future research. Some of these are extensions of work described here while others have a much broader scope. This section describes some of the possible directions for building upon the research described here.

8.2.1 System Extensions

For a successful adoption, a number of systems described in this dissertation need to be extended to support for a richer set of features. For example, in Cedar, hoarding data on a mobile client can have privacy, security, and regulatory concerns as the loss or theft of a mobile device can expose confidential data. While one can elide sensitive database fields, this can lead to a performance loss if queries frequently reference those fields. Another possible alternative for data protection is to encrypt sensitive data. Encryption could be implemented in a number of ways, including at the storage layer [116], within the client replica [65], or even within the client proxy [45]. These choices need to be carefully evaluated in order to minimize encryption's impact on resource-limited clients. Cedar's impact on power consumption has also not yet been considered. While reducing network transmission has been shown to save power, there is a tradeoff as Cedar uses computational cycles to achieve the reduction. A careful investigation is needed to determine how power should be factored into Cedar's adaptation mechanisms.

In the current DOT prototype, all transfers are treated equally. While this suffices for a large and important class of applications, applications might desire more control over their transfers. Examples of such applications include those that have specific encryption requirements, those that wish to give higher or lower priorities to transfers or take advantage of quality of service categories. Further, in the event of a widescale deployment of a transfer service, it is highly probable that different implementations would support overlapping methods of transferring data. In such scenarios, DOT's plugin configuration interface may need to evolve to support more sophisticated configurations that allow plugins to support feature negotiation, determine the capabilities of their upstream and downstream plugins, and dynamically configure plugin chains.

8.2.2 Studies of Similarity in the Real World

While this dissertation has looked at some studies of similarity in the real world, including the source code in CASPER and email trace data in DOT, I believe a more careful study of similarity is needed. Due to privacy and feasibility issues, previous studies have either limited themselves to metadata [2, 125, 132] or been restricted to publicly available datasets [89, 104]. I believe a wider commonality study is needed that covers both corporate and home users. Not only will this help in accurately identifying workloads that would benefit from the use of content-addressable techniques, but it will also quantify the degree of cross-user, or communal, sharing of data.

8.2.3 Provisioning Content Addressable Repositories

As the systems described in this dissertation depend upon finding similarity to improve performance, correctly populating content-addressable data sources can make a significant impact on system performance. While the increasingly large storage capacities of both hard drives will help, transfer time to populate remote networked stores might be limited by a client's available network bandwidth.

While tools such as the SQL log miner used by Cedar are a step in the right direction, more work needs to be done. First, the data selection should be automated as far as possible so that minimum input is needed from users. Machine learning techniques could be very useful to accomplish this goal as they can discover user behavior based on historical use and tailor caching based on these observations. When storage on the repository is limited, these observations can also be used to select the most frequently accessed data. Second, security concerns must be addressed so that either no personal or confidential data is injected into the network or that is it properly encrypted and secured.

8.2.4 Choosing between Content Addressable Repositories

Apart from the deciding what data should be cached, properly selecting data sources for use will also be very valuable in both improving performance as well as ensuring low overhead when content-addressable techniques are used. For example, as Cedar showed, due to the slow seek time of hard drives, even if the needed data is present on local storage, it might be faster to access it over the LAN if the server has it present in its cache. Similarly, if the network has a higher bandwidth than an attached flash-based storage device, lookaside caching showed that it might be faster to use the network. On the other hand, if the original source is overloaded, it might be beneficial to explore the use of external data repositories that might have a higher access time [106].

While these are simple examples, the problem becomes more complex as the number of potential sources increases. Using DOT as an example, there might be a number of different methods of transferring data, with each having different performance, latency, and overhead characteristics. As using all of them in parallel might not always be desirable or even feasible, the decision on what sources to use and when needs to be made into a dynamic process. Assuming that the costs and benefits of each resource are measurable, an optimization framework to choose between different networked and non-networked resources could prove to be very valuable.

8.3 Concluding Remarks

This dissertation has put forward the claim that content-addressable techniques could be used to optimize bulk data transfer in WAN-based client-server systems. By detecting similarity between different sources of data, these techniques allow external sources to be used without weakening any attributes, including consistency, of legacy systems with no or minor changes to the original system.

This claim has been empirically validated through the use of five case studies that encompass the two traditional forms of data storage, file systems and database systems, and then generalized in the form of a generic transfer service that can be shared by different applications. Each of these case studies showed that, even with the addition of content-addressable techniques, the resulting system still maintained the attributes and semantics of the original system. A careful evaluation of the prototypes with both synthetic and real benchmarks showed that the use of these content-addressable techniques had low overhead and provided a substantial performance improvement when compared to the original system.

Bibliography

[1] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 183–194, Tucson, AZ, May 1997.

[2] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, San Jose, CA, February 2007.

[3] Akamai. `http://www.akamai.com/`.

[4] Aditya Akella, Srinivasan Seshan, and Anees Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 101–114, Miami Beach, FL, October 2003.

[5] Rafael Alonso, Daniel Barbara, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transaction on Database Systems*, 15 (3):359–384, 1990.

[6] Mehmet Altinel, Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Bruce G. Lindsay, Honguk Woo, and Larry Brown. DBCache: Database caching for web application servers. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 612–612, Madison, WI, June 2002.

[7] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 718–729, Berlin, Germany, September 2003.

[8] Khalil Amiri, Renu Tewari, Sanghyun Park, and Sriram Padmanabhan. On space management in a dynamic edge data cache. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, pages 37–42, Madison, WI, June 2002.

[9] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. DBProxy: A dynamic data cache for web applications. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 821–831, Bangalore, India, March 2003.

[10] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *Proceedings of the 5th Annual IEEE International Workshop on Workload Characterization (WWC-5)*, pages 3–13, Austin, TX, November 2002.

[11] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, October 2001.

[12] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, Boston, MA, May 2005.

[13] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 43–56, Banff, Canada, October 2001.

[14] Nadia Ben Azzouna and Frbrice Guillemin. Analysis of ADSL traffic on an IP backbone link. In *Proceedings of the IEEE 2003 Conference on Global Communications (GlobeCom)*, pages 3742–3746, San Francisco, CA, December 2003.

[15] B. R. Badrinath and Shirish H. Phatak. On clustering in database servers for supporting mobile clients. *Cluster Computing*, 1(2):149–159, 1998.

[16] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of the 1997 USENIX Technical Conference*, pages 289–303, Anaheim, CA, January 1997.

[17] Daniel Barbará. Mobile computing and databases - a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[18] Daniel Barbará and Tomasz Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Minneapolis, MN, May 1994.

[19] Micah Beck, Terry Moore, and James S. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 339–346, Pittsburgh, PA, August 2002.

[20] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[21] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[22] J. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 85–90, Boston, MA, May 2006.

[23] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000.

[24] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):34–43, 2000.

[25] Christof Bornhvd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. DBCache: Middle-tier database caching for highly scalable e-business architectures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 662, San Diego, CA, June 2003.

[26] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. `http://www.w3.org/TR/REC-xml`.

[27] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[28] Laura Bright and Louiqa Raschid. Using latency-recency profiles for data delivery on the web. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 550–561, Hong Kong, China, August 2002.

[29] Andrei Broder, Steven Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference (WWW)*, pages 1157–1166, Santa Clara, CA, April 1997.

[30] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.

[31] Ramón Cáceres, Casey Carter, Chandra Narayanaswami, and Mandayam Raghunath. Reincarnating PCs with portable soulpads. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 65–78, Seattle, WA, June 2005.

[32] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS Version 3 Protocol Specification*, June 1995. RFC 1813.

[33] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, San Francisco, CA, December 2004.

[34] Guohong Cao. A scalable low-latency cache invalidation strategy for mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 15(5): 1251–1265, 2003.

[35] Justin Cappos and John Hartman. Why it is hard to build a long-running service on PlanetLab. In *Proceedings of the 2nd Workshop on Real, Large Distributed Systems (WORLDS)*, pages 61–66, San Francisco, CA, December 2005.

[36] Mark Carson and Darrin Santay. NIST Net: A linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[37] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, pages 242–261, Rio de Janeiro, Brazil, June 2003.

[38] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 9–18, Boston, MA, June 2004.

[39] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 128–136, Orlando, FL, June 1982.

[40] Rajiv Chakravorty, Suman Banerjee, Pablo Rodriguez, Julian Chesterfield, and Ian Pratt. Performance optimizations for wireless wide-area networks: Comparative study and experimental evaluation. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 159–173, Philadelphia, PA, September 2004.

[41] Boris Y. Chan, Antonio Si, and Hong V. Leong. A framework for cache management for mobile databases: Design and evaluation. *Distributed and Parallel Databases*, 10(1):23–57, 2001.

[42] Mun Choon Chan and Ramachandran Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. *Wireless Networks*, 11(1-2):81–97, 2005.

[43] Ludmila Cherkasova and Jangwon Lee. FastReplica: Efficient large file distribution within content delivery networks. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, March 2003.

[44] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, pages 102–111, Berkeley, CA, June 2003.

[45] Mark D. Corner and Brian D. Noble. Protecting applications with transient authentication. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 57–70, San Francisco, CA, May 2003.

[46] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, Boston, MA, December 2002.

[47] Paul Currion, Chamindra de Silva, and Bartel Van de Walle. Open source software for disaster management. *Communications of the ACM*, 50(3):61–65, 2007.

[48] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, Banff, Canada, October 2001.

[49] Louis Degenaro, Arun Iyengar, Ilya Lipkind, and Isabelle Rouvellou. A middleware system which intelligently caches query results. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 24–44, New York, NY, April 2000.

[50] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, August 2004.

[51] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 617–624, Vienna, Austria, July 2002.

[52] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 113–126, San Antonio, TX, June 2003.

[53] Peter Druschel and Ant Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 75–80, Schloss Elmau, Germany, May 2001.

[54] Edge Side Includes. `http://www.esi.org`.

[55] EMC Centera Content Addressed Storage System. EMC Corporation, 2003. `http://www.emc.com/`.

[56] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 27–34, Karlsruhe, Germany, August 2003.

[57] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1*. Internet Engineering Task Force, January 1997. RFC 2068.

[58] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and Mahadev Satyanarayanan. Data staging on untrusted surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 15–28, San Francisco, CA, March 2003.

[59] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. Application specific data replication for edge services. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, pages 449–460, Budapest, Hungary, May 2003.

[60] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995. ISBN 1-55615-815-7.

[61] Christos Gkantsidis and Pablo Rodriguez. Network coding for large scale content distribution. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 2235–2245, Miami, FL, March 2005.

[62] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[63] Tobias Groothuyse, Swaminathan Sivasubramanian, and Guillaume Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 301–310, Banff, Canada, May 2007.

[64] A. Gulbrandsen, P. Vixie, and L. Esibov. *A DNS RR for specifying the location of services (DNS SRV).* Internet Engineering Task Force, February 2000. RFC 2782.

[65] Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 216–227, Madison, WI, June 2002.

[66] Stephen Hemminger. Netem - emulating real networks in the lab. In *Proceedings of the 2005 Linux Conference Australia*, Canberra, Australia, April 2005.

[67] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 13–18, Lihue, Hawaii, May 2003.

[68] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[69] Ningning Hu and Peter Steenkise. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 21(6):879–894, August 2003.

[70] Jin Jing, Ahmed K. Elmagarmid, Abdelsalam Helal, and Rafael Alonso. Bit-sequences: An adaptive cache invalidation method in mobile client/server environments. *Mobile Networks and Applications*, 2(2):115–127, 1997.

[71] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–93, Asheville, NC, December 1993.

[72] Jayanthkumar Kannan, Ayumu Kubota, Karthik Lakshminarayanan, Ion Stoica, and Klaus Wehrle. Supporting legacy applications over i3. Technical Report UCB/CSD-04-1342, University of California, Berkeley, May 2004.

[73] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computing Systems*, 10(1):3–25, 1992.

[74] J. Klensin. *Simple Mail Transfer Protocol*. Internet Engineering Task Force, April 2001. RFC 2821.

[75] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 282–297, Bolton Landing, NY, October 2003.

[76] Michael Kozuch and Mahadev Satyanarayanan. Internet suspend/resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–46, Callicoon, NY, June 2002.

[77] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, MA, June 2004.

[78] Alexandros Labrinidis and Nick Roussopoulos. Balancing performance and data freshness in web database servers. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 393–404, Berlin, Germany, September 2003.

[79] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent mid-tier database caching in sql server. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 661–661, San Diego, CA, June 2003.

[80] George Lawton. What lies ahead for cellular technology? *IEEE Computer*, 38 (6):14–17, 2005.

[81] Robert Love. Kernel korner: Intro to notify. *Linux Journal*, 2005(139):8, 2005.

[82] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 600–611, Madison, WI, June 2002.

[83] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Francisco, CA, January 1994.

[84] David Mazières, Frank Dabek, Eric Peterson, and Thomer M. Gil. Using libasync. `http://pdos.csail.mit.edu/6.824-2004/doc/libasync.ps`.

[85] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., 2001.

[86] Migo, Forward Solutions. `http://4migo.com/`.

[87] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 267–277, San Francisco, CA, December 1968.

[88] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 181–194, Cannes, France, September 1997.

[89] Jeffrey C. Mogul, Yee Man Chan, and Terence Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–56, San Francisco, CA, March 2004.

[90] Robert J. T. Morris and Brian J. Truskowski. The evolution of storage systems. *IBM Systems Journal*, 42(2):205–217, 2003.

[91] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, Banff, Canada, October 2001.

[92] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, Boston, MA, December 2002.

[93] *MySQL 5.0 Reference Manual*. MySQL AB, October 2006.

[94] Partho Nath, Michael Kozuch, David O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 71–84, Boston, MA, June 2006.

[95] Shamkant B. Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

[96] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th Symposium on Operating*

System Design and Implementation (OSDI), pages 363–378, San Francisco, CA, December 2004.

[97] ObjectWeb Consortium. `http://www.objectweb.org`.

[98] Timo Ojala, Jani Korhonen, Tiia Sutinen, Pekka Parhi, and Lauri Aalto. Mobile kärpät: A case study in wireless personal area networking. In *In Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM)*, pages 149–156, College Park, MD, October 2004.

[99] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–191, Monterey, CA, June 1999.

[100] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 170–179, Monterey, CA, October 2003.

[101] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–44, San Jose, CA, May 2006.

[102] Daniel Pfeifer and Hannes Jakschitsch. Method-based caching in multi-tiered server applications. In *Proceedings of the 5th International Symposium on Distributed Objects and Applications*, pages 1312–1332, Catania, Sicily, Italy, November 2003.

[103] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 155–174, Toronto, Canada, October 2004.

[104] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, MA, June 2004.

[105] J. B. Postel and J. Reynolds. *File Transfer Protocol (FTP).* Internet Engineering Task Force, October 1985. RFC 959.

[106] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, Cambridge, MA, April 2007.

[107] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, pages 89–101, Monterey, CA, January 2002.

[108] Michael Rabin. Fingerprinting by random polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81*, 1981.

[109] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Chuck Kalmanek. Moving edge side includes to the real edge – the clients. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, March 2003.

[110] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, pages 443–454, New York, NY, May 2004.

[111] Murali Rangan, Ed Swierk, and Douglas B. Terry. Contextual replication for mobile users. In *Proceedings of the International Conference on Mobile Business*, pages 457–463, Syndey, Australia, July 2005.

[112] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 161–172, San Diego, CA, August 2001.

[113] George Reese. *Database Programming with JDBC and Java.* O'Reilly, 1st edition, June 1997.

[114] Sean Rhea, Kevin Liang, and Eric Brewer. Value-based web caching. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, pages 619–628, Budapest, Hungary, May 2003.

[115] Sean C. Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 73–84, Philadelphia, PA, August 2005.

[116] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, pages 15–30, Monterey, CA, January 2002.

[117] Riverbed Technology. `http://www.riverbed.com/`.

[118] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1):31–41, 1997.

[119] M. Rose. *On the Design of Application Protocols*. Internet Engineering Task Force, November 2001. RFC 3117.

[120] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[121] RPM Software Packaging Tool. *RPM Software Packaging Tool*. `http://www.rpm.org/`.

[122] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network file system. In *Proceedings of the Summer Usenix Conference*, pages 119–130, Portland, OR, June 1985.

[123] M. Satyanaranyanan, Michael A. Kozuch, Casey J. Helfrich, and David R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, 2005.

[124] M. Satyanarayanan. The evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.

[125] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 96–108, Pacific Grove, CA, December 1981.

[126] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC distributed file system: Principles and design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)*, pages 35–50, Orcas Island, WA, December 1985.

[127] Mahadev Satyanarayanan, Maria R. Ebling, Joshua Raiff, Peter J. Braam, and Jan Harkes. *Coda File System User and System Administrators Manual*. Carnegie Mellon University, 1995. `http://coda.cs.cmu.edu`.

[128] Secure Hash Standard (SHS). Technical Report FIPS PUB 180-1, NIST, 1995.

[129] Secure Hash Standard (SHS). Technical Report FIPS PUB 180-2, NIST, August 2002.

[130] Keng Siau, Ee-Peng Lim, and Zixing Shen. Mobile commerce: Promises, challenges and research agenda. *Journal of Database Management*, 12(3):4–13, 2001.

[131] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data replication in Mariposa. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 485–494, New Orleans, LA, February 1996.

[132] Tracy F. Sienknecht, Richard J. Friedrich, Joseph J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. In *Proceedings of the 16th IFIP Working Group 7.3 International Symposium on Computer Performance Modeling Measurement and Evaluation (Performance '93)*, pages 3–25, Rome, Italy, 1994.

[133] Swaminathan Sivasubramanian, Gustavo Alonso, Guillaume Pierre, and Maarten van Steen. GlobeDB: Autonomic data replication for web applications. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*, pages 33–42, Chiba, Japan, May 2005.

[134] Mike Snyder and Jim Steger. *Working with Microsoft Dynamics CRM 3.0*. Microsoft Press, Redmond, WA, USA, 2006. ISBN 0735622590.

[135] Sumeet Sobti, Nitin Garg, Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Wang. PersonalRAID: Mobile storage for distributed and disconnected computers. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, pages 159–174, Monterey, CA, January 2002.

[136] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilei Shao, Chi Zhang, Elisha Ziskind, Arvind Krishnamurthy, and Randolph Wang. Segank: A distributed mobile storage system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 239–252, San Francisco, CA, March 2004.

[137] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 97–95, Stockholm, Sweden, August 2000.

[138] SQLite. `http://www.sqlite.org/`.

[139] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Dallas, TX, January 1988.

[140] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applica-*

tions, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 149–160, San Diego, CA, August 2001.

[141] Sysinternals. http://www.sysinternals.com.

[142] Sysstat. http://perso.wanadoo.fr/sebastien.godard.

[143] Douglas B. Terry. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, 13(1):48–54, January 1987.

[144] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182, Copper Mountain, CO, December 1995.

[145] The Globus Project. GridFTP: Universal data transfer for the Grid. http://www-fp.globus.org/datagrid/deliverables/C2WPdraft3.pdf, September 2000.

[146] Walter F. Tichy. RCS - A system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[147] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/.

[148] Niraj Tolia and M. Satyanarayanan. No-compromise caching of dynamic content from relational databases. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, pages 311–320, Banff, Canada, May 2007.

[149] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Adrian Perrig, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[150] Niraj Tolia, Jan Harkes, Michael Kozuch, and Mahadev Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pages 227–238, San Francisco, CA, March 2004.

[151] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, San Jose, CA, May 2006.

[152] Niraj Tolia, M. Satyanarayanan, and Adam Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *Proceedings of the 5th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 71–84, San Juan, Puerto Rico, June 2007.

[153] Joe Touch and Steve Hotz. The X-Bone. In *Proceedings of the 3rd Global Internet Mini-Conference at Globecom '98*, pages 59–68, Sydney, Australia, November 1998.

[154] Avishay Traeger, Nikolai Joukov, Charles P. Wright, and Erez Zadok. A nine year study of file system and storage benchmarking. Under review.

[155] *TPC Benchmark App (Application Server): Specification*. Transaction Processing Performance Council, San Francisco, CA, 1.1.1 edition, August 2005.

[156] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[157] U3. `http://www.u3.com`.

[158] Wietse Venema. The Postfix home page. `http://www.postfix.org/`.

[159] Michael Walfish, Jeremy Stribling, and Maxwell Krohn. Middleboxes no longer considered harmful. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 215–230, San Francisco, CA, December 2004.

[160] Randolph Y. Wang, Sumeet Sobti, Nitin Garg, Elisha Ziskind, Junwen Lai, and Arvind Krishnamurthy. Turning the postal system into a generic digital communication mechanism. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 159–166, Portland, OR, August 2004.

[161] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The personal server: Changing the way we think about ubiquitous computing. In *Proceedings of the 4th International Conference on Ubiquitous Computing (UbiComp)*, pages 194–209, Goteborg, Sweden, September 2002.

[162] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, pages 255–270, Boston, MA, December 2002.

[163] Kun-Lung Wu, Philip S. Yu, and Ming-Syan Chen. Energy-efficient caching for wireless mobile computing. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 336–343, New Orleans, LA, February 1996.

[164] Qingsong Yao, Aijun An, and Xiangji Huang. Finding and analyzing database user sessions. In *Proceedings of the 10th International Conference Database Systems for Advanced Applications*, pages 851–862, Beijing, China, April 2005.

[165] Chun Yuan, Yu Chen, and Zheng Zhang. Evaluation of edge caching/offloading for dynamic content delivery. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, pages 461–471, May 2003.

[166] Michael Juntao Yuan. *Enterprise J2ME: Developing Mobile Java Applications*. Prentice Hall PTR, 2003. ISBN 0131405306.

[167] Markos Zaharioudakis, Michael J. Carey, and Michael J. Franklin. Adaptive, fine-grained sharing in a client-server OODBMS: A callback-based approach. *ACM Transactions on Database Systems*, 22(4):570–627, 1997.

[168] Bruce Zenel and Andrew Toy. Enterprise-grade wireless. *ACM Queue*, 3(4): 30–37, 2005.

[169] Zona Research, 2001. The Need for Speed II, Research Report.