Providing Safety and Visibility for Mobile Users

by

Minkyong Kim

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science & Engineering)
in The University of Michigan
2004

To my family.

ACKNOWLEDGEMENTS

Thinking about the time that I spent at the University of Michigan, I cannot believe how lucky I was to come to know a lot of great people. I could not have finished my dissertation without their help.

First and foremost, I would like to thank my advisor, Brian Noble, for his advice on my research. I was fortunate to be one of his first students, getting invaluable attention. He was always available for guidance and advice. I also learned a lot from him how to write. He has emphasized the importance of telling a story in writing, which I am still trying to improve. I have no doubt that he will be my role model in my professional life.

I thank Professor Peter Chen, Atul Prakash, and Dawn Tilbury for serving on my dissertation committee. Peter made perceptive suggestions through out the design process of the file system. Atul showed interests in my work and made valuable comments. I enjoyed working with Dawn in designing data propagation mechanisms. Dawn is also a very careful editor, correcting numerous mistakes.

I would like to thank Professor Pramod P. Khargonekar for giving me advice in designing network estimators. He spared time from his tight schedule, and led me through the work with patience. Being a person with deep insight in the area of process control, he made many incisive suggestions. I would also like to thank Professor Kang G. Shin for his encouragement and support.

I thank the Mobility group members. Sitting next to me, Dr. Mark Corner (now with University of Massachusetts Amherst) made my graduate life pleasant by sharing various thoughts. Landon Cox made countless suggestions on my research and on my writing. He always willingly reviewed my work and gave me feedback. I also would like to thank newer members of the group, James Mickens, Anthony Nicholson, and Sam Shah for providing many helpful suggestions and comments.

I am deeply indebted to my parents and family. Words cannot express my gratitude to my parents, Yong-Sup and Soo-Ja. They have taught me the importance of hard work and encouraged me to pursue knowledge. Without their constant support, I could not have reached where I am now. I also thank my mother-in-law for her encouragement.

I would like to offer special thanks to my husband, Songkuk, who has been the constant force that made me go forward. His love and support made my graduate life enjoyable. He supported me not only emotionally but also academically. He has listened to numerous ideas and gave me feedback. I also like to thank my daughter, Na-Yun, for enduring the time that I was not with her. I love both very much and look forward to sharing the future with them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Computing is moving into the mobile era. The number of mobile users is increasing and notebook sales reflect this trend.

- Gartner, a research and advisory firm, reported in 2002 that "Between 1998 and mid-2002, mobile PCs increase their share of the global PC market by over 50 percent. For every four PCs shipped, one is now a mobile, and this share is set to rise further, approaching one in three by 2006" [16]. In Japan, mobile PC shipments already outnumbered those of desktop PCs.
- During the first quarter of 2003, Apple shipped 711,000 Macs, 40 percent of which were notebooks. That was the highest percentage ever for Apple up to that point [15].
- In the spring of 2003, according to NPD Group, retailers made more money selling laptops than desktops for the first time: Portables accounted for 54 percent of May's nearly $500 million in PC sales [74].

Mobile users bring new challenges to distributed file systems. Due to mobility of the file system clients, network cost between the clients and the servers varies. Another problem is that the client machines are often unavailable and cannot be accessed from other hosts due to no network connectivity. The client machines are also unreliable because they are vulnerable to breakage and theft. We consider each challenge in the following paragraphs.

First, the network costs for mobile clients to reach their home servers vary. When a file system client travels away from its home server, the cost to reach the home server increases, because he needs to pay the wide-area network cost to reach it; the latency and congestion along such paths impose a substantial performance penalty. For instance, a simple test with `ping` utility shows that latency increases by three orders of magnitude when a machine at the University of Michigan pings a machine in Korea compared to the case in which it pings a local machine. In short, it is often costly for mobile clients to access remote servers.

Second, mobile clients are less available. Laptops can be unreachable from other computers due to the absence of network connectivity or if in a suspended mode for power savings. Therefore, data may be unavailable for other hosts if it is stored only at clients.

Third, mobile clients are unreliable. Laptops are highly susceptible to breakage and theft. Safeware, an insurance company, reported that in 2001 [68], of the total of 2.4 million losses of computers, over 96% were related to laptops. Of the total laptop claims, accidents such as dropping the laptop is the number one cause, accounting 60%. Theft was the number two reason, accounting 26%. Since file system clients are unreliable, it is not safe to store data only at clients.

Current file systems do not address these challenges adequately. Current wide-area and mobile file systems employ two techniques to reduce their use of the remote server. The first technique is *caching*. When workloads have good locality and infrequent sharing, caching can avoid most file fetches from the server. But, caching may be limited for mobile clients since they are resource poor relative to static machines. The second technique is *optimistic concurrency control* [53]. Clients defer the shipment of updated files until time and resources permit. Although deferring updates improves performance, it harms safety and visibility. In other words, clients can defer shipping updates to avoid high latency to access home servers, but this leaves new data vulnerable and increases the time for updates to reach other hosts.

1.1 Thesis Statement

To address the challenges described above, we designed and implemented a prototype of a distributed file system, called Fluid Replication. Through this system, we demonstrated the following thesis statement:

A distributed file system can provide safety and visibility to file updates over the wide-area with performance comparable to the local-area.

An update is *safe* if it survives the theft, loss, or destruction of the mobile client that created it. Safety requires that the contents of an update reside on at least one other host. An update is *visible* if every other client in the system knows it exists. Visibility requires that notification of an update reaches all replicas in the system.

Fluid Replication is a distributed file system based on a client-server architecture. Between the server and clients, intermediate servers, called *WayStations*, are placed to provide immediate and inexpensive **safety** for clients' update data. Each client chooses a nearby WayStation and propagates updates to the WayStation, but not to the server. This hides wide-area network cost from clients and provides update **performance** that is comparable to the local case. If the network condition to the current WayStation degrades, the client adaptively chooses a new WayStation that is closer. WayStations participate in the consistency protocol for replicas on behalf of clients. WayStations aggregate clients' updates and send them to the server periodically through *reconciliation*. During reconciliation, WayStations send only the knowledge of updates without update contents. This makes cost of reconciliation inexpensive. Thus WayStations can reconcile frequently, resulting in better **visibility**.

1.2 Structure of Dissertation

In this dissertation, we provide solutions to the challenges intrinsic to the mobile clients of distributed file systems. These solutions provide safety, visibility and performance to wide-area updates that are comparable to the local case. The dissertation is organized as follows.

We start with background and related work in Chapter 2. This chapter consists of three sections. Section 2.1 covers related work on network estimation, which is the key component in choosing a nearby WayStation. We discuss network bandwidth measurement methods and filtering techniques to smooth individual network measurements. Section 2.2 describes related work on distributed file systems. We discuss background on optimistic concurrency control and survey distributed file systems that use it. Section 2.3 presents related work on managing update data–whether and when to transfer data from WayStations to servers. We describe prefetching methods and the uses of feedback controllers to predict future file accesses.

Chapter 3 presents the overall design of Fluid Replication. Section 3.1 presents the overview of Fluid Replication. To make the system tolerant of wide-area network cost, we introduce a collection of intermediate servers, called WayStations. Clients send updates aggressively to a WayStation, providing immediate safety. WayStations disclose them to the server periodically through reconciliation, providing visibility. Section 3.2 presents our key design decisions and describes how these decisions collaborate to meet our design goals. Section 3.3 introduces three mechanisms needed for Fluid Replication. First, a client needs to choose a nearby WayStation. After choosing one, it monitors network conditions to the WayStation to detect performance degradation along the path. If it occurs, the client chooses a new WayStation that is closer. Second, WayStations need to participate in the consistency protocol for replicas. During reconciliation, WayStations notify the server of updates, but do not send the contents of update files. This separation makes the reconciliation cost low, allowing frequent reconciliation for better visibility. Third, update data should be propagated from WayStations to the server efficiently. Sending them too aggressively wastes network resources, while the opposite may penalize clients who share data. In Section 3.4, we discuss how useful the mechanisms of Fluid Replication could be without the presence of WayStations.

Chapter 4 presents techniques for estimating network conditions to choose a nearby WayStation. A client discovers a set of available WayStations, and measures latency and available bandwidth to these WayStations in order to select the nearest one. The challenge is that these individual measurements are noisy and change constantly. Thus, we need a filter that detects persistent changes in network conditions quickly but ignores transient ones. In other words, the filter should be agile when possible, but stable when necessary. We developed a filter that consists of two exponentially weighted moving average (EWMA) filters with a controller that selects between the two; the controller is based on a technique from statistical process control. We compare this filter with three other adaptive filters, including the well-known Kalman filter.

Chapter 5 describes the consistency mechanism for Fluid Replication. During reconciliation, WayStations send the server update notifications, and the server returns invalidations. Timestamps are used to support optimistic concurrency control. When a WayStation requests a file, the server first checks the WayStation's timestamp against its own to see whether the WayStation has missed any invalidations. Then, the server returns the most recent version known to the server.

Chapter 6 presents the ways to propagate data from WayStations to the server. During reconciliation, update meta-data is sent to the server, but the data transfer is deferred. We consider mechanisms to decide whether and when to transfer data from WayStations to the server. Two simple schemes, write-through and on-demand, are too extreme; the write-through scheme is too aggressive, sending too much data unnecessarily, and the on-demand scheme is too passive, penalizing clients who share. We developed a heuristic approach that is based on the observation that updates that invalidate cached objects elsewhere are likely to be shared. This mechanism is compared with write-through, on-demand, and two other schemes over traces capturing live file system uses.

Finally, Chapter 7 summarizes the thesis and outlines the directions for future research. Fluid Replication successfully met our design goal: providing safety and visibility to updates over wide-area with performance comparable to local-area. The directions for future work include automating client migration using the network estimator as a base component, exploring security-related problems, and getting more experience with Fluid Replication by deploying it in a real environment.

CHAPTER 2

Background and Related Work

This chapter introduces background and related work. As clients move, network conditions may change constantly. This may be due to topology changes, vertical handoff or wireless fading and shadowing. Dealing with these changes in network conditions is important for client-server file systems. In Section 2.1, we discuss network bandwidth measurement methods and filtering techniques to smooth individual measurements. Although file sharing is rare, distributed file systems need to provide a consistency mechanism for replicas. In Section 2.2, we provide background on optimistic concurrency control and survey distributed file systems that use it. The goal of prefetching data is to improve read performance in distributed file systems. To do so, data should be propagated from one node to another in advance of requests. In Section 2.3, we describe prefetching methods and the uses of feedback controllers to predict future file accesses.

2.1 Network Estimation

Adapting to changes in network conditions is important in many areas. For example, a content distribution replica can be selected based on client-initiated network measurements [9]. TCP congestion control can be improved using available network bandwidth [76]. Overlay networks use network information to decide when to switch from one overlay network to another [28]. To do these, the first step is to estimate the network conditions.

Many researchers have been working on measuring physical and available network bandwidth. Physical bandwidth is also referred as bottleneck link bandwidth. The bottleneck link bandwidth is the maximum rate that the path can provide to a flow in the *absence* of congestion traffic. It is an upper bound on how fast a connection can transmit data. The available bandwidth is the maximum rate that the path can provide to a flow without reducing the rate of the rest of the traffic [25]. Most prior systems focus on finding the physical bandwidth of the bottleneck link rather than the available bandwidth between two end hosts. While there are applications that can make use of bottleneck link information, adaptive mobile systems are concerned with the bandwidth that is actually available to them over time.

Individual measurements of available network conditions change constantly, due to cross traffic and users' mobility. Some changes are persistent while others are transient. To filter out noise (or transient changes) and estimate true network conditions, we need a filtering mechanism. When there is a persistent change in network conditions, a filter should detect it quickly and report the change to a higher level application, so that the application can adapt to the change. At the same time, the filter should ignore transient changes and save the application from mistakenly reacting to them. In short, the goal of a filtering mechanism is to be agile when possible, but to be stable when necessary.

Most of existing systems rely on *active* probing of the network: injecting measurement traffic in addition to or instead of passive observation of traffic already present. This is likely to be unacceptable during times of sharp decreases in network performance, as the probe traffic compounds the problem.

2.1.1 Bandwidth Measurements

Jacobson first noted the principle of the bottleneck spacing effect [23]. If two packets are sent close enough together, they will queue back-to-back at the bottleneck link. Thereafter, their spacing will remain same, and they will arrive at the destination while preserving the spacing.

Keshav [31] proposed the *packet pair* technique that is based on the bottleneck spacing effect. The packet pair technique sends packets back-to-back and estimates the bottleneck link bandwidth using the spacing between the acknowledgements. Keshav's work on flow control provides much of the groundwork in the area of active probing. Keshav discussed the use of a well-known Kalman filter for network estimation. The Kalman filter estimates the state of a linear system with given noisy measurements. If properly tuned, it generates an optimal estimate of the system state. But, Keshav rejected the Kalman filter because too little is known about the network state space to tune properly. Instead, he employed a fuzzy logic estimator based on a heuristic with a mechanism to resist occasional transient spikes. This mechanism would not improve performance in the presence of congestion-induced noise, which is endemic to cross-traffic. Keshav's estimator was designed for *rate-allocation servers* that exhibit much less noise than FCFS routers. Unfortunately, the current networking infrastructure consists primarily of FCFS routers, a domain Keshav's work explicitly excludes.

Variations on packet pair improve its ability to generate estimates of bottleneck link bandwidth. Bolot [5] uses pairs of UDP packets to explore network state. UDP probe packets are sent at regular time intervals and their round trip times (rtt) are measured. The rtt of successive packets are plotted as rtt_n and rtt_{n+1}. The inverse of slope of the line is the bandwidth. Unfortunately, these active probes generate substantial overhead.

Carter and Crovella [9] present tools to measure bottleneck and available bandwidth, called `bprobe` and `cprobe`, respectively. These tools rely on bursts of ICMP packets, sent in several phases. As with Bolot's scheme, this requires substantial overhead. They assume that network conditions do not change during this process, limiting the granularity of changes that they can detect.

Both studies are based on measurements made only at the sender. A problem with these approaches is that they cannot guarantee that a packet took the same path to and from the target. Thus, a packet may have experienced different bottleneck links in the forward and reverse paths.

Paxson [60] presents receiver-based packet pair (RBPP), which takes observations at the receiver. The receiver also uses the timing pattern in which the data packets were originally sent. RBPP is more accurate than sender-based packet pair, because it eliminates noise due to reasons including possible asymmetry of the forward and reverse paths.

Lai [42] developed a further refinement, called receiver-only packet pair (ROPP). It depends only on timing information taken at the receiver, but approaches the effectiveness of RBPP. Lai also incorporates a mechanism called *packet windows* that increases the agility of packet-pair schemes, but leaves them susceptible to noise and transients. Varying the size of packet windows can bias ROPP for agility or stability, but this size is chosen statically.

While the packet pair technique estimates end-to-end properties of a path, the single packet technique focuses on the characteristics of each link along a path. `pathchar` [24] is a tool that infers the characteristics of each link along a path. It uses the time-to-live (ttl) field in an IP packet. When a router receives a packet whose ttl has expired, it drops the packet and sends an ICMP packet to the sender. `pathchar` sends a series of probes with varying values of ttl and varying packet sizes. Using the time until the ICMP packet is received, it infers the latency and bandwidth of each link.

Downey's application of `pathchar` [12] uses ICMP packets with varying time-to-live fields. It reduces measurement traffic; while `pathchar` uses the same number of probes for each link, Downey's approach starts with a small number of packets for each link and increases the number of packets only if the estimated properties of a link have not converged. Unfortunately, it too suffers from heavy bandwidth consumption.

Lai's subsequent work [43] develops a more sophisticated network model that, combined with a technique called *packet tailgating*, estimates physical bandwidth of each link along a path, and the prototype of this approach is called `nettimer` [44]. As done in packet pair, tailgating sends two packets back to back, but unlike packet pair, the first packet is much larger than the second. Because the transmission delay of the first packet is larger than the second, the smaller packet queues continuously after the larger packet. The larger packet is dropped when the time-to-live expires, then the smaller packet continues without queueing to the target. Tailgating generates much less active network traffic and makes fewer assumptions about router behavior than prior schemes. It estimates link bandwidth by determining the minimum delay experienced at each link.

Balakrishnan's congestion manager [2] allows multiple conversations between two hosts—including those of protocols that normally do not provide congestion control—to effectively share bandwidth. The congestion manager uses a window-based additive-increase multiplicative-decrease (AIMD) scheme to modulate transmissions. A sender periodically sends probes to a receiver. Using the responses, the sender infers the state of the network and adjusts its congestion window. However, if it does not receive responses, it uses exponential aging; the transmission rate is halved every t seconds where t is defined as the minimum of all round-trip samples.

2.1.2 Filtering Techniques

Several systems attempt to discover available network performance using static-gain exponentially weighted moving average (EWMA) filters. These filters can be tuned to react to changes aggressively or conservatively. For example, the round-trip time estimator in TCP uses a stable EWMA filter [23]. Odyssey, a platform for application-aware adaptation, employs a network estimator using an agile filter [56]. Unfortunately, both of these static filters suffer from their biases. The round trip time estimator in TCP cannot track varying performance quickly, resulting in retransmission timeouts (RTOs) that are too aggressive. To compensate, RTOs are increased by a factor that accounts for observed variance. Odyssey suffers from the opposite problem. Occasionally, it is fooled into tracking transient changes in bandwidth and adapting too aggressively as a result. In Odyssey, applications are responsible for filtering out transient changes.

2.2 Distributed File Systems

In typical distributed file system workloads, the degree of write-sharing is low. Based on this, many systems allow reads and writes in any partitions, detecting conflicts after the fact. This is called *optimistic concurrency control*, and was first proposed by Kung and Robinson [40] in database systems. All of the file systems that we describe in this section use this mechanism.

In systems using optimistic concurrency control, all updates are made to local copies. These updates are not safe. An update is *safe* if it survives the theft, loss, or destruction of the mobile client that created it. Safety requires that the contents of an update reside on at least one other host. Updates are exchanged later between clients or servers and become visible. An update is *visible* if every other client in the system knows it exists. Visibility requires that notification of an update reaches all replicas in the system.

All current systems needlessly combine the knowledge of updates and the contents of updates. By doing this, the systems bind the safety and visibility of updates. This makes the reconciliation process expensive. Therefore, reconciliation cannot be done frequently.

2.2.1 AFS

The Andrew File System (AFS) [70] is based on the client-server model. Clients cache files and directories at their local disks and use a callback-based cache consistency mechanism [30]. In this mechanism, servers remember the list of nodes who are caching each file and send them invalidation messages when the file is updated elsewhere. All updates to a file are propagated to its server upon `close`, while directory modifications are propagated immediately. When clients are separated from their server by a slow network connection, performance suffers greatly.

Huston modified AFS to support disconnected operations [21]. When a client is disconnected from the server, the client's operations are satisfied by locally cached copies. These operations are also logged and later replayed to the file server when the client is reconnected. Logs contain both non-mutating and mutating operations. The former is used to detect stale reads and the latter is used to detect conflicting writes. Recording non-mutating operations may increase the size of the log quickly, and reporting stale reads during replay may be unnecessary since it would be hard for clients to cope with problems at that point.

2.2.2 Coda

Coda provides high availability through server replication [39]. Because write-sharing is infrequent, it uses optimistic replica control that permits reads and writes in any partition with a replica. This allows stale reads and conflicting writes.

Even with optimistic replica control, server replication cannot help if all servers crash or if a network failure isolates a client. To provide availability under these situations, Coda supports disconnected operation [34]. This also allows stale reads and conflicting writes.

Using optimistic replica control and disconnected operation, Coda can experience conflicts arising from updates to partitioned replicas of the same object by two different clients. Resolution of directory conflicts is done using a log-based strategy. Every replica has its *resolution log* that contains the entire list of directory mutations performed on a replica. During reintegration, conflicts are resolved by combining each resolution log and applying missed updates at each replica.

Coda also supports weakly-connected operation [54] to utilize low bandwidth between a client and its servers. During weakly-connected operation, Coda clients defer updates, and ship them later through a mechanism called *trickle reintegration*. Trickle reintegration propagates updates to the servers asynchronously, relieving users of the need to perform manual reintegration. During reintegration, clients send both the knowledge of an update and contents to the server. Since this is expensive, Coda defers reintegration of each update in the hopes that it will be canceled by later overwrites or deletions. This *aging window* is set to ten minutes to capture an acceptable fraction of possible optimizations, trading network bandwidth for more aggressive propagation.

2.2.3 Ficus

Ficus [19, 63, 58] shares Coda's goal of providing optimistic file access, but uses a peer-to-peer architecture. It allows updates as long as at least one replica is available; this is called *single-copy availability*. Updates are propagated asynchronously to other replicas, so Ficus cannot bound the time required to make updates visible.

In Ficus, a reconciliation process runs periodically. It compares all files and directories of the local replica with a remote replica, sending missed updates and detecting update conflicts. To avoid exchanging attributes of all files between two replicas, the disk at each replica is scanned to discover updates that have happened since the last reconciliation. This reduces network traffic, but a disk scan is a heavyweight process.

During reconciliation, replicas exchange both knowledge and content of updates. Since this is expensive, it is intended only for well-connected peers, making visibility dependent on the mobility pattern of replicas.

2.2.4 Bayou

Bayou [61] provides optimistic concurrency control for database applications. Each *data collection* is replicated at a number of servers. A client can perform an update to any of the servers, and need not wait until it is propagated to the remainder. Writes are exchanged pairwise among servers. Because the exchanges between peers rely on network connectivity, Bayou cannot bound the time required to propagate updates.

When an update is accepted by a server, it is initially marked as *tentative*. Updates are eventually committed by the server designated as the *primary* replica. The primary commits an update it receives by assigning a *commit sequence number* (CSN); the CSN defines the final total update order.

The Bayou system uses two mechanisms for automatic conflict detection and resolution to support arbitrary applications: a *dependency check* and a *merge procedure*. These mechanisms permit clients to indicate, for each individual update, how the system should detect conflicts involving the update and what steps should be taken to resolve any detected conflicts based on the semantics of the application. Each update includes a dependency check consisting of a query and its expected result supplied by an application. This dependency check is a precondition for performing an update. If the server does not return the expected value, a conflict is detected and the update is not performed. Then, the server invokes a procedure to resolve the detected conflicts.

2.2.5 OceanStore

OceanStore [37, 64] advocates an architecture for global storage. It envisions storage supported by a loose confederation of independent servers. It is a two-tiered storage system. The upper tier consists of powerful, well-connected hosts that serialize changes and archive results. The lower tier consists of less powerful hosts including users' workstations.

OceanStore allows client applications to specify consistency mechanisms. Updates are represented as an array of potential actions each guarded by a predicate supplied by an application. To update an object, the system first checks predicates; this is similar to the dependency checks in Bayou. If the predicate checks are successful, the actions are applied to the data object.

In OceanStore, a new version is created for each write, and it is kept forever. Because an object is a sequence of read-only versions, the replication of these blocks introduces no consistency issues. However, the mapping from the name of an object to the latest version does cause a consistency problem. To solve this, OceanStore assigns a *primary* replica to each object, as done in Bayou. The primary replica is responsible for serializing and committing updates. OceanStore's consistency mechanism, like that of Ficus and Bayou, is based on epidemic algorithms, so it cannot bound the time required for updates to become visible to all replicas.

2.3 Data Management

In Fluid Replication, meta-data at the server is updated during reconciliation, but data transfer is deferred. To reduce client latencies, data should be propagated from WayStations to the server in advance by anticipating future sharing among different nodes. This is related to prefetching algorithms developed to reduce file access latencies. Prefetching is a well-studied topic, but most of the previous work on prefetching files tries to predict file accesses from a single node, while our work focuses on predicting future sharing of files between different nodes.

2.3.1 Prefetching

Griffioen [18] developed a prefetching algorithm based on a probability graph where each node represents a file and a directed arc represents access order. Each arc is weighted by the number of times that the target is accessed after the source. Lei [46] used *access trees* to capture the relationships among files. The access trees are maintained for each program; when a program is re-executed, the current access tree is compared against the saved access trees. Kroeger [36] suggested a last-successor model, which predicts that each file access will be followed by the same file that followed the last time. Kuenning [38] used *semantic distance* between files to choose which files should be hoarded for later use when a mobile user is disconnected.

Distributed databases also face the data migration problem, but have very different access patterns—a lower degree of locality but a higher incidence of sharing. Mariposa applies an economic model for data migration [72]. Each query has a budget allocated to it, and it tries to minimize expenditures. Each replica site can either process a query or ship tuples; both actions accrue revenue but cost resources. By attempting to maximize revenue given fixed resources, Mariposa allocates these resources efficiently.

2.3.2 Feedback Control

Over the past several years, a number of different groups have been working to apply feedback control theory to problems in performance management of computing systems. Challenges exist both in defining the control framework for a problem and in developing a mathematical model of the computing system that can be used to design and analyze the control algorithm.

Feedback control has been applied to real-time scheduling problems, adapting admission control to account for poorly-estimated execution times by submitted jobs [50, 49]. Each real-time task submitted to a server for execution has an estimated and an actual execution time, and the system decides whether a task will be admitted. Without adaptation, the system will be underutilized if the actual times are shorter than estimated, and will be overloaded (and hence miss deadlines) if the actual times are longer than estimated. Feedback control is used to adjust the admission control criteria based on the actual number of jobs that miss their deadlines. A small tolerance for missed deadlines is assumed providing that the server can be nearly fully utilized.

Another application of feedback control is in caching for service differentiation [51]. The amount of cache space allocated to each class of jobs is adjusted based on the measured cache-hit rates of each class; a standard caching algorithm is used.

2.4 Summary

This chapter starts with network estimation techniques. These can be grouped into single-packet and packet-pair techniques. The single-packet technique focuses on estimating the characteristics of each link along a path using the time-to-live field. The packet-pair technique estimates end-to-end properties of a path by sending two packets back to back. Then, we surveyed distributed file systems that use optimistic concurrency control in Section 2.2. Because the degree of write-sharing is low, these file systems allow reads and writes in any partition. In Fluid Replication, meta-data is sent to the server during reconciliation, but the data transfer is deferred. To reduce latencies, data should be sent to the server before another node asks for it. Deciding when to ship data is related to prefetching algorithms developed to reduce file access latencies. We discussed several prefetching methods in Section 2.3.

CHAPTER 3

Design of Fluid Replication

In this chapter, we present a design overview of Fluid Replication with an eye towards separating the concerns of safety and visibility. We explain how our decisions combine to meet our design goals. Then, we introduce the three main challenges for Fluid Replication and outline our solutions.

3.1 Design Overview

In Fluid Replication, when a client is close to the server, it communicates with the server directly. When it moves away from the server, increasing network cost, it starts using a nearby WayStation. When the connectivity to the current WayStation degrades, the client adaptively selects a new, nearby WayStation. We envision that WayStations are commonly available. Each WayStation can serve multiple clients simultaneously. Since each client may have different home servers, WayStations need to communicate with multiple servers. Figure 3.1 shows how the components of the Fluid Replication system are connected.

In Fluid Replication, a client sends updates immediately to a WayStation. WayStations aggregate clients' updates and send them to the server periodically. During periodic

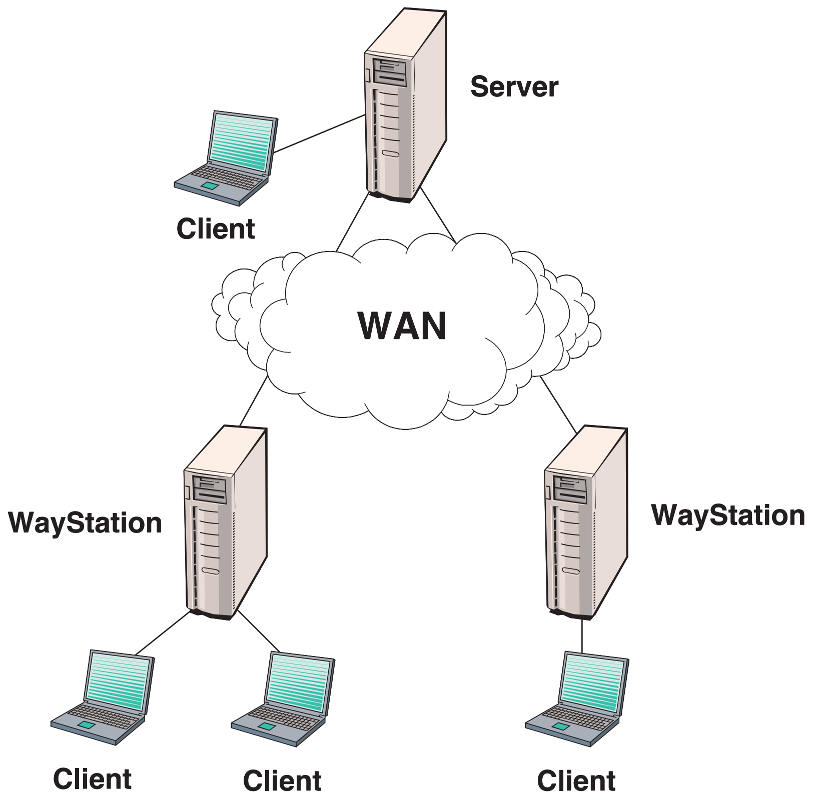


Figure 3.1: System Overview

reconciliation, update notifications are sent without the contents of those updates. Since contents are often much bigger than notifications, moving the contents is costly compared to notification. The contents of updates should be moved only when they are likely to be needed elsewhere.

3.2 Design Rationale

Our goal in designing Fluid Replication is to hide wide-area network cost from clients while providing safety and visibility of updates similar to that available in the local area. The system also should adapt to changing network conditions for good performance. In the following sections, we describe the rationale behind our design decisions and how these decisions combine to meet the goals.

3.2.1 Separating Safety and Visibility

Current file systems needlessly combine safety and visibility by propagating the contents of an update, implicitly notifying the destination that the update exists. Fluid Replication separates the concerns of safety and visibility through the addition of secondary replica sites, WayStations. These sites act as server replicas to nearby clients, serving clients' uncached reads and writes. Client writes are propagated to a WayStation immediately, providing safety. WayStations periodically reconcile their updates with the servers for which they act as replicas; this exchanges update notifications, but not the contents of those updates. Specifically, a WayStation sends the server an update log that contains mutating operations during reconciliation. For `store` operations, the meta-data is updated during reconciliation, but data transfer is deferred.

The idea of separating meta-data and data is similar to the concept used in cache coherence protocols in multiprocessor systems. In shared-memory multiprocessor systems, each processor has a local cache and multiple processors are connected to the main memory. The processor cache corresponds to a WayStation replica, and the main memory corresponds to the file server. When a processor updates a block of its cache, the corresponding block in the memory is invalidated; the notification of update is sent to the memory without the update contents.

There are several advantages to separating data propagation from the reconciliation process. First, by excluding costly data propagation, reconciliation can be done frequently, providing good visibility. Second, now that data propagation is separated from the consistency mechanism, the decision of when and whether to send data to the server affects only performance, not consistency. Third, the separation can reduce network traffic to the server. Because WayStations are reliable, data need not be sent to the server diligently; this does not penalize consistency.

3.2.2 Using WayStations

When a client moves away from its home server, network cost increases. To avoid these costs, the client can defer sending updates to the server. But, this has following disadvantages:

- **Low availability**: The mobile clients may be disconnected or tuned off at any time. Therefore, updates solely on them can be unavailable for other hosts.
- **Unreliability**: Mobile clients are unreliable compared to desktop machines. They are easily broken or stolen. Safeware, an insurance company, estimated that in 2001, 1.387 million laptops were broken due to accidents, and 591,000 laptops were stolen [68]. Thus, data stored only at mobile clients can easily be lost.

In Fluid Replication, we introduce intermediate servers, called WayStations. In contrast to clients, WayStations are carefully administered machines, and are relatively permanent members of the infrastructure. While they can be independently managed, they are not expected to be transient. They are reliably connected to servers. Thus, they provide availability and reliability.

Compared to clients that communicate directly with the server, clients communicating through WayStations are at a disadvantage: Visibility of updates is reduced, since updates take more time to reach other hosts. But, the advantages of using WayStations outweigh the disadvantage. The advantages are as follows:

- **Low-cost safety**: WayStations provide immediate safety of update data without clients paying wide-area network cost for each update to reach the home server.
- **Reduced wide-area traffic**: Because WayStations are reliable there is no reason to propagate update data to the remote server other than administrative reason and true sharing.
- **Server scalability**: WayStations make the server scalable by hiding individual clients from the server. WayStations aggregate clients' updates and send them to the server periodically.
- **Adaptability**: Clients can be adaptive to network changes by changing their corresponding WayStations. When network conditions to the current WayStation degrade, a client can select a new, nearby WayStation.
- **Caching**: WayStations provide caching for mobile clients, since mobile clients are resource poor relative to static machines.

3.3 Main Mechanisms

There are three main challenges that the Fluid Replication must address to support mobile clients:

- WayStation selection under changing network conditions,
- managing meta-data (update notifications) in high-latency wide-area network, and
- managing data (update contents) with limited network resource.

The rest of this section describes these challenges and outlines our solutions.

3.3.1 WayStation Selection

The success of Fluid Replication depends on WayStations, which act as replicas for servers. They provide caching and immediate safety to nearby clients without penalizing performance. A WayStation is able to host replicas of any service for any client, regardless of administrative domain, and can host replicas of multiple servers concurrently.

The decision of whether or not to use a WayStation is client-driven. Clients estimate the quality of the network between themselves and their current replica site. If the network quality becomes poor, the client looks for a WayStation close enough to improve matters, and initiates a replica on it. Such searches are carried out in a neighborhood near the client using Pastry [67], a peer-to-peer routing substrate. The WayStation informs the server of initiation.

To detect changes in the network quality, clients need to monitor the network conditions. The problem in estimating network conditions is that they change constantly. This may be due to network topology changes, vertical handoff across connection alternatives, or wireless fading and shadowing. These factors conspire to produce frequent changes in available latency and bandwidth, and induce substantial noise in individual network observations. Thus it is difficult to tell the “true” conditions of network.

To solve this problem, we need a network filter that produces current network conditions given noisy individual observations. The network filter should ignore transient changes but at the same time detect enduring changes quickly. In other words, it should be stable to noises but agile to true change.

We developed a filter, called *Flip-flop*, that uses a technique from statistical process control. It selects between an agile exponentially weighted moving average (EWMA) filter and a stable EWMA filter. We compare this filter with three other filters, including the well-known Kalman filter. We also compare it with two static-gain EWMA filters: the agile Odyssey filter and the stable TCP filter. It provides agility comparable to the Odyssey filter and stability comparable to the TCP filter in the majority of cases.

3.3.2 Meta-Data Management

WayStations have two important roles. The first is to participate in the data consistency protocol on behalf of clients. The second is to make the update data safe by storing it at another machine beside the client itself. In this section, we concentrate on the first role. The second role is discussed in the next section.

For consistency, optimistic concurrency control [53] is used based on the observations that a file is not likely updated simultaneously by more than one node. However, when two or more updates occur concurrently, the server must check whether the updates can be serialized. If not, the problem is reported to the client, and the client must manually fix it.

A WayStation periodically informs the server of its updates. Upon receiving the message from a WayStation, the server invalidates other WayStation replicas immediately. These WayStations then invalidate cached copies at clients.

We evaluate the overall performance of Fluid Replication in comparison to Coda and AFS. While performances of Coda and AFS decrease as network conditions degrade, Fluid Replication is largely unaffected by degradation. Fluid Replication successfully isolates clients from wide-area networking cost.

3.3.3 Data Management

When an optimistic update is made at a replica site, that site must eventually inform others of the update for consistency. However, file sharing is rare in file system workloads. So, while the meta-data describing an update needs to be propagated, the data comprising that update often need not be. However, this deferring of data shipment penalizes the clients who do share. For these clients, the cost of fetching updated files that are at other WayStation is high in a wide-area network.

To solve this problem, we developed a mechanism, called *Invalidation Heuristic*, to predict future file sharing. The goal is to propagate to the server the files that are likely to be shared in advance of others' requests, while the rest are not propagated at all. The Invalidation Heuristic scheme is based on the observation that files whose updates caused invalidations are likely to be used by others, and therefore should be propagated to the server in advance. We compare this scheme against several other schemes using real file system traces. It ships two orders of magnitude less data than the aggressive write-through scheme, while reducing the penalty of on-demand shipment by nearly a factor of two.

3.4 The Role of WayStations

The premise of ubiquitous WayStations is perhaps a tenuous one. These machines must be reliably connected to the rest of the network, and this requires a significant level of administrative care. This represents a significant deployment burden. So, it is reasonable to ask how useful the mechanisms of Fluid Replication could be without the presence of WayStations.

In this alternative design, clients take on the duties of WayStations. They notify servers of the presence of updates, but retain the contents of those updates. This provides good visibility and common-case performance, but poor safety. Because servers are given timely notification of updates, any shared references will be to the most recent version available. Because notifications are small and shipped asynchronously, the end user pays no performance costs to support them. However, in this model updates are stored only on the mobile client, subject to the increased chance of theft, loss, or destruction.

In addition to reduced safety, this alternate design also suffers from reduced availability in the presence of shared updates. In Fluid Replication, updated files are left on nearby WayStations; when other clients read those files, they are back-fetched. Because WayStations are reliably connected, back-fetches can be satisfied. In contrast, mobile clients are often disconnected from the network. They are often suspended to conserve power, and sometimes are not reachable by any networking technology. Thus, client-serviced back-fetches are much less likely to be successful, leading to user frustration.

To avoid this problem, data needs to be sent diligently to the server. To send data aggressively, clients can include data in logs and send logs frequently. This is precisely what Coda does in cases of weak connection [54], though Coda could be improved by sending invalidations and updates separately. However, there is a tension between log size and shipment frequency, which is exacerbated by the cost of communicating between client and server across the wide area. Furthermore, even if data propagation is done asynchronously, clients still need to stay connected until all the update data is received by the server to guarantee safety of data.

Thus, WayStations play an indispensable role in the design of Fluid Replication. Perhaps the most likely deployment scenario is for each Fluid Replication server to also act as a WayStation for foreign clients. One could also imagine using other nearby clients as repositories. However, this would require replication—any one of those clients is no more reliable than the client on which the file was written.

3.5 Summary

Fluid Replication separates the concerns of safety and visibility through the addition of secondary replica sites, WayStations. WayStations provide immediate safety of update data, make the server scalable by hiding individual clients, reduce overall network traffic by aggregating clients' updates, and allow clients to be adaptive to network changes by switching their corresponding WayStations.

During reconciliation, WayStations send the server the notification of updates without update data; data transfer is deferred. This separation reduces reconciliation cost. Thus reconciliation can be done frequently, providing better consistency. The separation also allows a variety of data propagation mechanisms, reducing overall network traffic.

There are three main challenges to Fluid Replication: selecting a nearby WayStation under changing network conditions, managing meta-data in a high-latency wide-area network, and managing data with limited network resources. The following three chapters explore these challenges, describe solutions, and evaluate their effectiveness.

CHAPTER 4

Mobile Network Estimation

4.1 Introduction

It is widely recognized that adaptation to changing networking conditions is critical to mobility [13, 27]. In Fluid Replication, clients must know the available network capacity to each WayStation to decide when to look for a new WayStation and to select among them. For example, when the network capacity to a WayStation is reduced, a client needs to tell whether it is a true, persistent change, or transient one. If it is a true change, the client looks for a closer WayStation and migrates to it. Otherwise, the client stays at the current WayStation to avoid the overhead of migration.

Discovering the available network capacity is a difficult problem, because the effective latency and bandwidth between a mobile host and other nodes changes constantly. This may be due to ad hoc topology changes, vertical handoff across connection alternatives, or wireless fading and shadowing [29, 62]. These causes are in addition to the more prosaic routing changes and congestion endemic to wired networks of even modest scale [41, 60]. Together, these factors conspire to produce frequent changes in available latency and bandwidth, and induce substantial noise in individual network observations.

Converting noisy observations into an estimate of available latency and bandwidth is an instance of the *filtering* problem. Observations are fed into a filter, which smoothes them in some way to produce estimates. Typically, systems employ an exponentially weighted moving average (EWMA) filter to solve this problem. Given a new observation, an EWMA filter produces a new estimate as a linear combination of the old estimate plus the new observation, each given some weight.

Unfortunately, the *gain*, which determines the proportional weight assigned to the new observation and the old estimate, is fixed in traditional EWMA filters. When old estimates are given more weight, the filter provides good *stability*; it resists noise in individual observations. When new observations are given more weight, the filter provides good *agility*; it is able to detect performance changes quickly. Neither property is desirable at all times. Ideally, one would like to have a filter that is agile when possible but stable when necessary, depending on current circumstances. In other words, filters must be adaptive, just as other components of the system must be.

This chapter describes our experiences designing and evaluating filters that trade stability for agility based on the prevailing situation. We have designed several candidate filters in an attempt to meet this goal. One, called *Flip-flop*, is a composition of an agile EWMA filter and a stable one. Flip-flop selects between them using a technique borrowed from statistical process control [52, 65, 78]. Two others, *Stability* and *Error-based*, use heuristics to vary the gain of an EWMA filter continuously, in order to select for agility or stability based on the behavior of observations and estimates, respectively. Finally, we have applied a variant of the well-known *Kalman* filter [17].

4.2 Locating Nearby WayStations

When the network connectivity to the current WayStation degrades, the client needs to look for a new nearby WayStation. We can use Pastry [67] to discover a list of nearby WayStations and use our filters to choose the nearest one from the list.

Pastry is a peer-to-peer routing substrate. Each node has a unique, randomly assigned nodeID in a circular 128-bit identifier space. Given a 128-bit destination address, Pastry routes the message towards the live node whose nodeID is numerically closest to the address using *prefix routing*. In prefix routing, a message with some address A is first sent to a host with a nodeID matching A in the first digit. This host forwards the message to a node with nodeID matching in the first two, and so on. Eventually, the message reaches the numerically closest node.

To support prefix routing, each pastry node maintains a routing table with a number of rows equal to the number of digits in a nodeID/address. The first row lists nodes whose nodeIDs differ in their first digit. The second lists nodes whose nodeIDs are identical in their first digit, but differ in their second, and so on. If a Pastry node has several peers which could fill a particular routing table entry, it chooses the node that is the fewest number of network hops away. When filling upper rows, Pastry typically has a large set of candidate nodes from which to choose. Thus, the first few rows in any Pastry routing table refer to nodes that are nearby in network terms. Pastry supports a mechanism by which new nodes can join the network, discovering nearby nodes in the process [10].

Fluid Replication can leverage Pastry as follows. Each WayStation is permanently part of the Pastry overlay. When a client wishes to find a set of candidate WayStations, it joins the overlay and then withdraws from it. In the process, the client fills its own Pastry routing table, and the first row represents the closest WayStations in terms of network hops. The client then uses these WayStations as candidates, selecting among them based on available network capacity.

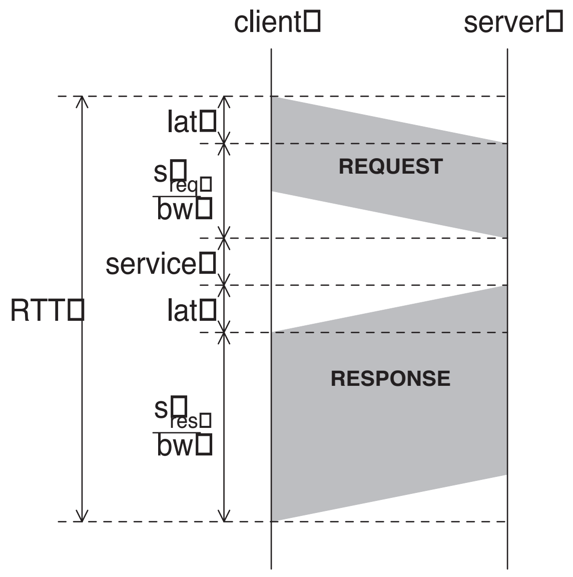
4.3 Making Observations

This section describes how we obtain individual observations of network performance. We first describe how we compute individual bandwidth and latency from round-trip-time measurements. Then, we explain how we discount queueing delay from these measurements.

4.3.1 Computing Bandwidth and Latency

We represent the end-to-end path between two hosts with a simple fluid-flow model [47]. In this model, the sequence of hops from source to sink can be represented as a single service queue with latency lat and bandwidth bw. A packet's delay is simply: $delay = lat + size/bw$. The terms lat and bw are considered to be the *effective* latency and bandwidth along a path rather than those of some physical link. As congestion along the path increases, effective latency increases and effective bandwidth decreases.

An individual client observes network performance to a server by sending a request to that host, receiving a response, and measuring the total elapsed time. We assume that such observations happen only as a side effect of normal traffic exchanged between one machine and another. They are not made through active probing to avoid overhead during times of decreasing network performance. Requiring more traffic to detect such situations



This figure illustrates the delays experienced by a request/response pair in our fluid-flow model.

Figure 4.1: Observing Network Performance

will only degrade conditions.

The total elapsed time consists of the time to transmit the request to the server, the time required to service the request, and the time required to transmit the response. This process is depicted in Figure 4.1. The round-trip time, RTT, with this model is:

$$RTT = (lat + \frac{s_{req}}{bw}) + service + (lat + \frac{s_{res}}{bw}) \tag{4.1}$$

where s_{req} and s_{res} are the sizes of request and response, respectively.

4.3.2 Discounting Self-Interference

Our goal is to determine the effective latency and bandwidth available to this host. Therefore, estimates should be independent of its behavior, but dependent on the other traffic present along the path. When a host transmits two requests very close to one another in time, the second will be queued behind the first, inflating the second's RTT. We must

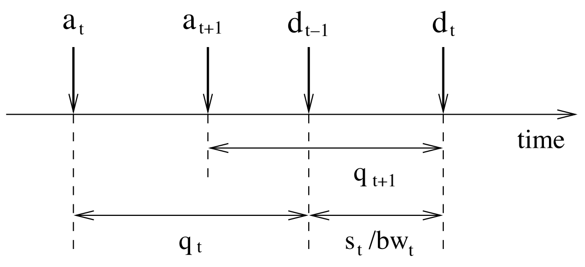


Figure 4.2: Self-Interference Queueing Delay

discount such *self-interference* [60]. To do so, we use the bandwidth estimate to determine self-imposed queuing delay.

Figure 4.2 illustrates how we compute the self-interference queueing delay. When message m_{t+1} arrives at the modeled queue at a_{t+1}, it must wait until the previous message m_t departs at d_t. So the queueing delay, q_{t+1}, is defined as: $q_{t+1} = d_t - a_{t+1}$. Because we do not explicitly measure the departure time, we compute q_{t+1} using the following:

$$q_{t+1} = max\left\{q_t + \frac{s_t}{bw_t} - (a_{t+1} - a_t), 0\right\} \tag{4.2}$$

where s_t is the size of message m_t, and bw_t is the bandwidth estimate generated at a_t.

4.4 Adaptive Filters

Typically, systems use exponentially weighted moving average (EWMA) filters to smooth noisy network observations. Such filters take the form:

$$E_t = \alpha E_{t-1} + (1 - \alpha) O_t \tag{4.3}$$

where E_t is the newly generated, smoothed estimate, E_{t-1} is the prior estimate, and O_t is the current observation. The term α is called the *gain*, and determines the filter's reactivity. If the gain is large, old estimates will dominate and the filter will be slow to change, making it stable. For example, TCP's RTT filter has a gain of 7/8. In contrast, filters with low gain will tend to be agile. Odyssey's network filters use gains as low as 1/8 to detect changes quickly. In the remainder of this section, we describe our four adaptive filters.

4.4.1 Flip-Flop Filter (FF)

The first filter, called Flip-flop, consists of two EWMA filters. One is agile, with a gain of 0.1, and the other is stable, with a gain of 0.9. A controller selects between the two. The underlying principle of this controller is to employ the agile filter when possible, but fall back to the stable filter when observations are unusually noisy. It employs a *control chart* [52] to make this decision.

Control charts are commonly used to provide statistical process control in manufacturing applications. They plot the sample mean, $\overline{x}$, of a controlled quantity against the desired population mean, μ, over time. The plot includes two *control limits*: the upper control limit (UCL), and the lower control limit (LCL). Usually, the control limits are defined to be $\mu \pm 3\sigma_{\overline{x}}$, where $\sigma_{\overline{x}}$ is the sample standard deviation. When a sample exceeds the control limits, the process is judged to be out of control. This is called the *3-sigma rule* [78].

We apply this technique to filter selection, but must make allowances for our domain. First, we do not know the true latency and bandwidth at any point; this is needed to generate a population mean. Second, those quantities are expected to change over time. Control charts are primarily used only to detect such shifts in mean, but we also want to recalibrate our control to the new mean value. Finally, we do not know the population standard deviation in advance, but need it to establish the control limits.

To address these shortcomings, we periodically change the center line and limits of

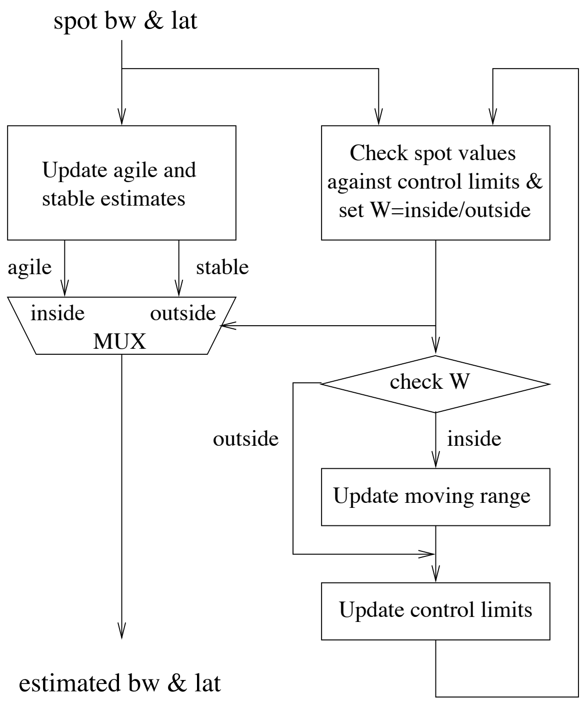


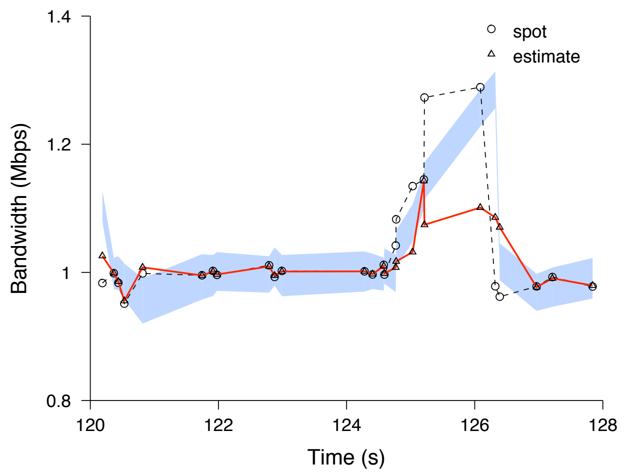
Figure 4.3: Flip-Flop Filter

the control chart, using the *moving range* to approximate the standard deviation. The moving range, $\overline{MR}$, is the average of the differences between adjacent points, $|x_i - x_{i-1}|$. The control limits use the moving range as a substitute for the sample standard deviation. Because the true value of $\overline{MR}$ may change over time, we smooth it using an EWMA filter with a gain of 7/8. The center line, which represents the population mean, is set to an exponentially weighted moving average of the estimated value, $\overline{x}$, with a gain of 1/8 to account for mean shifts. The control limits are then:

$$\overline{x} \pm 3\frac{\overline{MR}}{d_2} \tag{4.4}$$

where d_2 estimates the standard deviation of a sample given its range. When the range is from a sample of two, as it is for $\overline{MR}$, the value of d_2 is approximately 1.128 [52]. In the process control literature, this type of control chart is called the *individual-x chart* [65].

Figure 4.3 shows how the Flip-flop filter chooses between its two EWMA filters and



The shadowed area represents the values between the upper and lower control limits. Around 125 seconds, the Flip-flop filter switches from the agile filter to the stable one.

Figure 4.4: Flip-Flop Filter Example

when it updates its control values. As long as observations (also called *spot* values) fall within the 3-sigma limits, we use the agile filter. If the values fall outside the limits, we fall back to the stable one. In other words, when observations are unusually variable, the filter dampens its estimates. The moving range is updated only when the spot values fall within the 3-sigma limits. This prevents the moving range from becoming too wide. The center line and control limits are adjusted for each packet received.

Figure 4.4 shows an example of the Flip-flop filter. The shadowed area represents the values between the upper and the lower control limits, the dotted lines show the spot values, and the solid line plots the estimate over time. When the spot values are outside of the control limits, the controller selects the stable filter; at other times, it chooses the agile filter. For example, around 125 seconds, the Flip-flop filter switches from the agile filter to the stable one because the spot value is above the upper control limit, causing a drop in estimate. This drop is a result of maintaining the two static-gain filters independently; the stable filter is consistently more conservative in following changes.

4.4.2 Stability Filter (SF)

The second filter is called the Stability filter. Like the Flip-flop filter, the Stability filter dampens estimates in proportion to the variance of spot observations. However, rather than using variance to select between static-gain filters, we use a measure of the variance to dynamically change the gain of a single filter.

The goal of the Stability filter is to dampen estimates when the network exhibits unstable behavior. As consecutive observations diverge, the instability increases. This, in turn, increases the gain, making the filter stable. Our measure of instability is similar to the moving range used in the Flip-flop filter. To compute instability, U, we use a second EWMA filter:

$$U_t = \beta U_{t-1} + (1 - \beta)|x_t - x_{t-1}| \tag{4.5}$$

where x_t is the spot value measured at time t, and β is 0.6; this value was chosen empirically to minimize estimation error. We set the gain to be:

$$\alpha_t = \frac{U_t}{U_{max}} \tag{4.6}$$

where U_{max} is the largest instability seen in the ten most recent observations.

An example of the Stability filter is shown in Figure 4.5. The dotted line shows the changing spot values, and the solid line tracks estimated values. The filter is relatively robust against large changes in performance, but tracks small changes well.

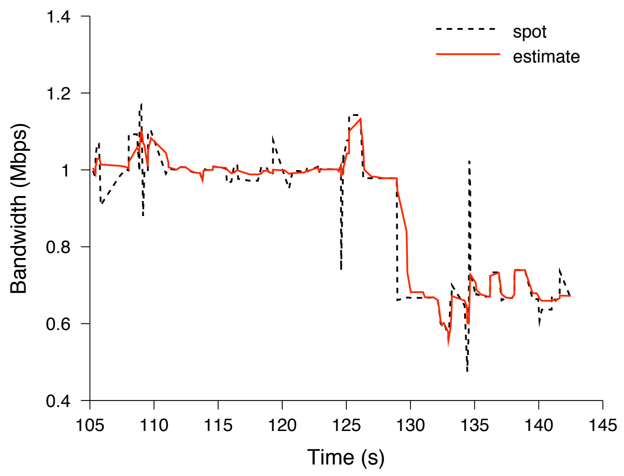


Figure 4.5: Stability Filter Example

4.4.3 Error-Based Filter (EF)

The Error-based filter follows the general form of the Stability filter, but takes a different approach in adapting its gain. Rather than vary gain based on the variance in network observations, it bases gain on the predictive power of its estimates. When the Error-based filter produces estimates that match well with reality, these estimates are given more weight through higher gain. When the filter does not accurately match observed values, we decrease its gain so that it can converge more quickly.

The error at an individual observation is the difference between the past estimate and the current observation: $|E_{t-1} - O_t|$. Rather than use raw error values at each step, we filter these errors through a secondary EWMA filter; it then plays a role similar to that of U_t in the Stability filter. Estimator error, Δ_t, is:

$$\Delta_t = \gamma\Delta_{t-1} + (1-\gamma)|E_{t-1} - O_t| \tag{4.7}$$

where γ is 0.6. This value was chosen empirically in the same manner as β. We then set

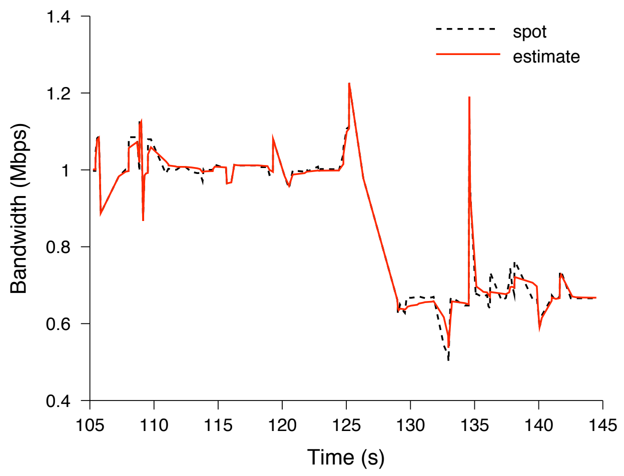


Figure 4.6: Error-Based Filter Example

the gain of the Error-based filter to be:

$$\alpha_t = 1 - \frac{\Delta_t}{\Delta_{max}} \tag{4.8}$$

where Δ_{max} is computed the same way as U_{max}.

An example of the Error-based filter is shown in Figure 4.6. The dotted line shows the changing spot values, and the solid line tracks the estimated value. In contrast to the Stability filter, the Error-based filter tracks large changes quickly, but is robust against small fluctuations in performance.

4.4.4 Kalman Filter (KF)

The final filter we have explored is an application of the Kalman filter. Kalman filters, if properly applied to a linear system, are *optimal* in that they minimize mean squared estimation error. While an optimal Kalman filter requires significant knowledge of the system—knowledge that is not available when estimating network performance—one can employ reasonable guesses that give a good result.

Kalman filters describe a system in terms of *state space notation*. For our model, this is:

$$\mathbf{X}(t+1) = \Phi(t)\mathbf{X}(t) + \mathbf{W}(t) \tag{4.9}$$

where $\mathbf{X}$ is the system state vector, Φ is a constant matrix combining the state variables, and $\mathbf{W}$ is a matrix representing system noise.

Our filter estimates latency and bandwidth given round-trip time measurements; latency and bandwidth are the state variables. Recall that the round-trip time is proportional to $1/bw$. Therefore, in order to make the system linear, we use $1/bw$ rather than bw as the second state variable. So, the system state vector is written as:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} lat \\ \frac{1}{bw} \end{bmatrix} \tag{4.10}$$

The state equations are then written as:

$$x_1(t+1) = x_1(t) + w_1(t) \tag{4.11}$$

$$x_2(t+1) = x_2(t) + w_2(t) \tag{4.12}$$

where w_1 and w_2 are the (unknown) system noise in latency and bandwidth, respectively. Putting these together, we have:

$$\Phi = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.13}$$

$$\mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \tag{4.14}$$

In order to apply a filter to the system state, one must measure it:

$$\mathbf{Z}(t) = \mathbf{H}(t)\mathbf{X}(t) + \mathbf{V}(t) \tag{4.15}$$

Here, $\mathbf{Z}$ is a matrix containing the measured state values, $\mathbf{H}$ is a constant matrix combining the system state, and $\mathbf{V}$ is a matrix representing the (unknown) measurement error.

Our measurement, RTT, is a scalar, so $\mathbf{Z}$ is the scalar z, and $\mathbf{V}$ is the scalar v. Our network model says that $RTT = 2lat + s/bw$, so the measurement equation is:

$$z(t) = 2x_1(t) + s(t)x_2(t) + v(t) \tag{4.16}$$

where $s(t)$ is the sum of the sizes of the request and response pair at time t. Hence, matrix H is:

$$\mathbf{H} = \begin{bmatrix} 2 & s(t) \end{bmatrix} \tag{4.17}$$

To apply a Kalman filter, we must know the process noise covariance matrix $\mathbf{Q}$ and the measurement error covariance matrix $\mathbf{R}$. $\mathbf{Q}$ is the covariance matrix of $\mathbf{W}$; $\mathbf{R}$ is the covariance matrix of $\mathbf{V}$. Because $\mathbf{W}$ and $\mathbf{V}$ are not known, their covariances are unavailable. Therefore, we assume that the noise in latency and bandwidth is independent, making their products zero.

$$\mathbf{Q}(t) = \begin{bmatrix} {w_1}^2 & 0 \\ 0 & {w_2}^2 \end{bmatrix} \tag{4.18}$$

$$\mathbf{R}(t) = [v^2] \tag{4.19}$$

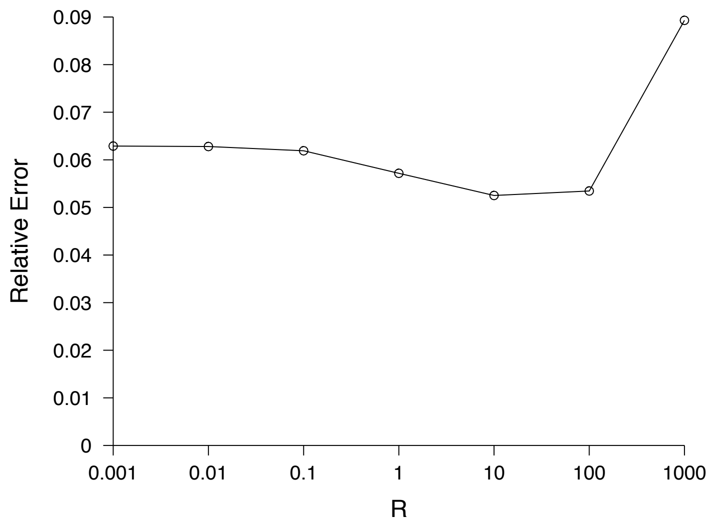
Intuitively, $\mathbf{Q}$ represents the degree of variability in latency and bandwidth. The terms ${w_1}^2$ and ${w_2}^2$ describes the degree to which one is more volatile than the other. $\mathbf{R}$ describes the measurement uncertainty. If measurements are uncertain, system state estimates should not change drastically with individual measurements. In other words, the filter becomes stable when $\mathbf{R}$ is large.

Unfortunately, we do not know the relative variances in latency and bandwidth, nor do we have an accurate picture of measurement noise. Although good guesses for $\mathbf{Q}$ and $\mathbf{R}$ will not provide optimal performance, they will make the filter perform reasonably well [31]. The real impact of $\mathbf{Q}$ and $\mathbf{R}$ on the filter's performance is determined by the relative magnitudes of each. For matrix $\mathbf{Q}$, we assume that the variances of noise in latency and bandwidth are the same, and set both ${w_1}^2$ and ${w_2}^2$ to 1.

With $\mathbf{Q}$ fixed, we then empirically determine $\mathbf{R}$ using a simple ad hoc network simulation, which is also used to evaluate the performance of filters. There are two reasons that we chose this simulation to determine the best $\mathbf{R}$ value. First, the nominal bandwidth values are known, allowing us to compute the relative error, $|estimate - true|/true$. Second, the noise endemic to mobile networks tests the stability of the filter. In other words, the error values are not only affected by how fast each filter settles, but also how stable it is once settled. In contrast, if we tune the parameter in an environment that does not have much noise after each persistent change, the error is mainly determined by how fast the filter settles to a new value. In this case, a smaller $\mathbf{R}$, which makes the filter agile, would always be preferable.

Figure 4.7 shows the relative error of the Kalman filter with $\mathbf{R}$ values varied from 0.001 to 1000. The results show that $[10]$ is the best value for $\mathbf{R}$, and that this value is relatively insensitive to the performance of the Kalman filter. In other words, any value except 1000 does not make a large difference in performance. This holds true for all of our experiments. The resulting matrices are:

$$\mathbf{Q}(t) \;=\; \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.20}$$



This figure shows relative error of Kalman filter with different R values under the simple ad hoc network simulation.

Figure 4.7: Kalman Parameter

$$\mathbf{R}(t) \;=\; [10] \tag{4.21}$$

Given matrices $\mathbf{Q}$ and $\mathbf{R}$ with the state equations, applying the Kalman filter is straightforward [8].

4.5 Evaluation

In evaluating the candidate filters, we set out to answer the following questions:

- How agile are the filters in the face of idealized, persistent changes in network capacity?
- How stable are the filters in reaction to transient changes in network capacity?
- How do filters react in the presence of changing network congestion?
- How do filters react in the presence of topology changes in a mobile network?
- How do filters compare in a complex, ad hoc network?

To answer these questions, we subjected each filter to a variety of networking scenarios. Many of these are based on simulations that vary latency, bandwidth, cross traffic, or node topology over time. For the simulations, we used ns [6], a packet level network simulator, with the Monarch extensions for mobile, ad hoc networking [7]. The Monarch extensions include near/far propagation models, packet capture, and the complete IEEE 802.11 MAC implementation [22]. The 802.11 MAC layer incorporates collision avoidance and link-level acknowledgement and retransmission. The nominal transmission range is 250 m. For the ad hoc network routing protocol, we used dynamic source routing [26].

We further modified ns to implement links whose performance can change according to a profile we provide. For each experiment, we generate a profile that specifies the topology, link characteristics, traffic, and how each of these change over time. This gives us two important benefits. First, since we know the objective state of the network, we can precisely quantify the behavior of our estimator. Second, these changes can be made arbitrarily taxing, stressing the adaptive estimators.

In the situations we examined, the long-term estimates for latency and bandwidth produced by all the filters are roughly the same. The filters differ only in how quickly they arrive at this estimate, and how well they hold to that estimate once there. In other words, they differ in agility and stability. We use *settling time* as a measure of agility. Settling time measures the elapsed time it takes a filter to produce an estimate within 10% of some new nominal value. Lower settling times are better. We use two different measures to describe stability, depending on the context. The first, *coefficient of variation* (CV), is used during times when network performance is nominally stable. It is the ratio of standard deviation to mean. It measures the degree to which measurement noise affects a filter's estimates; lower CV values are better. The second, *mean squared error*, is used to measure resistance to transients. We use this metric because it penalizes filters that are disturbed by large amounts for a short time more than those disturbed by small amounts for a longer time. An error that is large in magnitude is more likely to cause an adaptive system to make a poor decision.

4.5.1 Detecting Persistent Changes

In the first experiment, we subject each filter to two *step functions*: an immediate change from one bandwidth to another. A client and a server are connected by a single link, with performance that varies over time. We explored two different changes. In the first, called *step-up*, we increase the available link bandwidth from 1 Mb/s to 10 Mb/s. In the second, called *step-down*, the change is in the other direction. Our goal in this experiment is to test each filter's agility; how long it takes each filter to recognize a persistent change in bandwidth.

Since the filters rely only on passive measurements, their agility depends heavily on the rate of the underlying network traffic. To explore this, our traffic generator uses a Poisson process with means varying between one and sixteen packets per second; the fastest Poisson process will saturate the link when it is at low bandwidth. All the measurements are done at the UDP layer, so the sizes of requests and responses can be larger than the MTU, which is 1500 bytes for Ethernet. For our experiment, each request is small, while each response is randomly chosen to be either small (512 bytes) or large (8 KB) with equal probability; we used 8 KB responses to represent NFS traffic. Large responses will be

Experiment	FF	SF	EF	KF	Ody	TCP
step-up	7.4	10.8	4.5	8.5	6.0	35.5
step-down	6.5	12.6	14.8	3.5	8.8	86.8

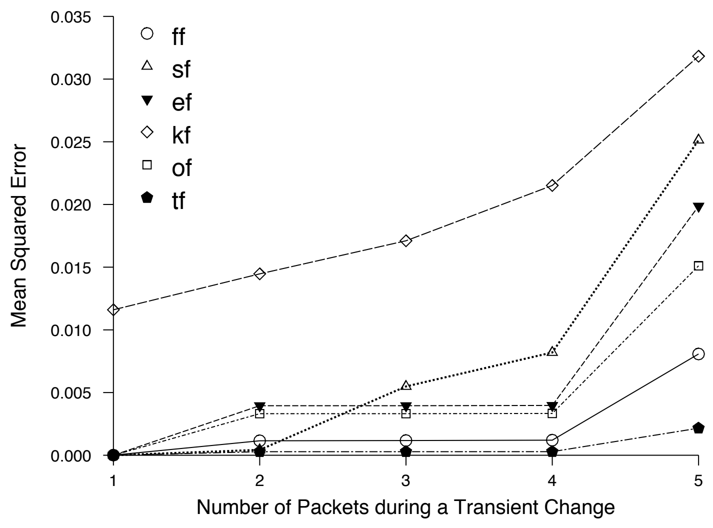
This table depicts the agility of each filter under a sharp, ideal change in bandwidth. It shows the average number of packets until filters settle. In both cases, the Odyssey filter and the Flip-Flop filter perform well.

Table 4.1: Agility: Settling Time under Varying Rates

fragmented by IP. Five trials of each experiment were taken with differing random seeds for the request generator.

The results for this experiment are shown in Table 4.1. Each number represents the average packets required for each filter to settle. It is the average of five scenarios: 1, 2, 4, 8, and 16 packets per second. The first line of data shows the filters' ability to detect increases in bandwidth; the second line quantifies detection of decreases. The static EWMA filters behave as expected. The stable TCP filter is the slowest to converge, while the agile Odyssey filter is among the fastest. Only the Flip-flop filter is comparable to the Odyssey filter in both cases.

The relatively slow response of EWMA-based filters to detect decreases compared to increases is not surprising, and agrees with the evaluation of the Odyssey filter [56]. For example, the Odyssey filter settles within 11.25% of the goal (10 Mb/s) with one accurate observation for the increase in bandwidth, while it settles within 112.5% of the goal (1 Mb/s) for the decrease. The Kalman filter does not follow this, and detects decreases faster than increases for all packet numbers.



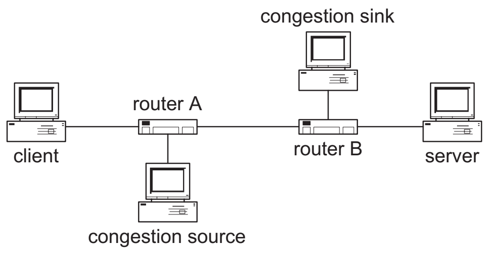
This figure depicts each filter's stability in the face of a transient drop in bandwidth. The X axis shows the number of packets each filter observes during the performance drop; the Y axis gives the mean squared error. The TCP filter and the Flip-Flop filter are the least perturbed.

Figure 4.8: Stability: Resistance to Transient Change

4.5.2 Resisting Transient Changes

The second experiment gauges the filters' resistance to short-term drops in performance. In this experiment, each filter is subjected to a short decrease in bandwidth from 10 Mb/s to 1 Mb/s. In order to fairly compare the filters, we must ensure that the same number of packets experience each transient change. So, unlike the agility experiments, we use a constant transmission rate, and vary the length of the transient from 1 to 5 packets. The sizes of these packets are chosen as before; requests are small, with responses randomly small or large. Since they are random, we perform five trials at each duration.

To evaluate stability of filters, we compute the mean squared error with a constant nominal value of 10 Mb/s. The results for bandwidth estimation are shown in Figure 4.8. As expected, the TCP filter is the most resistant to change. The Flip-flop filter is also very



This figure illustrates the network topology for the congestion experiments. The client and server exchange requests and responses, through routers A and B. During the experiment, the congestion source begins a traffic stream to the congestion sink.

Figure 4.9: Topology for Congestion Experiments

stable. The Kalman, Error-based and Odyssey filters are susceptible to changes, while the Stability filter follows longer transients more aggressively than shorter ones. As the duration of the transient increases, the Stability filter judges the transient value to be stable, and follows it.

4.5.3 Detecting Congestion

The previous two experiments explored changes in link capacity, as might happen during a vertical handoff. However, link congestion also determines the end-to-end bandwidth that is ultimately available to a client and a server. This experiment characterizes each filter's ability to detect and estimate the effects of congestion.

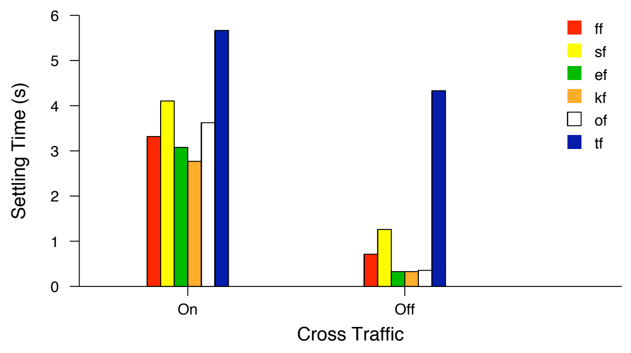
The client and the server are part of a six-node network, depicted in Figure 4.9; each link has a capacity of 10 Mb/s. They exchange request-response traffic with a Poisson process at an average rate of 140 Kb/s. The experiment lasts a total of 150 seconds. 50 seconds into the experiment, the congestion source sends a constant bit rate (CBR) stream of 5 Mb/s in 8 KB chunks to the congestion sink, reducing the available bandwidth along the client-server path. This congestion traffic lasts for 50 seconds, and then stops. The results reflect five trials with different random seeds.

During congestion, observed round-trip times are exceptionally noisy. This is because cross traffic may or may not interfere with any particular request-response pair. In other words, the observed traffic is only partially perturbed by congestion. If a router has cross-traffic queued when a request or response arrives, it will be delayed, but otherwise it will not. While each filter produces the same long-term estimates, they are consistently optimistic, at just over 6 Mb/s. This optimism is a direct result of the fact that congestion traffic only partially perturbs the observed round trips.

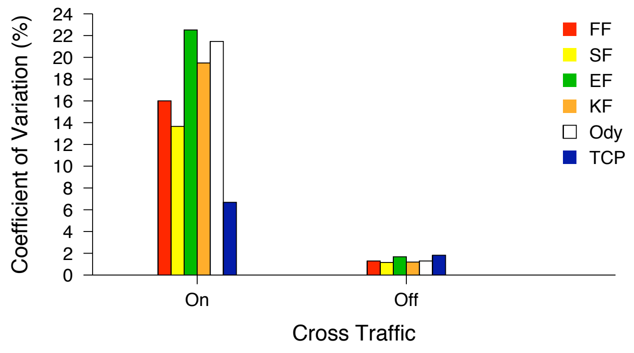
Figure 4.10 shows the agility and stability of each filter. The first set of bars shows the result from 50 seconds to 100 seconds when the congestion traffic is present; the second set shows it from 100 seconds to 150 seconds. To compute the settling time, one needs the nominal bandwidth during each period. We determined this value by taking the average estimate produced by all filters over the last half of each 50 second interval. As explained above, this estimate will tend to be optimistic, but is the best one can do without measurement within the network. As shown in Figure 4.10(a), the TCP filter is clearly the least agile of any of the filters. The other five filters are very agile. As shown in Figure 4.10(b), all filters except the TCP are not stable when the congestion traffic is on. Among the five, the Stability and Flip-flop filters provide more stable estimates than the others.

4.5.4 Wide-Area Networks

To gauge whether instability is endemic to these filters in the presence of cross traffic, we subjected them to two different, real-world networking scenarios. In them, a client and a server exchanged ICMP ECHO packets, where each packet was either 512 bytes or 8 KB with equal probability. The requests were generated by a Poisson process with an



(a) Agility: settling times

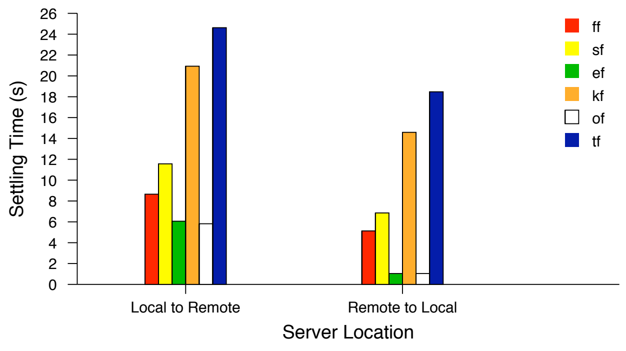


(b) Stability: coefficient of variation

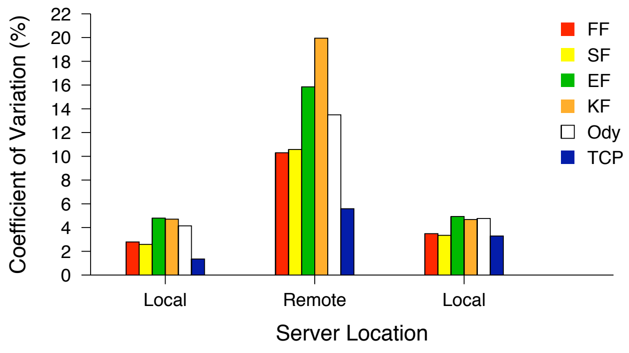
This figure depicts the filters' reaction to congestion. There are two categories across each X axis; when congestion is introduced, and when it is removed. Figure 4.10(a) gives the settling time for each filter; the Y axis is given in seconds. Figure 4.10(b) shows the coefficient of variation exhibited by each filter after it has settled.

Figure 4.10: Detecting and Tolerating Congestion

average rate of one packet per second. We took measurements between one client and two servers. One server was in the same subnet, and the other was located twelve hops and roughly 100 ms away. We repeated this for five trials to each server; each trial was 300 seconds long. In each trial, the client starts communication with the local server, switches to the remote one at 100 seconds, and switches back to the local at 200 seconds. The raw observations from each trial were recorded, and each filter was subjected to precisely the same set of observations.



(a) Agility: settling times



(b) Stability: coefficient of variation

This figure depicts the filters' reaction to routing changes in wide-area networks. The client first requests data from a co-located server, switches to a remote one, and then switches back. Figure 4.11(a) gives settling times for the two route changes. Figure 4.11(b) presents coefficients of variation in each distinct experiment phase after settling.

Figure 4.11: Performance over Wide-Area Networks

Since this environment is real, we do not know the nominal bandwidth. However, we need some measure of nominal to compute agility and stability for our filters. We used the average bandwidth of the second half of each 100-second period as the nominal bandwidth.

Figure 4.11(a) shows the settling times for each filter. The Odyssey and Error-based filters are the most agile ones. Although the Flip-flop and Stability filters are not as agile as the best performers, they are much better than both the Kalman and TCP filters.

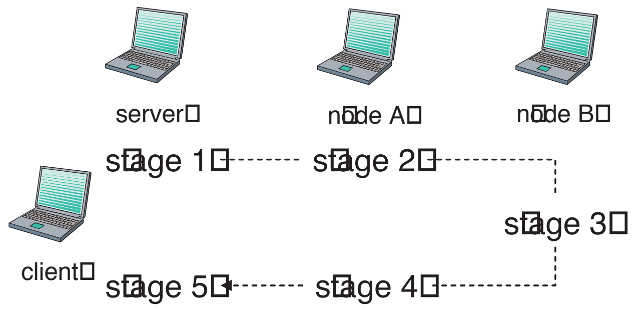


Figure 4.12: Topology Changes

Figure 4.11(b) shows the stability results for each 100-second period separately. The stability for all filters is much better when observing traffic from the local server than from the remote one. This is most likely due to the smaller number of hops combined with the lower likelihood of encountering cross traffic in the local case. The Kalman filter performs particularly poorly in the communication with the remote server. As expected, the TCP filter is very stable, but the Flip-flop and Stability filters also perform reasonably well.

4.5.5 Topology Changes in Mobile Network

The next experiment explores how our filters react to changing topologies in simple ad hoc networks without the presence of cross traffic. The topology for this simple network is shown in Figure 4.12. Three wireless nodes—one of them a server—are arranged in a line, each separated by 200 m. The client moves from the server's neighborhood to the vicinity of the far node, and then back, comprising five different stages. The client completes this circuit in five minutes; we use handoff times as the breakpoints between stages. The traffic rate was Poisson, with an average of four requests per second; requests were small and responses were 512 bytes or 8 KB with equal probability. We took five trials for each filter.

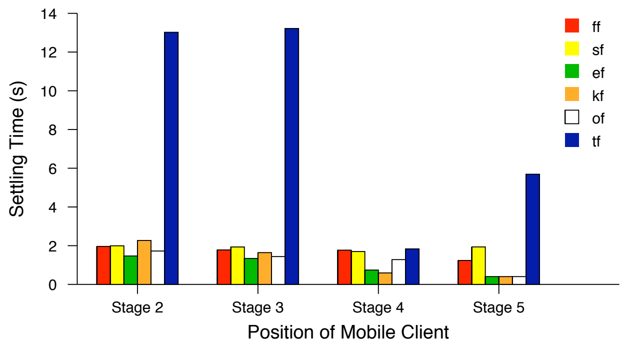
There are interesting effects even in this simple topology. First, although there is no cross traffic, the effective bandwidth changes as the client moves through the stages. This is because all nodes share the same physical channel. While the bandwidth of the physical device is 2 Mb/s, the effective bandwidth between the client and the server is divided by two when routed through node A, and by three through node B.

Second, the wireless MAC protocol allows collisions even when the client and the server communicate directly; the rate of collisions goes up with hop count. This leads to substantial variability in RTT observations, increased noise in all of the estimators, a higher average latency, and lower average bandwidth.

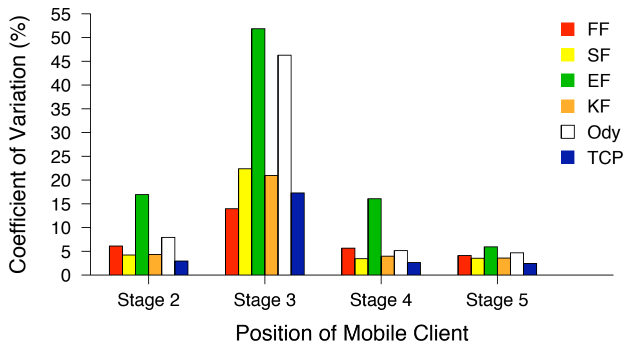
Figure 4.13(a) shows the agility of each stage. All of the filters except for TCP settle reasonably well, though the Error-based filter holds a slight advantage. Figure 4.13(b) shows the stability. All of the filters except the Error-based and Odyssey filters show comparable stability. The Flip-flop filter has an advantage in the worst situation: stage three, where collisions and link-level retransmissions are most frequent. This is because the infrequent retransmissions in the mobile experiment lie outside the Flip-flop control limits, causing the Flip-flop filter to choose the estimates generated by the stable filter.

4.5.6 Ad Hoc Wireless Networks

In the final experiment, each filter is given the task of predicting available capacity in an ad hoc network with substantial cross traffic. This is a demanding scenario, as the network topology—and hence route and cross traffic—are constantly changing. In such a network, past observed performance is at best a loose predictor of future results. While no filter can predict perfectly in such an environment, many applications require the best



(a) Agility: settling times



(b) Stability: coefficient of variation

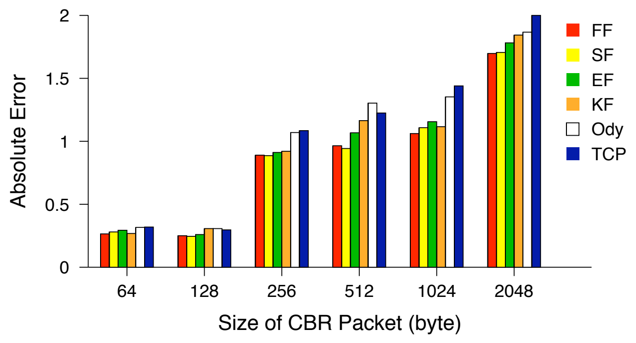
This figure shows the filters' performance when moving through a simple ad hoc network. Figure 4.13(a) shows the time each filter requires to determine that a handoff has occurred. Figure 4.13(b) presents the stability during each stage after settling.

Figure 4.13: Performance under Changing Topology

predictions they can get.

We placed 50 nodes in a space 1500 meters by 500 meters. Initial node locations are distributed throughout the space uniformly randomly. Each node follows the random waypoint model [26] with a pause time of 20 seconds and a maximum speed of 20 meters per second. Each trial simulated 500 seconds.

The 50 nodes were formed into 25 pairs. One pair served as the estimating client and the corresponding server, exchanging requests and responses with Poisson distribution, averaging four requests per second. The remaining pairs are CBR connections, each source



This figure shows the filters' ability to correctly predict round trip times in a large ad hoc network. The X axis lists the packet sizes of background traffic. The Y axis shows the average error produced by each filter's estimates, in seconds; lower errors are better. In general, increased background traffic leads to less accurate predictions. At relatively low levels of background traffic, each filter is comparable. As background traffic increases, the Flip-Flop and Stability filters begin to show advantages.

Figure 4.14: Accuracy in an Ad Hoc Network

transmitting four packets per second. Since the amount of traffic substantially affects noise in observations, we repeated the simulations with six different packet sizes. The size varied from 64 bytes to 2 KB. We ran five different trials for each packet size.

In this experiment, we cannot easily compute the nominal bandwidth. Thus, we cannot use the usual metrics. Instead, we computed average absolute error, the difference between the measured RTT and the estimated RTT based on the bandwidth and latency estimates. Figure 4.14 shows the results. For every packet size, the Flip-flop and Stability filters work the best. They show advantages more clearly as the background traffic increases.

Interestingly, neither the TCP filter nor the Odyssey filter performed well in the ad hoc network. This is because one was incapable of detecting changes quickly enough, while the other was overly sensitive to noise. Either problem is enough to disqualify a filter from consideration in this setting.

	Filter	FF	SF	EF	KF	Ody	TCP
Agility	Step Up	$\circ$	$\times$	$\circledcirc$	$\times$	-	$\times$
	Step Down	$\circledcirc$	$\times$	$\times$	$\circledcirc$	-	$\times$
	Congestion	$\circ$	$\circ$	$\circ$	$\circledcirc$	-	$\times$
	Wide-Area	$\times$	$\times$	$\circ$	$\times$	-	$\times$
	Mobile	$\circ$	$\circ$	$\circledcirc$	$\circ$	-	$\times$
Stability	Transient	$\circ$	$\times$	$\times$	$\times$	$\times$	-
	Congestion	$\times$	$\times$	$\times$	$\times$	$\times$	-
	Wide-Area	$\circ$	$\circ$	$\times$	$\times$	$\times$	-
	Mobile	$\circ$	$\circ$	$\times$	$\circ$	$\times$	-

This table summaries the performance of six filters, compared with the reference filter. As the reference filter, we used the Odyssey filter for agility, and used the TCP filter for stability. For each experiment, filters that worked better than the reference are marked $\circledcirc$, those that are comparable to the reference are given $\circ$, and those that performed poorly are marked $\times$.

Figure 4.15: Evaluation Summary

4.5.7 Overall Performance

Figure 4.15 summaries the results of the experiments. Clearly, neither the Odyssey nor TCP filter is a good choice overall. However, the Odyssey filter is always very agile, and the TCP filter is always very stable. So, we use them as the references against which to compare. For agility, the filters that are comparable to Odyssey are marked $\circ$. Those that work better are marked $\circledcirc$, while those that perform substantially worse are marked $\times$. For stability, we compare against TCP, with the same conventions. The top half of the table shows the agility results, and the bottom half shows the stability results. We did not include the results from the complex ad hoc network experiment presented in Section 4.5.6 because neither static-gain filter worked well.

The Flip-flop filter is clearly the winner. It often produced agility comparable to the agile Odyssey filter and stability comparable to the stable TCP filter. In the few settings in which it did not compare with the reference filter, it often outperformed its adaptive peers.

For the wide-area experiment, the Flip-flop filter was more agile than the Stability and Kalman filters, but not the Error-based one. For the congestion experiment, the Flip-flop and Stability filters were much more stable than the other two adaptive filters.

The Stability filter also worked well in most of the experiments. It provided stability comparable to Flip-flop, but was always slower than Flip-flop in detecting persistent changes. While the Error-based filter provided excellent agility, it was too unstable to be considered useful. The Kalman filter was also very agile, but has a tendency to follow transient changes in bandwidth. We suspect that the Kalman filter could be tuned to behave more reasonably, but do not have confidence that such tuning would apply to all possible circumstances.

4.6 Summary

Adaptation to changing network conditions is critical to the success of mobile systems. However, knowing how those conditions change is a difficult problem. Individual observations of network performance may have little bearing on available capacity. Historically, systems have filtered such observations with static-gain EWMA filters. Such filters are biased either toward agility or stability. Ideally, a network estimator would be agile when possible, but stable when necessary; it should adapt to the prevailing network conditions.

We have developed four candidate filters with this goal in mind. Each one examines the behavior of the network, and tunes itself in an attempt to provide accurate and timely estimates. The first uses techniques from statistical process control to select between two static-gain EWMA filters, one agile and the other stable. Two of these filters are modifications of an EWMA filter that use heuristics to vary the filter's gain dynamically. The fourth is an application of the well-known Kalman filter.

We evaluated these filters in a variety of contexts, including persistent and transient capacity changes, the presence of cross traffic, and changes in topology. We compared each adaptive filter to two static-gain EWMA filters, one agile and the other stable. We find that in the majority of scenarios, the filter based on statistical process control provides agility comparable to the agile Odyssey filter and stability comparable to the stable TCP filter. We believe that this filter presents the best choice for adaptive, mobile systems.

CHAPTER 5

Meta-Data Management

Once a client has chosen a nearby WayStation to act as a replica, the client interacts with that WayStation as if it were the client's home server—all uncached read requests are satisfied by the WayStation, and modified files are shipped to the WayStation immediately on close. This model of WayStation-client interaction borrows heavily from AFS [20].

In AFS, servers provide the centralized point of administration. They are expected to be carefully maintained to provide the best possible availability to their clients. Consistency between client caches and server state is maintained by *callback*.

When referencing an uncached file, an AFS client must obtain the file from its server. As a side effect, the server establishes a callback on that file. The client is then free to use the cached copy. If an AFS client modifies a file, the new version is sent to the server on `close`; this is called a *store* event. The server *breaks callback* on any other clients caching this file, invalidating their copies, before accepting the store.

Clients using the same WayStation will see callback-based consistency between one another. However, providing this guarantee to far-off clients at other WayStations is too expensive, particularly given the low incidence of write sharing.

Thus, the consistency mechanism between WayStations and servers is asynchronous. WayStations batch and periodically ship notifications of updates to the central server, which relays those advertisements to other WayStations that store old versions. While these notifications are asynchronous, they are inexpensive, and can be performed frequently, limiting the window of potential inconsistency to the point where the overwhelming majority of clients will never see conflicting updates.

5.1 Design

In Fluid Replication, each WayStation maintains a log of mutating operations, called an *update log*, and sends it to the server periodically during reconciliation. Upon receiving the message, the server invalidates other copies at WayStations immediately. This reconciliation protocol is designed to be fault tolerant. After sending a message, a node need not wait for an acknowledgement for the message. It proceeds with other work, assuming the message has arrived at the destination safely. If the message is lost, it will be detected at the next interaction.

5.1.1 File States

In Fluid Replication, a file is in one of four states: *modified*, *safe*, *visible*, or *stable*. When a client modifies a file, the file is in the *modified* state. If the client is connected to a WayStation, the update is written-through and the file becomes *safe*. When the file reaches this *safe* state, the update will survive destruction or loss of the client machine. The update becomes *visible* to the server when the WayStation reconciles with the server. The WayStation is responsible for an update until its contents are propagated to the server.

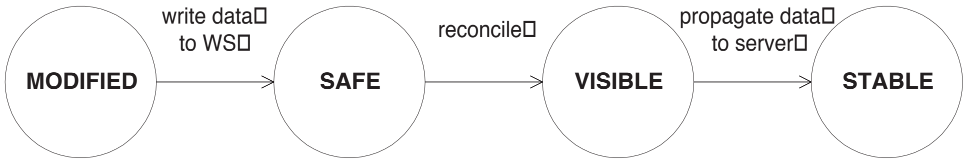


Figure 5.1: States of File Versions

Operation	Fields
common fields	**client, credentials, id, time, ws**
create	attribute, child fid, flag, mode, name, parent fid
mkdir	attribute, parent fid, child fid, name
rename	destination fid, destination name, source fid, source name, target, type
rmdir	fid, name, parent fid
setattr	attribute, fid, parent fid
symlink	attribute, destination fid, destination name, link fid, source name
unlink	fid, name, parent fid
store	count, fid, offset, parent fid

Table 5.1: Update Log Entries

When the data are at the server, the state of the file becomes *stable*. Figure 5.1 summarizes the states of a file.

5.1.2 Reconciliation

Once a client has selected a WayStation, that WayStation is treated as the client's file server. Clients cache files from WayStations, which manage callbacks for those clients. Dirty files are written back to WayStations on `close`. Each WayStation maintains an update log, which is similar to Coda's *replay log* [34]. It contains file stores, directory operations, and meta-data updates. Table 5.1 shows log entries with their corresponding fields. Note that the log entry for the `store` operation does not have a data field; data is sent separately. Redundant log records are removed via cancellation optimizations [33].

The server maintains an update log whenever one or more WayStations hold replicas of files on that server. This log is a merged log of all replica sites. The server also tracks which files are cached at the WayStation, called the *interest set*. The bookkeeping for interest sets is similar to that for callbacks, but the information is used only during reconciliation.

Each WayStation checks periodically if it has any update that has not been sent to the server. If it has unsent updates, it sends the server its update log and a list of files that it has evicted from its cache. Upon receiving the reconciliation message, the server removes from the WayStation's interest set the files that the sender evicted from its cache, and checks each WayStation log record to see if it is serializable. If it is, the server invalidates the modified object, and records the WayStation as the replica holding that version. If it is not, conflicts need to be manually fixed. The server also invalidates the files at other replicas immediately instead of waiting until other nodes reconcile. The server responds to the WayStation with a set of files that are now in conflict, if any. This response concludes the reconciliation process.

WayStations truncate their update logs on successful reconciliation, but the server cannot truncate its log immediately because it maintains a merged log of all replica sites. To apply the updates in its log to the files in disk, the server first needs to have logs of all replica sites. To check this, the server maintains the last reconciliation time for each WayStation holding a replica; let t_{oldest} be the earliest such time. The server then applies the updates that happened prior to t_{oldest}. After this, the server truncates its log; it need only hold log records from t_{oldest} forward. Unfortunately, slow WayStations can force the server to keep an arbitrarily large log.

This policy is in contrast to that taken by Coda's implementation of optimistic server replication. In Coda, only the tails of large update logs are retained; objects with discarded log records are marked in conflict and will need manual repair. Given Coda's presumption of tightly-coupled and jointly-administered replicas, this is an appropriate design choice. However, since WayStations are administered by independent entities, it would be unwise to allow a WayStation's absence to necessitate many manual repairs.

If each WayStation reconciles at a constant rate, all updates are globally visible within the longest reconciliation period. In order to provide clients with well-bounded visibility, reconciliation must be a lightweight operation. This is why reconciliations exchange only notification of updates, not their contents. Because sharing is rare, aggressively exchanging file contents increases reconciliation time without improving client access time. Leaving shared accesses to pay the full cost of wide-area network delays preserves performance, safety, and visibility for common-case operations.

When a WayStation requests a file, the server always provides the most recent version of the file that is known to the server. If the server has not applied the log to the requested file, the server applies the log to the file in memory and sends the updated file. If the proper version is not at the server, the server first fetches it from the WayStation that is responsible for the file, and then sends it to the requesting node. We call this fetch by the server a *back-fetch*.

5.1.3 Fault Tolerance

To initiate reconciliation, a WayStation sends the server a reconciliation message, m_R. Upon receiving this message, the server invalidates other copies by sending out invalidation messages, m_I. Because these messages are sent over a wide-area network, the cost of waiting for confirmation is high. Furthermore, the sender may need to wait indefinitely if the sent message is lost due to network or machine failures.

Our goal in designing the protocol is to be tolerant of failures. To achieve this, a WayStation does not wait for confirmation for a reconciliation message, m_R, and the server does not wait for confirmation for an invalidation message, m_I. As soon as a WayStation sends m_R, it assumes that the message is received successfully and continues processing client requests. The server sends m_Is to replicas and assumes they are received successfully; the server need not send m_I repeatedly for unavailable replicas. This design decision is made to reduce network traffic and server load.

Because messages are not reliable, Fluid Replication must detect and resend missing messages. We accomplish this through an extension to logical clocks [45]. Each party—the WayStation and the server—tracks the logical time at which it sent its last message. Likewise, each party records the sender's logical time for each message it receives. These recorded stamps are piggy-backed onto the next message sent to the originator, providing receipt confirmation as a side effect.

There are two message types: WayStations send reconciliation messages, m_R, to servers, and servers send invalidation messages, m_I, to clients. A WayStation i associates a timestamp, denoted $f_{i,j}$ for the most recent m_R to corresponding server j; this timestamp is increased with each accepted update operation. The server records the last received m_R timestamp from that client, $F_{i,j}$. Servers likewise maintain invalidation timestamps, $V_{i,j}$, which are incremented before each invalidation message is sent. These timestamps are recorded by WayStations, denoted as $v_{i,j}$.

Whenever a server contacts a WayStation, it includes its most recent reconciliation time, $F_{i,j}$. If this timestamp is earlier than the most recently-disclosed updates, the WayStation resends them along with the response to the server's request. As a consequence, a record cannot be pruned from a WayStation's update log until the WayStation receives a timestamp greater than the record's from the server. The server likewise resends any needed invalidation messages in responses to client requests.

Because no response from either party is seen without the complete reconciliation and invalidation history, this scheme guarantees that all parties see all updates in order. This consistency guarantee is called *monotonic writes* in the literature [73].

5.2 Implementation

Our Fluid Replication prototype consists of three components: a client cache manager, a server, and a WayStation. Each of these components is written primarily in Java. There are several reasons for this, foremost among them Java's clean combination of thread and remote procedure call abstractions and the benefits of writing code in a type-safe language. As will be shown, this decision has some performance implications.

5.2.1 Client

The bulk of the Fluid Replication client is implemented as a user-level cache manager, supported by a small, in-kernel component called the *MiniCache* [71]. The MiniCache implements the vnode interface [35] for Fluid Replication. It services the most common operations for performance, and forwards file operations that it cannot satisfy to the user-level cache manager.

Calls are forwarded from the kernel to the cache manager across the Java Native Interface (JNI). The calls are then satisfied by one of a set of worker threads, either from the local disk cache or via Remote Method Invocation (RMI) to the appropriate replica. Fluid Replication uses write-back caching; a close on a dirty file completes in parallel with the store to the replica.

5.2.2 WayStation and Server

WayStations and servers share much of the same code base, because their functionality overlaps. Data updates are written directly to the replica's local file system. Meta-data is stored in memory, but is kept persistently. Updates and reconciliations are transactions, but we have not yet implemented the crash recovery code.

We use Ivory [3] to provide transactional persistence in the Java heap; used in this way, it is similar to Recoverable Virtual Memory [69]. Originally, Ivory is a toolkit that traps updates to its data structures and propagates modified objects to peer replicas. It is used to automate state management for replicated web services.

5.2.3 Java

The decision to write the client in Java cost us some performance, and we took several steps to regain ground. Our first major optimization was to hand-serialize RMI messages and Ivory commit records. The default RMI skeleton and stub generator produced inefficient serialization code, which we replaced with our own. This reduced the cost of a typical RMI message by 30%.

The second major optimization concerned the crossing of the C-Java boundary. Each method call across this boundary copies the method arguments onto the Java stack, and returned objects must be copied off of the Java stack. We were able to avoid making these copies by using *preserialized objects*, provided by the Jaguar package [77]. Jaguar allows objects to be created outside the Java VM, and still be visible from within. We used this to pass objects, copy-free, between our C and Java code.

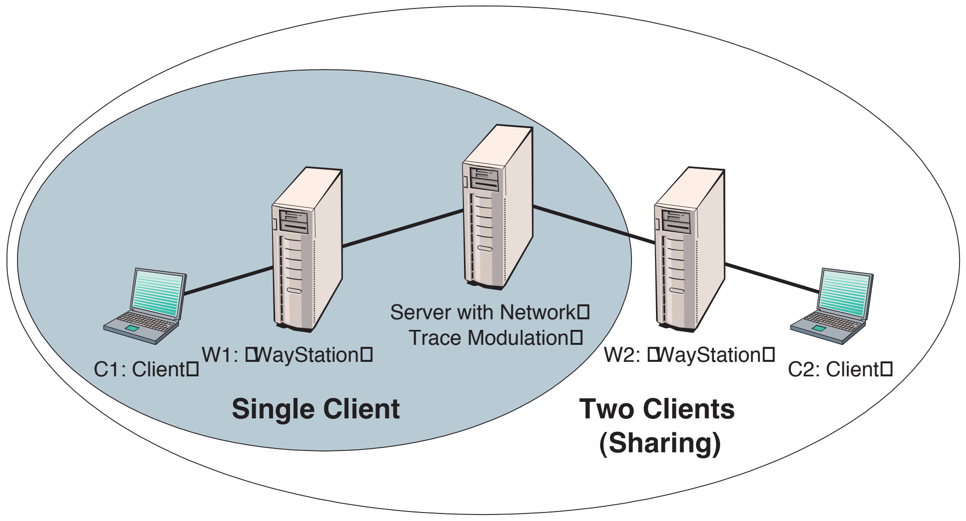
5.3 Evaluation

In evaluating Fluid Replication, we set out to answer the following questions:

- Can Fluid Replication insulate clients from wide-area networking costs?
- What is the impact of sharing on performance?
- How expensive is reconciliation?
- Can Fluid Replication provide the consistency expected by local-area clients to those in the wide area?

These questions concern the performance seen by individual clients and the behavior of the system as a whole. To measure client performance, we subjected our prototype to a set of controlled benchmarks. We explored system behavior through the use of trace-based simulation.

Our benchmarks ran on the testbed depicted in Figure 5.2. The WayStations are connected to the server via a *trace modulated* network. Trace modulation performs application-transparent emulation of a slower target network over a LAN [56]. We have created modulation traces that emulate the performance of a variety of different wide-area networking scenarios, listed in Table 5.2. Latency numbers in these traces are in *addition* to baseline latency, while bandwidth specifies the bottleneck capacity. All testbed machines run the Linux 2.2.10 kernel. The server and WayStations have 550 MHz Pentium III Xeon



This figure illustrates our benchmarking topology. Each client is well connected to its WayStation, but traffic between a WayStation and the server is subject to trace modulation.

Figure 5.2: Benchmark Topology

Scenario	Latency (ms)	Bandwidth
local area	0.0	10 Mb/s
small distance	15.0	4 Mb/s
medium distance	33.5	3 Mb/s
large distance	45.0	1 Mb/s
intercontinental	55.0	1 Mb/s
low bandwidth	0.0	56 Kb/s
high latency	200.0	10 Mb/s

This table lists the parameters used in each of our trace modulation scenarios. The local area scenario is the baseline against which we compare. The next four were obtained by measuring small `ping` and large `ftp` performance to four different sites. Bandwidth numbers are increased over `ftp` throughput by 20% to account for the difference in metric. The last two are used only to determine trends as latency or bandwidth worsen, and modify parameters orthogonally. Latencies are one-way; bandwidths are symmetric.

Table 5.2: Trace Modulation Parameters

Processors, 256 MB of RAM, and 10K RPM SCSI Ultra Wide disks. The clients are IBM ThinkPad 570s; these machines have 366 MHz Mobile Pentium IIs with 128 MB of memory.

For the trace-based studies, we collected traces comprising all activity on a production NFS server over one week. This server holds 188 users' home directories, plus various collections of shared data, occupying 48 GB. The users are graduate students, faculty, and staff spread throughout our department. They come from a variety of research and instructional groups, and have diverse storage needs. Generally speaking, the clients are not mobile, so they may not be wholly representative of our target domain. However, prior studies suggest that, at the operation level captured by our traces, mobile and desktop behavior are remarkably similar [55].

Traces were collected using `tcpdump` on the network segment to which the server was attached. These packet observations were then fed into the `nfstrace` tool [4], which distilled the traces into individual fetch and store operations. Note that this tool does not record operations satisfied by a client's cache. However, since NFS clients do not use disk caches, this will overstate the amount of read traffic a Fluid Replication client would generate. For the purposes of our analyses, we assume that each client host resides on a separate WayStation. The traces name 84 different machines, executing 7,980 read operations and 16,977 write operations. There are relatively few operations because most of our client population did not materially contribute to the total. Seven hosts account for 90% of all requests, and 19 hosts account for 99% of all requests.

5.3.1 Wide-Area Client Performance

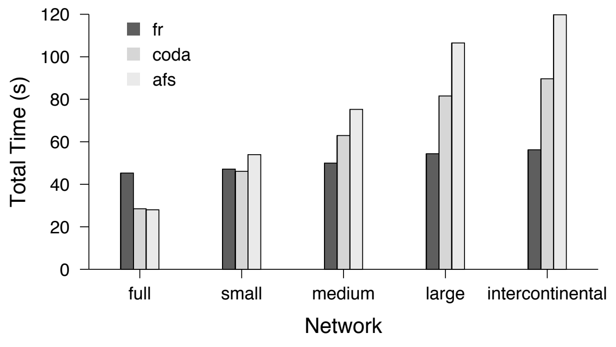
How effectively does Fluid Replication isolate clients from wide-area networking costs? To answer this question, we compare the performance of Coda, AFS, and Fluid Replication in a variety of networking conditions. For Coda, we ran Coda 5.3.13 at the server and the client. For AFS, we used OpenAFS, a descendant of AFS 3.6, as the server, and Arla 0.35.3 as the client. To provide a fair comparison, we set up our Coda volume on a single server, rather than a replicated set.

We have employed a modified version of the Andrew Benchmark [20]. Andrew Benchmark consists of five phases: MakeDir, Copy, ScanDir, ReadAll, and Make. Our benchmark is identical to the Andrew Benchmark in form; the only difference is that we use the *gnuchess* source tree rather than the original source tree. Gnuchess is 483 KB in size; when compiled, the total tree occupies 866 KB. We pre-configure the source tree for the benchmark, since the configuration step does not involve appreciable traffic in the test file system. Since the Andrew Benchmark is not I/O-bound, it will tend to understate the difference between alternatives. In the face of this understatement, Fluid Replication still outperforms the alternatives substantially across wide-area networks.

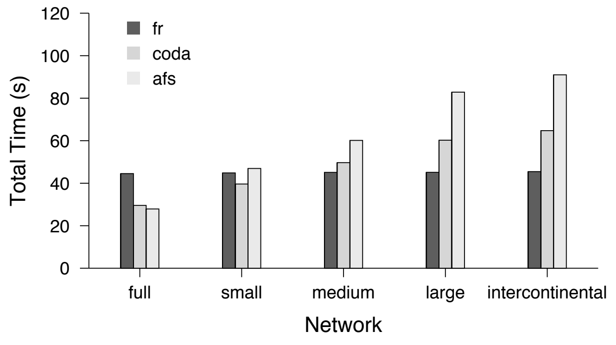
We tested each file system with both cold and warm caches. A "cold cache" means that the source tree is not at clients' cache; it is at the server. In the case of AFS and Coda, a "warm cache" means that the clients already hold valid copies of the *gnuchess* source tree. In the case of Fluid Replication, the source tree is cached on the WayStation.

Figures 5.3 compares the total running times of the Fluid Replication, Coda, and AFS clients under different network environments. Figure 5.3(a) gives the performance with a cold cache, and Figure 5.3(b) shows it with a warm cache. Each experiment comprises five trials, and the standard deviations are less than 2% of the mean in all cases.

With a cold cache and a well-connected server, Coda and AFS outperform Fluid Replication. We believe this is due to our choice of Java as an implementation language and



(a) cold cache



(b) warm cache

This figure compares the total running times of Coda, AFS and Fluid Replication under a variety of network environments. The upper figure gives the results for a cold cache, and the lower figure shows them for a warm cache. With a cold cache, Fluid Replication is least affected by network costs among three systems. With a warm cache, the performance of Fluid Replication does not appreciably degrade as network costs increase.

Figure 5.3: Overall Client Performance

the general maturity level of our code. We see no fundamental reason why Fluid Replication's performance could not equal Coda's. They have identical client architectures, and the client/WayStation interactions in Fluid Replication are similar to those between client and server in Coda.

As the network conditions degrade, the cold-cache times of Coda and AFS rapidly increase while those of Fluid Replication increase slowly. All systems must fetch source tree objects from the server, and should pay similar costs to do so. The divergence is due to the systems' different handling of updates. In Fluid Replication, all updates go only to the nearby WayStation. In Coda and AFS, however, updates must go across the wide area to the server.

With a warm cache, the total running time of Fluid Replication remains nearly constant across all network environments. This is because the updates are propagated only to the WayStation; invalidations are sent to the server asynchronously. The running time of AFS and Coda increases as the network degrades. They must propagate updates to the server during the *copy* phase and write object files back to the server during the *make* phase. The Coda client never entered weakly-connected mode, since the bandwidths in our sample traces were well above its threshold of 50 KB/s. Had Coda entered weakly-connected mode, its performance would be identical to Fluid Replication's, but its updates would be neither safe nor visible for many minutes.

Table 5.3 shows the normalized running time of Fluid Replication, with the local-area case serving as the baseline. While the running time increases by as much as 24.1% when the cache is cold, it remains nearly constant when the cache is warm. Looking only at the four scenarios generated by `ping` and `ftp` experiments, there appears to be a slight growth trend, but it is within observed variance. Only the results for Small and Intercontinental are not identical under the t-test; all other pairs are statistically indistinguishable. To conclusively rule out a true trend, we also ran the warm-cache Fluid Replication experiment across two more-demanding networking scenarios. The first decreased bandwidths to 56 Kb/s, but added no additional latency. The second increased one-way latency to 200 ms, but placed no additional bandwidth constraints. In both cases, Fluid Replication's running

Trace	Cold	Warm
Small	1.040	1.007
Medium	1.103	1.012
Large	1.200	1.013
Intercontinental	1.241	1.020
Low Bandwidth		1.014
High Latency		1.007

This figure shows the normalized running time for Fluid Replication over each of the networking scenarios in Table 5.2. The local-area case served as the performance baseline in each case. When the WayStation cache is cold, performance decreases as wide-area networking costs increase. However, no consistent trend exists when the WayStation cache is warm.

Table 5.3: Fluid Replication over Wide-Area Networks

time was less than that for the Intercontinental trace. Therefore, we conclude that that Fluid Replication's update performance does not depend on wide-area connectivity. Of course, this is due to deferring the work of update propagation and the final reconciliation—neither of which contribute to client-perceived performance. Sections 5.3.4 and 5.3.5 quantify any reduction in consistency or availability due to deferring work.

5.3.2 Costs of Sharing

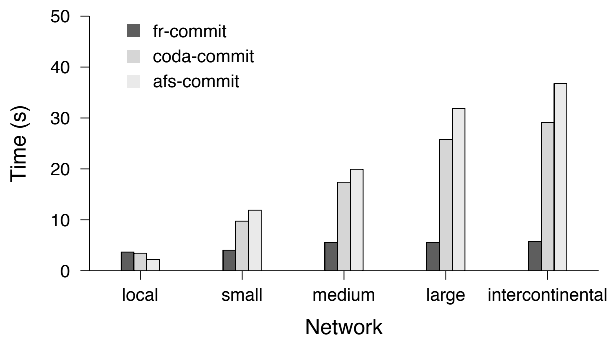
Our next task is to assess the potential impact that deferred propagation has on sharing between wide-area clients. We devised a benchmark involving two clients sharing source code through a CVS repository. In the benchmark, clients C1 and C2 are attached to WayStations W1 and W2, respectively. In the first phase, C1 and C2 each check out a copy of the source tree from the repository. After changing several files, C1 commits those changes to the repository. Finally, C2 updates its source tree. Both the repository and working copies reside in the distributed file system. When the benchmark begins, both clients have the repository cached.

We used the Fluid Replication source tree for this benchmark. The changes made by C1 are the updates made to our source tree during the four of the busiest days recorded in our CVS log. At the beginning of the trial, the source tree consists of 333 files totaling 1.91 MB. The committed changes add four new files, totaling seven KB, and modify 13 others, which total 246 KB. We re-ran this activity over the two-client topology of Figure 5.2. Figure 5.4 shows our results for the commit and update phases, run over Fluid Replication, AFS, and Coda; each bar is the average of five trials. We do not report the results of the checkout and edit phases, because they exhibit very little sharing.

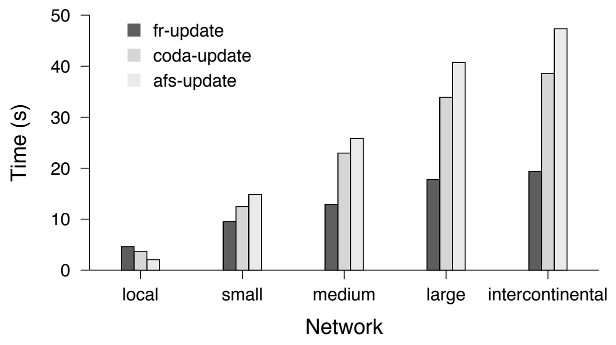
Unsurprisingly, the cost of committing to the repository was greater for AFS and Coda than for Fluid Replication. This is because Fluid Replication clients ship updates to a nearby WayStation. AFS and Coda must ship data and break callbacks over the wide area.

What is surprising is that the update phase is also more costly for AFS and Coda. One would think that Fluid Replication would perform poorly, since file data has to traverse two wide area paths: from the first WayStation to the server, and then to the second WayStation. The unexpected cost incurred by AFS and Coda stems from the creation of temporary files. The latter are used to lock the repository and back up repository files in case of failure. WayStations are able to absorb these updates, and later optimize them away, since they are all subject to cancellation. AFS and Coda clients, on the other hand, send every update to the server. Furthermore, these updates break callbacks on the far client.

It is important to note that hiding the creation of temporary locking files may improve performance, but it renders the intended safety guarantees useless. The best that Fluid Replication can offer in the face of concurrent updates is to mark the shared file in conflict;



(a) commit



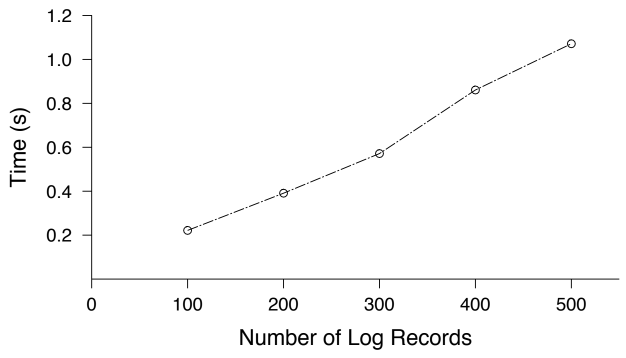
(b) update

Our sharing benchmark replayed sharing from the CVS log of our Fluid Replication source tree. The benchmark consisted of five phases: a checkout at C1, a checkout at C2, an edit at C1, a commit at C1, and an update at C2. This figure shows the time to complete the commit and update phases for AFS, Coda, and Fluid Replication.

Figure 5.4: Sharing Benchmark

the users must resolve it by hand. Coda would be faced with the same dilemma if it had entered weakly-connected mode during this benchmark. However, we believe that in practice, optimism is warranted. Even in the case of CVS repositories, true concurrent updates are rare. Our own logs show that commits by different users within one minute occurred only once in over 2,100 commits. A commit followed by an update by different users within one minute happened twice.

This benchmark illustrates the cost of obtaining deferred updates. However, in prac-



This figure shows the reconciliation times spent at a server as the number of log records varies. These times determine the number of WayStations a server can handle.

Figure 5.5: Reconciliation Time at Sever

tice, these files are likely to have migrated to the server. Our logs show that the median time between commits by different users was 2.9 hours, and that the median time between a commit and an update by different users was 1.9 hours. This would provide ample opportunity for a WayStation to asynchronously propagate shared data back to the server before it is needed.

5.3.3 Reconciliation Costs

To be successful, Fluid Replication must impose only modest reconciliation costs. If reconciliations are expensive, WayStations would be able to reconcile only infrequently, and servers could support only a handful of WayStations. To quantify these costs, we measured reconciliations, varying the number of log records from 100 to 500. To put these sizes in context, our modified Andrew Benchmark reconciled as many as 148 log records in a single reconciliation.

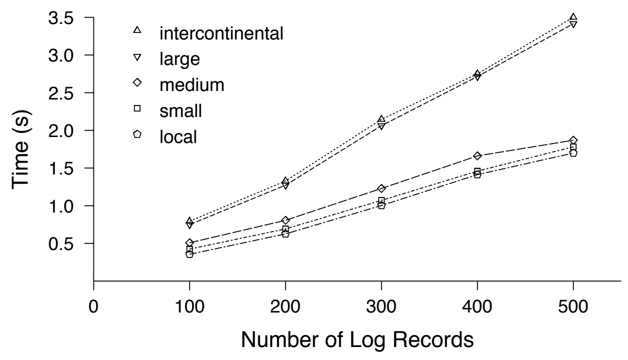
Figure 5.5 shows the reconciliation time spent at the server as the number of log records varies; each point is the average of five trials. This time determines the number of WayStations a server can handle in the worst case. Server-side time increases to just under 1.1 seconds for 500 log records. In practice we expect the costs to be much smaller. The week-long NFS trace would never have generated a reconciliation with more than 64 log records.

Figure 5.6 shows the total time for a reconciliation measured at a WayStation as the number of log records varies. This includes the server-side time, RMI overheads, and network costs. The total reconciliation times for the local area, small distance, and medium distance traces do not vary significantly. This means that, at these speeds, bandwidth is not the limiting factor. Profiling of the reconciliation process suggests that RMI—even with our hand-optimized signatures—is the rate-limiting step. However, at 1 Mb/s, the bandwidth of the large distance and intercontinental traces, the bottleneck shifts to networking costs. In any event, the reconciliation times are much shorter than our default reconciliation period of 15 seconds, allowing the WayStation to reconcile more frequently if sharing patterns warrant.

5.3.4 Consistency

The consistency offered by Fluid Replication depends on two factors: the frequency of reconciliation and the time between uses of a shared object. Section 5.3.3 quantified the former. In this section, we address the incidence of sharing observed in a real workload.

To determine how often sharing happens and the time between shared references, we examined our week-long NFS client traces. For this analysis, we removed references to user mail spools from our traces. A popular mail client in our environment uses NFS rather than IMAP for mail manipulation. Many users run these clients on more than one

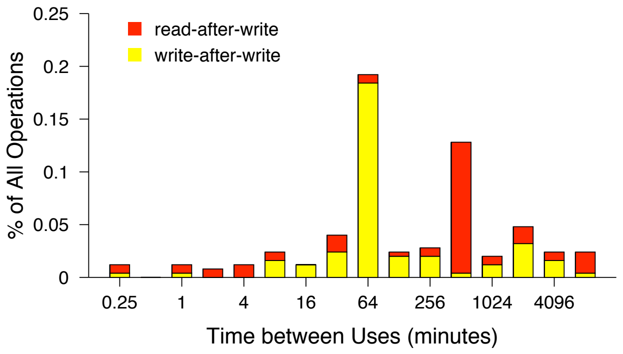


This figure shows the total reconciliation times in seconds as the number of log records varies. These times determine the upper limit on how frequently a WayStation can reconcile with the server. For all sizes, they are much shorter than our default reconciliation period of 15 seconds.

Figure 5.6: Reconciliation Time at WayStation

machine, despite NFS's lack of consistency guarantees [59], generating spurious shared references. Since we would expect mobile users to use IMAP instead, we excluded these references.

Figure 5.7 shows the percentage of all references to objects written previously at another replica site. The top part of each bar shows read-after-write sharing, and the bottom part shows write-after-write. As expected, sharing is not common, especially over short periods of time. Only 0.01% of all operations caused sharing within 15 seconds. The total fraction of references that exhibited sharing during the week was just over 0.6% of all operations. Note that these numbers are pessimistic, as we have assumed that each client uses a distinct WayStation. The graph shows some interesting periodic behavior; unfortunately, with network-level traces, we are unable to identify the processes causing it.



This figure shows the percentage of operations that caused sharing. The x-axis shows the time between uses of a shared objects in minutes, and the y-axis shows the percentage of total operations that exhibited sharing. The top part of bar shows the read-after-write sharing; the bottom part shows the write-after-write. Only 0.01% of all operations caused sharing within 15 seconds. Just over 0.6% of all operations exhibited any form of sharing.

Figure 5.7: Sharing

5.3.5 Availability

Because WayStations do not propagate file contents to the server immediately, a failed WayStation could keep a client from retrieving a needed update. To gauge how likely such a scenario might be, we fed our NFS traces into a Fluid Replication simulator. We augmented the trace with WayStation reconciliations and failure/recovery pairs. Reconciliations were scheduled every 15 seconds from the WayStation's first appearance in the trace. We assume a mean time to failure of 30 days and a mean time to repair of one hour, both exponentially distributed. These parameters were chosen to represent typical uptimes for carefully administered server machines; they are the same as those used in the xFS study [11]. Note that we did not model server failures; our intent is to explore Fluid Replication's *additional* contribution to failures.

For each trial of the simulation, we varied the random seed controlling failures and repairs. We then counted the number of back-fetches that fail due to the unavailability of a WayStation. We took over five million trials of the experiment in order to provide reasonable confidence intervals for a very small mean.

Recall that during our trace there were 24,957 total requests made of the WayStations. Of those, 243 requests required a back-fetch from one WayStation to another. We first simulated Fluid Replication when updates that had not been overwritten or back-fetched were propagated by the WayStation after 1 hour. Across five million trials, 187,376 requests failed, for an average of 1.46×10^{-6} failures per operation, with a 90% confidence interval of $\pm 8.83 \times 10^{-7}$; this is equivalent to five nines' availability. Expressed in time, we observed an average of 0.037 failures per week, or roughly one failed access every 27 weeks for our client population.

5.4 Summary

As mobile clients travel, their costs to reach back to home filing services change. To mask these performance problems, current file systems reduce either *safety*, *visibility*, or both. This is a result of conflating safety and visibility into a single mechanism. They have different requirements, and so should be provided through different mechanisms.

Fluid Replication separates the concerns of safety and visibility. While traveling, a mobile client associates itself with a nearby *WayStation* that provides short-term replication services for the client's home file system. Updates are sent to the nearby WayStation for safety, while WayStations and servers frequently exchange knowledge of updates through *reconciliation* to provide visibility. Reconciliation is inexpensive and wait-free.

An analysis of traffic in a production NFS server validates our design decisions. A modest reconciliation interval of fifteen seconds limits the rate of stale reads or conflicting writes to 0.01%. Measurements of a Fluid Replication prototype show that it isolates clients from most wide-area networking costs. Update traffic is not affected at bandwidths as low as 56 Kb/s or latencies as high as 200 ms. These gains offset increased sharing costs, even for workloads with substantial degrees of sharing. Despite the fact that update propagation is deferred, availability does not suffer. Our traced clients could expect five nines' availability.

CHAPTER 6

Data Management

6.1 Introduction

Fluid Replication has adopted optimistic concurrency control; copies of files are replicated across many WayStations and any copy can be updated at any time. While this allows the possibility of conflicting updates, Chapter 5 shows that conflicts are so rare that the infrequent cost of merging versions is repaid by gains in performance and availability.

When an optimistic update is made at a replica site, that site must eventually inform others of the new version's existence for consistency. However, it is likely that a file written by a particular user will be read only by that user. In other words, sharing is rare in file system workloads. Consequently, while the meta-data *describing* an update needs to be propagated, the data *comprising* that update often need not be.

This observation suggests an obvious optimization: avoid propagating contents of updates whenever possible. Replica sites must then move data only for administrative reasons—such as backup—or in instances of true sharing. This drastically reduces data movement compared to aggressive schemes, charging the cost of sharing to those who require it.

Unfortunately, these costs can mount quickly. In a wide-area deployment, there can be substantial latency to fetch a file from the replica on which it was last updated. Given the instabilities seen across paths with large bandwidth-delay products [48], this can have profound effects in cases of active sharing.

These concerns raise the question of when to propagate data objects. We will show that propagating aggressively wastes server resources, while propagating only on-demand unduly penalizes clients who share. In this chapter, we explore three alternative mechanisms for deciding when to propagate an update, and their impact on Fluid Replication.

6.2 Simple Propagation Schemes

To put our propagation schemes in context, we first describe three naive attempts at deciding when to ship updates. The first, *write-through*, aggressively ships all updates as they are made. The second, *on-demand*, ships them only when they are needed elsewhere. The third, *periodic*, batches updates and ships them every t minutes.

A simple way to propagate updates is to send them as soon as they are made. We call this the write-through scheme. Using this scheme, we get the upper bound for the amount of data propagated from WayStations to the server. Because all updates are at the server, no file is back-fetched. The problem with this scheme is that many updates are propagated unnecessarily, increasing server load and network traffic.

At the other end of the spectrum, updates are propagated only when they are needed elsewhere. This is called the on-demand scheme. Using this scheme, we get the lower bound for the amount of data propagated, because files are shipped only when necessary. The problem with this scheme is that clients may experience slow response time when requested files are at remote WayStations.

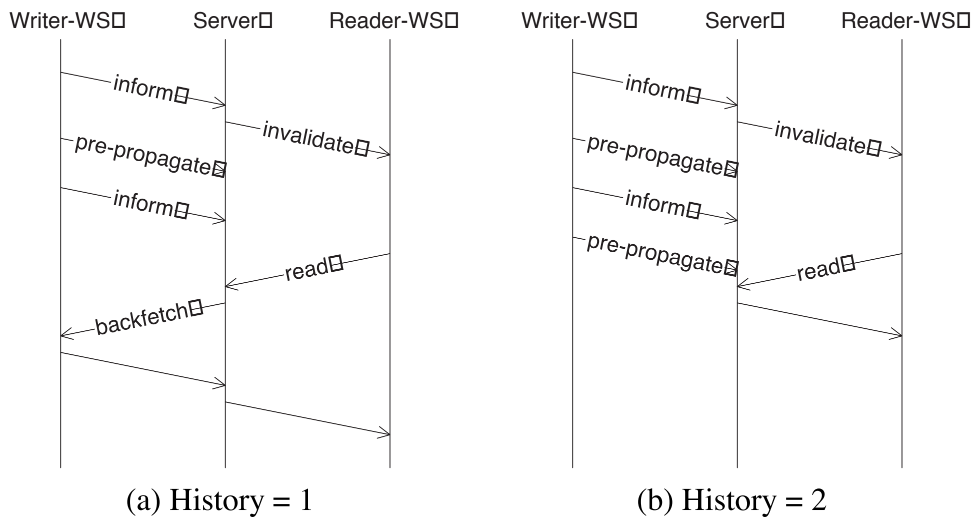
A simple improvement to the write-through scheme is to defer propagation, batching updates together and sending them periodically. This scheme is used in many systems, including an early Fluid Replication prototype [32]. Periodic schemes with long periods can ship substantially less data than the write-through scheme. This is because writes are bursty [75]—they tend to be clustered together in time. If an update is made but not shipped, a later update can overwrite it. So deferring shipment can lead to savings. Unfortunately, these periodic schemes are still not very precise, because they propagate files indiscriminately; many of the propagated files are never accessed by other clients.

6.3 Making Propagation Decisions

The on-demand scheme gives rise to back-fetches, while the write-through scheme propagates files needlessly. The periodic schemes reduce this, but are either still too aggressive or not able to eliminate most back-fetches. To address these problems, we introduce three new propagation schemes: Invalidation Heuristic, Time to Next Use, and Adaptive Time Delay.

6.3.1 Invalidation Heuristic Scheme

File accesses tend to have strong temporal locality. If files were shared in the recent past, they are likely to be shared in the near future. Based on this observation, updates should be propagated if the file is shared. When a WayStation informs the server of its updated files, the server invalidates copies at other WayStations. Updates that cause invalidations are propagated to the server immediately, because they are likely to be read again



This figure shows a reader accessing a file after a writer updates it twice. With history of 1, a back-fetch occurs since only the first update to the file is propagated. With history of 2, a back-fetch is avoided since the first two updates are propagated.

Figure 6.1: Example of IH Scheme

where they have been invalidated. We call this the *invalidation-heuristic* (IH) scheme. Note that this scheme requires no extra state at the server; the list of files to be propagated can be computed from the state used to issue invalidations.

This simple approach has a drawback. Consider a scenario in which files are always written twice before other nodes access them. When the WayStation informs the server of its first write, the server invalidates all the other copies. At this point, the file is no longer shared. Therefore, only the first write of two is propagated, and every read requires back-fetching. Figure 6.1(a) illustrates this scenario.

To solve this problem, we keep the same sharing status for a fixed number of bursty writes. This number is called *history* and determines how many writes of each burst are propagated to the server. The *burst* counter is reset when another node reads the file or writes to it. We considered *history* values from one to four in our evaluation. Figure 6.1(b)

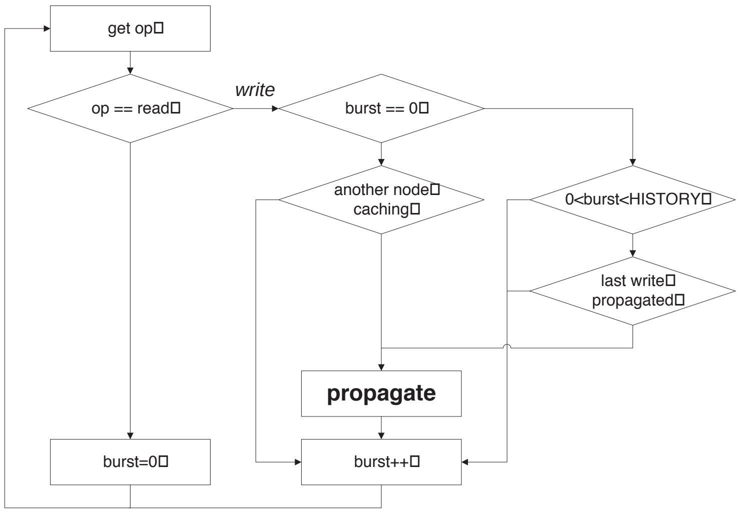


Figure 6.2: Decision Making in the IH Scheme

shows what happens in the above scenario when history is set to two. Compared to Figure 6.1(a) where history is one, the server need not back-fetch the update, and the read request is served faster. Figure 6.2 summarizes the decision making process.

Another optimization is to consider spatial locality. If many files are back-fetched from a particular WayStation, the server should prefetch other files that have been updated then, but not yet propagated. This mechanism is especially effective for producer-consumer relationships; one node has updated many files, and others read them later. More specifically, pending writes are prefetched if the number of back-fetches to the WayStation, b, excesses the number of writes overwritten by the WayStation, o, by a certain limit, L:

$$b - o > L$$

When this condition is met for a WayStation, pending writes that happened within the last 30 minutes are propagated to the server. We call this modified scheme *invalidation-heuristic-with-spatial-locality* (IHSL).

6.3.2 Time to Next Use Scheme

Rather than using past sharing to predict future sharing, one can use the times of recent accesses to predict the next time that a read or write might occur. The time-to-next-use (TTNU) scheme decides whether to propagate or not based on the past history of file access patterns. The basic idea is that a modified file should be propagated if the expected time to the next read by another node, E_r, is earlier than the expected time to the next write by the current node, E_w. The propagation decision, D, is defined as follows:

$$D = \begin{cases} \text{propagate} & if\ E_r < E_w \\ \text{do not propagate} & otherwise \end{cases} \tag{6.1}$$

The expected next access times are computed based on the past access pattern using exponentially weighted moving average (EWMA) filters. EWMA is the weighted sum of the previous estimate and the new value. We keep two separate filters for the read estimate, E_r and the write estimate, E_w:

$$E_r(t+1) = \alpha E_r(t) + (1-\alpha)n_r(t) \tag{6.2}$$

$$E_w(t+1) = \alpha E_w(t) + (1-\alpha)n_w(t) \tag{6.3}$$

where α is the weight and n is the new value. We explored a variety of values for α in our evaluation.

For reads, we update the estimate when a read by a node other than the last writer arrives, and use the time passed since the last write as the new value (n_r). For writes, we update the estimate when a write is overwritten by the same node using the time between consecutive writes as the new value (n_w). The estimate for writes is reset when the writer changes.

6.3.3 Adaptive Time Delay Scheme

Static periodic schemes can be too aggressive, not effective, or both. This is because a single, static period is not capable of capturing the different access patterns between files and over time. The *adaptive-time-delay* (ATD) scheme is based on the periodic scheme; it maintains a period estimate for each file, and adjusts this estimate based on the utility of past propagation decisions. If we can adjust the propagation period of each file based on the WayStation operations on it, we expect to get better performance.

It is often hard to predict when a user will start or stop using a file, but once a user has started using the file, she is likely to use it for awhile. Based on this, we expect that when a WayStation reads or writes a file, it will do so again in near future. So, if we adjust our propagation scheme based on current activities, this scheme should perform well.

Because most files are never shared, files are given an initial period of infinity. So, the first shared access will result in a mandatory back-fetch. On this back-fetch, the period of sharing is estimated to be just less than the interval between the last update and this back-fetch.

If a file scheduled for propagation is updated again before the timer expires, it is reset. If a file is propagated, but not used before it is overwritten, the period estimate is doubled. If a file is back-fetched before its period estimate expires, the estimate is halved. By observing the benefit (or lack thereof) of prior propagation decisions, this feedback controller produces accurate, per-file estimates of need.

6.4 Evaluation

6.4.1 Trace Collection

To understand how propagation schemes work in a real environment, we use two sets of traces, one from the University of Michigan (UM) and another from Harvard University. The first set of traces was collected on an NFS server in our department for three months (April - June). This machine serves the home directories for 86 users; they occupied 35.7 GB as of August 12, 2002. Using `tcpdump`, we collected NFS requests to the server and the corresponding responses. Then, we used Blaze's `nfstrace` [4] to convert the traces to the user level file system commands (reads and writes). The second set of traces comes from the EECS department at Harvard University [14]. They are taken from September to November of 2001. The NFS server is the primary file system for the department. Because traces are not available for all of these days, we use two consecutive weeks from each month to evaluate the propagation schemes.

The traced machines were generally not mobile clients, and may not provide a workload matching our intended client population. However, prior studies suggest that, at the operation level captured by our traces, mobile and desktop behavior are remarkably similar [55].

Table 6.1 shows the number of reads and writes, the number of distinct machines that accessed the NFS server in question, and the number of files accessed during each month. The number of writes are also the number of propagations if we use the write-through scheme. Note that for the UM traces, the number of different machines that accessed the NFS server during each month is bigger than the number of user accounts which have

Month	Reads	Writes	Machines	Files
UM Trace				
4	95,427	301,905	197	98,588
5	88,295	211,058	199	96,687
6	180,448	306,233	197	57,793
Harvard Trace				
9	439,256	895,804	117	276,494
10	434,259	945,294	108	285,848
11	254,019	854,775	142	312,049

Table 6.1: Summary of Traces

home directories at the server. This is likely due to users using different machines from one session to another.

In applying these traces, we assume that each client machine is in a different location, and uses a different WayStation. This will tend to over-emphasize the costs of wide-area operations, but is a conservative choice. A discrete event simulator replayed the traces, counting updates, reads, propagations, and back-fetches. One can apply this simulator to different propagation policies, measuring their potential gain.

Using these file system traces, we set out to answer the following questions:

- What percentage of writes are read by others?
- How much does a client suffer from sharing when the on-demand scheme is used?
- What is distribution of sharing over machines?
- For each scheme, how many files are propagated compared to write-through?
- For each scheme, how many files are back-fetched compared to on-demand?
- How effective is each scheme in terms of precision and recall?
- Which scheme produces the best overall performance?

Month	Shared	Back-fetch	Back-fetch to writes
UM Trace			
4	1,223	2,438	0.81%
5	463	1,142	0.54%
6	738	1,648	0.54%
Harvard Trace			
9	12,273	34,595	3.86%
10	6,808	29,419	3.11%
11	3,477	23,361	2.73%

Table 6.2: Back-fetch

6.4.2 Aggressive Propagations

Table 6.2 shows the back-fetches when the on-demand scheme is used. There are two things to note from this data. First, the number of different files that are back-fetched is smaller than the number of back-fetches. From this, we can infer that the files that are once shared are likely to be shared again. Second, less than 4% of writes are read by others. So propagating all updates introduces unnecessary network traffic between the server and WayStations. In short, the write-through scheme is too aggressive.

6.4.3 Passive Propagations

Our goal for propagation schemes is to help those clients who suffer under passive propagations due to sharing. Before exploring new schemes, we need to consider whether clients actually suffer or not with the on-demand scheme. To answer this, we chose one hour with substantial sharing from the Harvard traces and one from the UM traces, and measured the time a client spends to fetch updated files from other nodes.

The network condition between the server and WayStations are slow (latency of 82.15 ms and bandwidth of 3 Mb/s), while that between WayStations and clients are fast (100 Mb/s). The WayStations are connected to the server via a *trace modulated* network. Trace mod-

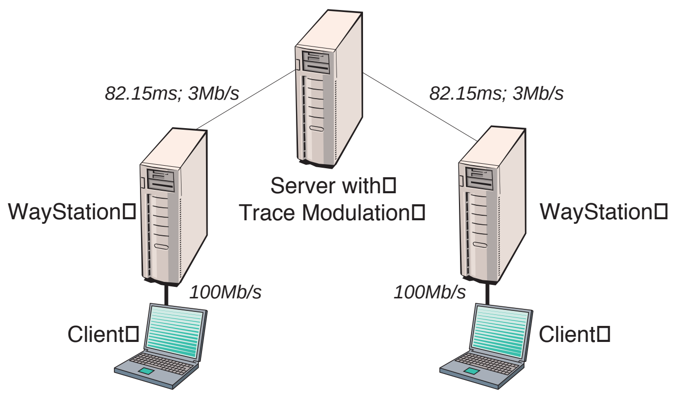


Figure 6.3: Setup for Passive Propagation Experiment

ulation performs emulation of a slower network over a LAN [57]. To get the latency and bandwidth parameters, we ran `ping` from a machine in our department to a machine at Hewlett Packard with different packet sizes. The average round trip time with packet size of 20 MB was 217.9 ms; the time with packet of eight bytes was 164.3 ms. We used half of 164.3 ms, which is 82.15 ms, as the one-way latency. This gives us bandwidth of 3 Mb/s using the equation $delay = latency + size/bandwidth$ [5]. Figure 6.3 shows the experiment setup.

Using the traces, we measured the time that one client spends to fetch files. The updated file data is never sent to the server unless other nodes request it, causing back-fetches. The goal of this experiment is to answer the question: What percentage of time is spent to back-fetch files from other WayStations?

Table 6.3 shows the time with a cold cache; no files are cached at the WayStation. *Read-hit* shows the time to fetch files cached locally. *Fetch-from-server* shows the time to fetch files that are at the server. *Back-fetch* shows the time to fetch files that require back-fetching from the writer, because the server does not have the up-to-date copy. Back-fetch time is thus the time to ship a file from a remote WayStation to the server, plus the

	Count	Size(MB)	Time(s)
UM Trace			
Read hit	0	0	0
Fetch from server	5	0.0287	2.2
Back-fetch	138	0.2132	197.8
Total	143	0.2419	200.0
Back-fetch to total			98.9%
Harvard Trace			
Read hit	487	0.8761	0.077
Fetch from server	1607	1.0762	696.5
Back-fetch	1808	30.2044	2382.1
Total	3092	32.1567	3078.7
Back-fetch to total			77.4%

Table 6.3: Total Fetch and Back-fetch Time

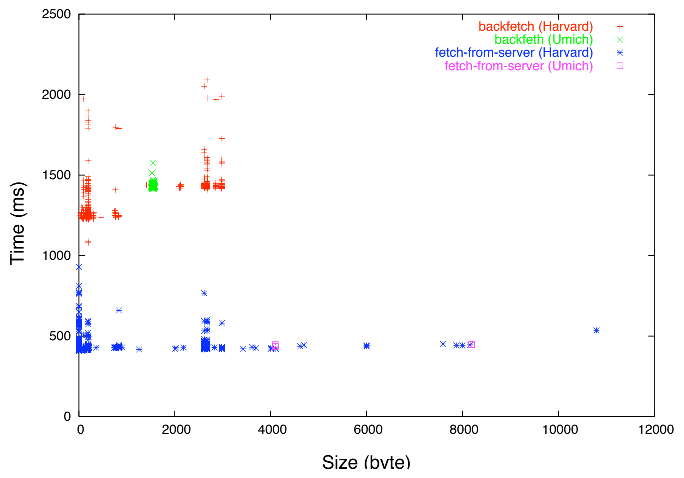
time to ship the file to the local WayStation, and finally to the client. In the UM trace, the client spent 98.9% of total fetch time waiting for files that needed to be back-fetched from writers to the server. It spent 77.4% for the Harvard trace. We can predict that the ratio of back-fetch to total time will be even higher for a warm cache. So the on-demand scheme penalizes the clients who share.

Figure 6.4 shows the fetch and back-fetch times measured at the client for individual files. Back-fetching takes about three times longer than fetching from the server, for files of the same size.

6.4.4 Burstiness

Figure 6.5 shows the distribution of back-fetches over machines. The x-axis shows the number of machines; the y-axis shows the number of back-fetches each machine experienced. The machines that did not experience any back-fetches are excluded from these graphs.

The back-fetches are not distributed evenly over all machines. They are highly concen-



This figure shows the fetch and back-fetch time measured at the client for individual file under latency of 82.15 ms and bandwidth of 3 Mb/s.

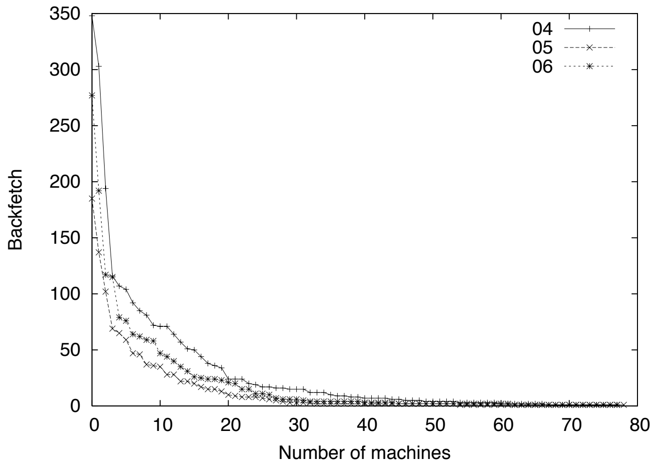
Figure 6.4: Fetch and Back-fetch Time for Individual File

trated among a few machines. For the UM trace, the top three machines are responsible for more than one third of all back-fetches: 34.7%, 37.1% and 35.6% for the months of April, May, and June, respectively. For the Harvard trace, the top three are responsible for more than one fourth: 29.3%, 26.9%, and 30.4% for the months of September, October, and November, respectively. Thus, most machines experience no or few back-fetches, while some machines experience many.

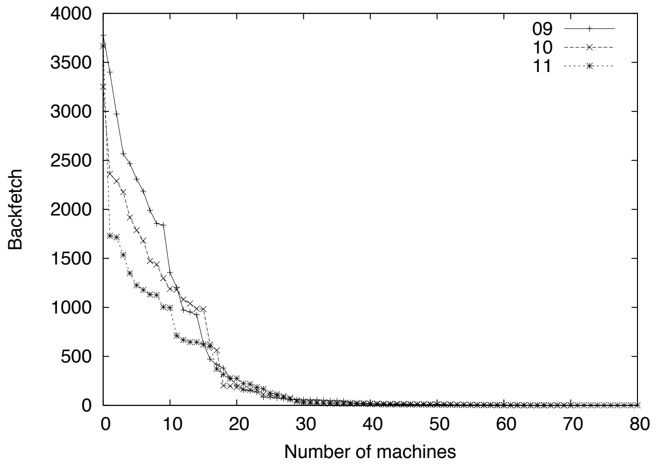
6.4.5 Files Propagated and Back-fetched

We have built a simulator to compare the different propagation schemes. The simulator takes our traces as input and returns a summary of data transferred between the server and WayStations.

To evaluate the family of periodic schemes, we subjected several of them to our traces. We tried periods of 15 seconds, 1 minute, 1 hour, and 1 day. Figure 6.6 and Figure 6.7 show the percentage of propagations normalized over the write-through scheme, and the



(a) UM



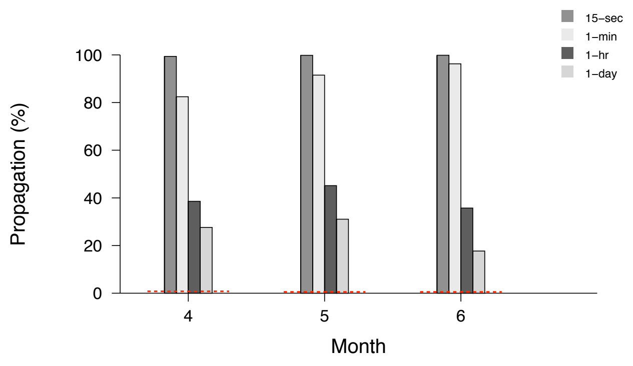
(b) Harvard

This figure shows the distributions of back-fetches over machines. The x-axis shows the number of machines; the y-axis shows the number of back-fetches each machine experienced.

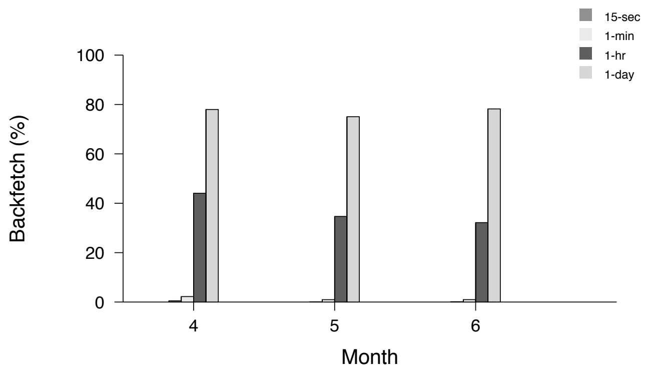
Figure 6.5: Back-fetch Distribution

percentage of back-fetches normalized over the on-demand scheme. Note that the propagations include both propagations done before the data is actually needed elsewhere (pre-propagations) and propagations caused by back-fetches. The horizontal dotted lines in Figure 6.6(a) and Figure 6.7(a) represent the values for the on-demand scheme that serve as the lower bounds.

The periodic-15-second scheme propagates only slightly less than the write-through



(a) Propagations



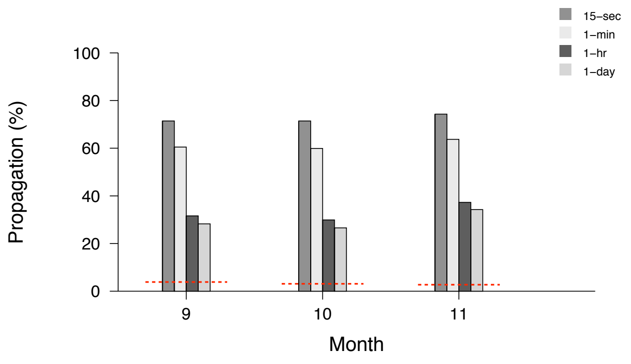
(b) Back-fetches

This figure shows the percentage of propagations normalized over the write-through scheme, and the percentage of back-fetches normalized over the on-demand scheme. Updates are propagated every 15 seconds, 1 minute, 1 hour, and once a day. The dotted lines denote lower bounds.

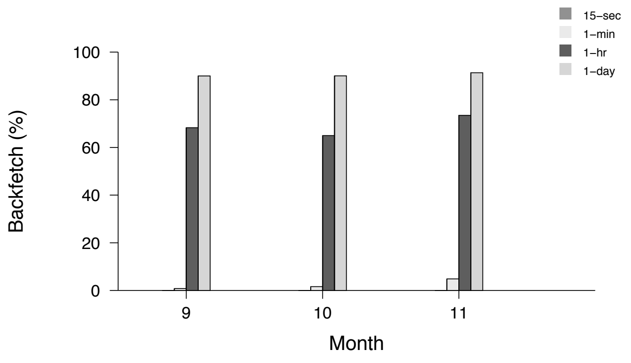
Figure 6.6: Performance of Periodic Scheme for the UM Trace

scheme for the UM trace; it propagated 99.7% of the write-through scheme on average. It does better for the Harvard trace, propagating 72.4%. The periodic-1-day scheme sends much less data, but causes almost as many back-fetches as the on-demand scheme. Specifically, this scheme propagates 25.5% and 29.7% on average for UM and Harvard traces respectively, but these are still much more than the on-demand scheme, which propagates 0.6% and 3.2%.

For IHSL, we tried the history from one to four. For TTNU, we tried 0.25, 0.5 and



(a) Propagations



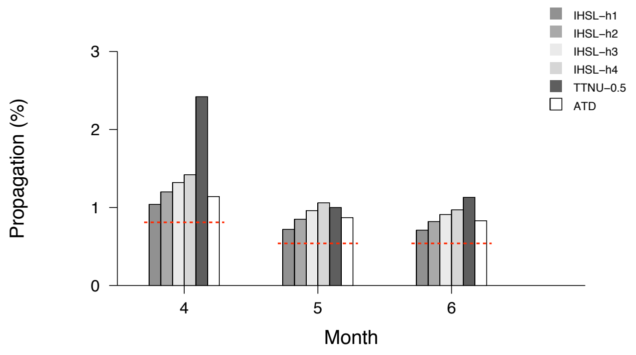
(b) Back-fetches

This figure shows the percentage of propagations normalized over the write-through scheme, and the percentage of back-fetches normalized over the on-demand scheme. Updates are propagated every 15 seconds, 1 minute, 1 hour, and once a day. The dotted lines denote lower bounds.

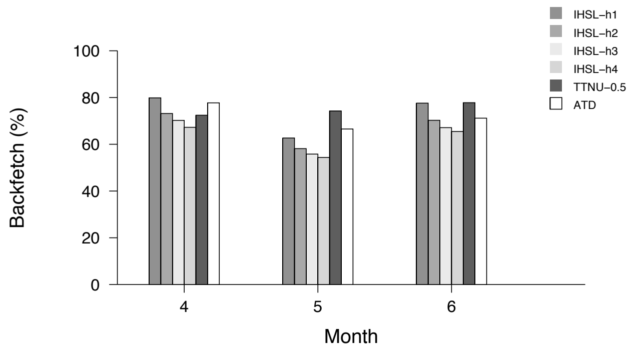
Figure 6.7: Performance of Periodic Scheme for the Harvard Trace

0.75 as weights of the EWMA filter. Because the different weights in this scheme did not make much difference and the scheme did not perform well, we included only the single weight of 0.5 in the results.

Figure 6.8 and Figure 6.9 shows total propagations and back-fetches during each month for our three new propagation schemes. IHSL, ATD and TTNU propagate less than 2.5% of the data shipped by the write-through scheme for the UM trace. For the Harvard trace, it is less than 17.2%. All of these schemes propagate less data than the 1-day periodic



(a) Propagations



(b) Back-fetches

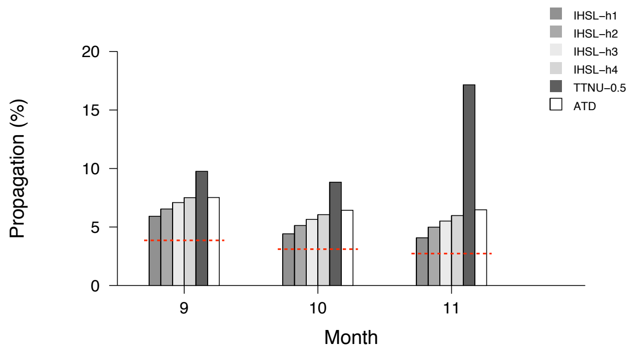
This figure shows the percentage of propagations to the write-through scheme and the percentage of back-fetches to the on-demand scheme. The horizontal dotted lines denote lower bounds.

Figure 6.8: Overall Performance for the UM Trace

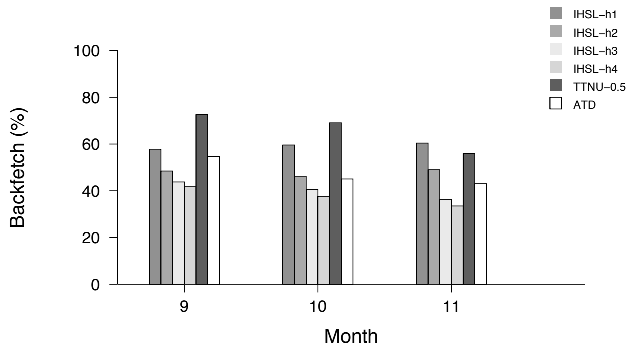
scheme, but leave fewer back-fetches to clients.

The on-demand scheme propagates the minimum amount of data—two orders of magnitude less than the write-through scheme—but penalizes clients with substantial sharing. In contrast, IHSL propagates less than twice that of the on-demand scheme for all months except for November, but reduces back-fetches by 43.5% on average. For November, the IHSL scheme with history of three and four propagated a little more than twice that of the on-demand scheme.

All three propagation schemes clearly outperformed the simple schemes (write-through,



(a) Propagations



(b) Back-fetches

This figure shows the percentage of propagations to the write-through scheme and the percentage of back-fetches to the on-demand scheme. The horizontal dotted lines denote lower bounds.

Figure 6.9: Overall Performance for the Harvard Trace

on-demand, and periodic), but it is hard to compare them to each other because schemes with less propagations often cause more back-fetches. How can we tell which scheme is better? In the following section, we introduce new metrics to compare our schemes.

6.4.6 Effectiveness

Our goal is to minimize both server load and client response time, but improving one often penalizes the other. For example, a scheme that propagates aggressively may reduce client response time, but increase server load. So it is not easy to tell which propagation scheme is better.

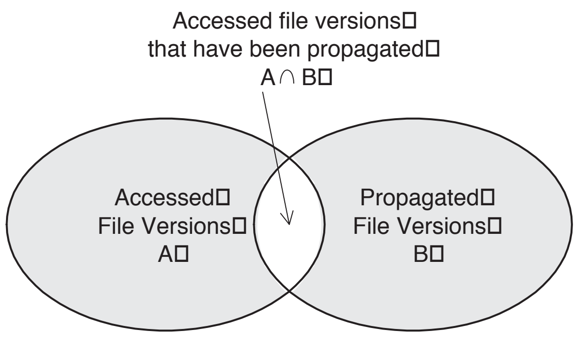


Figure 6.10: Precision-Recall Diagram

To measure effectiveness, we introduce two metrics, *precision* and *recall*, which are often used in information retrieval. In the context of information retrieval, precision is the fraction of the relevant documents that have been retrieved, and recall is the fraction of the retrieved documents that are relevant [1].

In our system, precision refers to the fraction of accessed file versions that have been pre-propagated, and recall refers to the fraction of pre-propagated files that are accessed. Note that we used term *version* to refer a file at a specific time. We excluded all the accesses that did not require propagations. In other words, we do not include reads to files that are never updated and reads to files that are only updated locally.

Let Set A be the accessed file versions that are modified elsewhere, and Set B be the file versions that are pre-propagated to the server. Figure 6.10 illustrates these basic sets. Then, precision (p) and recall (r) are defined as follows:

$$p = \frac{|A \cap B|}{|B|} \tag{6.4}$$

$$r = \frac{|A \cap B|}{|A|} \tag{6.5}$$

One way to combine precision and recall is the harmonic mean of two, called the *F-measure*. F-measure was introduced by van Rijsbergen [66], and is an established metric for information retrieval. The degree to which two sets do not match is defined as the *E-measure*. It is the shaded area in Figure 6.10. With normalization, it is written as:

$$E = \frac{|A \cup B - A \cap B|}{|A| + |B|} \tag{6.6}$$

In term of p and r, E is written as:

$$E = 1 - \frac{2rp}{r + p} \tag{6.7}$$

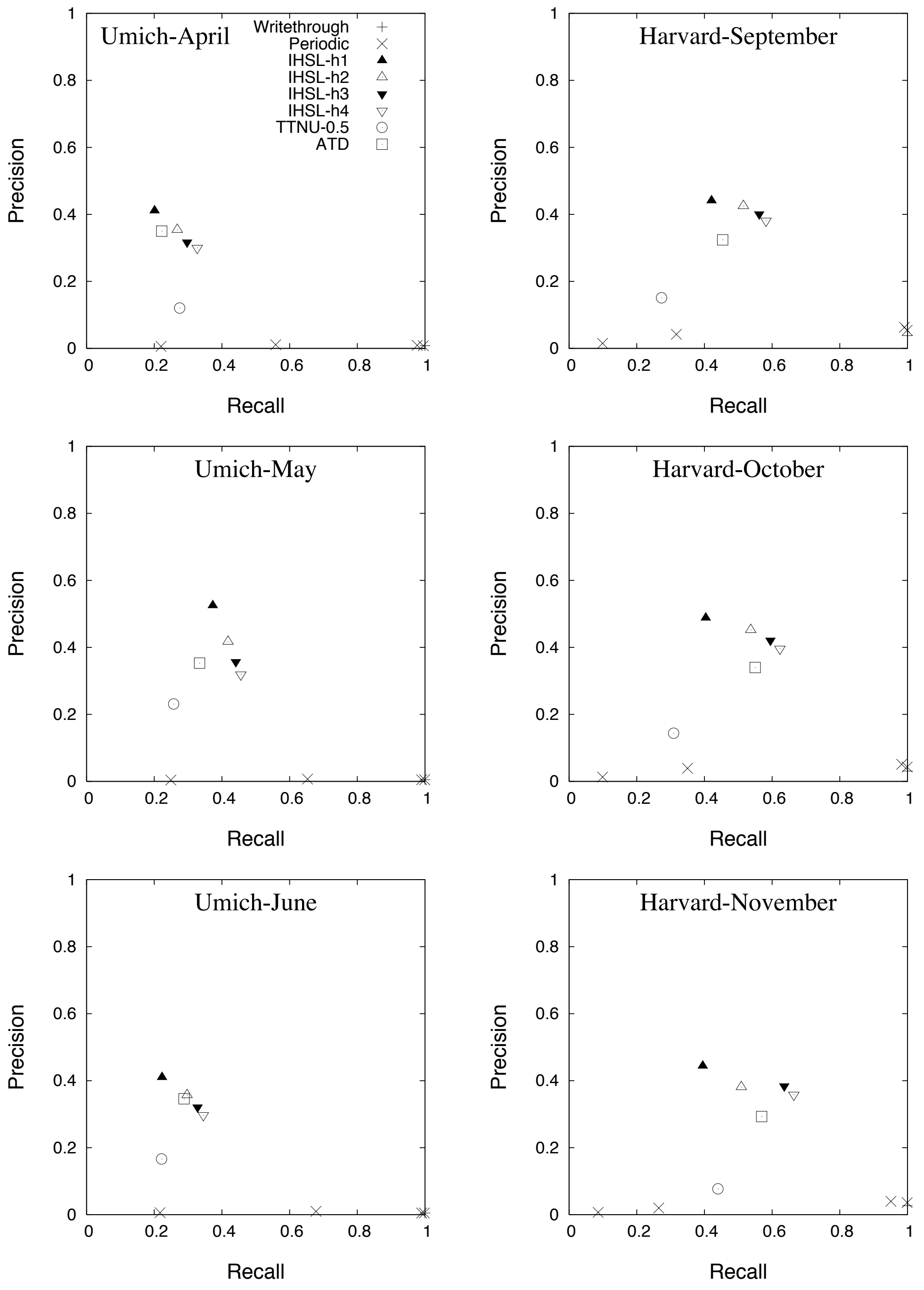
Then, F-measure is defined as $1 - E$. It denotes area of $A \cap B$ in Figure 6.10.

$$F = \frac{2rp}{r + p} \tag{6.8}$$

Figure 6.11 shows the precision-recall graphs for six months. These graphs include every scheme that we have considered except the on-demand scheme; for the on-demand scheme, precision is undefined because it does not propagate anything in advance.

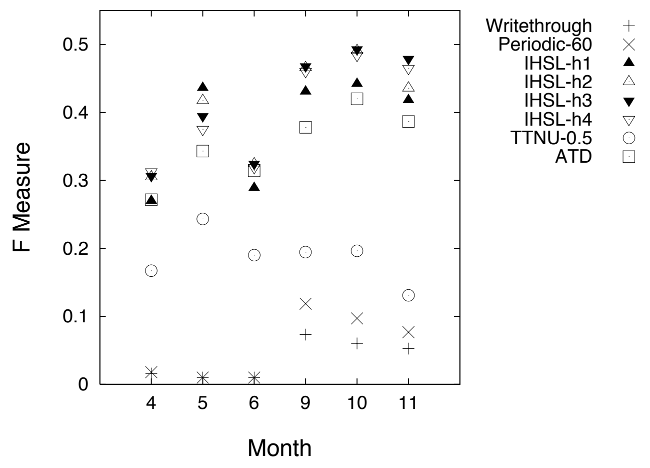
As expected, the write-through scheme has recall equal to one, but precision close to zero. Four points for the periodic scheme denotes period of 15 seconds, 1 minute, 1 hour, and 1 day. As the period increases recall decreases. The periodic scheme with 15-second period depicts results similar to the write-through scheme. The periodic scheme with 1-day period has both low precision and low recall, meaning it propagates many files without reducing the number of back-fetches compared to the on-demand scheme.

For IHSL, increasing history improves recall but reduces precision; it is more liberal in deciding which files to ship. IHSL provides better overall performance than our other adaptive schemes by detecting when sharing no longer holds. The ATD scheme performed second to the IHSL scheme. TTNU performed the worst of our three schemes.



This figure shows the precision-recall graphs. Four points for the periodic scheme denotes period of 15 seconds, 1 minute, 1 hour, and 1 day. As the period increases recall decreases. For the IHSL scheme, increasing history improves recall but reduces precision.

Figure 6.11: Precision and Recall



Months from 4 to 6 represent the UM trace; those from 9 to 11 denotes the Harvard trace.

Figure 6.12: F-Measure

Figure 6.12 shows the F-measure. For all months, the relative order of propagation schemes remains about the same. IHSL with histories of 2, 3 and 4 performed better than other schemes. Different history values in the IHSL scheme did not make much difference in performance, but histories of two and three worked slightly better than one and four. In short, our experiments show that the IHSL scheme with history value of two or three provides the best balance between recall and precision.

6.5 Summary

Wide-area file systems benefit greatly from optimistic concurrency. The low incidence of sharing allows such systems to simplify their consistency mechanisms, with substantial performance benefits. However, one must then decide when or whether to propagate updated data from one replica site to others. Simple schemes either ship many unneeded files, or suffer from long back-fetch delays, or both.

In this chapter, we have presented three schemes to intelligently decide whether or not to propagate an update. Each depends on the past predicting the future. The most successful of these ships updates whenever they cause invalidations elsewhere. This simple heuristic ships two orders of magnitude less data than aggressive schemes, while reducing the penalty of on-demand shipment by nearly a factor of two.

CHAPTER 7

Conclusion and Future Work

We summarize this dissertation and present several directions for future work.

7.1 Summary

As the number of mobile users increases, distributed file systems need to deal with new challenges. First, the network cost between clients and servers varies. When a client moves away from its home server, the cost to reach the server increases. Second, mobile clients are less available due to no network connectivity. Thus, data stored only at clients may be unavailable to others. Third, mobile clients are unreliable; they are highly susceptible to destruction and theft. Therefore, data stored only at clients may be lost.

The Fluid Replication file system addresses these challenges. It hides wide-area network costs from clients by introducing intermediate servers, called WayStations. A client sends updates immediately to a nearby WayStation, providing safety. The WayStation aggregates updates and notifies the server of them periodically over a wide-area network, providing visibility.

To cope with changes in network conditions, Fluid Replication clients monitor network conditions. When the conditions to the current WayStation or to the server become bad, a client looks for an alternative WayStation to alleviate the problem. But, under changing network conditions, it is difficult to tell apart the true changes from noise. We have developed a network filter that ignores transient changes but detects persistent ones quickly. Based on the estimation of network conditions, a client chooses a new WayStation.

For replica consistency, Fluid Replication uses optimistic concurrency control. Update notification is sent to the server periodically during reconciliations. Such notifications are sent without update file data; data shipment is deferred. This separation makes the reconciliation cost low, allowing frequent reconciliations for better visibility.

Update data is shipped to the server in a way that balances network load and client overhead. Propagating data aggressively wastes network and server resources, while doing it passively penalizes clients who share data. Our data propagation mechanisms predict data sharing among different nodes. Data is shipped to the server in advance only when it is likely to be used by another node. Our propagation techniques reduce both network traffic and server load while keeping the clients' penalty low.

7.2 Future Work

There are several directions for future work.

- **Automatic client migration**: We presented a network estimator to detect degradation of network capacity between WayStations and clients. Using this estimator, combined with a cost-benefit analysis of migration, a client should be able to migrate from one WayStation to another automatically when it moves too far from the old one.

- **Trust between clients and WayStations**: Before a client will agree to use a WayStation, it must be assured of the privacy and integrity of cached data, and of non-repudiation of updates. Preventing exposure through encryption is straightforward, though managing keys can be subtle. It is also easy to reliably detect unauthorized modifications using cryptographic hashes. However, guaranteeing that a WayStation will correctly forward updates is difficult, if not impossible.

- **Experience with Fluid Replication**: While the NFS traces are informative, they do not capture precisely how users will use Fluid Replication. It would be useful to gain experiences with Fluid Replication by deploying a server in a school department for day-to-day storage requirements, and provide users with WayStations and wireless gateways at home. This will allow client laptops to seamlessly migrate between locales, providing valuable insights into the system's use.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[2] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proceedings of ACM SIGCOMM '99*, pages 175–87, Cambridge, MA, August 1999.

[3] G. C. Berry, J. S. Chase, G. A. Cohen, L. P. Cox, and A. Vahdat. Toward automatic state management for dynamic web services. In *Network Storage Symposium*, Seattle, WA, October 1999.

[4] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the Winter 1992 USENIX Conference*, pages 333–343, Berkeley, CA, January 1992.

[5] J.-C. Bolot. Characterizing end-to-end packet delay and loss behavior in the Internet. *Journal of High Speed Networks*, 2(3):305–23, 1993.

[6] L. Breslau, D. Estrin, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[7] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of MobiCom'98*, pages 85–97, Dallas, TX, October 1998.

[8] R. B. Brown and P. Y. C. Hwang. *Introduction ot Random Signals and Applied Kalman Filtering*. John Wiley & Sons, Inc., 1997.

[9] R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of INFOCOM '97*, pages 1014–21, Kobe, Japan, April 1997.

[10] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. *Technical Report MSR-TR-2002-82*, 2002.

[11] M. D. Dahlin, C. J. Mather, and R. Y. Wang. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of ACM Sigmetrics*, Nashville, TN, May 1994.

[12] A. Downey. Using pathchar to estimate Internet link characteristics. In *Proceedings of ACM SIGCOMM '99*, pages 241–250, August 1999.

[13] D. Duchamp. Issues in wireless mobile computing. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 2–10, Key Biscayne, FL, April 1992.

[14] D. Ellard, J. Ledlie, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the Second Annual USENIX File and Storage Technologies Conference*, pages 203–216, San Francisco, CA, March 2003.

[15] I. Fried. Laptop sales lift Apple earnings. *CNET*, April 2003.

[16] B. Gammage. The changing spectrum of mobile PC usage. *Gartner*, October 2001.

[17] A. Gelb. *Applied Optimal Estimation.* M.I.T. Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

[18] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Parallel and Distributed Computing Systems*, pages 165–170, September 1995.

[19] J. S. Heidemann, T. W. Page, Jr., R. G. Guy, and G. J. Popek. Primarily disconnected operation: experience with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, November 1992.

[20] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayana, R. Sidebothamn, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, Feburary 1988.

[21] L. B. Huston and P. Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Mobile and Location Independent Computing Symposium*, pages 1–10, Cambridge, MA, August 1993.

[22] IEEE. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Std 802.11-1999.

[23] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[24] V. Jacobson. pathchar—a tool to infer characteristics of Internet paths. ftp://ftp.ee.lbl.gov/pathchar/, 1997.

[25] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In *Proceedings of ACM SIGCOMM '02*, Pittsburgh, PA, August 2002.

[26] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, T. Imielinski and H. Korth, editors. Chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[27] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[28] R. H. Katz. Bay area research wireless access network: Towards a wireless overlay internetworking architecture. In *ARPA GloMo PI Meeting*, Menlo Park, CA, November 1995.

[29] R. H. Katz and E. A. Brewer. The case for wireless overlay networks. In *Multimedia Computing and Networking 1996*, pages 77–88, San Jose, CA, January 1996.

[30] M. Kazar. Synchronization and caching issues in the Andrew File System. In *Proceedings of USENIX Conference*, pages 27–36, 1988.

[31] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM '91*, pages 3–15, September 1991.

[32] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002.

[33] J. J. Kistler. Disconnected operation in a distributed file system. *Ph.D. Thesis*, May 1993.

[34] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems 10*, Feburary 1992.

[35] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.

[36] T. M. Kroeger and D. D. E. Long. The case for efficient file access pattern modeling. In *Workshop on Hot Topics in Operating Systems*, pages 14–19, 1999.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[38] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 264–275, Saint-Malo, France, October 1997.

[39] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the 1995 USENIX Technical Conference*, pages 95–106, New Orleans, LA, January 1995.

[40] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2)(213-226), June 1981.

[41] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. *IEEE/ACM Transactions on Networking*, 6(5):515–28, October 1998.

[42] K. Lai and M. Baker. Measuring bandwidth. In *Proceedings of INFOCOM '99*, pages 235–45, New York, NY, March 1999.

[43] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[44] K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link bandwidth. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, pages 122–133, San Francisco, CA, March 2001.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[46] H. Lei and D. Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX 1997 Annual Technical Conference*, pages 275–288, Anaheim, CA, January 1997.

[47] S. Low. *Traffic management of ATM networks: service provisioning, routing, and traffic shaping*. PhD thesis, University of California, Berkeley, 1992.

[48] S. H. Low, F. Paganini, J. Wang, S. Adlakha, and J. C. Doyle. Dynamics of TCP/RED and a scalable control. In *Proceedings of IEEE INFOCOM'02*, pages 239–248, New York, NY, June 2002.

[49] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Markley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the IEEE Real Time Systems Symposium*, Orlando, 2000.

[50] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the IEEE Real Time Systems Symposium*, Phoenix, 1999.

[51] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services; a control-theoretic approach. In *Proceedings of the International Conference on Distributed Computing Systems*, Phoenix, 2001.

[52] D. C. Montgomery. *Introduction to statistical quality control*. John Wiley & Sons, Inc., 3rd edition, 1997.

[53] S. J. Mullender and A. S. Tanenbaum. A distributed file serivce based on optimistic concurrency control. In *the tenth ACM Symposium on Operating Systems Principles*, pages 51–62, Orcas Island, Washington, 1985.

[54] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain Resort, CO, December 1995.

[55] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. In *Proceedings of 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138–149, Nashville, TN, May 1994.

[56] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–87, Saint-Malo, France, October 1997.

[57] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM '97*, pages 51–61, Cannes, France, September 1997.

[58] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software — Practice and Experience*, 28(2):155–180, February 1998.

[59] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137–152, Boston, MA, June 1994.

[60] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of ACM SIGCOMM '97*, pages 139–52, Cannes, France, September 1997.

[61] K. Petersen, M. J. Spreitzer, K. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[62] T. S. Rappaport. *Wireless Communications:Principles and Practice*. Upper Saddle River, New Jersey:Prentice Hall, 1996.

[63] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of USENIX Conference*, pages 183–195, June 1994.

[64] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003.

[65] S. E. Rigdon, E. N. Cruthis, and C. W. Champ. Design strategies for individuals and moving range control charts. *Journal of Quality Technology*, 26(4):274–87, October 1994.

[66] C. V. Rijsbergen. *Information Retrieval*. Butterworth, London, 1979.

[67] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM*

International Conference on Distributed Systems Platforms, Heidelberg, Germany, November 2001.

[68] Safeware. Human error still poses greates threat to PCs. *safeware insurance surevey shows*, February 2002.

[69] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994. Corrigendum: 12(2):165–72, 1994.

[70] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[71] D. C. Steere, J. J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the Sun Vnode interface. In *Proceedings of the 1990 Summer USENIX Conference*, pages 325–331, Anaheim, CA, 1990.

[72] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 58–67, Austin, TX, September 1994.

[73] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, September 1994.

[74] C. Thornton. Laptop era dawns. *PC World*, October 2003.

[75] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[76] R. Wang, M. Valla, M. Sanadidi, and M. Gerla. Adaptive bandwidth share estimation in TCP Westwood. In *Proceedings of IEE/ISCC 2002*, Taipei, Taiwan, November 2002.

[77] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, June 1999.

[78] Western Electric. *Statistical Quality Control Handbook.* Western Electric Corporation, Indianapolis, Inc., 1956.

ABSTRACT

Providing Safety and Visibility for Mobile Users

by

Minkyong Kim

Chair: Brian D. Noble

Mobile users bring new challenges to distributed file systems. First, the network costs between clients and servers vary due to mobility of clients. Second, mobile clients are less available due to the absence of network connectivity or being suspended for power savings. Third, mobile clients are unreliable because they are highly susceptible to breakage and theft.

Current file systems needlessly combine safety and visibility; a client propagates the contents of an update, implicitly notifying the server that the update exists. Fluid Replication separates them through the addition intermediate servers, called *WayStations*. While traveling, a client associates itself with a nearby WayStation that provides replication services. Updates are sent to the nearby WayStation for safety, while WayStations and servers frequently exchange knowledge of updates through *reconciliation* to provide visibility. In this way, Fluid Replication provides safety and visibility to file updates over the wide-area with performance comparable to the local-area.

To choose a nearby WayStation, clients need to know the available network capacity to WayStations. This is particularly difficult in mobile networks where capacity changes frequently. Current systems depend on static exponentially weighted moving average (EWMA) filters. These filters are either able to detect true changes quickly or to mask transients, but cannot do both. This motivated the design of a new network filter. This filter is agile when possible and stable when necessary; it adapts to the prevailing network conditions.

During reconciliation, a WayStation sends the server update notifications without the contents of those updates. While sending notifications frequently is important to improve consistency, sending data should be deferred to reduce server load. However, deferring of data shipment penalizes clients who share because to serve these clients, the server must fetch updated files from other WayStations, called *back-fetch*, over a wide-area network. To solve this problem, a heuristic that predicts future file sharing has been developed. It is based on the observation that past instances of sharing are likely to lead to future ones. It reduces data shipped by orders of magnitude compared to aggressive schemes, while reducing the number of back-fetches by nearly half compared to on-demand shipment.