Total Recall: System Support for Automated Availability Management

Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Availability is a storage system property that is both highly desired and yet minimally engineered. While many systems provide mechanisms to improve availability – such as redundancy and failure recovery – how to best configure these mechanisms is typically left to the system manager. Unfortunately, few individuals have the skills to properly manage the trade-offs involved, let alone the time to adapt these decisions to changing conditions. Instead, most systems are configured statically and with only a cursory understanding of how the configuration will impact overall performance or availability. While this issue can be problematic even for individual storage arrays, it becomes increasingly important as systems are distributed – and absolutely critical for the wide-area peer-to-peer storage infrastructures being explored.

This paper describes the motivation, architecture and implementation for a new peer-to-peer storage system, called *TotalRecall*, that automates the task of availability management. In particular, the *TotalRecall* system automatically measures and estimates the availability of its constituent host components, predicts their future availability based on past behavior, calculates the appropriate redundancy mechanisms and repair policies, and delivers user-specified availability while maximizing efficiency.

1 Introduction

Availability is a storage system property that is highly desired in principle, yet poorly understood in practice. How much availability is necessary, over what period of time and at what granularity? How likely are failures now and in the future and how much redundancy is needed to tolerate them? When should repair actions be initiated and how should they be implemented? These are all questions that govern the availability of a storage system, but they are rarely analyzed in depth or used to influence the dynamic behavior of a system.

Instead, system designers typically implement a static set of redundancy and repair mechanisms simply parameterized by resource consumption (e.g., number of replicas). Determining how to configure the mechanisms and what level of availability they will provide if employed is left for the user to discover. Moreover, if the underlying environment changes, it is again left to the user to reconfigure the system to compensate appropriately. While this approach may be acceptable when failures are consistently rare, such as for the individual drives in a disk array (and even here the management burden may be objectionable [23]), it quickly breaks down in large-scale distributed systems where hosts are transiently inaccessible and individual failures are common.

Peer-to-peer systems are particularly fragile in this respect as their constituent parts are in a continual state of flux. Over short time scales (1-3 days), individual hosts in such systems exhibit highly transient availability as their users join and leave the system at will – frequently following a rough diurnal pattern. In fact, the majority of hosts in existing peer-to-peer systems are inaccessible at any given time, although most are available over longer time scales [4, 19]. Over still longer periods, many of these hosts leave the system permanently, as most peer-to-peer systems experience high levels of churn in their overall membership. In such systems, we contend that availability management must be provided by the system itself, which can monitor the availability of the underlying host population and adaptively determine the appropriate resources and mechanisms required to provide a specified level of availability.

This paper describes the architecture, design and implementation of a new peer-to-peer storage system, called *TotalRecall*, that automatically manages availability in a dynamically changing environment. By adapting the degree of redundancy and frequency of repair to the distribution of failures, *TotalRecall* guarantees user-specified levels of availability while minimizing the overhead needed to provide these guarantees. We rely on three key approaches in providing these services:

- *Availability Prediction*. The system continuously monitors the current availability of its constituent hosts. This measured data is used to construct predictions, at multiple time-scales, about the future availability of individual hosts and groups of hosts.

- *Redundancy Management*. Short time-scale predictions are then used to derive precise redundancy requirements for tolerating transient disconnectivity. The system selects the most efficient redundancy mechanism based on workload behavior and system policy directives.

- *Dynamic Repair*. Long time-scale predictions coupled with information about current availability drive system repair actions. The repair policy is dynamically selected as a function of these predictions, target availability and system workload.

TotalRecall is implemented in C++ using a modified version of the DHash peer-to-peer object location service [8]. The system implements a variety of redundancy mechanisms (including replication and online coding), availability predictors and repair policies. However, more significantly, the system provides interfaces that allow new mechanisms and policies to describe their behavior in a unified manner – so the system can decide how and when to best use them.

The remainder of this paper describes the motivation, architecture and design of the *TotalRecall* system. The following section motivates the problem of availability management and describes key related work. In Section 3 we discuss our availability architecture – the mechanisms and policies used to ensure that user availability requirements are met. Sections 4 describes the design of the *TotalRecall* Storage System and its implementation. Section 5 describes the *TotalRecall* File System, a NFS file service implemented on the core storage system. In Section 6, we quantitatively evaluate the effectiveness of our system and compare it against existing approaches. Finally, Section 7 concludes.

2 Motivation and Related Work

The implicit guarantee provided by all storage systems is that data, once stored, may be recalled at some future point. Providing this guarantee has been the subject of countless research efforts over the last twenty years, and has produced a wide range of technologies ranging from RAID to robotic tape robots. However, while the efficiency of these techniques has improved over time and while the cost of storage itself has dropped dramatically, the complexity of managing this storage and its availability has continued to increase. In fact, a recent survey analysis of cluster-based services suggests that the operational cost of preparing for and recovering from failures easily dominates the capital expense of the individual hardware systems [17]. This disparity will only continue to increase as hardware costs are able to reflect advances in technology and manufacturing while management costs only change with increases in human productivity. To address this problem, the management burden required to ensure availability must be shifted from individual system administrators to the systems themselves. We are by no means the first to make this observation.

A major source of our inspiration is early work invested by HP into their AutoRAID storage array [23]. The AutoRAID system provided two implementations of storage redundancy – mirroring and RAID5 – and dynamically assigned data between them to optimize the performance of the current workload. While this system did not directly provide users with explicit control over availability it did significantly reduce the management burden associated with configuring these high-availability storage devices. A later HP project, AFRAID, did allow user-specified availability in a disk array environment, mapping availability requests into variable consistency management operations [20].

In the enterprise context, several researchers have recently proposed systems to automate storage management tasks. Keeton and Wilkes have described a system designed to automate data protection decisions that is similar in motivation to our own, but they focus on longer time scales since the expected failure distribution in the enterprise is far less extreme than in the peer-to-peer environment [10]. The WiND system is being designed to automate many storage management tasks in a cluster environment, but is largely focused on improving system performance [2]. Finally, the PASIS project is exploring system support to automatically make trade-offs between different redundancy mechanisms when building a distributed file system [24].

Not surprisingly, perhaps the closest work to our own arises from the peer-to-peer (P2P) systems community. Due to the administrative heterogeneity and poor host availability found in the P2P environment, almost all P2P systems provide some mechanism for ensuring data availability in the presence of failures. For example, the CFS system relies on a static replication factor coupled with an active repair policy, as does Microsoft's FARSITE system (although FARSITE calculates the replication factor as a function of total storage and and is more careful about replica placement) [1, 8, 9]. The Oceanstore system uses a combination of block-level erasure coding for long term durability and simple replication to tolerate transient failures [12, 22]. Finally, a recent paper by Blake and Rodrigues argues that the cost of dynamic membership makes cooperative storage infeasible in the transiently available peer-to-peer environments [5]. This finding is correct under certain assumptions, but is not critical in the environments we have measured and the system we have developed.

What primarily distinguishes *TotalRecall* from previous work is that we allow the user to specify a specific availability target and then automatically determine the best mechanisms and policies to meet that request. In this way, *TotalRecall* makes availability a first-class storage property – one that can be managed directly and without a need to understand the complexities of the underlying system infrastructure.

3 Availability Architecture

There are three fundamental parameters that govern the availability of any system: the times at which components *fail* or become unavailable, the amount of redundancy employed to tolerate these outages and the time to detect and repair a failure.

The first of these is usually considered an independent parameter of the system, governed primarily by the environment and external forces not under programmatic control.[1] The remaining variables are dependent – they can be controlled, or at least strongly influenced, by the system itself. Therefore, providing a given level of availability requires predicting the likelihood of component failures and determining how much redundancy and what kind of repair policies will compensate appropriately.

The remainder of this section discusses the importance of these issues in turn, how they can be analyzed and how they influence system design choices. The following section then describes our concrete design and implementation of this overall architecture.

3.1 Availability Prediction

At the heart of any predictive approach to availability is the assumption that past behavior can be used to create a stochastic model of future outcomes. For example, "mean-time-to-failure" (MTTF) specifications for disk drives are derived from established failure data for similar components over a given lifetime. This kind of prediction can be quite accurate when applied to a large group of fail-stop components. Consequently, the future availability of single homogeneous disk arrays can be statically analyzed at configuration time.

However, in a distributed storage system – particularly one with heterogeneous resources and administration – this prediction can be considerably more complex. First, since the hosts composing such systems are rarely identical, the availability distribution cannot be analytically determined *a priori* – it must be measured empirically. Second, unlike disks, individual hosts in a distributed system are rarely fail-stop. Instead, hosts may be transiently unavailable due to network outages, planned maintenance or other local conditions. Consequently, such hosts may become unavailable and then return to service without any data loss. Finally, as such systems evolve, the number of hosts populating the system may grow or shrink – ultimately changing the availability distribution as well.

Nowhere is this issue more pronounced than in the public peer-to-peer environment. In such systems, the availability of an individual host is governed not only by failures, but more importantly by user decisions to disconnect from the network. Several recent studies of peer-to-peer activity have confirmed that individual hosts come and go at an incredible rate. In one such study of hosts in the Overnet system, we have observed that each host joined and left the system over 6 times per day on average [4]. In a similar study of Kazaa and Gnutella, Saroiu et al. found that the median session duration of a peer-to-peer system was only 60 minutes [19]. In addition to the transient availability found in these systems, public peer-to-peer populations exhibit a high rate of long-term churn as well. The previously-mentioned Overnet study found that approximately 20 percent of active hosts permanently departed from the system each day and roughly the same number of new hosts joined as well.

Consequently, in the peer-to-peer context, storage systems face two prediction requirements. First, the system must empirically measure the short-term availability distribution of its host population on an ongoing basis. We use this to model the probability of transient disconnections – those typically having no impact on the durability of data stored on disconnected hosts. From this distribution we estimate the *current* likelihood that a set of hosts will be available at any given time and subsequently determine the proper amount of redundancy needed. Our second prediction requirement focuses on non-transient failures that take stored data out of the system for indefinite periods. Since hosts are leaving the system continuously, redundancy is insufficient to ensure long-term storage availability. Instead the system must predict when hosts have "permanently" left the system (at least for long enough a period that they were no longer useful in the short term) and initiate a repair process.

3.2 Redundancy Management

In a peer-to-peer environment, each host may only be transiently available. When connected, the data stored on a host contributes to the overall degree of redundancy and increases the data's availability; when disconnected, both the degree of redundancy and data availability decreases. With sufficient redundancy across many hosts, at any moment enough hosts will be in the system to make a given data item available with high probability. However, it is not trivially clear how much redundancy is necessary for a given level of availability or what redundancy mechanism is most appropriate for a given context. We discuss both issues below.

There are a wide range of mechanisms available for producing redundant representations of data. However, each mechanism has unique trade-offs. For example, the simplest form of redundancy is pure replication. It has low run-time overhead (a copy) and permits efficient random access to sub-blocks of an object. However, replication can be highly inefficient in low-availability environments since many storage replicas are required to tolerate

[1] This is not always true, since processes that impact human error or opportunities for correlated failures can have an impact. However, we consider these issues outside the scope of this paper.

potential transient failures. At the other extreme, optimal erasure codes are extremely efficient. For a constant factor increase in storage cost, an erasure-coded object can be recovered at any time using a subset of its constituent blocks. However, the price for this efficiency is a quadratic coding time and a requirement that reads and writes require an operation on the entire object. By comparison, "non-optimal" erasure codes sacrifice some efficiency for significantly reduced on-line complexity for large files. Finally, it is easy to conceive of hybrid strategies as well. For example, a large log file written in an append-only fashion, might manage the head of the log using replication to provide good performance and eventually migrate old entries into an erasure coded representation for provide higher efficiency.

However, for all of these representations another question remains: how much redundancy is required to deliver a specified level of availability. More precisely: given an known distribution for short-term host availability and a target requirement for instantaneous data availability, how should these mechanisms be parameterized? Below we provide analytic approximations to these questions for pure replication and pure erasure coding. In both cases, our approach assumes that host failures are independent over short time scales. In previous work, we have provided a detailed explanation of our stochastic analysis and its assumptions [3], as well as experimental evidence to support our independence assumption [16]. Consequently, the *TotalRecall* system is not designed to survive catastrophic attacks or widespread network failures, but rather the availability dynamics resulting from localized outages, software crashes, disk failures and user dynamics.

Replication. Given a target level of availability A (where A represents the probability a file can be accessed at any time) and a mean host availability of μ_H, we can calculate the number of required replicas, c, directly.

$$A = 1 - (1 - \mu_H)^c \tag{1}$$

Solving for c,

$$c = \frac{log(1 - A)}{log(1 - \mu_H)} \tag{2}$$

Consequently, if mean host availability is 0.5, then it requires 10 complete copies of each file to guarantee a target availability of 0.999.

Some systems may choose to perform replication for individual blocks, rather than the whole file, as this allows large files to be split and balanced across hosts. However, this is rarely an efficient solution in a low-availability environment since every block (and hence at least one host holding each block) must be available for the file to be available. To wit, if a file is divided into b blocks, each of which has c copies, then the availability of that file is given by:

$$A = (1 - (1 - \mu_H)^c)^b \tag{3}$$

Consequently, a given level of availability will require geometrically more storage (as a function of b) in the block-level replication context.

Erasure coding. Given the number of blocks in a file b, and the stretch factor c specifying the erasure code's redundancy (and storage overhead) we can calculate the delivered availability as:

$$A = \sum_{j=b}^{cb} \binom{cb}{j} {\mu_H}^j (1 - \mu_H)^{(cb-j)} \tag{4}$$

If cb is moderately large, we can use the normal approximation to the binomial distribution to rewrite this equation and solve for c as:

$$c = \left(\frac{k\sqrt{\frac{\mu_H(1-\mu_H)}{b}} + \sqrt{\frac{k^2\mu_H(1-\mu_H)}{b} + 4\mu_H}}{2\mu_H} \right)^2 \tag{5}$$

More details on this equation's derivation can be found in [3]. For the same 0.999 level of availability used in the example above, an erasure-coded representation only requires a storage overhead of 2.49.

3.3 Dynamic Repair

However, the previous analyses only consider short-term availability – the probability that at a given instant there is sufficient redundancy to mask transient disconnections and failures. Over longer periods, hosts permanently leave the system and therefore the degree of redundancy afforded to an object will strictly decrease over time – ultimately jeopardizing the object's availability. In response, the system must "repair" this lost redundancy by continuously writing additional redundant data onto new hosts.

The two key parameters in repairing file data are the degree of redundancy used to tolerate availability transients and how quickly the system reacts to host departures. In general, the more redundancy used to store file data, the longer the system can delay before reacting to host departures.

Below we describe a spectrum of repair policies defined in terms of two extremes: eager and lazy. Eager repair uses a smaller degree of redundancy to maintain file availability guarantees by reacting to host departures immediately, but at the cost of additional communication overhead. In contrast, lazy repair uses additional redundancy, and therefore additional storage overhead, to delay repair and thereby reduce communication overhead.

3.3.1 Eager Repair

Many current research peer-to-peer storage systems maintain data redundancy pro-actively as hosts depart from the system. For example, the DHash layer of CFS replicates each block on five separate hosts [8]. When DHash detects that one of these hosts has left the system, it immediately repairs the diminished redundancy by creating a new replica on another host.

We call this approach to maintaining redundancy *eager repair* because the system immediately repairs the loss of redundant data when a host fails. Using this policy, data only becomes unavailable when hosts fail more quickly than they can be detected and repaired.

The primary advantage of eager repair is its simplicity. Every time a host departs, the system only needs to place redundant data on another host in reaction. Moreover, detecting host failure can be implemented in a completely distributed fashion since it isn't necessary to coordinate information about which hosts have failed. However, the eager policy makes no distinction between permanent departures that require repair and transient disconnections that do not. Consequently, in public peer-to-peer environments, many repair actions may be redundant and wasteful. In Section 6 we show via simulation that this overhead is very high for contemporary peer-to-peer host populations.

3.3.2 Lazy Repair

An alternative to eager repair is to defer immediate repair and use additional redundancy to mask and tolerate host departures for an extended period.[2] We call this approach *lazy repair* since the explicit goal is to delay repair work for as long as possible. The key advantage of lazy repair is that, by delaying action, it can eliminate the overhead of redundant repairs and only introduce new redundant blocks when availability is threatened.

However, lazy repair also has disadvantages. In particular, it must explicitly track the availability of individual hosts and what data they carry. This is necessary to determine when an object's availability is threatened and a repair should be initiated. Consequently, the system must maintain *explicit* metadata about which hosts hold what data. By contrast, eager implementations can make use of the implicit mappings available through mechanisms like consistent hashing [8]. For small objects, this can significantly increase the overhead of repair actions.

For lazy repair, the system must incorporate additional redundancy for files so that it can tolerate host departures over an extended period. Hence while the analysis in the previous section gives us the *short-term* redundancy factor used to tolerate transient failures, each file needs to use a larger *long-term* redundancy factor to accommodate host failures without having to perform frequent file repairs.

As mentioned in Section 3.1, the system requires an availability predictor that will estimate when a file needs to be repaired. A simple predictor for lazy repair periodically checks the total amount of available redundancy for a given file. If this value falls below the short-term redundancy factor for the file, then the system triggers a repair. Thus we also refer to the short-term redundancy factor as the *repair threshold* for the file.

Section 6 compares the repair bandwidth required by each policy using an empirical trace of peer-to-peer host availability patterns.

3.4 System Policies

The combination of these mechanisms – prediction, redundancy and repair – must ultimately be combined into a system-wide strategy for guaranteeing file availability. Minimally, a system administrator must specify a file availability target over a particular lifetime. From these parameters, coupled with an initial estimate of host availability, an appropriate level of redundancy can be computed. In addition to repair actions triggered by the disappearance of individual hosts, the system may occasionally need to trigger new repair actions to compensate for changes in the overall availability of the entire population. For example, a worm outbreak may reduce the average host availability system-wide or the expansion of broadband access may increase the average uptime of connected hosts.

However, there is significant room for more advanced policies. For example, there is a clear trade-off between random access performance and storage efficiency in the choice of redundancy mechanism. A system policy can make this trade-off dynamically in response to changing workloads. For instance, a file might use an erasure coded base representation, but then replicate frequently accessed sub-blocks independently. As well, system policies could easily specify different availability requirements for different portions of the file system and even calculate availability as a function of file dependencies (e.g., a user may wish to request a given level of availability for the combination of the mail program and the mail spool it uses).

4 *TotalRecall* Storage System

This section describes the design and implementation of the *TotalRecall* Storage System. The *TotalRecall* Storage System implements the availability architecture described in Section 3 in a cooperative host environment. It provides a simple read/write storage interface

[2] This is similar, in spirit, to Oceanstore's *refresh* actions which are meant to ensure data durability in the face of disk failures [12].

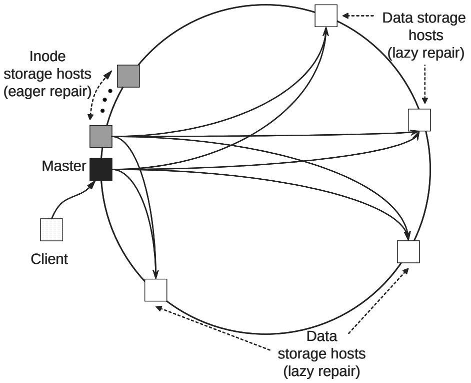

Figure 1: *TotalRecall* system architecture.

for maintaining data objects with specified target availability guarantees.

4.1 System Overview

Hosts in *TotalRecall* contribute disk resources to the system and cooperatively make stored data persistent and available. Figure 1 shows an overview of *TotalRecall* with participating hosts organized in a ring ID space. *TotalRecall* stores and maintains *data objects*, conveniently referred to as *files*. Files are identified using unique IDs. The system uses these IDs to associate a file with its *master* host, the host responsible for maintaining the persistence, availability, and consistency of the file. *Storage* hosts persistently store file data and metadata according to the repair policy the master uses to maintain file availability. *Client* hosts request operations on a file. Clients can send requests to any host in the system, which routes all requests on a file to its master. As a cooperative system, every *TotalRecall* host is a master for some files and storage host for others; hosts can also be clients, although clients do not need to be *TotalRecall* hosts.

A *TotalRecall* server runs on every host in the system. As shown in Figure 2, the *TotalRecall* host architecture has three layers. The *TotalRecall* Storage Manager handles file requests from clients and maintains file availability for those files for which it is the master. It uses the Block Store layer to read and write data blocks on storage hosts. The Block Store in turn uses an underlying distributed hash table (DHT) to maintain the ID space and provide scalable lookup and request routing.

4.2 Storage Manager

The *TotalRecall* Storage Manager (TRSM) implements the availability architecture described in Section 3. It has three components, the policy module, the availability monitor, and the redundancy engine (see Figure 2).

The TRSM invokes the policy module when clients create new files or substantially change file characteristics such as size. The policy module determines the most efficient strategy for maintaining stored data with a target availability guarantee. The strategy is a combination of redundancy mechanism, repair policy, and number of blocks used to store coded data. It chooses the redundancy mechanism (e.g., erasure coding vs. whole-file replication) based on workload characteristics such as file size and the rate, ratio, and access patterns of read and write requests to file data (Section 3.4). The repair policy determines how the TRSM maintains data availability over long-term time scales to minimize repair bandwidth for a target level of availability (Section 3.3). Although redundancy and repair are orthogonal, for typical workloads *TotalRecall* uses replication and eager repair for small files and erasure coding and lazy repair for large files (Section 6.3.1). Finally, with lazy repair the policy module also determines the number of blocks to use with erasure coding to balance file availability and communication overhead; more blocks increases availability but requires the TRSM to contact more storage hosts to reconstruct the file [3].

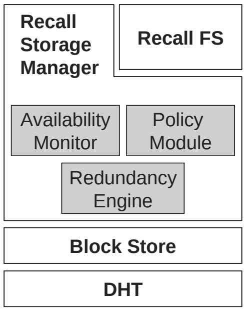

Figure 2: *TotalRecall* host architecture.

The TRSM dynamically adapts its mechanisms and policies for efficiently maintaining data according to the availability of hosts in the system. To do this, the availability monitor (AM) tracks host availability, maintains host availability metrics that are used by other components, and notifies the redundancy engine when the system reaches availability thresholds that trigger repair. The AM tracks the availability of the storage hosts storing the metadata and data for those files for which it is the master. Based upon individual host availability, the AM maintains two metrics: *short-term host availability* and *long-term decay rate* (Section 3.1). Short-term host availability measures the minimum average of all tracked hosts that were available at any given time in the past 24 hours (e.g., 50% of hosts were available at 4am). It is a conservative prediction of the number of hosts available over the course of a day. Long-term decay rate measures the rate at which hosts leave the system over days and weeks, and is used to predict the frequency of repair. Finally, the TRSM registers to receive events from the AM whenever the availability of a set of storage hosts drops below a specified threshold to trigger repairs.

Whereas the policy module decides what kind of redundancy mechanism to use based upon high-level workload characteristics, the redundancy engine (RE) implements the redundancy mechanisms and determines how much short-term and long-term redundancy to use for a file based upon current system conditions. The TRSM invokes the redundancy engine when writing and repairing files. The RE currently supports simple replication and erasure coding. For replication, the RE uses Equation 2 in Section 3.2 to determine the number of replicas r to create when storing the file. It uses the target availability A associated with the file and the short-term host availability from the AM as inputs to the equation. For erasure coding, the RE uses Equation 5 to determine the short-term redundancy r (also called repair threshold) for encoding the file. It uses the target availability A associated with the file, the short-term host availability from the AM, and the number of blocks b determined by the policy module as inputs to the equation.

4.3 Storage Layout

For every file, the *TotalRecall* Storage Manager uses *inodes* as metadata to locate file data and maintain file attributes such as target availability, size, version, etc. It stores inodes for all files using replication and eager repair. The master stores inodes itself and a set of replicas on its successors, much like DHash blocks [8], and the redundancy engine determines the number of replicas (Section 4.2). Figure 1 shows an example of storing an inode for a lazily repaired file. The master updates inodes in place, and it serializes all operations on files by serializing operations on their inodes (e.g., a write does not succeed until all inode replicas are updated).

The TRSM stores data differently depending upon the repair policy used to maintain file availability. For files using eager repair, the TRSM on the master creates a unique file data ID and uses the DHT to lookup the storage host responsible for this ID. It stores file data on this storage host and its successors in a manner similar to inodes. The inode for eagerly repaired files stores the file data ID as a pointer to the file data.

For files using lazy repair, the TRSM stores file data on a randomly selected set of storage hosts (Section 3.3.2). Figure 1 also shows how the master stores file data for lazily repaired files. It stores the IDs of the storage hosts in the file's inode to explicitly maintain pointers to all of the storage hosts with the file's data. It also uses the redundancy engine to determine the number of storage hosts to use, placing one block (erasure coding) or replica (replication) per storage host.

File data is immutable. When a client stores a new version of a file that is lazy repaired, for example, the TRSM randomly chooses a new set of storage hosts to store the data and updates the file's inode with pointers to these hosts. The TRSM uses the version number stored in the inode to differentiate file data across updates. A garbage collection process periodically reclaims old file data, and a storage host can always determine whether its file data is the latest version by looking up the inode at the master (e.g., when it joins the system again after being down).

4.4 Storage API

The *TotalRecall* Storage Manager implements the storage API. The API supports operations for creating, opening, reading, writing, and repairing files, and similar operations for inodes. All request operations on a file are routed to and handled by the file's master. Lacking the space to detail all operations, we highlight the semantics of a few of them.

Clients use `tr_create` to create new files, specifying a target availability A for the file upon creation. It is essentially a metadata operation that instantiates a new inode to represent the file, and no data is stored until a write operation happens. `tr_read` returns file data by reading data from storage hosts, decoding erasure-coded data when appropriate. `tr_write` stores new file data or logically updates existing file data. It first sends the data to storage hosts and then updates the inode and inode replicas (see Section 4.5). For lazily repaired files, encoding and distributing blocks for large files can take a considerable amount of time. To make writes more responsive, the master uses a background process that performs the encoding and block placement offline. The master initially eagerly repairs the blocks using simple replication, and then erasure codes and flushes these blocks out to the storage hosts.

The TRSM also implements the `tr_repair` operation for repairing file data, although its execution is usually only triggered internally by the availability manager. For eager repair, `tr_repair` repairs data redundancy immediately when a host storing data departs. For lazy repair, it only repairs data when the number of hosts storing file data puts the file data at risk of violating the file's target availability. Since this occurs when much of the file's data is on hosts that are not available, `tr_repair` essentially has the semantics of a file read followed by a write onto a new set of hosts.

4.5 Consistency

Since the system maintains replicas of inodes and inodes are updated in place, the master must ensure that inode updates are consistent. In doing so, the system currently assumes no partitions and that the underlying DHT provides consistent routing — lookups from different hosts for the same ID will return the same result.

When writing, the master ensures that all data writes complete before it updating the inode. The master writes all redundant data to the storage hosts, but does not start updating the inode until all the storage hosts have acknowledged their writes. If a storage host does fail during the write, the master will retry the write on another storage host. Until all data writes complete, all reads will see the older inode and, hence, the older version of the file. As the master makes replicas of the inode on its successors, it only responds that the write has completed after all successors have acknowledged their writes. Each inode stores a version number assigned by the master ordered by write requests to ensure consistent updates to inode replicas. Once a successor stores an inode replica, eager repair of the inode ensures that the replica remains available. If the master fails as it updates inodes, the new master will synchronize inode versions with its successors. If the master fails before acknowledging the write, the host requesting the write will time out and retry the write to the file. A new master will assume responsibility for the file, receive the write retry request, and perform the write request. As a result, once a write completes, i.e., the master has acknowledged the write to the requester, all subsequent reads see the newest version of the file.

4.6 Implementation

We have implemented a prototype of the *TotalRecall* storage system on Linux in C++. The system consists of over 5,700 semi-colon lines of new code. We have also reused existing work in building our system. We use the SFS toolkit [14] for event-driven programming and MIT's Chord implementation as the underlying DHT [21]. Files stored using eager repair use a modified version of the DHash block store [8].

The prototype implements all components of the *TotalRecall* Storage Manager, although some advanced behavior remains future work. The prototype policy module currently chooses the redundancy mechanism and repair policy solely based on file size: files less than 32 KB use replication and eager repair, and larger files use erasure coding and lazy repair. For lazy repair, files are fragmented into a minimum of 32 blocks with a maximum block size of 64 KB. To erasure code lazily repaired files, the redundancy engine implements Maymounkov's online codes [13], a sub-optimal linear-time erasure-coding algorithm. The redundancy engine also uses a default constant long-term redundancy factor of 4 to maintain lazy file availability during the repair period.

The availability monitor tracks host availability by periodically probing storage hosts with an interval of 60 seconds. This approach has been sufficient for our experiments on PlanetLab, but would require a more scalable approach (such as random subsets [11]) for tracking and disseminating availability information in large-scale deployments. The TRSM uses the probes to storage hosts for a file to measure and predict that file's availability. Based upon storage host availability, the TRSM calculates the amount of *available redundancy* for the file. The available redundancy for the file is the ratio of the total number of available data blocks (or replicas) to the total number of data blocks (replicas) needed to read the file in its entirety. When this value drops below the repair threshold, the AM triggers a callback to the TRSM, prompting it to start repairing the file. The prototype by default uses a repair threshold of 2. With a long-term redundancy factor of 4 for lazy repair, for example, when half of the original storage hosts are unavailable the AM triggers a repair.

In building our prototype, we have focused primarily on the issues key to automated availability management, and have not made any significant effort to tune the system's runtime performance. Addressing run-time overheads, as well as implementing more advanced performance and availability tradeoffs in the policy module, remains ongoing work.

5 *TotalRecall* File System

The *TotalRecall* Storage System provides a core storage service on which more sophisticated storage services can be built, such as backup systems, file-sharing services, and file systems. We have designed one such service, the *TotalRecall* File System (TRFS), an NFSv3-compatible file system [7]. To provide this service, the *TotalRecall* File System extends the Storage Manager with the TRFS Manager (see Figure 2). The TRFS Manager extends the storage system with file system functionality, implementing a hierarchical name space, directories, etc. It extends the *TotalRecall* Storage Manager with an interface that roughly parallels the NFS interface, translating file system operations (e.g., mkdir) into lower-level TRSM operations.

Clients use a TRFS loopback server to mount and access TRFS file systems. The loopback server runs on the client as a user-level file server process that supports the NFSv3 interface [14]. It receives redirected NFS operations from the operating system and translates them into RPC requests to *TotalRecall*.

We have implemented TRFS as part of the *TotalRecall* prototype, adding 2,000 lines of code to implement the TRFS Manager and loopback server. It currently supports all NFSv3 operations except hard links.

6 Experimental Evaluation

In this section, we evaluate *TotalRecall* using both trace-driven simulation and empirical measurements of our prototype implementation. We use simulation to study the effectiveness of our availability predictions, the be-

Hosts	5500
Files	5500
No. of Blocks Before Encoding	32
File Sharing File Sizes	4 MB (50%), 10 MB (30%), 750 MB (20%)
File System File Sizes	256 B (10%), 2 KB (30%), 4 KB (10%), 16 KB (20%), 128 KB (20%), 1 MB (10%)

Table 1: File workloads used to parameterize simulation.

havior of the system as it maintains file availability, the tradeoffs among different repair policies, and *TotalRecall*'s use of bandwidth resources to maintain file availability. And we evaluate the prototype implementation of the *TotalRecall* File System on a 32-node deployment on PlanetLab and report both per-operation microbenchmarks and results from the modified Andrew benchmark.

6.1 Simulation Methodology

Our simulator, derived from the well-known Chord simulation software [21], models a simple version of the *TotalRecall* Storage System. In particular, it models the availability of files across time as well as the bandwidth and storage used to provide data and metadata redundancy. The simulator is designed to reveal the demands imposed by our system architecture and not for precise prediction in a particular environment. Consequently, we use a simple model for host storage (infinite capacity) and the network (fixed latency, infinite bandwidth, no congestion).

To drive the simulator we consider two different file workloads and host availability traces. The two file workloads, parameterized in Table 1, consist of a *File Sharing* workload biased towards large files [18] and a more traditional *File System* workload with smaller files [6]. Similarly, we use two corresponding host availability traces. The File Sharing trace is a scaled down version of a trace of host availability in the Overnet file sharing system [4], while the File System availability trace is synthetically generated using the host availability distribution in [6]. The two availability traces are both one week long and differ primarily in their dynamics: the File Sharing trace has an average host uptime of 28.6 hours, compared to 109 hours in the File System trace.

The simulations in this section execute as follows. Hosts join and leave the system, as dictated by the availability trace, until the system reaches steady-state (roughly the same number of joins as leaves). Then files are inserted into the system according to the file workload. Subsequent joins and leaves will cause the system to trigger repair actions when required. The system repairs inodes eagerly, and data eagerly or lazily depending on policy (Section 4.6). From the simulation we can then determine the delivered file availability and bandwidth usage: the two primary metrics we evaluate.

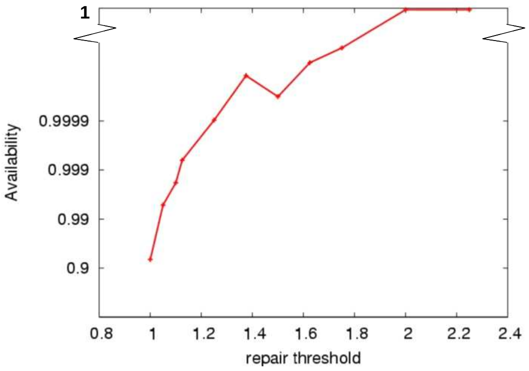

Figure 3: Empirical file availability calculated for the File Sharing host availability distribution.

6.2 Delivered Availability

It is critical that *TotalRecall* is able to deliver the level of availability requested. To verify, we specify a target availability of 0.99 and from this compute the required repair threshold. Using Equation 4 with an average host availability of 0.65, we compute that an erasure coded file with lazy repair will require a repair threshold of at least 2 to meet the availability target.

To see how well this prediction holds, we simulate a series of periodic reads to all 5500 files in the File Sharing workload. Using the associated host availability trace to drive host failures, we then calculate the average file availability as the ratio of completed reads to overall requests. Figure 3 shows how this ratio varies with changes in the repair threshold (this assumes a constant long-term redundancy factor of 4). From the graph, we see that files with a repair threshold of 2 easily surpass our 0.99 availability target. For this trace a lower repair threshold could also provide the same level of availability, although doing so would require more frequent file repairs.

To provide better intuition for this dynamic, Figure 4 shows the repair behavior of *TotalRecall* over time at a granularity of 60 minutes. We use the File Sharing workload parameters in Table 1 to parameterize the system and the File Sharing host availability trace to model host churn. The three curves on the graph show the number of available hosts, the bandwidth consumed by the system, and the average *normalized available file redundancy* across all files in each time interval. Available file redundancy measures the amount of redundant data that the system has available to it to reconstruct the file. For each file, we normalize it with respect to the long-term redundancy factor used for the file, so that we can compute an average over all files.

Looking at the curves over time, we see how *TotalRecall* uses lazy repair to maintain a stable degree of data

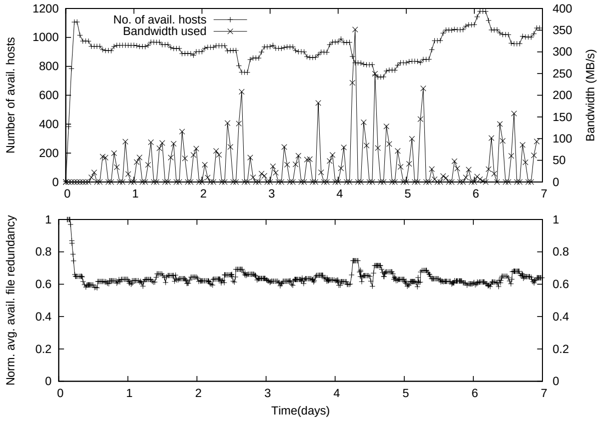

Figure 4: System behavior on the File Sharing workload.

redundancy as host availability varies substantially over time. Note that though the total number of hosts available at any time is roughly between 800 and 1000, new hosts are constantly joining the system while old hosts leave, causing substantial amounts of host churn. We make three observations of the system behavior.

First, we see that system bandwidth varies with host availability. As hosts leave the system, *TotalRecall* eagerly repairs inodes and lazily repairs data blocks for those files whose predicted future availability drops below the lazy repair threshold. Consequently, relative system bandwidth increases as the number of hosts decreases. As hosts join the system, *TotalRecall* eagerly repairs inodes but does not need to repair data blocks. Consequently, relative system bandwidth decreases as the number of hosts increases.

Second, the normalized average degree of available redundancy reaches and maintains a stable value over time, even with substantial host churn. This behavior is due to the design of the lazy repair mechanism. Files stored using a lazy repair policy experience cyclic behavior over time. When the system first stores a file using lazy repair, it places all of the redundant data blocks on available hosts. At this time, the file has maximum available redundancy (since we create all files at time 0, all files have maximum available redundancy at time 0 in Figure 4). As hosts leave the system, file blocks become unavailable. As hosts join the system again, file blocks become available again. As a result, available file redundancy fluctuates over time. But the long-term trend is for blocks to become unavailable as hosts depart the system for extended periods of time (possibly permanently). Eventually, based upon *TotalRecall*'s prediction of future host availability and current available file redundancy (Section 3.1), enough blocks will become inaccessible that the system will trigger a lazy repair to ensure continued file availability. Lazy repair will replace missing redundant blocks and raise available file redundancy back to its maximum, and the cycle continues.

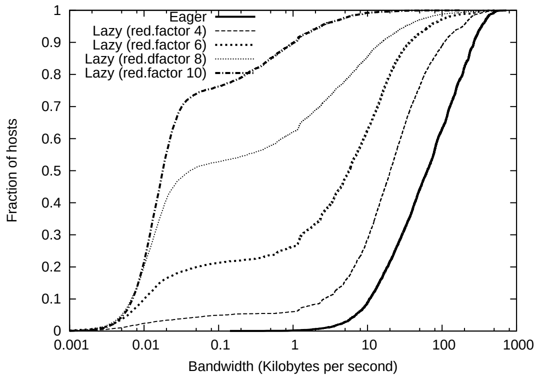

Figure 5: CDF of the bandwidth usage of hosts in *TotalRecall* for different repair policies.

Third, the overall average system repair bandwidth for the entire time duration is 35.6 MB/s. Dividing by the number of files, the average repair bandwidth per file is 6.5 KB/s. While this is not insignificant, we believe that, given the large file sizes in the File Sharing workload (20% 750MB), this figure is reasonably small. Also, note that using larger long-term redundancy factors has the effect of reducing the bandwidth usage of the system (Figure 5). Breaking down bandwidth overhead by use, overall 0.6% of the bandwidth is used for eager repair of inodes and 99.4% is used for lazy repair of data blocks

.

6.3 Repair Overhead

A key design principle of *TotalRecall* is to adapt the use of its repair policies to the state of the system. These policies have various tradeoffs among storage overhead, bandwidth overhead, and performance, and interact with the distributions of host availability and file sizes as well. Finally, *TotalRecall* efficiency hinges on accurate prediction of future failures. We investigate these issues in turn.

6.3.1 Repair Policy

To illustrate the tradeoff between storage and bandwidth, we simulate the maintenance of the File Sharing workload on *TotalRecall* and measure the bandwidth required to maintain file availability using the File Sharing host availability trace to model host churn. (Note that we do not include the bandwidth required to write the files for the first time.) We measure the average bandwidth required by each node to maintain its inode and data blocks across the entire trace for five different repair policies: eager repair, and lazy repair with erasure coded data using four different long-term redundancy factors.

Figure 5 shows the cumulative distribution function of the average bandwidth consumed by the hosts in the system over the trace for the different repair policies. From the graph, we see that eager repair requires the most

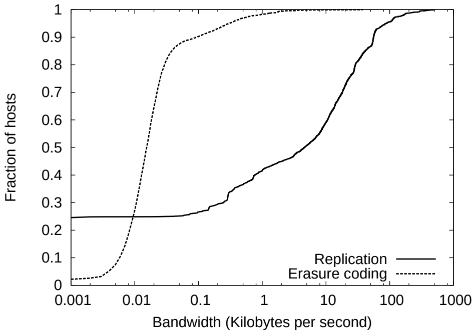

Figure 6: CDF of the bandwidth usage of hosts comparing replication and erasure coding for lazy repair.

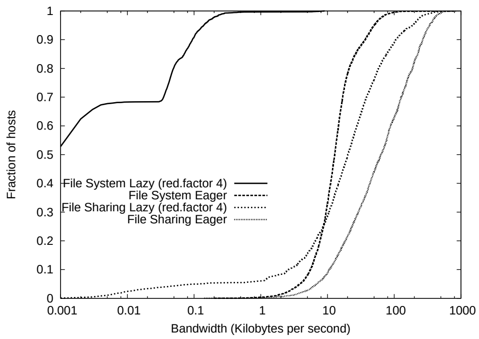

Figure 7: CDF of the bandwidth usage of hosts comparing eager and lazy repair on different workloads.

maintenance bandwidth across all hosts, lazy repair with a long-term redundancy factor of 4 requires the second-most bandwidth, and larger long-term redundancy factors require progressively less bandwidth. These results illustrate the fundamental tradeoffs between redundancy and repair bandwidth. Eager repair, which uses convenient but minimal redundancy, cannot delay repair operations and requires the most bandwidth. Lazy repair, which uses more sophisticated redundancy to delay repair, requires less bandwidth, especially with significant host churn as in the File Sharing scenario. Lazy repair with lower long-term redundancy factors require less storage, but more frequent repair. Higher long-term redundancy factors delay repair, but require more storage.

The shapes of the curves in Figure 5 show how the bandwidth requirements vary across all of the hosts in the system for the different repair policies. Eager repair essentially has a uniform distribution of bandwidth per node across all hosts. This is mainly due to the fact that hosts are assigned random IDs. Consequently, hosts leave and join the system at random points in the DHT, and the load of making replicas of inodes and file blocks is well distributed among all the hosts in the system.

In contrast, lazy repair essentially has two categories of hosts. The first is all the hosts that store some file data blocks. The second are hosts that join and leave the system before any file repairs are triggered, do not receive any file data, and participate only in the eager repair of inodes. As a result, the bandwidth usage of these hosts is smaller than those that store data. For larger long-term redundancy factors such as 8 and 10, file repairs are not that frequent, and hence there are a significant number of hosts that fall into the second category. Curves for these long-term redundancy factors in Figure 5 have a sharp rise around 30 bytes per second, demonstrating the presence of an increasing number of such hosts with increasing long-term redundancy factor.

So far we have concentrated on evaluating lazy repair with erasure coding. We now study how lazy repair with coding compares with lazy repair with replication. The question that this experiment seeks to answer is that for the same level of file availability and storage, how does the bandwidth usage of lazy repair with coding compare to the bandwidth usage of lazy repair with replication.

To maintain a file availability of 0.99, Equations 3 and 4 estimate that lazy repair with erasure coding has a repair threshold of 2 and lazy repair with replication requires 5 replicas. In other words, the system needs to repair files with erasure coding when the redundancy (degree of coded data) falls below 2, and the system would have to perform repairs with replication when the available redundancy (number of replicas) falls below 5. Lazy repair with replication therefore potentially uses more bandwidth than lazy repair with erasure coding.

To quantify how much more bandwidth replication uses, we repeat the bandwidth measurement simulation experiment but assign a long-term redundancy factor of 10 to each file. For lazy repair with coding, the system performs a file repair when the file redundancy falls to 2 and, for lazy repair with replication, the system repairs the file when the redundancy falls to 5. Figure 6 shows the CDF of bandwidth required per host for these two cases. From the graph, we see that the system bandwidth requirements to perform lazy repair with replication are far higher than that required for lazy repair with erasure codes. The average bandwidth per host for lazy repair with erasure coding is 655 Bps, while lazy repair with replication is 75 KBps. Our conclusion from these experiments is that for large file size distributions, and for highly dynamic and highly unavailable storage components, lazy repair with erasure coding is the more efficient availability maintenance technique.

6.3.2 Host Availability

To study the affect of different host availability distributions on bandwidth usage, we compared the bandwidth consumed for each host for the File Sharing host availability trace with that of the File System trace.

Figure 7 shows that the File System availability trace requires less bandwidth and that lazy repair works partic-

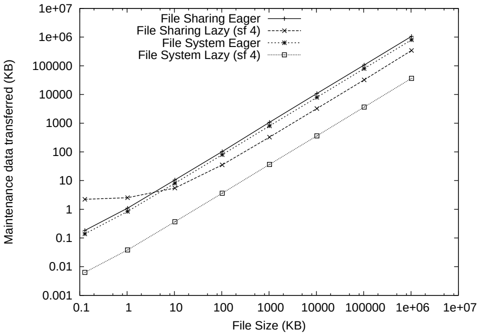

Figure 8: Per-file bandwidth required for repair.

ularly well. Since the availability of hosts is higher for this trace, the host churn is lower. Moreover, the source of this trace was collected from a workplace, and as a result we see a more cyclic pattern of availability for some hosts. These hosts contribute to eager repair bandwidth usage since they increase the churn in the system. However, since they cyclically re-appear in the system, they do not trigger lazy repair.

6.3.3 File Size

The repair policy for a file depends on file size (Section 4.2). To illustrate the tradeoff between eager and lazy repair on file size, we measure the bandwidth usage per file in the system with both eager and lazy repair, for both File Sharing and File System host availability traces, for various file sizes.

Figure 8 shows the average system bandwidth for maintaining each file for the entire trace for a range of file sizes. For each host availability trace, the graph shows two curves, one where the system maintains files using eager repair and the other where the system uses lazy repair. From the graph, we see that, for the File Sharing trace, eager repair requires less bandwidth to maintain small files less than approximately 4 KB in size, but that lazy repair requires less bandwidth for all larger files. This crossover between eager and lazy is due to the larger inodes required for lazy repair. For the File System trace however, we do not see a crossover point. Since the trace has less churn, fewer repairs are required and less bandwidth is consumed for eager repair. Eager repair is better for smaller file sizes and higher host availability.

To see the effect of using a hybrid repair policy, i.e., using eager repair for files smaller than 4 KB and lazy for all others, we simulated the File System workload on *TotalRecall* using the File Sharing and the File System host traces. Figure 9 shows the CDF of average bandwidth usage per host for pure lazy repair and the hybrid policy, and for both host availability traces. There is very little difference in the bandwidth usage between the two curves for the same host trace. From this we conclude that, for small files, *TotalRecall* should use eager repair. While bandwidth usage is comparable to that for lazy repair, the performance will be better since the system avoids the computational overhead of encoding/decoding these files and also avoids the communication overhead of distributing them over many storage hosts.

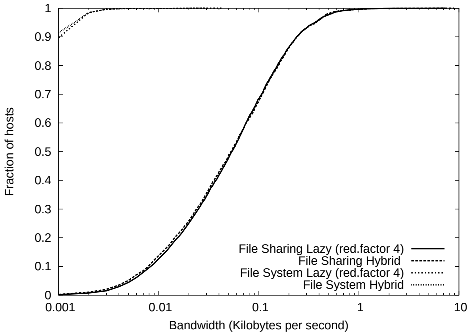

Figure 9: CDF of the bandwidth usage per host for lazy repair and the hybrid policy.

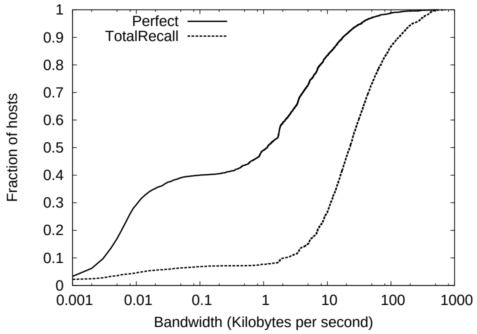

Figure 10: CDF of the bandwidth usage per host comparing *TotalRecall* with an optimal system.

6.3.4 Prediction

Though we have established that lazy repair with erasure coding is the most efficient availability maintenance technique in our system, we would like to see how close *TotalRecall* comes to optimal bandwidth usage in the system. The question we address is, if there existed an Oracle that would repair a file $just$ before it becomes unavailable, how would the system's bandwidth usage characteristics compare to those of *TotalRecall*?

To answer this question, we compare *TotalRecall*'s bandwidth consumption using lazy repair and erasure coding to that of an optimal system that also uses lazy repair and erasure coding. The optimal system minimizes bandwidth by performing repairs just before the files become unavailable. Note that a file becomes unavailable when its redundancy drops below 1 (less data available than originally in the file). To model the optimal system, we modified the simulator so that whenever the availability monitor detected that a file's redundancy dropped below 1, it would initiate a repair. In contrast, *TotalRecall*

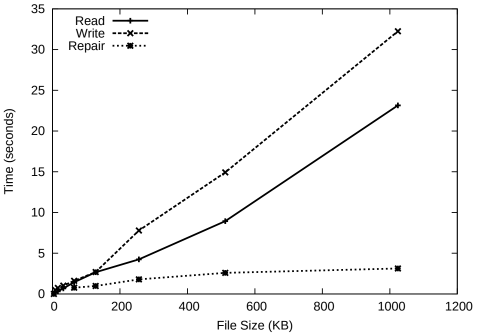

Figure 11: File system performance on PlanetLab.

initiates file repair whenever the file's redundancy drops just below the repair threshold of 2. Both *TotalRecall* and the optimal simulator use a long-term redundancy factor of 4 for this experiment.

Figure 10 shows the results of this experiment. The bandwidth usage in *TotalRecall* is almost an order of magnitude more than the optimal system. While the average bandwidth usage in *TotalRecall* is 49 KBps, the optimal system is 7 KBps. The difference is due to two reasons. First, it is very difficult to predict host behavior accurately given the strong dynamics in the system. Second, *TotalRecall*'s main goal is to guarantee availability of files, and in doing so, it tends to make conservative estimates of when file repairs are required. We believe this to be a suitable design decision given the system's goals. However, this experiment does show that there is room for the system to improve its bandwidth usage by using more sophisticated techniques to predict host failures and file availability.

6.4 Prototype Evaluation

The simulation experiments focused on the file availability and bandwidth overhead of providing available files in the *TotalRecall* Storage System. Next, we evaluate the performance of the prototype *TotalRecall* File System implementation. To perform our measurements, we ran TRFS on a set of 32 PlanetLab hosts distributed across the U.S. We used a local machine as a client host mounting the *TotalRecall* File System via the TRFS loopback server. In all experiments below, the system uses an eager repair threshold of 32 KB, i.e., the system replicates and eager-repairs all files smaller than 32 KB, and it uses erasure coding and lazy repair for all files of size greater than 32 KB. For lazy repair, the system uses a long-term redundancy factor of 4.

6.4.1 Microbenchmarks

We first evaluate the *TotalRecall* File System by measuring the performance of file system operations invoked via the NFS interface. Figure 11 shows the sequential file read, write, and repair performance for lazily repaired files in TRFS running on the 32 PlanetLab hosts for a range of file sizes. First, we measured the performance of using the NFS interface to write a file of a specified size using a simple C program. We then measured the performance of the same program reading the file again, making sure that no cached data was read. Finally, we forced the master node responsible for the file to perform a lazy repair. Lazy repair roughly corresponds to a combined read and write: the master node reads a sufficient number of file blocks to reconstruct the file, and then writes out a new encoded representation of the file to a new set of randomly chosen nodes.

From the graph, we see that write performance is the worst. Writes perform the most work: writing a file includes creating and storing inodes, encoding the file data, and writing all encoded blocks to available hosts. Read performance is better because the master node need only read a sufficient number of blocks to reconstruct the file. Since this number is smaller than the total number of encoded blocks stored during a write read operations require fewer RPC operations to reconstruct the file.

Finally, lazy repair performs the best of all. Although this might seem counterintuitive since the lazy repair operation requires more work than read or write, lazy repair operates within the *TotalRecall* Storage System. As a result, it is able to operate in parallel on much larger data aggregates than the read and write operations. NFS serializes read operations, for example, in 4KB block requests to the master. However, when the master reads blocks to perform a lazy repair, it issues 64KB block requests in parallel to the storage hosts.

6.4.2 Modified Andrew Benchmarks

Phase	Duration (s)
mkdir	12
create/write	60
stat	64
read	83
compile	163
Total	392

Table 2: Wide-area performance of the modified Andrew benchmark on 32 PlanetLab nodes.

We also ran the modified Andrew benchmarks on TRFS on 32 hosts on PlanetLab. We chose hosts that were widely distributed across the U.S. Table 2 shows the results of running these benchmarks on *TotalRecall*. We see that the read phase of the benchmark takes longer than the write phase. Since the benchmark primarily consists of small files that are eagerly-repaired and replicated in our system, the writes take less time than if they were erasure coded. The compile phase, however, takes a fair amount of time since the final executable is large and it is erasure-coded and lazy-repaired. The total time of execution of the benchmarks was 392 seconds. As one point of rough comparison, we note that in [15] the authors evaluated the Ivy peer-to-peer file system on 4 hosts across the Internet using the same benchmark with a total execution time of 376 seconds.

The absolute performance of the *TotalRecall* File System is not remarkable, and not surprising since we have not focused on performance. In part this is due to the wide variance in the underlying network performance of the PlanetLab hosts used in our experiments (e.g., 25% of the nodes have RPC latency over 100 ms) and time spent in software layers underneath *TotalRecall* (e.g., 87% of the time writing 4 KB files in Figure 11 is spent in Chord lookups and block transfers from storage hosts). Given that our implementation is an unoptimized prototype, we are also exploring optimizations to improve performance, such as aggregating and prefetching data between clients and the master to improve NFS performance.

7 Conclusions

In this paper, we have argued that storage availability management is a complex task poorly suited to human administrators. This is particularly true in the large-scale dynamic systems found in peer-to-peer networks. In these environments, no single assignment of storage to hosts can provide a predictable level of availability over time and naive adaptive approaches, such as eager replication, can be grossly inefficient.

Instead, we argue that availability should become a first class system property – one specified by the user and guaranteed by the underlying storage system in the most efficient manner possible. We have proposed an architecture in which the storage system predicts the availability of its components over time, determines the appropriate level of redundancy to tolerate transient outages, and automatically initiates repair actions to meet the user's requirements. Moreover, we have described how key system parameters, such as the appropriate level of redundancy, can be closely approximated from underlying measurements and requirements. Finally, we described the design and implementation of a prototype of this architecture. Our prototype peer-to-peer storage system, called *TotalRecall*, automatically adapts to changes in the underlying host population, while effectively managing file availability and efficiently using resources such as bandwidth and storage.

Acknowledgments

We would like to thank the reviewers for their valuable comments, Barbara Liskov for being our shepherd, and Marvin McNett for the system support during the development of *TotalRecall*. Support for this work was provided in part by DARPA FTN Contract N66001-01-1-8933 and AFOSR MURI Contract F49620-02-1-0233.

References

[1] A. Adya et al. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of OSDI*, 2002.

[2] A. C. Arpaci-Dusseau et al. Manageable storage via adaptation in WiND. In *IEEE CCGrid*, 2001.

[3] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer systems. Technical Report CS2002-0726, UCSD, Nov 2002.

[4] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proc. of IPTPS*, 2003.

[5] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of HotOS*, 2003.

[6] W. J. Bolosky et al. Feasibility of a serverless distributed file system depoloyed on an existing set of desktop PCs. In *Proc. of SIGMETRICS*, 2000.

[7] B. Callaghan. *NFS Illustrated*. Addison Wesley, 1999.

[8] F. Dabek et al. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[9] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proc. of SRDS*, 2001.

[10] K. Keeton and J. Wilkes. Automating data dependability. In *Proc. of the ACM SIGOPS European Workshop*, 2002.

[11] D. Kostic et al. Using random subsets to build scalable services. In *Proc. of USITS*, 2003.

[12] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[13] P. Maymounkov and D. Mazieres. Rateless codes and big downloads. In *Proc. of IPTPS*, 2003.

[14] D. Mazieres. A toolkit for user-level file systems. In *Proc. of the USENIX technical conference*, 2001.

[15] A. Muthitacharoen et al. Ivy: a read-write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[16] Overnet website, http://www.overnet.com.

[17] D. A. Patterson et al. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB-CSD-02-1175, UC Berkeley, 2002.

[18] S. Saroiu et al. An Analysis of Internet Content Delivery Systems. In *Proc. of OSDI*, 2002.

[19] S. Saroiu et al. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN*, 2002.

[20] S. Savage and J. Wilkes. AFRAID – a frequently redundant array of independent disks. In *Proc. of the USENIX Technical Conference*, 1996.

[21] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM*, 2001.

[22] H. Weatherspoon et al. Silverback: A global-scale archival system. Technical Report UCB-CSD-01-1139, UC Berkeley, 2001.

[23] J. Wilkes et al. The HP AutoRAID hierarchical storage system. In *Proc. of SOSP*, 1995.

[24] J. J. Wylie et al. Survivable information storage systems. *IEEE Computer*, 2001.