Stony Brook University

Repairable File and Storage Systems

A Dissertation Presented

by

Ningning Zhu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2008

Stony Brook University

The Graduate School

Ningning Zhu

We, the dissertation committee for the above candidate for
the Doctor of Philosophy degree,
hereby recommend acceptance of this dissertation.

Professor Tzi-cker Chiueh, Dissertation Advisor
Department of Computer Science

Professor Erez Zadok, Chairman of Defense
Department of Computer Science

Professor Michael Bender
Department of Computer Science

Dr. Daniel Ellard
BBN Technologies

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Repairable File and Storage System

by

Ningning Zhu

Doctor of Philosophy

in

Computer Science

Stony Brook University

2008

The data contents of an information system may be corrupted due to human errors, malicious attacks or untrusted software. The financial loss of such corruption is typically proportional to the amount of time required to recover the system's data/service. Recognizing that it is impossible to build absolutely secure computer systems and that human errors are inevitable, this dissertation proposes a repairable system framework which greatly reduces data loss and system downtime with minimum cost and performance penalty. We illustrate that repairability is affordable and ready to be integrated into main stream file/storage system.

Repairable file/storage system needs to perform two tasks. First it has to maintain all the raw data so that every update is undoable. Secondly it has to keep track of data updates due to errors and attacks so that only the data affected by mistakes or attacks are rolled back to their last known consistent state. We develop two novel comprehensive versioning schemes for repairable NFS file server and for repairable

SAN storage system. We design a simple solution for dependency tracking and integrate it with both schemes. We also developed an NFS trace play toolkit and gained experience on trace driven file system evaluation.

For the repairable file system, we focus on the performance optimization in the absence of failures and errors. Empirical measurements show that the performance overhead due to repairability is less than 10%. For the repairable storage system, more focuses are shifted to the integration with traditional fault tolerance techniques. Evaluation results show that the repairable storage system is available upon any single point of failure, including disk failure with and without data losses, power failure, network failure and software crash failure.

Dedicated to my mother and father

Contents

List of Tables

List of Figures

Acknowledgments

First of all I would like to thank my advisor Professor Tzi-cker Chiueh. He taught me the essential skills of doing concrete systems research, from how to read papers and to how to design a system, and especially how to evaluate it – yes the ”numbers”. His strong passion and inspiration by hard work were very motivating. His insightful comments kept me on track throughout. His high standards pushed me to a level that I could have never reached by myself. He also instilled in me the basic values of honesty and sincerity of attitude towards research which I will forever cherish.

Sincere thanks go to my committee members. When I arrived at Stony Brook, Professor Michael Bender's classes on algorithms were incredibly fun and intriguing. Every Thursday, I looked forward for the homework, something I've never done in my life earlier. These classes instilled my love for algorithms since then and I still feel Prof. Bender's influence in the protocol design in my final dissertation. I first got to know Dr. Daniel Ellard when he confessed his anxiety about mailing the hard disk full of NFS traces from Boston to Stony Brook in the rough weather in early spring. Through the years, we have developed a strong professional collaboration as well as a warm personal friendship. He gave me great advice in the TBBT paper and in the fault tolerance protocol design of Mariner system. I

am impressed with his broad knowledge, great research attitude and sense of humor. Professor Erez Zadok was a valuable source of feedback on my dissertation. The FSL lab that he directs was also very helpful during my years in Stony Brook. I have been fortunate to exchange file system developing tools and experiences, benchmarks and traces with the students there.

It was an unforgettable experience to share five years of my life with my ECSL colleagues. Many of them have given me great support and have been of much inspiration in many aspects - from kernel debugging to proof-reading, from fixing cars to setting up a testbed, and from SAC lunch puzzles to late night conversations, I owe deep thanks to Srikant Sharma, Pradipta De, Jiawu Chen, Gang Peng and Ashish Raniwala. I'd also like to thank Maohua Lu and Shibiao Lin who worked closely with me in the Mariner project and gave me a lot of support after I left Stony Brook campus. Many senior students have helped me to get started with system research and provided me the survival tips in ECSL, including Lan Huang, Anindya Neogi, Kartik Gopalan, Prashant Pradhan and Lap-chung Lam. Other members of ECSL, Fanglu Guo, Jui-Hao Chiang, Kyung Hoon Kim, Fu-hau Hsu, Alexey Asimov, Wei Li, have enriched my life in one way or another. Again thanks Professor Tzi-cker Chiueh for leading such a wonderful research lab.

Special thanks go to my manager Eisar Lipkovitz in the Google indexing team who was very understanding and generously reduced my workload and approved my leave from work. It would have been very hard to finish my dissertation without this support. I deeply appreciate the help from my old and current colleague Chi Zhang all through the years, from applying to the Ph.D program in US together 10 years ago, to design discussions, to the tricky issues in operator duty. Also a big thank to other colleagues in the Google indexing team, Joachim Kupke, Sitaram Iyer, Hao Wu, Kekoa Proundfoot, Chuck Wu, Abhishek Gupta, Ram Janakiraman, Nikhil Sarin, Hui Xu, David Ziegler, Li xiao, Andrea Chu, Paul Chien and John

Huang for their helps in the office so that I have more time for the thesis.

Thanks to Stony Brook Computer Science System Administrator Brian Tria for much needed help with hardware, especially in helping me to recover very important pieces of source code after a disk failure towards the end of my PhD study. Thanks Kathy Germa for her prompt help with the paper work and administration issues, especially after I left Stony Brook.

The PhD life would not have been as fun, warm and comforting without my friends: V.N. Venkatakrishnan, Bixia Ji, Heng Xue, Hua Qian, Ajay Gupta, Diptikalyan Saha, Yang Feng, Yuanyuan Zhou, Xiaolan Zhang, Geng Liu, Rui Yang, Danxia Ke, Yuhua Wang, Nita and Kefei Lu.

Thanks Mr Lin Cai and Ms Jine Hu for taking care of my daughter every day since she was 5 month old. While I kept working, it was such a big relief to know that my daughter was in caring and loving hands. Also thanks to Bixia Ji and her parents for taking care of my daughter with great attention in hours of need.

Many people have helped me in the final push of the dissertation. Ming Zhang provided me valuable advices on the implementation details of UNH ISCSI. Wei Xu provided a lot of help with the vmware-based testbed setup. Srikant Sharma provided a thorough reading and critique of the lengthy Mariner Chapter. Ram Janakiraman helped me polish the RFS chapter even during the busy time of a new project launch. Rdennis Hayes provided insightful comments on several chapters. In addition, Srikant Sharma, Sitaram Iyer, Chi Zhang, and Joachim Kupke provided detailed, comprehensive, valuable feedback with the presentation slides. Jiawu Chen and Heng Xue provided the critical help with the testbed setup during the final performance evaluation time which I had to do it remotely.

My mother and father seeded me with the interest for science since early childhood. They never intended that I should do a PhD, but they whole-heartedly supported me on my decision to do one. Throughout the long Ph.D. journey, they were

always been there on the other side of the phone line, ready to listen and to encourage. I am truly thankful to my little sister Jingjing Zhu. Growing up together, we share almost everything closely with each other - except that she took most of the responsibilities taking care of our parents during my busy study period.

Last but not the least, I thank Andrei Vvedenski for knowing me better than myself. For the past 6 years, with all kinds of difficulties, the question that ”Shall I finish my PhD?” has never stopped haunting me. Every time I was faced with this question (seriously or not), I always got a firm ”Yes” from him. A dear whisper to my 2-year old Paulina Vvedenskaya ”You taught me how to prioritize and do time management. Before you were born, I was not able to enjoy a free Sunday without feeling guilty. I am proud of myself that this dissertation is not based on any major sacrifice from your part”.

Chapter 1

Introduction

Traditional fault tolerance research has been largely focused on tolerating two kinds of failures. First is the hardware or site failure which may occur to any computer system. Second is the specific issue of an untrusted peer in a distributed environment. In recent years, other failure scenarios, including human mistakes, security breaches, and untrusted software have drawn the attention of both the research community and industry. In this dissertation these failures are called "soft failures." They are distinguished from "hard failures," which refer to hardware, network, or site failures that completely halt the system and render it unavailable so that no useful work can be performed unitl the stricken system revovers. By their nature, hard failures are relatively easy to detect. In contrast, soft failures may not render the system immediately unavailable and may even allow some useful tasks to execute. Soft failures therefore may take longer to detect. Although preventing soft failures is desirable, the preponderance of experience, distilled into the facts and statistics of failure analysis, show that soft failures are as unavoidable as hard failures.

Despite a growing volume of research focused on improving computer security, there is no such thing as an unbreakable system. At the same time, according to

Gartner Group's estimate, on average more than 50% of the cost associated with a computer security break-in is attributable to lost productivity or revenue due to data loss, service disruption, or the additional work required to repair the damages that intruders wrought. These two facts - that computer breaches are inevitable and that the real cost of an intrusion lies in the post-attack data or service unavailability - argue strongly for an intrusion-tolerance direction to computer security research. It is time to shift at least some of the research focus from intrusion prevention to the design and development of system techniques that can minimize losses by facilitating post-intrusion system clean-up and restoration.

Soft failures are by no means exclusively the result of premediated attacks. Even absent malicious attacks, humans make mistakes, which can and do lead to data damage or service outages. James Reason [10, 35] published a study of the types of computer-related human errors and concluded that they are, even when involving simple tasks, unavoidable. Interestingly,he found that humans tend to self-detect errors. According to Reason, people detect about 75% of errors immediately after they make them. Another study [11],which analysed PSTN and Internet sites' operational statistics, underscores a consistent pattern: that operator's errors are the leading cause of system failures, as compared with other software issues, hardware faults, or overloads. While some application software supports the "'undo"'- reversal - of certain user actions, it is neither feasible nor possible to enhance all software to cover all user mistakes.

Untrusted software comprises another threat to computer systems. Attracted by its promised functionality or low price, users consciously accept the risks of downloading and then running untrusted software, but often find it hard to clean up the mess once it is proven unsafe and damaging.

1.1 Tolerance of soft failures

One common fault tolerance approach is redundancy. RAID, local and remote replications, and alternative network paths are typical redundancy techniques for hardware and site failures. These assume a variety of forms. In P2P environemnts, the peer collaborative design may not amount to redundancy, strictly speaking, but relies on a similar scheme and the assumption that not all peers are malicious. Another example of a redundant approach is deploying multiple versions of software, each developed independently, to protect the system from bugs. Still another strategy is requiring two operators to be present simultaneously for important operations, thereby decreasing the liklihood of an operator mistake. To effectively protect the system from one kind of failure, the redundancy must be deployed at the same level as the failure, and the probability that each redundant unit fails should be derived independently. Therefore, hardware redundancy cannot protect systems from user mistakes; instead, it may simply repeat the mistakes in the redundant system. In the case of security breaches, the redundancy should be built in such a way that breaking into one system will not help the attacker break into the redundant one. Redundancy is not always practical, affordable, or legal. While the cost of low-level redundancy for hardware and site failure is modest, the cost of higher level redundancy is often unaffordable except for mission critical tasks. In the case of human mistakes, redundancy is often preempted by privacy concerns.

While redundancy tolerates failures by executing the same task multiple times, expecting that at least one will be successful, we explore an orthogonal approach in which tasks execute only once while the system maintains the capability to undo the effects of failures. This capability is called repairability. Traditional backup helps to provide primitive system repairability, but it can not meet the current requirements of “soft failure” tolerance.

In summary, a system may be damaged due to malicious attack, inadvertent human error, or untrusted software. It is fundamentally difficult to stop all malicious attacks, to completely prevent mistakes, or to restrain users from trying untrusted software. So the next best thing one can hope for is to repair the damaged system and return it back to a functional state as soon as possible while preserving its most useful tasks and related data. The file system image is the most important system state; it is the prime repair target. Repairing other system states, such as memory or network states, is out of the scope of this dissertation. The goal of a repairable file/storage system is to minimize the system downtime and data loss upon *soft failures*.

The protection provided by a repairable system is complementary to that of the recoverable system, where redundancy (replication, mirroring) is used to protect against site or hardware failures. The repairable system is not meant to replace the documentation control systems or information archival systems which have different goals, provide different features, and are under different performance constraints (Chapter 2. But we do expect that repairability can be added extensively to existing systems without significant performance overhead or extra hardware, software, or management costs. To protect data from soft failures , repairable file/storage systems must perform two tasks:

- Comprehensive Versioning. Maintain all the raw data so that every update is undoable.
- Dependency Tracking. Keeping track of the dependencies between the soft failure and data updates so that only the data affected by soft failures are rolled back to their last known consistent state.

1.2 Comprehensive versioning

Versioning functionality has been built into many systems. Although all versioning techniques save old data, the goals can be numerous and varied. In addition to protecting data against soft failures , such goals include version control documentation, facilitating backups, resolving concurrency conflicts (in distributed systems), or archiving information.They also have different workload assumptions and performance requirements. In a repairable file/storage system, the desirable features of versioning are:

- Versioning should be comprehensive. Soft failures are unavoidable and unpredictable. This requires that the protection mechanism be deployed extensively and activated by default. This contrasts with versioning systems that are periodical or triggered by file closure or upon user request, etc.

- Versioning should be simple to configure. The task of configuring the versioning itself should not be prone to user mistakes. Versioning simplicity also tends to reduce the TCO (total cost of ownership) The simplest configuration uses the repairable file/storage system as the default file (storage) system and protects all data automatically. More complicated configurations may create various policies for different files or file types (e.g., files with common suffixes), which may require significant effort.

- Versioning tasks should not signficantly degrade the performance of the repairable system. If the versioning overhead is low enough to allow the repairable system to achieve benchmark performance for most workloads, the repairable system can be used extensively. When used extensively, the repaiarable system is most effective at protecting user data. Low overhead also

correlates with simple configuration; when extensive versioning creates significant overhead, many systems provide diverse versioning polices or rely on application-aware solutions.

- The performance of accessing old version is not of great importance. We assume that old version access is rare and needed only for undo or repair operations. However, this is in comparison with current data access. Compared to traditional data restoration from backup, our data repair procedure or old version access are still much faster. When soft failure does occur, typically the most time-consuming task in the repair process is identifying which previous version to use (Dependency Analysis), not accessing it.

While the first three features pose great challenges to the design of appropriate versioning techniques, the last feature allows us to optimize the versioning structures for current data access. Moreover, the per byte price of disk storage has been continuously dropping in recent years. With easily accessible large and cheap disk storage, comprehensive versioning is feasible and cost-effective.

1.3 Dependency Tracking and System Repairing

Comprehensive versioning makes arbitrary point-in-time images available and forms the basis of a repairable system. The point-in-time image itself can be easily used to correct simple errors, such as a careless deletion. In more complicated scenarios, soft failures interleave with useful jobs. By “useful jobs’ we mean those which create important data for users. Jobs which are not useful (in the above sense) are called “neutral jobs”. Neutral jobs may update temporary files, cached files, automatically generated files, or any files that are considered unimportant by the user or which do not result in updates to the file system.

To repair effectively, we need to 1) identify the parts of the system that are compromised directly or indirectly by the soft failures , or 2) identify the parts of the system that are updated by the most recent useful job and not corrupted by the soft failure . In case 1), the repair could undo the effects of a soft failure from the current image. In the case 2), the repair could first roll back the system to the last clean snapshot and then redo the updates of useful jobs. While sometimes the data affected by the soft failure or useful job can be easily identified by the user and the repair process is simple, it can be time consuming and complicated to repair when there are large amounts of interleaving data updates. The second task of a repairable system - dependency tracking – helps to correlate data updates with a soft failure or useful job. Note that similar recovery functionality that is supported by the application software itself could be fairly easy because of complete application knowledge. In contrast, the repair scheme in repairable file/storage system is not application 'aware.'

1.4 Trace based system performance evaluation methodology

The performance of comprehensive versioning is essential for a repairable system. However, the effectiveness of many design decisions and performance optimizations are workload dependent. An ideal file system evaluation workload is representative of real application requirements, effective in predicting system performance in target environments, easy to use, scalable to stress the system under evaluation, and reproducible. Currently the most commonly used workloads for file system evaluation are synthetic workload benchmarks. These benchmarks are

designed specifically to re-create the characteristics of particular operating environments. Most synthetic benchmarks are parameterizable, making it possible to tailor the resulting workload to specific requirements. Although in recent years synthetic benchmarks have improved significantly in terms of realism and the degree to which they can be tailored to a specific application, it is not always possible for a synthetic file system benchmark to mimic file access traces collected from a real-world environment. Firstly, there are many time-varying and site-specific factors in a workload that are very difficult, if not impossible, for a benchmark to capture. Secondly, because the time required to develop a high-quality benchmark is often on the order of months or years, benchmarks cannot always keep up with the dynamic changes in the workload of the target environment.

In contrast, traces taken from a system are, by definition, representative of that system's workload as long as they are collected carefully and over a period time long enough to ensure that the characteristic workload has been captured. Therefore we believe that file access traces can serve as a basis for file system evaluation benchmarks. Even though file access traces have been used for workload characterization and guided the development of many file system design techniques, they are rarely used in the evaluation of file systems or servers. Given that disk, network, and web access traces have been used extensively to evaluate storage systems, network protocols, and web servers respectively, we do not see why file access traces should not be used to evaluate file systems. The reason that this has not been done already is because replaying file access traces is more difficult that replaying other types of traces: we must take into account the related facts that the file system is stateful and that access requests are dependent on one another.

Another issue independent of synthetic benchmark or trace play-back is file system aging. File systems that have been in use for a period of time have different performance characteristics than new file systems. Therefore, "aging" a file system

is an important part of file system performance evaluation.

1.5 Contribution

In this dissertation, we identify several important computer system failure scenarios which have not been addressed at the system level by traditional fault tolerance research, namely user mistakes, malicious attacks, and untrusted software. We propose to enhance existing systems with repairability to tolerate these soft failures . The solution incorporates comprehensive versioning, dependency tracking, user knowledge, and external assistance from intrusion detection software, system integrity checking software, etc.

None of the existing versioning schemes can satisfy the stringent performance requirements for comprehensive versioning in a repairable system. Accordingly, we designed two novel comprehensive versioning schemes customized specifically for two repairable systems that we built. The first scheme works with NFS protocol. It focuses on transparency and portability. The second scheme works at the block device level. It focuses on seamless integration with other advanced features of a high performance large storage system (replication, multicast, low latency write) and fault tolerance design.

We bring forth the notion of dependency tracking in the context of tolerating soft failures . We explored design choices of dependency tracking in general and present effective solutions for the two repairable systems that we built.

We advocate a trace-driven file system evaluation methodology. En route to this methodology, we identified many challenges in playing file system traces that do not exist for other traces such as network traces and disk traces. We developed the first general purpose, fully-fledged network file system trace player. We also developed a unique artificial file system aging technique that ages a file system

much faster than any other file system aging technique. We publish here for the first time the results of a file system evaluation that employs long duration traces from commercial servers using the tools we developed.

The rest of the dissertation is organized as follows. Chapter 2 analyzes the design choices of versioning systems and reviews related work in the area of logging, versioning and system repair. In Chapter 3, we explore the design choices for dependency tracking. Chapter 4 describes the design, implementation and evaluation of RFS, a repairable file system. Chapter 5 describes Mariner, a high performance storage system with repairability, recoverability and robust fault tolerance design. Chapter 6 describes TBBT, an NFS trace play toolkit. Chapter 7 concludes the dissertation.

Chapter 2

Related Work

In this chapter we review the related work of repairable file and storage systems. The related work for TBBT - the NFS trace player - is described in Chapter 6. The related works presented in this chapter focus on versioning and dependency tracking techniques. In addition we discuss general logging techniques and data mirroring/replication techniques.

2.1 Versioning

In this section, we provide an overview of the versioning system design choices. Then we compare the techniques and trade-offs of versioning systems at the file system level and at the storage system level. Finally we describe each versioning system separately.

In general, a versioning system can be characterized by: (1) The level that a versioning system is built at, (2) the interface that a versioning system provides for old data access, and (3) the frequency of versioning, i.e., continuous or discrete.

Versioning functionality can be built at different levels, namely, the application

level, the file system level, or the storage system level. In this chapter, storage system refers to block-based storage systems only; object-based storage interfaces such as NASD are classified as file system level interfaces because, with the excepton of directory hierarchy, object-based approaches retain most of the file system semantics. File system level versioning can be further subdivided and includes systems built at these levels: the system call interface (Alcatraz), the VFS interface (VersionFS), the physical file system (Elephant, CVFS, WAFL), distributed file systems (Google File System, Oceanstore), or object storage (S4). Storage system level versioning also has subtypes including block device driver (clotho) or iSCSI. In the case of the Frangipani [68] distributed file system, which is built upon Petal [39] (a distributed storage device), the versioning functionality is provided by the collaboration of the two.

The versioning system design is also affected by the historical version access interface, specifically, whether each version is object-based or (logical) partition-based, and whether the access to the old version is through a point-in-time rollback or through a point-in-time snapshot.

Versioning frequency directly affects system performance, versioning data organization and the historical data access interface. As versioning frequency increases, journaling and logging techniques are favored over checkpointing techniques, and historical access may become less direct.

As we can see, these three characteristics are closely related to each other and to versioning data organization and system performance. In general, high level versioning is characterized by a smaller scope of protection, less frequency, and a better historical data access interface. Low level versioning delivers a higher scope of protection and frequency but with a worse historical data access interface.

2.1.1 Versioning Systems at file system level

Physical file system level versioning provides great flexibility for optimization. The object store level is similar to the physical file system level except that there are no directory operations. S4 is the first fine grain versioning system built for security purposes at the object store level. CVFS, which solves the metadata versioning inefficiency in Elephant and S4, achieves good performance. While the performance of normal operations (i.e., access to current data) gets high priority in our repairable file system, CVFS tries to strike a balance between current data and old data access. A disadvantage of physical file system level versioning is its poor portability, which also affects performance in the long run because such a versioning system can not easily take advantage of the new advances in file system development. Another disadvantage is implementation complexity.

VFS level versioning systems sacrifice performance for better portability and reduced complexity. For example, due to the limitations of the VFS interface, VersionFS [50] generates a new inode and a new directory entry for each version of a file.

The file system syscall interface differs from the VFS interface mainly in that an object is referred by name (or file descriptor) instead of by inode. While the VFS interface is more natural for versioning, the syscall interface is well-defined and portable. In many systems, syscall can be intercepted without changing the OS kernel. Alcatraz [64] is such a user-level Isolation Execution environment for untrusted code. Strictly speaking, Alcatraz itself is not a versioning system because it maintains only a clean file system image and a delta image as result of the execution of the untrusted code. At the end of each execution, the user must decide immediately whether to discard the execution result or commit the result to the clean file system image before the next execution.

2.1.2 Versioning at storage system level

The main disadvantage of storage level versioning is the storage inefficiency in metadata versioning. Specifically, file system operations such as *setattr*, *create* and *delete* trigger updates to the inode and directory entry. The size of an inode or a directory entry is usually small, but the update at the storage level is at least one disk sector. The advantages to storage level versioning are as follows:

- It is simple and file system agnostic. Therefore its implementation can be relatively easy and portable.

- Storage level versioning does not have the cascading block update problem that earlier file system versioning had. The cascading problem occurs when file block updates are saved to a new disk location for versioning purposes. If this block is referred to by an indirect block, there will be an update to the indirect block which also needs to be saved to a new location. The updates propagate through the indirect block chain until they finally reach the inode. This problem is better addressed in the more recent CVFS versioning file system. At the storage level, this problem does not exist because the versioning is transparent to the file system. File system data updates do not propagate to indirect blocks at all.

- In recent years, the cost of disk space has dropped faster than the cost of disk bandwidth. This means that the disadvantage of storage level versioning over file system level versioning, in terms of disk inefficiency (considering both space and bandwidth factors), is constantly diminishing. In terms of bandwidth efficiency, storage level versioning is about the same as file system level versioning.

- There have been many file system optimizations designed to reduce the disk

load. Small in-place metadata updates can be delayed and coalesced for better performance. As a side effect this improves the space efficiency for metadata versioning. Therefore, file system performance optimization and storage level versioning space efficiency are two consistent goals.

This good news suggests that the metadata versioning inefficiency issue may not be severe enough to affect the feasibility of versioning at the storage level. This suggestion is confirmed by a simulation study using one day of the Harvard trace [17]. The result shows that for the workload of that day, 90% of the disk write is for file data updates; only 10% of the disk write is for metadata updates.

A seeming limitation of block based storage level versioning is that it is not possible to set different versioning policies for different files. While this may be a constraint for other versioning systems, it is consistent with the goals of our repairable system.

2.1.3 Discrete Versioning Systems

Elephant [56] is a versioning file system that defines versions when files close but does not distinguish updates within an open-close session. Elephant features a flexible versioning policy. It is the end-user's responsibility to set up the policy properly. Due to its relatively high versioning cost, Elephant does not expect a large scope of files to be protected with frequent versioning. Elephant extends the inode structure for each file into a log that holds multiple inodes, each corresponding to one version. The authors point out that metadata storage for versioned files can be 24 times larger than for non-versioned files because of the cascading problem.

VersionFS [50] is a versioning file system implemented using a stackable file system technique [73]. Similar to that of Elephant [56], the version is based on open-close sessions and the versioning policy is flexible. VersionFS also provides

a friendly interface for users to access old versions and to customize versioning policies. VersionFS still incurs non-negligible performance overhead - about 100% - when measured by Postmark.

WAFL [32] is a general-purpose high performance file system versioning product with snapshot support developed by Network Appliance. WAFL allows a limited number (32 originally) of snapshots. Its block bitmap contains a 32-bit entry for each block, with each bit corresponding to a snapshot. The snapshot is taken at a coarse granularity and the cost is amortized over many file updates. Its architecture is not scalable to allow increased snapshot frequency.

Petal [39] is a distributed storage service which provides automatically managed virtual disks. Virtual disk addresses are translated into physical disk addresses according epoch-number. Petal supports a snapshot feature that could provide fast, efficient support for backup and recovery. Petal itself is not a data versioning system. But it is easy to build a data versioning system above it using its snapshot feature.

Frangipani [68] is a scalable distributed file system built on top of Petal. It can directly generate backup versions with crash consistency. Crash consistency means that the snapshot image is not a consistent state for the file systems, though it is a consistent state of the virtual disk. When Frangipani wants to mount a virtual disk version with crash consistency, file system utilities (similar to the unix command *fsck*) can restore to a state with file system consistency. Frangipani can also flush all the updates to ensure that the virtual disk is at the file system consistency level after which it can mount the old virtual disk version without executing the fsck-like utility.

2.1.4 Comprehensive Versioning/Logging Systems

Wayback [14] is a user-level versioning file system for Linux. It is implemented with a user-level file system implementation toolkit FUSE(Section 2.2). For data updates, Wayback [14] uses an undo logging scheme similar to our first repairable file system prototype RFS-O(Section 4.1.2). For metadata updates, Wayback's versioning scheme incurs higher overhead than the logging scheme of our repairable file system. For old data access, Wayback supports an easy-to-use user interface. But for normal file system updates, the performance of Wayback is quite poor compared with that of traditional file systems. When compared with EXT3, the data read/write overhead ranges from -2% to 70% and the metadata update overhead ranges from 100% to 400%. In contrast, the performance of our third and fourth repairable file system prototypes(RFS-I and RFS-I+, Section 4.1.2) are comparable to that of an generic NFS server running on top of EXT3.

S4 [63] is a secure network-attached object store design for protection against malicious attacks. S4 logs every update and minimizes the space explosion. S4 uses log structured design to avoid overwriting of old data. S4 improves the inode/indirect-block logging efficiency by encoding their changes in a logging record. Performance evaluation shows that S4 logging is very lightweight without cleaning. But the cost of cleaning can be as high as 50%.

CVFS [61] is a kernel versioning file system focused on improving metadata-logging space efficiency. *Journal-based Meta-data* is used for inode/indirect-block updates and *multiversion b-tree* [70] is used for directory updates. In contrast, RFS performs logging at the NFS level, which preempts the space inefficiency problem. Both CVFS and our logging scheme favor current data access performance over historical data access performance but we favor it more: CVFS binds the access latency to historical data but its current data access performance cannot completely match non-versioning file systems. We provide matching performance for current

data access but the user may experience delays for very old historical data.

Clotho [25] is a storage abstraction layer that allows transparent and automatic data versioning at the block level. It strives to improve versioning metadata management efficiency. The performance measurements affirm the feasibility of block level versioning. Compared to Mariner's trail logging server, Clotho is geared more towards periodic versioning rather than comprehensive versioning; Clotho does not attempt to provide low-latency writes. Clotho's historical data access also involves more disk seeks than trail.

SVSDS [65] is a block level versioning system that performs selective, flexible and transparent versioning of disk data. It leverages the idea of Type-Safe Disks: by making small modification to the file system, the storage system can distinguish between file system data and file system metdata. The updates to file system metadata are always versioned. Other flexible policies can be enforced for file data, executables and log files. The evaluation shows that the block level versioning has minimal space and performance overhead. This is similar to our findings. The main difference between SVSDS and Mariner is that Mariner automatically does versioning for all storage blocks instead of only the selective ones. In addition, Mariner developped techniques to integrate block level versioing with traditional fault tolerance techniques.

ReVirt [16] is a virtual machine-based logging scheme that logs each non-deterministic system event(not just file updates). Starting from an initial checkpoint, ReVirt logs enough input information (keyboard, mouse, network traffic, etc.) to replay a long-term execution of the virtual machine. ReVirt logging is secure, general, and comprehensive but not file system oriented. If applied to a network file server environment, ReVirt logging will be a heavyweight scheme(with a worse case overhead of 58%).

David Patterson [11,20] advocates a radical shift from a performance-dominated

research focus to what he refers to as Recovery Oriented Computing (ROC). With ROC, the focus is on improving system availability by reducing MTTR and eventually the overall system ownership cost. One of the applications that was pursued, an undoable email system [20], shares a similar approach with RFS but focuses specifically on email message protection rather than on file system data in general.

CDP(Comprehensive Data Protection) is a hot topic in recent years. There have been many start-ups on this frontier. Some of them have gone out of business(Mendocino). Many have been acquired by larger companies (Revivio by Symantec, Kashya by EMC, Alacritus and Topio by NetApp, TimeSpring by Double-Take, Storactive by Atempo, FilesX by IBM, LassoLogic by SonicWALL, ConstantData by BakBone, Availl by GlobalScape or XOsoft by CA). The independent, pure-CDP players now include only Asempra and InMage Systems. Below we selectively introduce CDP products.

Enterprise Rewinder [72], is a file system level CDP product suite of Xosoft, Inc. The versioning could be either by operation or by open-close session. The interception of file system traffic is implemented as a file system-level operation filter. The products are application aware. They are customized to work with Exchange, Microsoft SQL, Oracle, or NFS, respectively. They can achieve application level consistency and automatic application failover. Data protection is through undo logging. Redundant hardware is used.

RealTime [48], was a product from Mendocino, Inc. It provides comprehensive block-level versioning. RealTime has an efficient logging data structure (both redo and undo logging) for fast recovery speed. It can restore a 1-tera to 1-petabyte database in 20 minutes.

RTP(RealTimeProtection) is a CDP product originally from Revivio. The software was later integrated into a Veritas backup product [66]. RTP provides instant (close to zero) point-in-time storage image upon request. RTP bears similarity to

Mariner: the write requests are duplicated at the storage client to both primary storage and backup/versioning storage. The main difference is that Mariner naturally supports N-way mirroring and M-way logging depending on system performance and reliability requirements. Moreover, in Mariner the logging nodes and the primary nodes are closely integrated and interchangeable (primary storage can do logging too, however with lower performance); the write requests are committed to logging storage first, which provides a low-latency write. In contrast, RTP is usually configured with one set of primary storage and one set of backup/versioning storage. RTP uses a side-band architecture; the I/O processing on the versioning/mirroring node is removed from the critical path of I/O processing. However RTP still pays a small cost of dirty region block logging on the critical path for write requests. One good feature of RTP is that its point-in-time image is fully read/write capable, which allows file systems/databases/applications to perform their own recovery procedure on a block level image with crash-consistency. On Mariner, instead of running a file system fsck, we provide a faster in-house fsck to recover a historical snapshot to consistent state [43].

Finally Table 2.1 summaries the design choices of the versioning systems described in this section.

2.2 User Level File System Implementation Toolkits

Fist [73] is a language for generating stackable file systems. Fist can output kernel code for Solaris, Linux and FreeBSD, and this provides some amount of portability. Its stackable file system attempts to strike a balance between portability and efficiency. The stackable file system exploits the VFS layer. It manipulates syscall

System	Level	Granularity	Old Data Access	Frequency	Portability
CVS	application	object	Direct	D	high
Elephant	file system	object	Direct	D	low
VersionFS	file system	object	Direct	D	near low
S4	object store	object	Direct& Rollback	C	low
CVFS	file system	object	Direct& Rollback	C	low
WAFL	file system	partition	Direct	D	low
WANsync	syscall interface	object	Direct& Rollback	C	low
RFS	file system	partition	Rollback	C	high
Petal	storage	block device	Direct	D	low
Clotho	storage	block device	Direct	Near C	near low
SVSDS	storage	block device	Rollback	Partial C	near low
RealTime	storage	block device	Direct& Rollback	C	unknown
Revivio	storage	block device	Unknown	C	unknown
Mariner	storage	block device	Direct& Rollback	C	near low

Table 2.1: The design choices of various versioning file/storage systems. ”C” stands for ”Comprehensive”, ”D” stands for ”Discrete”

parameters before calling the underlying physical file system interface and manipulates returned values before returning to the application. In comparable fashion, RFS logging works between an NFS client and an NFS server by modifying NFS requests and replies.

David Mazières developed a toolkit [44] to facilitate UNIX file system extensions. The toolkit exposes the NFS interface, allowing new file systems to be implemented and ported at the user-level. The NFS protocol gives the software an asynchronous, low-level interface to the file system that can greatly benefit the performance, security, and scalability of certain applications. The toolkit uses an asynchronous I/O library to build large, event-driven programs. The default configuration of this toolkit has one NFS proxy daemon on the client side and another on the server side. Our user-level file update logging uses a similar technique but incurs less overhead because we have only one NFS proxy daemon on the server side.

FUSE [67] is a package that facilitates user-level file system development. It redirects file system related system calls from the kernel to the user space. FUSE is implemented as a kernel module available on Linux systems. It is simple, stable, and secure. It is very useful in scenarios where functionality is preferred over performance. According to the measurements from Wayback [14], which is a FUSE implementation, FUSE causes significant performance overhead (around 20% to 50%).The repairable file system (RFS) project had user-level implementation but did not use FUSE. The reason is because FUSE is developped after RFS. Moreover, the overhead of FUSE would have been too high for RFS anyways.

2.3 Other Application of Logging Techniques

LFS [55] uses append logging schemes similar to those of repairable file and storage systems but for somewhat different purposes. Accordingly, there are different trade-offs. The main purpose of ”append logging” in repairable file system and storage systems is to preserve old data. In LFS, updates are written to new locations in large batches to improve small write performance. Cleaning is essential and more frequent in LFS to maintain disk access efficiency. In repairable file system and storage systems, the data is kept for longer periods and metadata manages different versions of data. The garbage collection is not as aggressive.

Journaling file system logs metadata updates to maintain file system metadata consistency and for fast recovery upon crashes. The log record in repairable systems can serve the same purpose in addition to the capability of providing comprehensive versioning.

Trail [13] is a track-based disk logging technique that can reduce the latency of synchronous disk writes to the level of 1.5 msecs. Trail also provides fast recovery upon failures. For low-latency writes, the data should be written where the disk head is at that instant. Commodity disk drivers do not provide such flexibility. Trail developed techniques to predict where the disk head is. Its logging scheme does not yield a contiguous log. Trail developed recovery software that can identify where, in each track, the log records are, and can quickly locate the most recent log records without scanning the entire log device. Mariner’s trail node is extended from the original trail node, but has a more complex log device usage pattern and consequently needs more complex recovery software.

2.4 Related Works in Dependency Tracking

2.4.1 Taser

Taser [29] is an intrusion recovery system that is very similar to our repairable file system - each with its own advantage and complementing each other. Taser has better and more flexible dependency analysis policies and an optimized recovery algorithm. RFS has a more efficient logging mechanism integrated into its versioning file system. RFS also provides better support for NFS.

Regarding dependency analysis, Taser [29] can be customized to specify whether to consider or ignore IPCs, signals, file attribute reads, file name reads, and file content reads. In addition, there is a "whitelist" that specifies the files whose write operations are also ignored. RFS considered but did not actually implement support for policy customization. Instead, RFS provides only one practical policy that is neither too conservative nor too aggressive. Taser also optimized the recovery algorithm, which includes a special conflict resolving mechanism. RFS's recovery algorithm is simpler but it may run more slowly and there is no requirement for a special conflict resolving component.

Regarding system call logging or auditing, Taser [29] uses the Forensix system [28] as its auditor to log all kernel operations related to dependency analysis including process management, file system syscalls, etc. Its evaluation results show high auditing overhead. This is similar to our findings in the first RFS prototype [75]. The main reason for the high auditing overhead is that write request logging is very expensive due to the large data size. In their future work, Taser pursued integration with journaling and versioning file systems to improve scalability and reduce logging overhead. RFS implemented its own syscall logging module. In the second RFS prototype [76], a comprehensive versioning file system is integrated with logging and recovery capabilities. Taser also disclosed that it currently

does not support network file systems such as NFS because of the limitation of its auditing module. In contrast, RFS is built on top of NFS by design.

2.4.2 Post-intrusion database damage repair

There have been several research projects on post-intrusion database damage repair. Ammann et al. [22,23] proposed a transaction models and protocols that allow normal transactions to proceed against a database that had sustained partial damage from an intrusion event. The proposal is largely a theoretical exercise without detailed system-level considerations.

Peng Liu [23, 24, 41] described a concrete intrusion-tolerant database system that can continue its transaction processing service even in the presence of active attacks. It logs database updates as SQL-based transactions. Instead of tracking inter-transaction dependencies at run time, it identifies them at repair time by analyzing the SQL log. To support continuous operation, this system incorporated several schemes to detect, assess, and repair damaged databases on the fly without completely halting incoming transaction processing. However, during repair time, the effective throughput of DBMS is degraded. This high-availability design is similar to Mariner. RFS takes a simpler "stop and repair" model, which is consistent with common system administration practice after an attack.

Fastrek [53, 59] is a dependency tracking approach for adding intrusion resilience in database systems. It shares similar principles with RFS but is designed for DBMS. The main difference is that RFS maintains the undo log but Fastrek utilizes the undo log maintained by DBMS for each transaction for abort purpose. Another difference is that Fastrek maintains an inter-transaction dependency graph on the fly to achieve fast recovery. File system operation granularity is smaller than transaction granularity. For RFS, the overhead and performance penalty incurred with such an approach would be too high. During normal processing states, RFS

keeps raw log records only. During repair time, only those records which are related to the corruption are analyzed and used to build the dependency graph.

2.5 Complementary Data Protection Techniques

File system backup protects data for a potentially longer duration but in much coarser granularity than our repairable system. The backup may use either operating system utilities or commercial software such as Amanda [1]. File system replication or mirroring (such as SnapMirror [21]) provides protection against disasters and failures but not against intrusions and user mistakes. Documentation versioning control systems(such as CVS [30]) gives users full control over the protection of a smaller number of important files. It also provides easy access to older versions. Tripwire [2] is an auditing tool that keeps track of changes to important system configurations files and is capable of recovering files to a known good status when soft failures occur. These changes are usually rare and small but could have a significant impact up to and including system malfunction.

Chapter 3

Dependency Tracking

In this chapter, we describe the framework of dependency tracking in a repairable file and storage system. Conceptually, the output of dependency tracking is a set of file system objects whose state needs to be restored to a particular version. This output is provided as input to the recovery algorithm. The recovery for the repairable file system is straight forward. For the repairable storage system, we build a user level versioning file system on top of the versioning storage system, and then run the recovery algorithm on top of the user level versioning file system. The details of the recovery algorithm are described in Chapter 4 and Chapter 5 respectively.

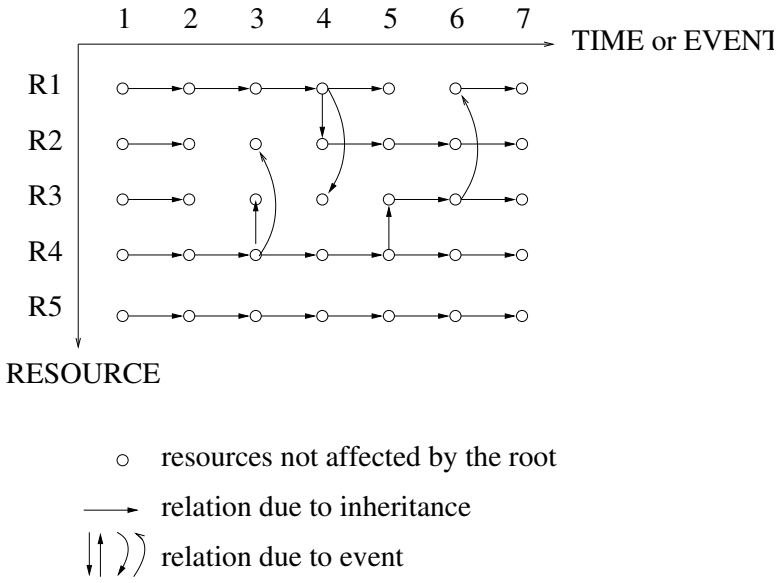
The goal of dependency tracking is to discover all the data updates involved in a soft failure or useful work. The distinction between soft failure and useful work is from the user's viewpoint. From a system viewpoint, both involve tasks that change the system state. Each task may involve many resources, such as processes, data, and userids, but the task is usually initiated by only a few of them, called "root resources." While it is usually not possible to identify all the resources in a task, it is much easier to pinpoint only a few "root processes." Other resources can be discovered through the dependency relationships with the root resources. These

dependencies are caused by certain system events. The discovery of all resources involved in a task is a transitive closure problem. Next we give the formal description of the dependency tracking framework.

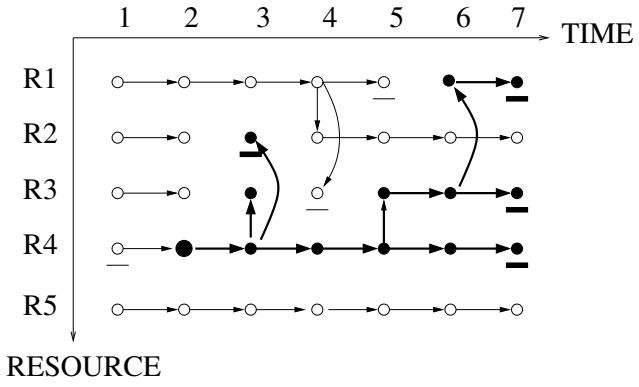
3.1 Dependency Tracking Framework

Figure 3.1 (a) is a directed graph G=$< V, E >$ that represents the state change of a system for a certain duration. The system is made of a set of resources RS=$R1, ..Rm$ and a set of events TS=$T1, ...Tn$. Each event has a unique timestamp. Each vertex $V =< R, T >, R \in RSandT \in TS$, represents the snapshot of resource R when event T happens. To better illustrate this, we arrange the vertices in rows and columns. Each row represents a resource and each column represents the system snapshot when an event occurs. An event correlates with dependency relations among vertices in the same column, shown as the vertical edges $<< Ri, T >, < Rj, T >>$. All vertices in G (except those in the first column) have exactly one incoming edge. Any vertex $< R, T >$ which does not have an incoming vertical edge inherits status from its previous snapshot $< R, T - 1 >$, shown as a horizontal edge $<< R, T - 1 >, < R, T >>$. In practice, resources may have a finite lifetime and may be dynamically added or removed. For the sake of simplification, we assume that they always have a snapshot at any time T. If the lifetime of a resource R is from Ti to Tj, then the vertices $< R, T > |T < Ti$ represent the pre-born status, and $< R, T > |T > Tj$ represent the after-delete status.

Given a set of resource snapshots $Vr = \{< R, T >, ...\}$ selected by the user as **task roots**, the transitive closure of Vr represents all the resource snapshots that are involved in this task. The bold vertices and edges show Vc, the transitive closure of Vr, and the edges that connect them. In the example, Vr contains only one task root $< R4, T2 >$, but it could have more.



(a)



(b)

Figure 3.1: Dependency Analysis

The output of dependency analysis is a set of vertices that will be the input for the recovery algorithm. The output could be in one of two forms depending on the nature of the task. If the task roots represent a soft failure , the output is $Vundo$, which represents the last clean snapshot of corrupted resources. If the task roots represent a useful job, the output is $Vredo$, which represents the last good image of the resources involved in the useful work. This is, in formal notation, as follows:

$Vundo = \{< R, T > | T < Tmax, < R, T > \in (V - Vc), \forall < R, T' > \in (V - Vc), T >= T'\}$, $Vredo = \{< R, T > | < R, T > \in Vc, \forall < R, T' > \in Vc, T >= T'\}$.

Note that $Tmax$ is the last event that happened to the system. Correspondingly, $< R, Tmax >$ represents the current image of resource R. In Figure 3.1(b), $Vundo$ vertices are shown with thin underlines and $Vredo$ vertices are shown with bold underlines. This formal description can be applied to system resource repair in general. In this thesis, we are only interested in data repair. Accordingly, the final output will filter out non-data vertices, such as vertices for processes.

3.2 Design choices

Given such a framework, to build an actual system, we must identify the vertices (resources, events) and the vertical edges (dependency relations). This boils down to five specific questions:

- What are the resources that constitute a task?
- What are the relations that create dependency among two resources?
- What are the events that cause such relations?
- How to log such events?

suitable tasks soft failure/ useful job	**relations & resources**	**event**	**where to log the event**
both	parent-child (P→P)	fork	syscall
both	same process group, session, user (P→P)	fork	syscall
both	IPC(P→P)	signal,pipe, shared memory,lock, semaphore,socket	syscall
soft failure	network communication ({PUHL}→{PUHL})	send, recv	syscall, network
soft failure	exec program (D→P)	exec	syscall
soft failure	data access (D→P)	read	syscall, network data server
both	data update (P→D)	write	syscall, network data server
both	customized ({PUHLD}→{PUHLD})	reuse existing events	reuse existing logs

Table 3.1: Design space. P(rocess), U(ser), H(ost), L(an) and D(ata) is the resources. Each row is a dependency relation (X→Y) between two resources X and Y. $\{X_1..X_p\} \rightarrow \{Y_1...Y_q\} means \{X_i \rightarrow Y_j | 1 \leq i \leq p, 1 \leq j \leq q\}$

- How can users pinpoint the task roots?

These questions are not independent from each other. The answer to one question may affect the answer to another. The first and second questions are the most important ones. The resources and relations should be extensive and at a granularity fine enough to distinguish a soft failure from a useful job. In the mean time the resources should be at a (not too low) level so that it is feasible for users to pinpoint the root resources. The events that cause the dependency relations should be "trackable" both in terms of implementation feasibility and performance overhead.

Process is a straightforward candidate for resources. We can use a set of processes to represent a useful job or a soft failure , and redo or undo their operations in the repair phase. Two processes could be related from the parent-child relationship. They may also be related in a producer-consumer relationship where one process generates some information to be consumed by another process, such as IPC communication, network communication, file system data read/write, etc. These relations can be tracked at the system call interface level. Syscall tracing has been used extensively for various purposes, dependency tracking is yet another application. Syscall tracing requires modification to the host.

When fine granularity dependency tracking is not available, there can be resources at coarser granularity such as "user" and "host." The relations among user and host are much simpler. The dependency can only be due to network communication and file system data read/write operations. In the WAN environment, the granularity of resources can be even coarser and based on the "local area network." It is arguable that very fine grain resources such as "thread" may also be used in dependency tracking, but we haven't considered it seriously.

In the above description, data are considered as media for two resources to have dependencies. Another way of expressing this kind of dependency is to consider data a resource, and to have two extra relations: data read and data write. This way,

the data dependency among two processes can be expressed in two dependency relations: first the data is affected by a process, then another process is affected by the data. This greatly simplifies the dependency analysis. Data can be "root resources," too. For example, an intrusion can be detected by the fact that one important file got modified. How the file got modified is probably not important or can not be identified. What is important is that many subsequent operations are affected by this corrupted file and need to be rolled back.

Identifying "root" resources Vr can be tricky. A general solution is difficult and out of scope for this dissertation. Taser [29] proposed some practical strategies to alleviate the problem. Here we provide only sample solutions for some simple and common scenarios. A user mistake may consist of one or more commands, each being executed through one or more processes. To undo the effect of user mistakes, all file system updates from these processes should be rolled back. If the user mistake occurs during the interaction with an application process, the subsequent updates from this process must be rolled back. In such a case, a timestamp is required to describe the moment when the user mistake occurs. In malicious attacks, the intruder may hijack an existing process and then spawn new ones. All updates made by the hijacked process after it is hijacked and by the new processes must be rolled back. Similarly a timestamp is required to specify the hijack time. In the case of untrusted software, the software will be run as one or more processes. If the software is proven to be unsafe, all updates from these processes should be rolled back. Useful jobs can be described in a similar way as a set of processes and associated timestamps.

Table 3.1 illustrates the dependency tracking on a UNIX like system. It lists the potential resources, possible dependency relations, the events that may cause these dependencies, and where to track these events.

The dependency relations to be considered for soft failures and for useful jobs can be different. For example, if the root resources represent a soft failure , a process that reads the polluted data might be presumed contaminated and therefore included in the transitive closure. On the other hand, if the root resources represent a useful job, a process that reads some data produced by the root resources is not necessarily considered part of the useful job.

For maximum flexibility, in addition to general dependency relations, we allow customized dependency relations at repair time for individual tasks. This is shown as the last row in the table. The events log must be able to support all possible dependency relations.

3.3 False Positives and false negatives

The output of dependency analysis, $Vredo$ and $Vundo$, may have both false positive and false negative results, either because the root resources Vr are not identified precisely, or because the dependency relations used in the analysis are inaccurate. These two issues are not completely independent: if more resources are included in the root resources, there is no need to discover them through dependency relations. The extreme example is that no dependency relationship is required if $Vr = Vredo$ or $Vr = Vundo$. We have mentioned such scenarios in the introduction, where dependency analysis would not be required if users have complete knowledge about the updates involved in a task. On the other hand, if the dependency relations are comprehensive, users need to pinpoint very few root resources.

Table 3.1 does not attempt to establish a standard. Rather it serves only as guidance for the design of dependency tracking. The relations listed in the table are only strong hints of dependency. It may be neither complete nor necessary. First we discuss false negatives. Processes on the same machine share much common

system status. In addition to explicit communication through the IPC or loop back network, there can be implicit communication. Consider, for example, one process allocated a lot of disk space, which then triggers aggressive garbage collection by another process, which consequently deletes many files unnecessarily. Such dependencies cannot be tracked unless there is a dependency relation that says that any process which measures free disk space depends on all processes that allocate or de-allocate any disk space. But then this dependency would cause false positives in most scenarios. Hopefully such a sample scenario is not common in practice. Another source of a false positive is implicit communication through users. Users may provide input to a process according to the screen output of another process. Unless the user marks both processes as root resources, it is very hard to track their dependencies.

False positives can occur, too. Firslty, they can be occasioned by false data sharing, which is a common problem in distributed file systems and SMP (share memory multiprocessor) systems. A sample case is access to the /etc/passwd file. A compromised process can modify the /etc/passwd to add a new user account or disable the root passwd. Later many other good login processes may access other lines of the /etc/passwd file. While these processes are probably not affected by the attacker, block-based data dependency will assume they are compromised. Secondly, even true data sharing, IPC, and network communication are not sufficient to prove the dependency between two processes. For example, process A may send process B a signal or some compromised network data, but process B may have a strong integrity check and therefore the signal is ignored and the data is discarded.

The consequence of false positives to useful jobs is that corrupted data remains in the system. The consequence of false negatives is that some useful work will get lost. The consequences of a soft failure are just the opposite. In practice, users

may chose a redo-based repair (identify useful work) or an undo-based repair (identify soft failure) depending on 1) which would have more accurate results, and 2) whether it is more important to preserve all useful work or it is more important to avoid corrupted data.

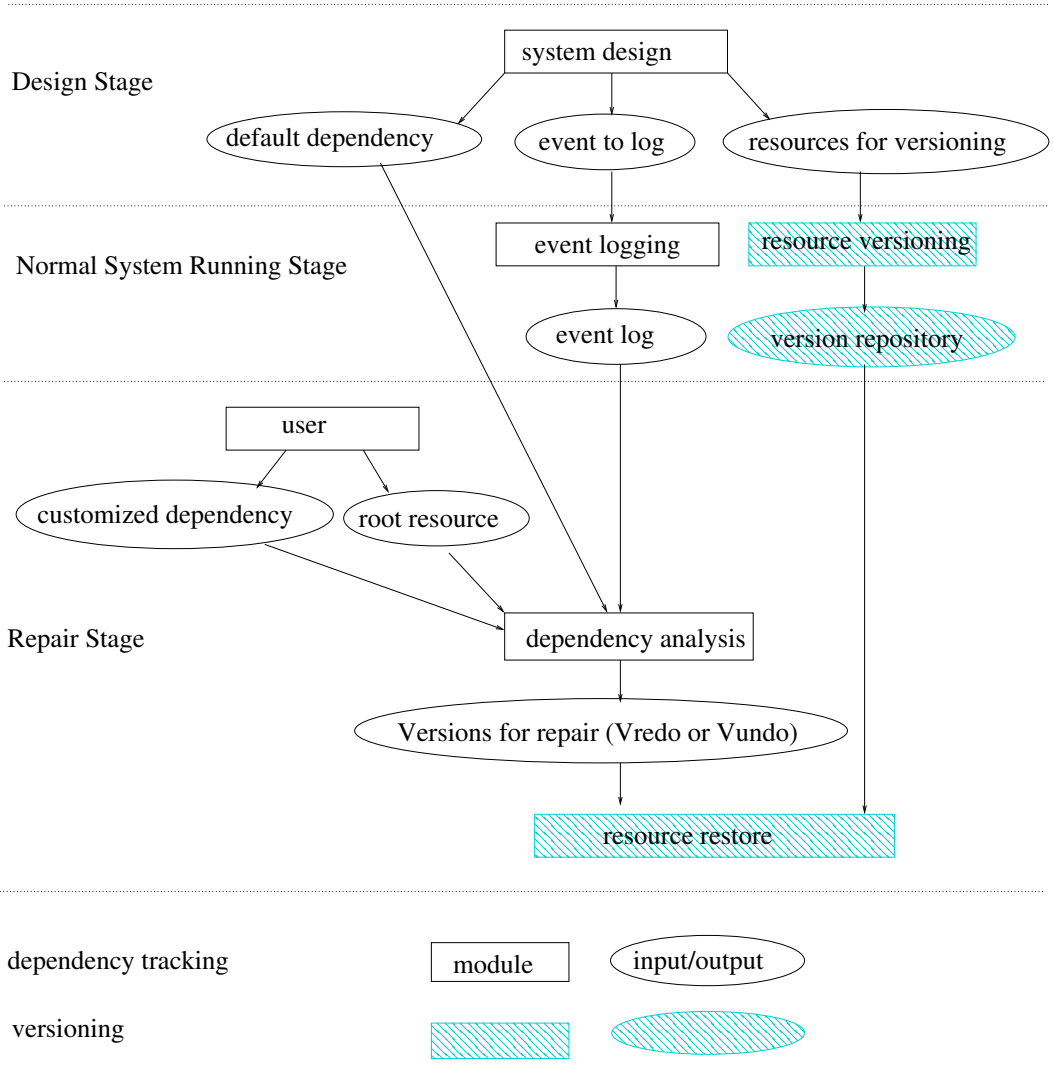


Figure 3.2: Dependency Tracking Overview

3.4 Summary

Figure 3.2 gives an overview of dependency tracking in a repairable system. The default dependency relationships, the events to be logged, and the resources for comprehensive versioning are decided at system design stage. Events are logged during normal system processing. The customized dependency relations and root resources are specified at repair time. The events log is of the format: $< timestamp, eventcode, R_1, ..R_p >$. Both default dependency relations and customized dependency relations are of the format $< eventcode, R_1, ..., R_p > \rightarrow \{R_x \rightarrow R_y | x, y \in \{1, ..., p\}\}$ Upon soft failure , the dependency analysis algorithm incorporates the default dependencies, the customized dependencies, and the event log to build the system state evolution graph as shown in Figure 3.1(a). Then it calculates the transitive closure from the root resources specified by the user. The output of dependency analysis is used for recovery. $Vredo$ and $Vundo$ specify the final valid versions of the resources, which can be retrieved from the version repository.

Chapter 4

Repairable File System

The Repairable File System (RFS) is a general framework for protecting networked file servers from irrevocable damage caused by errors or attacks. It has two operating modes: In the normal mode, RFS maintains a file update log and an inter-process dependency log. In the repair mode, RFS first determines the exact extent of system damage, and then performs selective roll-back of those data blocks that are considered contaminated. There is a limit on how much old data could be maintained, which decides the **protection window** provided by the repairable file system. [1]

RFS focuses on speeding up the system repair process after an intrusion. It does not perform intrusion detection by itself. At repair time RFS assumes that the processes that start an attack are already identified, either through an intrusion detection system, or through manual inspection of the system log. Given these processes, RFS can partially or completely automate the subsequent damage repair. In addition to the goal of fast and fine-granularity repair with minimum data loss,

[1]The protection window is the maximal interval between the time when an attack occurs and when it is detected such that an RFS installation can ensure lossless recovery.

RFS is designed with performance, flexibility, and portability in mind.

- RFS should not introduce significant logging overhead. The goal is to match the performance of standard file server systems, such as an NFS.
- RFS should be a flexible system that can work well with existing legacy file servers.
- The system architecture of RFS should be sufficiently modular that the components independent of the underlying network file access protocol (e.g., NFSv2) should be reusable across different network file access protocols.

Our prototype implementation of RFS is on top of NFS, with the following assumptions:

1. There are no intrusions and user errors occurring on NFS server.
2. The data on the NFS server can only be accessed by NFS client, i.e., there is no local access from the NFS server. This means that intrusions and user errors only occur on NFS clients and all file updates are through the NFS protocol.
3. The system call log (used for dependency analysis and collected on NFS client) cannot be corrupted.

In general it is efficient to do logging at the NFS command interface. It leads to a more compact log and simpler design because one NFS operation could map to multiple inode/indirect-block/data-block updates. For example, an NFS *create* request triggers the following local file system operations on the NFS server: (1) a new inode is created, (2) the file name is added to the parent directory, (3) the parent directory file may be expanded with a new block, (4) the block pointer of

the parent directory file is added to point to the new block, and (5) the attributes of the parent directory's inode are updated. As a consequence, a *create* operation may generate multiple log records if logging is done at the inode/block level. But in RFS it generates only one log record.

4.1 Overview

In this section we start with the design issues of a repairable file system. We have built four different prototypes (RFS-O/RFS-A/RFS-I/RFS-I+) for the repairable file system. We describe their basic architectures and characteristics. We also discuss the common software components shared by different prototypes. The four prototypes differ mainly in the logging scheme, which is discussed in detail in subsequent sections.

4.1.1 Design Issues

The dependency analysis has been discussed in-depth in Chapter 3. Repair is a relatively simple issue that naturally follows the design of logging and dependency analysis. In this section we focus on the logging design. The main research questions are:

- Should we use redo logging or undo logging?
- What information needs to be logged?
- In a distributed environment that consists of an NFS server and multiple NFS clients, how to consolidate the logs from multiple nodes?
- How to log the data in the write request?

In the event of a failure, forward recovery resets the system state to the last clean snapshot, and selectively applies *redo* operations to retain uncontaminated information. Backward recovery, on the other hand, rolls back the system from the current state by undoing contaminated operations until the entire system is clean. Backward recovery avoids the overhead of checkpointing, and the repair time is proportional to the interval before intrusion or mistakes are detected. Forward recovery avoids the overhead of reading the prior image to construct undo records, and the repair time is proportional to the interval between intrusion time and the time of the last clean snapshot. We assume that with current intrusion detection techniques, intrusions and errors can mostly be detected within a short period of time after their occurrence. RFS uses the backward recovery approach, and logs file system updates to an undo log. As we may observe later in this chapter and in the next chapter, the distinction between forward recovery and backward recovery is blurred if the underlying file system or storage system support comprehensive versioning.

System call logging provides both the information for the dependency analysis(Chapter 3) and the data for repair. The information to be logged includes the system call type, the timestamp, the system call parameters, and sometimes, the data. Most system calls do not contain data, except file system read/write and network send/receive. For dependency analysis, we do not need to log the data. But for file system recovery, we need to log the data in the *write* system call. Since we are not doing application level recovery like [37], the data in network send/receive is not needed.

Since files are stored on the NFS server and intrusions/user errors occur on NFS clients, the logging needs to be done on both the client side (called **syscall log**) and the server side (called **file update log**). The dependency analysis algorithm(Chapter 3) requires a global order among all syscall log records. It is not

straightforward to impose a global order among the syscall logs from multiple NFS clients, especially with NFS client caching, since there need not be a one-to-one mapping between a file system syscall on the NFS client and an NFS request to the NFS server.

In the absence of a global timestamp, RFS solves this problem by inserting an **RPC message entry** to both the client-side syscall log and the server-side file update log. On the NFS client, the entry is added before a request is sent to NFS server. On the NFS server, the entry is added when a NFS request is received. By analyzing the syscall log entries immediately before an *RPC message entry*, RFS can determine the process or processes responsible for a particular NFS request. The RPC message ids serve as synchronization points between the syscall logs and the file update log. Since RFS assumes that two NFS clients can only affect each other through NFS data dependency, this coarse granularity synchronization is enough for the dependency analysis. If we also consider network dependencies, the network communication itself can serve as extra synchronization points. The intuition behind this is that if there is a communication that might cause dependencies, the communication itself serves as a synchronization point. If there is no communication for a certain period of time, we wouldn't care about the global ordering because there would be no dependencies.

Logging the data in the write request is tricky because it incurs a very large logging overhead. Most syscalls and NFS requests do not take more than 128 bytes and the logging overhead is negligible. However, the data in the write syscall is at least 4K and often bigger, up to 64K. Storing data directly in the log record incurs a heavy disk load. With undo logging, the overhead doubles because we need to retrieve and store the prior image data. This approach is called **overwrite-logging**. One way to avoid the overhead of reading/writing the prior image is to use a comprehensive versioning file system that keeps all versions of data and only refers to

the location of the data in the log record. We call this approach **append-logging**. We have experimented with both logging approaches in different prototypes.

4.1.2 System Architecture

We have experimented with two system structures. The **decoupled** structure has a legacy file server. The file update logging is done by a separate logging server. There is a *traffic interceptor* dispatching the NFS traffic in front of the legacy file server and the logging server. The *decoupled* structure can take advantage of the benefits of a legacy file server, such as superb performance and reliability. This structure also make it easier to apply the RFS framework to different network file system technologies, such as SUN's NFS protocol [4] or Microsoft Server Message Block (SMB) [3] or CIFS [38] protocol. The logging server is invisible to both the network file server and its clients – it only monitors NFS traffic. There is little chance for it to be attacked. In the *decoupled* structure, the read performance and write latency are decided by the legacy file server, and the write throughput is decided by the slower of the legacy file server and the logging server. In our system, if the logging server is slow and cannot keep up with the legacy server, eventually the *traffic interceptor* will stop dispatching requests.

In the **integrated** structure, instead of the legacy file server, an NFS-Processor is added to the logging server to handle all NFS requests. The *traffic interceptor* is no longer needed. The *decoupled* structure has better performance isolation while the *integrated* has reduced hardware cost.

We have built four prototypes with different logging approaches and system structure, as shown in Table 4.1. In RFS-O, to compose the undo record, the logging server needs a *mirror file system* to serve the prior image. In RFS-A, there is also a *base image* to help with the append logging. There is no combination

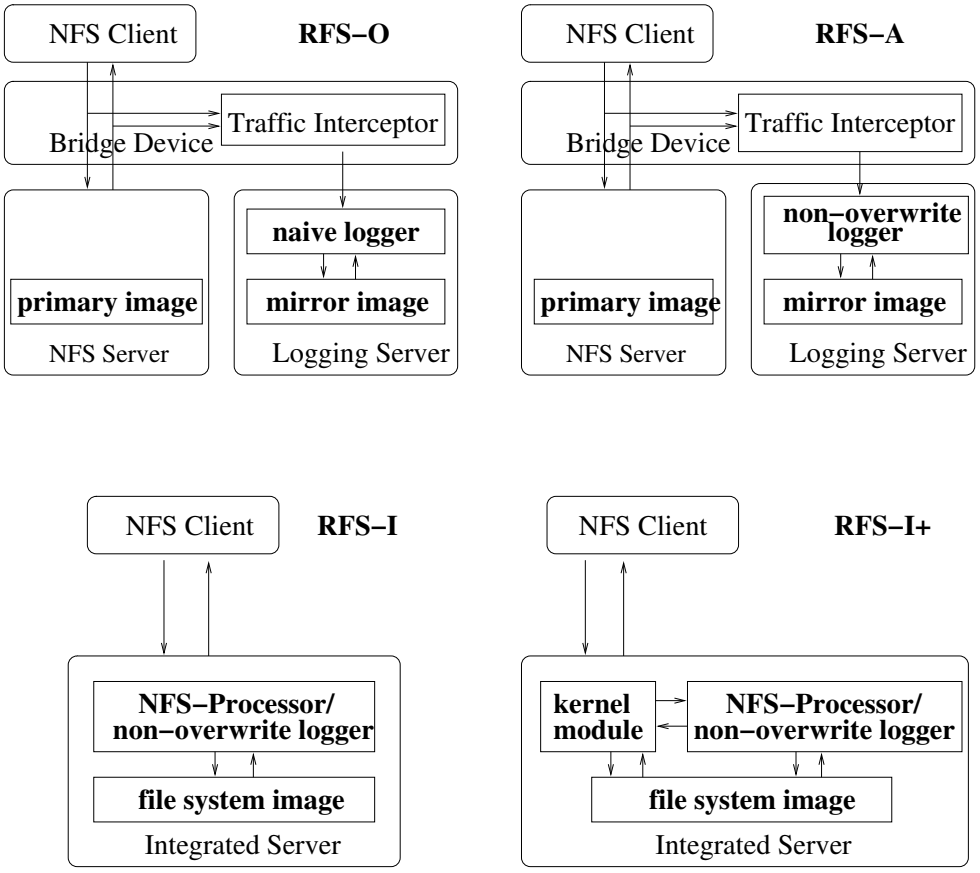


Figure 4.1: System architectures of four RFS prototypes. The *traffic interceptor* intercepts NFS requests and responses. It runs on a bridge device in front of the protected NFS file server. The user-level *non-overwrite Logger* records all file updates. Both RFS-O and RFS-A require separate nodes for traffic interception, file update logging, and NFS request processing. RFS-I integrates NFS processing and file update logging into one host and eliminates the interception device. RFS-I+ incorporates an in-kernel packet interception mechanism to reduce context switch and memory copy overhead.

	overwrite-logging	**append-logging**
decoupled	RFS-O	RFS-A
integrated		RFS-I, RFS-I+

Table 4.1: Four RFS prototypes

of overwrite-logging and integrated structure because the overhead of overwrite-logging turned out to be very high. Even with the decoupled structure, we observed that the write throughput dropped significantly. The integrated structure would perform even worse. In RFS-A and RFS-I, the file update logging is implemented at the user-level, incurring some memory copy and context switch overhead. RFS-I+ is largely the same as RFS-I, except that it has additional optimizations to reduce the memory copy and context switching overhead. Figure 4.1 shows the architecture of the four prototypes. Figure 4.2 shows the data structures that are needed in each prototype.

4.1.3 Contamination Analysis and Repair

The dependency analysis and repair run in the repair mode. Given a set of client syscall logs and the server file update log, the dependency analysis algorithm coalesces them into a single log by the *RPC message entries*(Section 4.1.1. Both the syscall logs and the file update log are stored on the logging server. Among many choices in dependency analysis(Chapter 3), we chose a set of practical and relatively simple policies. The dependency analysis outputs a set of undo records. The repair module executes these undo records to restore system data status. The repair component will vary slightly with different system structures and file update logging schemes.

In RFS, the dependency analysis is damage oriented rather than useful work

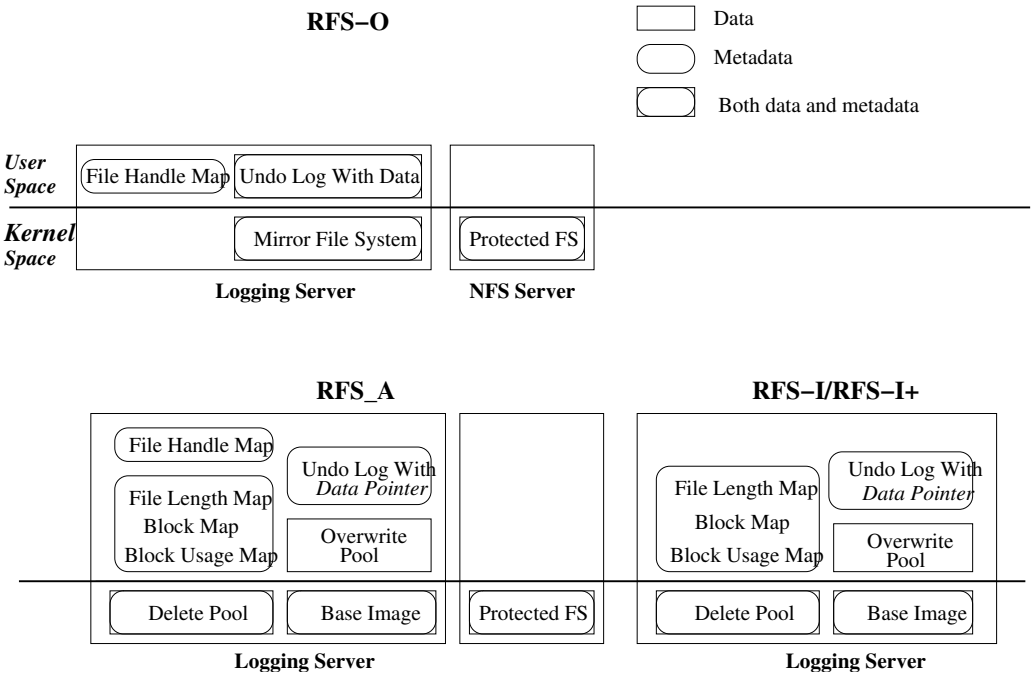


Figure 4.2: This figure shows the data structures of the four prototypes. It also shows whether they are maintained at user-level or at kernel-level. Data refers to the regular file blocks. Kernel level meta-data refers to superblock, inode, indirect block and directory, which are maintained by the underlying file system. User-level metadata includes various versioning data structures maintained by the user-level file update logging module.

oriented(Chapter 3). Therefore we also call it contamination analysis. The contamination analysis considers all mutable file system syscalls (and corresponding NFS requests) unless they only affect the last access time attribute. One reason is that these operations are very frequent and cause a blow-up in the log size. Another reason is that we suspect these operations will lead to a lot of false positives. As future work, this conjecture needs to be verified. Another option is to provide contamination analysis at different security levels. We always log these operations, but only use them in the analysis at a high security level.

RFS distinguishes between a *contaminated file* and a *contaminated file block*. If a file is contaminated, all its blocks are contaminated. The converse is not true – even if all the blocks are contaminated, a file may still not be contaminated (the attributes are clean). A file created by a corrupted process is a contaminated file. If a corrupted process writes into a file block, only that file block is contaminated. With this distinction, RFS considers a process contaminated if it

- is a child of a contaminated process,
- reads contaminated file blocks, or
- performs any operation that depends on the existence of a contaminated object. This includes read, write, get_attributes, and set_attributes calls on the contaminated object. It also includes any operations on the objects that are descendants of the contaminated object in the file system hierarchy.

According to the above rules, if a process just writes to a contaminated file block, the process is not considered contaminated. However, if a process writes to any block or manipulates the attributes of a contaminated file, the process becomes contaminated. The rationale of the third rule is that processes that touch contaminated files could not have continued because contaminated files will be deleted in the repair process.

Conceptually the contamination analysis assumes that an individual process is the unit of repair. Update operations of a process are either all preserved or all removed. In practice, the output of the contamination analysis is a set of undo records from the file update log. It is essential for RFS to determine the initiating process of each NFS request. Unfortunately, NFS requests only contain user ID. Again the *RPC message entry*(Section 4.1 helps to determine the process or processes that are responsible for a particular NFS request. RFS sends each undo operation as a normal NFS request. As a result, *the undo operation is also undoable*.

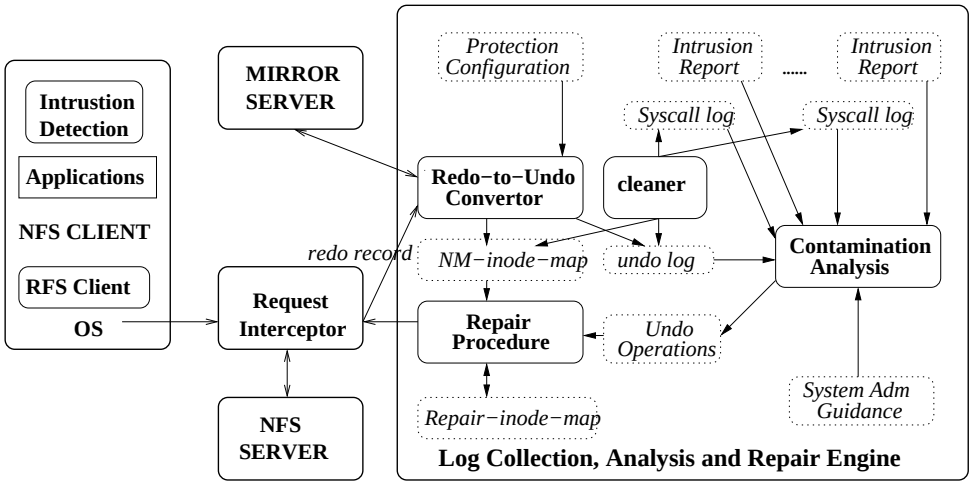


Figure 4.3: The detailed software architecture of the RFS-O system. Only the redo-to-undo converter is protocol dependent. All the other modules are reusable across different network file access protocols.

As described in Chapter 3, dependency analysis can have false positives and false negatives. The policy that we selected could be too liberal or too strict for a particular situation. It would be nice to have an interactive exploration tool for system administrators to interactively examine the validity of the output of contamination analysis. Along similar lines, RFS allows system administrators to specify

the *scope of protection* in terms of a set of file partitions, directories, or files on the NFS server.

4.1.4 Client-Side Syscall Logging

As determined by the contamination analysis policy, the client-side syscall logging records mutable file system syscalls except those that only modifies last access time. It also records the syscalls that create parent-child relationships, e.g., fork() and exit(). The implementation is similar to the *syslogd* in Unix system. The only difference is that instead of being stored locally, client-side logs are sent to the RFS server to be more secure. The client-side logging module is pretty thin, consisting of 203 lines of changes to the Linux kernel code and 273 lines of new code to maintain the logging buffer.

4.1.5 Traffic Interceptor

A *traffic interceptor* is needed in RFS-O and RFS-A. The requests from the NFS client are forwarded to the legacy NFS server immediately. The replies from the legacy NFS server are forwarded to the NFS client immediately. If a request is mutable (write, setattr, mkdir etc) and its reply is successful, they are put into a *redo record* and sent to the logging server. The interceptor maintains a buffer for all pending update requests that are waiting for replies. The interceptor interacts with the NFS server and the logging server through NFS protocol which is based on UDP. Therefore it needs to overcome packet loss and duplication issues.

nfs operations	mirror operations	undo record	NM-inode-map	repair-inode-map
R:create a	R:create a	R:remove a		
A:(10)	A:(20)	A:NULL	(10, 20)	
R:write(10), 0,2,xx	R:write(20), 0,2,xx	R:setattr(10), size 0		
A:success	A:success	A:NULL		
R:remove a	R:lookup a	R:create a		
A:success	A:(20)	A:(10)		
	R:read(20), 0,2	R:write(10), 0,2,xx		
	A:xx	A:NULL		
R:create a	R:create a	R:remove a		
A:(11)	A:(22)	A:NULL	(11, 22)	(10, 11)
R:write(11), 0,2,xx	R:write(22), 0,2,xx	R:setattr(11), size 0		
A:success	A:success	A:NULL		

Table 4.2: In this example three requests (create, write, remove) are sent to the protected NFS server. The last (remove) is malicious and needs to be undone. The table presents the operations that are applied to the protected NFS file system and the mirror file system on the logging server, together with the associated changes to the NM-inode-map and repair-inode-map. (10) means file with inode number 10, write(10),0,2,xx means write to file(10), offset 0, count 2, with data xx. R means REQ, A means ACK. In this example, file "a" is associated with four inode numbers: 10 on the protected file system before repair, 20 on the mirror file system before repair, 11 on the protected file system after repair, and 22 on the mirror file system after repair.

4.1.6 Inode Mapping Issue

In the *decoupled configuration*, there are two file systems accepting the NFS requests, the protected file system and the mirror file system. In RFS-O, the mirror file system is another NFS file system. In RFS-A, the mirror file system is a user-level versioning file system based on the underlying NFS file system. Conceptually the protected and mirror file system are identical. They are initialized with the same state and accept the same update requests. However they are not completely identical on a byte-by-byte basis. One difference that matters to RFS is that inode number for the same file could be different on the two file systems.

To carry out a update request to the same file on both file systems, RFS maintains a **NM-inode-map** to keep track of the one-to-one mapping. We call it an "inode" map because conceptually inode is the identifier of a file system object. However, the map actually stores NFS file handles instead of inode numbers. The reason is that in the NFS protocol each file is identified by an NFS file handle although the most important information in the NFS file handle is the inode. The file handle of a redo record is specified with respect to the protected file system. On the logging server, it is translated to the corresponding file handle in the mirror file system through the NM-inode-map.

In addition to the NM-inode-map, RFS also needs another inode map for repair time. Some undo operations create new file system objects, for example, the undo of *rm* and *rmdir*. The file handle of the new file object created by the undo operation may not be the same as the file handle being deleted. Therefore, in the undo process, RFS maintains another inode map (**repair-inode-map**) to keep track of the association between a file that is eventually deleted and its compensation copy. Table 4.2 presents an example to illustrate these inode mapping issue.

4.2 RFS-O: basic prototype

4.2.1 Undo Logging

File system updates can be classified into three categories: file block updates, directory updates, and attribute updates. To log a file block update, the undo logging first reads the prior image of the target block, updates the target block, and then appends the prior image to the undo record.

For directory updates, the undo logging does not need to save the old directory explicitly. For example, the undo operation for *create* is *remove*, which can directly be put into the undo record without reading any prior image. The same holds for *mkdir*, *rmdir*, *symlink*, *link* where the corresponding undo operations are *rmdir*, *mkdir*, and *remove*, respectively. The only exception is *remove*, for which the undo operation is not very simple. If a hard link is removed, the undo operation is *link*. If a symbolic link is removed, the undo operation is *symlink*, for which we need to read the contents of the symbolic link. If the object is a regular file, the undo operation is to create a new file, and write it to the full length for which the logging system needs to read the whole file and store the content in the undo record.

For file attribute update, i.e., *setattr*, usually the undo logging just need to save the old attribute to the undo record. Because the NFS protocol already includes the old attribute in the NFS reply, the undo logging does not need to issue another *getattr* request to retrieve the old attribute. The only exception is when a file is truncated for which the truncated data needs to be read and written to the undo record.

File block update, file truncate, and regular file delete are the most expensive NFS commands in terms of the undo logging overhead, and thus are the major targets for performance optimization.

4.2.2 Retrieve before image

A naive way to retrieve the prior image is to issue a file read or get_attribute request to the protected file system before it is updated. This approach adds significant delay to the original update operation and increases the load on the protected file server. Another small issue is that if the NFS request fails on the protected file server, the efforts to get the prior image are wasted. RFS-O uses a different approach to keep the logging activity isolated from the the interaction between the NFS client and the protected NFS server by assuming the existence of a separate *mirror file system,* (Figure 4.2), which services the requests to get the prior image. Typically the load on the mirror file system is lower than the protected file system because it does not need to serve non-update file system requests (read, getattr, lookup, access, readdir etc.) In addition the prior image accesses can be serviced asynchronously and scheduled for better disk access efficiency. The mirror file system could reside either on a separate server or on the logging server itself. Figure 4.2 shows the data structures used in RFS-O:

- **Protected file system** stores the current file system image. It is managed by the operating system and exported through NFS protocol.
- **Mirror file system** stores the mirrored file system image. It is also managed by the operating system and exported through NFS protocol.
- **Undo log** consists of a list of undo records, each of which stores the prior image required for an undo operation, including old data blocks, directory entries and/or attributes. It is managed by the user-level logging daemon.
- **File handle map** associates the file system objects in the protected file system to those in the mirror file system. There is also another temporary file handle map used only at repair time(Section 4.1.6).

4.2.3 Redo-to-undo conversion

The *redo record* produced by the *Traffic Interceptor* is converted to an *undo record* by a redo-to-undo converter. The redo-to-undo conversion is crucial to the system performance. Since the protected NFS server has multiple nfsd processes, the redo-to-undo converter uses a similar multi-threading structure to match the NFS server's processing capacity. In addition, the converter maintains a file attribute cache to reduce the frequency of contacting the mirror file system.

As shown in Figure 4.4, the converter has a dispatcher thread, three I/O threads (receive-redo thread, receive-reply thread and flush-undo thread), and a configurable number of processing threads. The central data structure is the request queue. Each queue will be in one of four status: *Free, Ready, Processing, Waiting-Reply*. All requests in the same queue operate on the same file system object. A *Free* queue is labeled as *Ready* after the dispatcher puts a request into the queue. Any processing thread can take a *Ready* queue, label it as *Processing* and start processing the requests in that queue. Whenever a processing thread needs to wait for a reply from the mirror file system, the queue is labeled as *Waiting-Reply* and will be set back to *Processing* after the reply is received. The queue will be labeled as *Free* when all the requests in the queue have been processed.

The redo-to-undo conversion must respect the dependencies among NFS requests. For example, a write request for a certain file cannot be dispatched before the create request for the same file has been processed. Similarly the remove request for a certain file cannot be dispatched before all the preceding write requests on the same file have been dispatched.

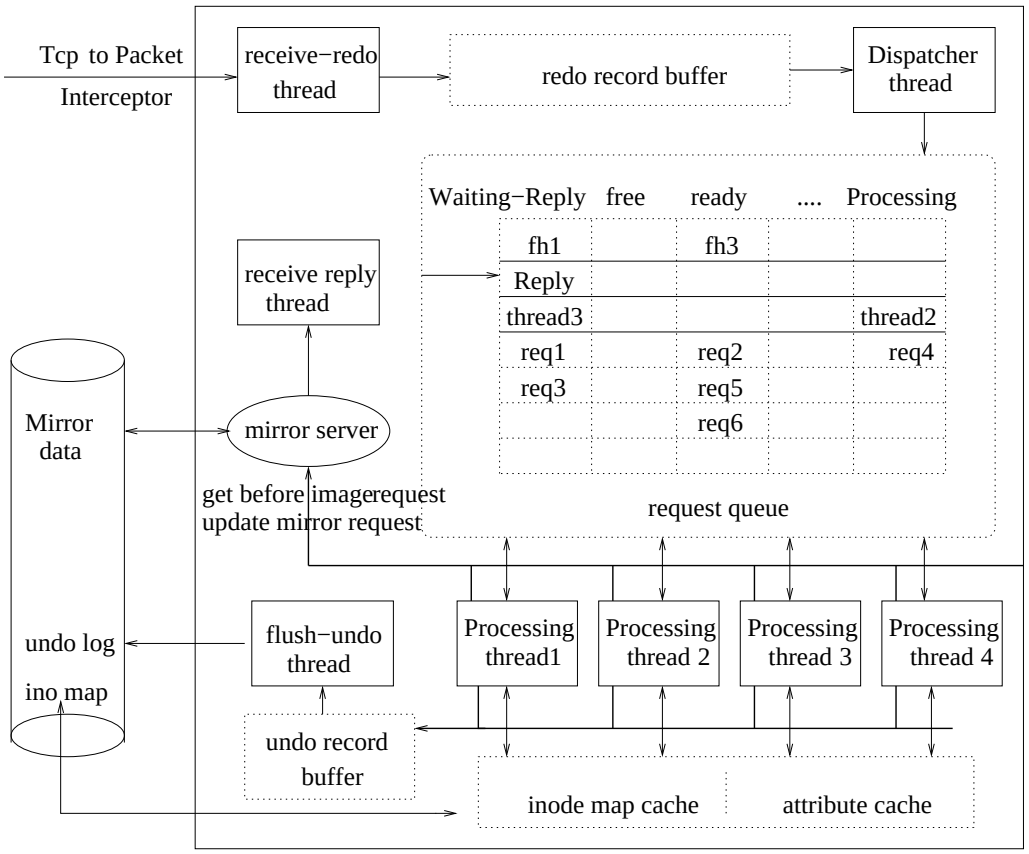


Figure 4.4: The redo-to-undo log converter uses a multi-thread software architecture to maximize the concurrency between disk I/O and log processing on the RFS machine, which hosts the mirror NFS server as well as the log converter.

4.3 RFS-A: improve logging efficiency

The *overwrite-logging* used in RFS-O is simple but expensive. For each file update operation, RFS-O reads the prior image of the written block to compose an undo record, (2) applies the write operation in place to the mirror file system, and (3) flushes the undo record onto disk. RFS-O performs undo logging asynchronously to hide most of its overhead. However, in the case of a long burst of file update operations, the system performs poorly compared to a standard NFS file server.

RFS-A solves this problem by using the more efficient *append-logging* scheme. However it requires significant modifications to file system metadata, as is the case with existing kernel-level versioning file systems [56, 61]. The *append-logging* is implemented at the user-level for ease of debugging and better portability. With *append-logging*, the undo log is much smaller than with *overwrite logging* because the undo record stores pointers to the old data rather than the actual prior image.

4.3.1 Overwrite logging

When a file block is updated, RFS-A allocates a new file block to hold the new version. Unlike kernel-level versioning file systems, which can directly modify file metadata (such as inode) to point to the new version, RFS-A needs to maintain a separate user-level metadata called **block map** to achieve the same purpose. The old data is kept intact during the *protection window* and recycled only when the corresponding undo record expires. The first version of every file block is stored in a place called **base image**. The *base image* has a similar role to the mirror file system in RFS-O. It has the same directory hierarchy and inode attribute values (except the file length attribute) as the protected file system. However, the *base image* is *not* an exact replica of the protected file system as we will explain later.

RFS-A uses a separate disk block pool, called the **overwrite pool**, to hold the

second and later versions of each file block. Each file block in the *overwrite pool* is identified by a virtual block number *Vblkno*. The pool is physically organized into multiple regular files in the local file system. Each such file is called a *stripe*. A **block usage map** is used to keep track of the usage of the *overwrite pool*. The map stores an *Oldtime* for each virtual block. A virtual block becomes old when the file block is overwritten, truncated or deleted; and its *Oldtime* is set to the timestamp of the corresponding undo record. The *Oldtime* of a virtual block containing current data is infinity. The *Oldtime* of a free virtual block is 0.

A **block map** is used to keep track of difference between the *base image* and the protected file system. For each block in the *base image* that contains an old image, there is an entry in the *block map*. The map entry is of the form <*Oldtime, Fid, Blkno, Vblkno*>, which indicates that the newest version of block *Blkno* of file *Fid* is stored at virtual block *Vblkno*. If *Vblkno* is -1, it means the target file block has been truncated. The *Oldtime* is the time when the file block <*Fid, Blkno*> in the *base image* becomes old.

In summary, when a logical file block is created, it is created in the *base image*. When a logical file block is overwritten for the first time, a virtual block is allocated from the *overwrite pool*, and an entry is added to the *block map*. The undo records uses a special location value of -1 to indicate that the old version is in the *base image*. When a logical file block is overwritten the second time, the undo record uses the *Vblkno* value in its *block map* entry to indicate the location of the old version. A new block from the *overwrite pool* is allocated and the *block map* entry is updated with the *Vblkno* of the new block.

Essentially, RFS-A distinguishes between write-once file blocks and overwritten file blocks. When a file contains only write-once file blocks, all its blocks are stored in the *base image*. However, as soon as some of them are overwritten, they will be stored in the *overwrite pool*. As a result, this design reduces the size of

the *block map* and improves the hashing and caching efficiency. For write-once file blocks, this scheme also preserves the disk proximity among adjacent file blocks.

For file truncate operations, RFS-A does not physically truncate the file in the *base image*, instead it maintains a **file length map** to distinguish the file length of a logical file and that of the corresponding physical file in the *base image*. Each *file length map* entry is of the form <*Fid, Userlen, Baselen*>. The truncate operation is executed as an update to the *Userlen* field. With append logging, whether a *write* operation is an "append" or an "overwrite" is based on *Baselen* and not *Userlen*. Although *Baselen* can be retrieved from the *base image* file attributes, it is stored in this map for the ease of frequent access. All other file attributes in the *base image* (except file length) are correct. The undo record for truncate operation contains only a pointer to the truncated data.

File delete operation is replaced by the *rename* operation. The deleted file is moved to a special directory called ***delete pool***. It has a flat structure and assigns each file a unique name generated from the inode number. Accordingly, the undo operation is another *rename* operation that brings the file back to the original directory. Figure 4.5 illustrates the lifetime of a file under RFS-A, starting from the time when it was created, then appended, overwritten, truncated, and finally till it is deleted.

As shown in Figure 4.2, the logging data structures used by RFS-A are:

- **Protected file system** is the same as that in RFS-O
- **Base image** stores the current file system hierarchy and most file attributes except file length. The file blocks it contains could correspond to either current or older versions.
- **Overwrite pool** stores blocks that have been overwritten at least once.
- **Delete pool** stores deleted files and their attributes.

- **Block map** stores the location of the current version of each file block if it is not in the *base image*. It also stores the associated timestamp for reclaiming storage.
- **Block usage map** is used for allocation of virtual blocks in the overwrite pool.
- **File length map** stores each file's length both as it is perceived by user and as it is in the *base image*.
- **Undo log** is similar to RFS-O except that each undo record contains a pointer to the prior image rather than the prior image itself.
- **File handle map** is the same as that in RFS-O.

4.3.2 Garbage Collection

The undo log and the client-side syscall log is recycled whenever they fall out of the protection window. Old versions of file attributes are stored and reclaimed with the undo records.

Each truncated file block and overwritten file block is pointed to by some undo log entries. The *Oldtime* values of these blocks and files are the timestamps of the corresponding undo records, which can be reclaimed after they fall outside the *protection window*. The *Oldtime* of each virtual block in the *overwrite pool* is stored in the *block usage map*. It is natural to integrate the garbage collection with virtual block allocation while scanning through the *block usage map*, permitting expired virtual blocks to be re-allocated on the fly.

It is slightly more subtle to reclaim the old blocks in the *base image*. The *Oldtime* of a file block in the *base image* is kept in the corresponding *block map* entry,

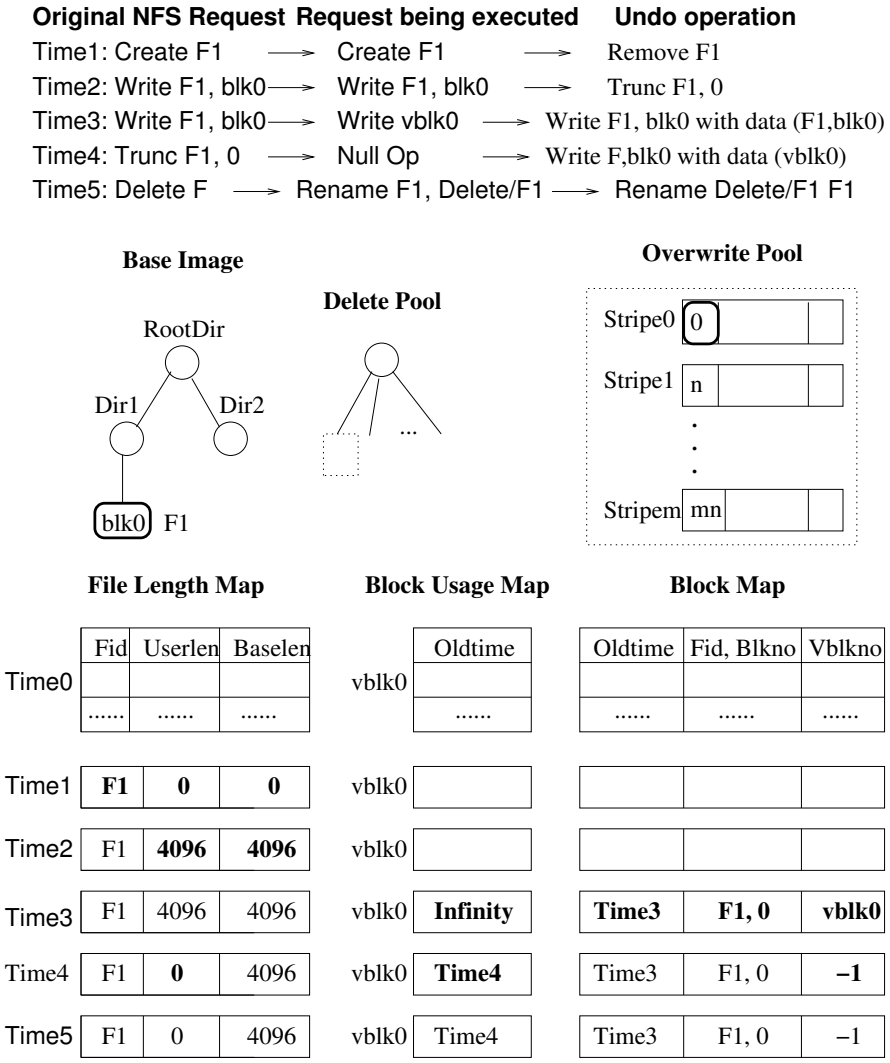


Figure 4.5: Evolution of a file block under RFS-A. This example includes 5 NFS requests. The figure shows the operations that are executed, their undo operations, and the metadata modifications associated with each request. The **Time3** is an overwrite operation, vblk0 is allocated to store the new data, and a new *block map* entry is added. The Oldtime is set to "Time3 ". The **Time4** is a truncate operation, the *Userlen* of corresponding file is set to 0. The *Oldtime* of the truncated block (vblk0) is set to Time4. The *Vblkno* of the *block map* entry is set to -1. The **Time5** is a delete operation, which moves F1 to the *delete pool*.

if it exists. [2] When a file block is overwritten, RFS-A checks the corresponding *block map* entry to see if the associated file block in the *base image* has already expired. If so, the new data is written to the *base image* instead of the *overwrite pool*. As a result the expired block gets recycled. However, this approach cannot reclaim all expired blocks in the *base image*. If an old block in the *base image* never gets overwritten after it expires, it cannot be recycled. One solution is to add a background cleaner which periodically checks if any file block in the *base image* has expired, and if so, move the current version from the *overwrite pool* to the *base image*. This block migration incurs extra overhead, hence it should be done when the system load is light and when heuristics indicate that the file block might not be overwritten soon.

The background cleaner also checks the last modified time of the files in the delete pool. Expired files are physically deleted and the corresponding entries in the *file handle map* and *file length map* are freed. Finally the background cleaner periodically scans through the *block map* to look for any entry with a *Vblkno* of -1, indicating that the block has been logically truncated. If a virtual block is the last block according to the file's *Baselen* and it has expired, the file in the *base image* is physically truncated and its *Baselen* is modified accordingly.

The other three prototypes(RFS-O, RFS-I and RFS-I+) have only have a subset of the data structures that RFS-A has. Their garbage collection can be easily adapted from the garbage collection scheme described above.

[2] Actually such entries in *block map* are kept even if the file is deleted and moved to the *delete pool*.

4.4 RFS-I: reduce hardware cost

RFS-O and RFS-A double the hardware costs of a file system by having both a protected file server and a logging server that maintains a versioned mirror image. From a performance perspective, with *append logging* it is no longer absolutely necessary to use a separate file server. In RFS-I, we experimented with integrating the NFS processing into the logging server.

Compared to RFS-A, RFS-I introduces three changes: (1) RFS-I no longer needs the *file handle map*. (2) RFS-I needs to process both read and write requests as well as their responses. In contrast, RFS-O and RFS-A only need to process the write requests, and do not need to touch the NFS replies. (3) The undo logging in RFS-I becomes synchronous logging. This logging overhead is added to the latency of normal request processing.

As shown in Figure 4.6(II), the file update logging module acts as an NFS proxy. Each NFS request first goes to RFS-I's user-level file update logging module, which modifies the request properly and sends it to the local NFS daemon in the kernel, which in turn sends a reply back. The file update logging module massages the reply into a response packet and sends it back to the requesting NFS client.

If an NFS request involves only one data block, RFS-I needs to determine whether the request should be directed to the *base image* or to the *overwrite pool*. If it should go to the *overwrite pool*, the request parameters (file handle, offset, count) need to be modified. If the request involves more than one block, RFS-I needs to check each block and if necessary, split the request into multiple requests. After receiving a reply, RFS-I may need to modify the file handle and attribute information if the request has been directed to the *overwrite pool*. If an incoming request is split into multiple requests, RFS-I needs to reassemble their replies into one reply and send the whole reply back to the requesting NFS client. In case some of these

replies are successful and some are not, RFS-I resolves the inconsistency and returns a coherent reply. Things become more complicated if the read/write requests are not aligned with the block boundaries.

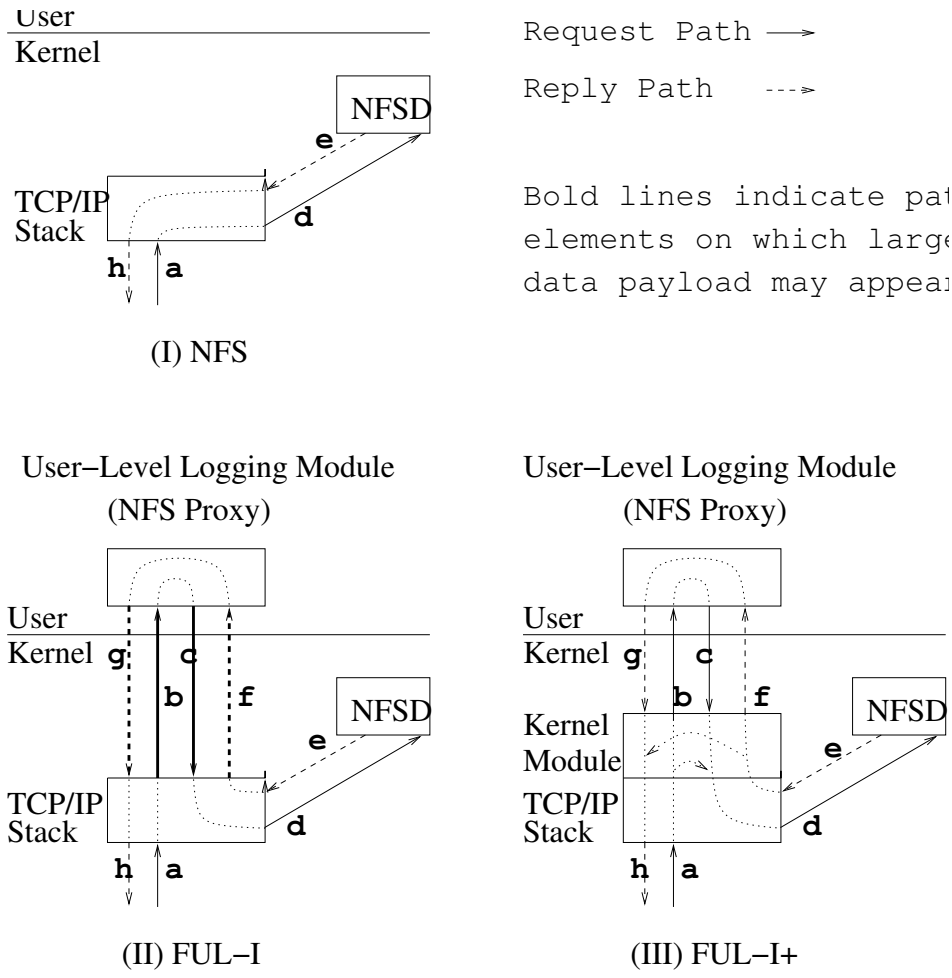


Figure 4.6: This figure illustrates the packet processing path in RFS-I and its overhead due to context switch, memory copy and user-level processing. In RFS-I+, an in-kernel packet interception mechanism reduces the overhead by providing a short-cut path and eliminating the need to copy large data payloads.

Figure 4.2 shows the data structures used in RFS-I. The data structures are similar to RFS-A except that there is neither a *protected file system* nor a *file handle map*.

4.5 RFS-I+: reduce overhead of user level implementation

In RFS-I, an NFS request and its reply are passed between the kernel and user space multiple times. To reduce the data copying and context switching overhead associated with user-level file update logging, we implemented another prototype RFS-I+ which incorporates an in-kernel packet interception mechanism to eliminate most of these overheads. This change has no effect on the logging scheme. RFS-I+ requires exactly same logging data structures as RFS-I. Figure 4.6 illustrates the difference in packet processing path between RFS-I and RFS-I+. Upon receiving an NFS request/reply, the kernel module has three possible ways to process it:

- Path-0: Forward the request/reply to the in-kernel NFS daemon/NFS-client directly (a→d / e→h in Figure 4.6), if the user-level file update logging module does not need to modify the request/reply, e.g., *readdir*.

- Path-1: Forward the request/reply to the in-kernel NFS daemon as well as the user-level file update logging module (a→b and a→d in parallel / e→f and e→h in parallel in Figure 4.6), if the request/reply does not need to be modified, but needs to be recorded, e.g., *create*.

- Path-2: Forward the request/reply to the user-level file update logging module if the request/reply potentially needs to be modified (a→b→c→d / e→f→g→h in Figure 4.6), e.g., *read* or *write*.

Path-0 represents the zero-overhead path, which is as fast as the standard NFS request processing time. Path-2 involves two context switches/memory copies because the original request and reply have to go through the user-level module. It affects not only the CPU utilization but also the end-to-end latency. Path-1 involves only one context switch/memory copy for sending the request packet to the user-level daemon. The overhead affects only the CPU utilization but not the end-to-end latency because the user-level processing is not on the critical path of NFS processing,

The **intelligent demultiplexing** scheme described above eliminates the processing overhead of those NFS requests and replies that are not at all relevant to file update logging. However, in many cases an NFS reply only requires very simple modification. For example, the *getattr* reply has completely correct content except the file length field, which needs to be changed from *Baselen* to *Userlen* according to the *file length map*. It is the same for many of the *read* and *write* replies where the requests are directed to the *base image*. Therefore we introduce another optimization called **in-kernel reply modification**. When the user-level logging module sends a request to the NFS daemon (step c in Figure 4.6), whenever possible it also sends an instruction on how to modify the associated reply. With this optimization, many NFS replies that used to take Path-2 can now take the less expensive Path-0 or Path-1. This optimization is particularly effective for NFS replies that contain large data payloads.

The last optimization in RFS-I+ is **write payload bypassing**, which decreases the memory copying overhead associated with *write* requests. A *write* request always needs to be processed by the user-level logging module. However, because user-level processing rarely touches a write request's payload, it is feasible to forward only the request's header to the user-level logging module (step b). When the user-level logging module sends the modified header back, the old header of the

request is replaced with the new header. In case the user-level logging module does need to touch the write's payload, for example if the request needs to be split, it can make another system call to explicitly retrieve the payload.

4.6 Fault tolerance considerations

RFS's fault tolerance design is very basic. It ignores the issue of high availability and focuses on restoring the system to a consistent state after failures. We assume that RFS system stops upon failures and when the failure reason has been fixed, restarts and initiates a consistency check. RFS is built on top of the kernel file system. All the metadata are stored as files/directories maintained by the operating system. The first step in the consistency check is to restore the underlying file system consistency with conventional fsck. The second step is only needed for RFS-A, RFS-I and RFS-I+. It restores the consistency of the user-level versioning file system data structures including undo log, base file system, overwrite pool, delete pool, file length map, block map and allocation map. The third step applies only to RFS-O and RFS-A. It restores the consistency between the protected NFS server and the logging server. It involves the protected file system, the mirror file system on the logging server and the file handle map. The fourth step restores the consistency between the client-side syscall log and the undo log. Next we will describe the second, third and fourth step in detail.

4.6.1 Consistency check among versioning metadata

To understand the consistency check algorithm, first we summarize how the user-level versioning metadata and data are updated during normal operations. The base image, overwrite pool, and delete pool are updated through file system interface upon each NFS request. It is up to the operating system to decide when these

updates go to the disk. Usually the operating system has an upper limit for the maximum delay before flushing, say, OS-SYNC-INTERVAL. The situation is similar for the undo log. The other metadata (file handle map, file length map, block map, allocation map) are updated in memory upon each NFS request. Periodically they are write to the metadata files and flushed to the disk. Let's call it RFS-SYNC-INTERVAL. Usually the RFS-SYNC-INTERVAL is bigger than the OS-SYNC-INTERVAL.

Depending on whether there was data loss caused by disk failure or unsuccessful kernel fsck in the first step, different approaches are used to restore logging server consistency. If there is no data loss, assuming the system crashed at time T, the consistency can be restored by first rolling the system backward to T - OS-SYNC-INTERVAL, and then replaying the updates to the versioning metadata from T - RFS-SYNC-INTERVAL to T - OS-SYNC-INTERVAL. This approach is similar to the fsck of journaling file system. Note that the distinction between redo and undo logging is only valid for overwrite logging. With the user level versioning file system, the information in the log record can be used for both redo and undo recovery.

If there is some data loss, the consistency check is conducted in a brute-force style. It restores the file system data using a best effort strategy. This approach is similar to the fsck of non-journaling file system such as ext2, The steps to be performed are:

1. Traverse the *base image*, and check (1) for each file system object whether there is an entry in the *file length map* and the size of the object equals the *Baselen* in *file length map*.

2. Examine each <*Oldtime, Fid, Blkno, Vblkno*> entry in the *block map*. If the *Vblkno* is not -1, verify if the *Blkno* is within the *Userlen* of the file *Fid*,

and the *Oldtime* of the *block usage map* entry for *Vblkno* infinity. If the *Vblkno* is -1, check if the corresponding file block is truncated according to the *Userlen* attribute of the file.

3. Check that for each entry in the *block usage map* whose *Oldtime* is infinity, there is a corresponding entry in the *block map*.

4.6.2 Consistency check between protected file system and mirror file system

This step applies only to RFS-O and RFS-A. In the absence of a disk log that records each NFS request before forwarding it to the protected file server, the only way to restore the consistency between the protected file system and the mirror file system is by brute-force.

Either of the protected file system or mirror file system could be used as the correct image. We chose to use the protected file system unless there is major data loss on it. A ”diff -r” can be used to find all the differences between the two file systems, and file system commands such as cp, mkdir, rmdir, remove can be used to correct the differences. The *file handle map* is updated along the way.

4.6.3 Consistency check between client syscall log and file update log

It is relatively easy to restore the consistency between client-side syscall log and the undo log. As described in Section 4.1, the syscall log and the undo log can be synchronized with RPC message entries. If after the last synchronization point, there are additional syscall log entries, they are simply discarded. If instead there are additional undo records entries, we could chose either to do nothing or roll back

those undo records. If we chose to do nothing, we get to keep a little bit more data but lose some accuracy in the contamination analysis accuracy (which in any case has false positives and false negatives).

4.7 Implementation

We have implemented RFS-O, RFS-A, RFS-I and RFS-I+ on Redhat 7.2 with kernel 2.4.7-10. These four file update logging schemes share a common code base, in total about 16000 lines of C code. The kernel module of RFS-I+ contains another 1500 lines.

We have ported RFS-O, RFS-A, and RFS-I to FreeBSD. The only part of the code that is OS-dependent is the NFS file handle interpretation. All the four logging schemes need to extract a unique file object id (including device id, inode number and generation number) from the NFS file handle. According to the NFS protocol specification [4], the NFS file handle is an opaque and implementation-dependent data structure. It is not supposed to be interpreted by NFS clients or any other third party. The NFS file handle structure is different on Linux and on FreeBSD; therefore the NFS file handle interpretation code needs to be changed. The modification is about 200 lines of code. We have not ported the kernel module of RFS-I+ to FreeBSD. The OS-dependent code in the kernel module of RFS-I+ includes memory allocation/deallocation, turning on/off interrupt, sleep/wake up operations, operations on SKB data structure, and some interactions with the network stack.

The RFS-O was implemented for NFSv2. It took 3 person weeks to convert it to NFSv3. This demonstrates the portability of the RFS architecture. Most modifications are within the processing thread of the redo-to-undo log converter. We had to rewrite the parser for NFS requests and replies, and the routines to convert redo record into undo record. Although the total number of lines of code involved

in this porting is about 5000 lines, most of the data structures and parser code were borrowed from the Linux kernel code with minor changes.

Some of NFSv3's features helped to simplify RFS implementation. In NFSv2, the NFS request to create a symbolic link does not return the associated file handle; in NFSv3, it does. Another example is that NFSv3 returns file attributes much more frequently than NFSv2. This reduces the number of get_attribute requests required in the redo-to-undo log conversion process.

As for NFSv4, most of the protocol changes are related to scalability and security, and therefore do not affect RFS. The request batching feature can be easily accommodated by a minor modification to RFS's request interceptor. For a batching request and reply, multiple log entries will be constructed in the file update log. We do not expect much effort will be needed to port RFS to NFSv4. So far we do not have any concrete experiences to report about porting RFS to network file servers based on AFS or CIFS.

4.8 Performance Evaluation

RFS facilitates the damage repair process at the expense of runtime overhead and additional resource consumption. The viability of the RFS approach thus depends on how expensive this additional performance/hardware cost is and how much RFS can speed up the repair process. In this section, we evaluate the general aspects of a repairable file system. We also compare the performance of the four prototypes in detail.

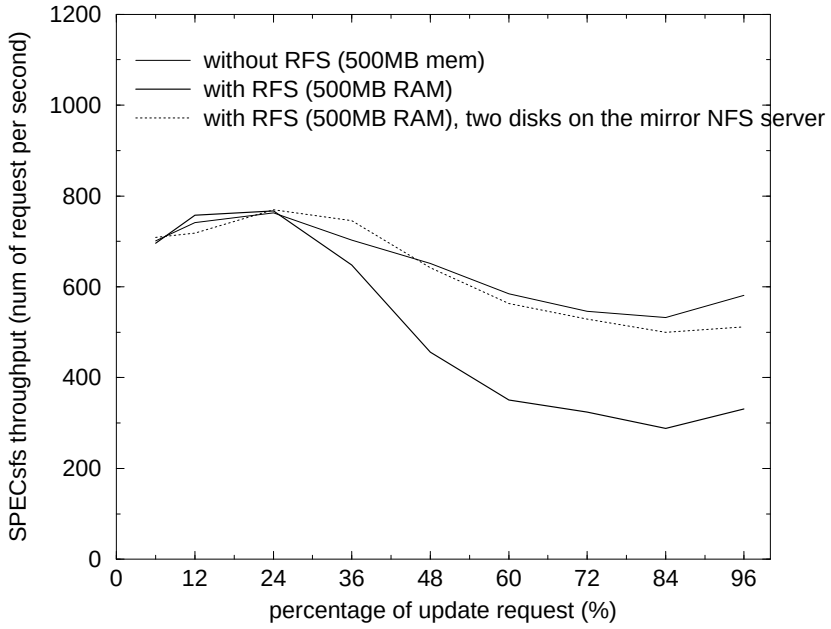


Figure 4.7: The system throughput with regard to the update request percentage in the SPECsfs benchmark under different running conditions

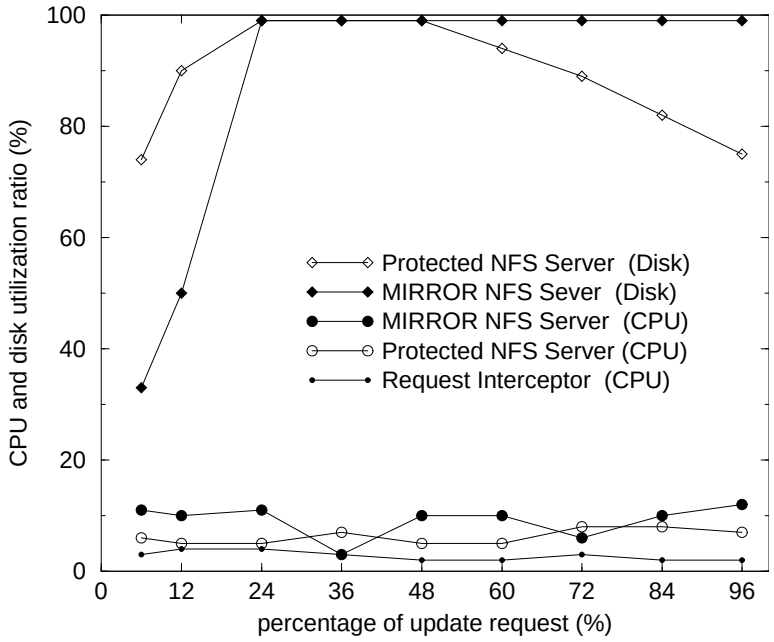


Figure 4.8: The CPU/disk load comparison among the protected NFS server, the logging server and the traffic interceptor when configured with 500MB RAM. The disk load of the traffic interceptor is 0 and hence omitted in this figure.

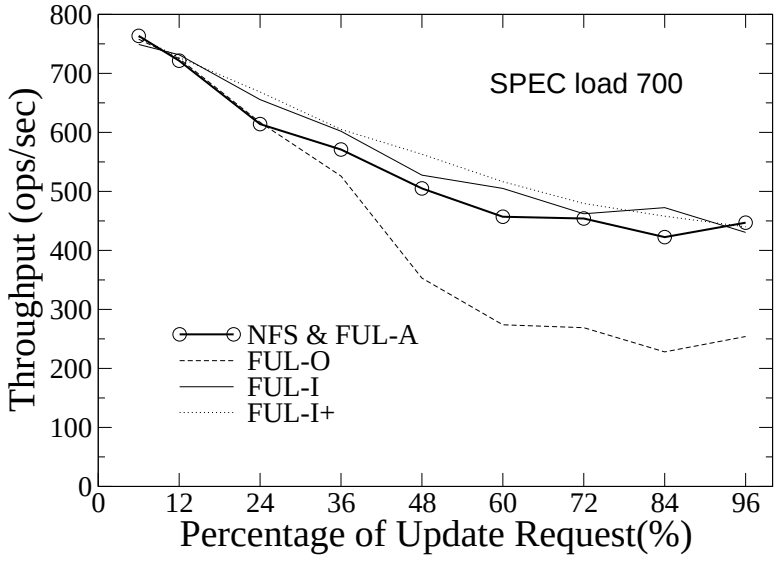


Figure 4.9: Throughput comparison among file update logging schemes as the percentage of update requests in the input workload is varied.

4.8.1 Testbed setup and evaluation workload

There are five machines in the testbed, all of which run Redhat 7.2 with Linux kernel 2.4.7-10. There are two NFS clients, one NFS server to be protected, one machine running the traffic interceptor and the last machine (logging server) running log converter, contamination analysis, repair engine and the mirror file system all together. The NFS clients and the traffic interceptor machine are connected through a Fast Ethernet switch. The protected NFS server and the logging server are connected to the traffic interceptor through a crossover cable. Other than one client machine which has 400MHz CPU and 128-MByte memory, all other machines are 1.4GHz Pentium IV machine with 500-MByte memory. Both the protected NFS server and the logging server server are configured with a 40GB ST340016A ATA disk drive with 2-MByte disk cache. The protected NFS server is only used in the evaluation of RFS-O and RFS-A.

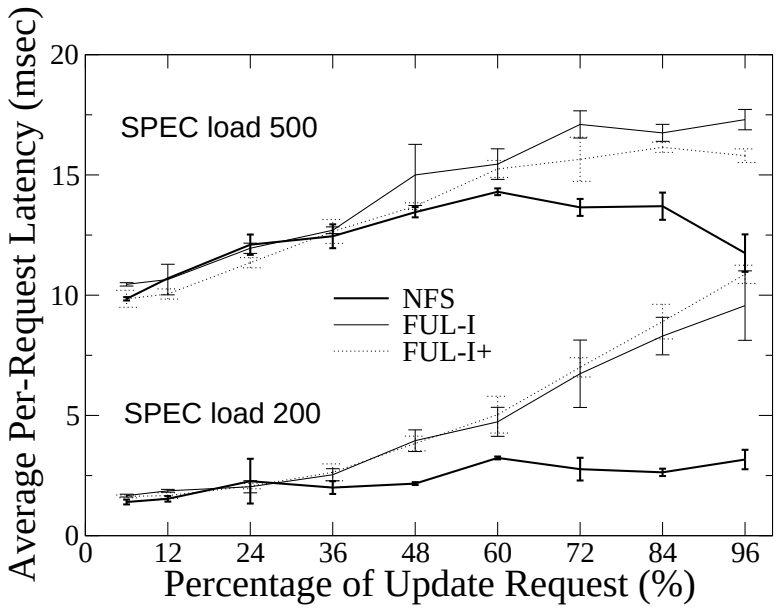


Figure 4.10: Latency comparison among file update logging schemes as the percentage of update requests in the input workload is varied.

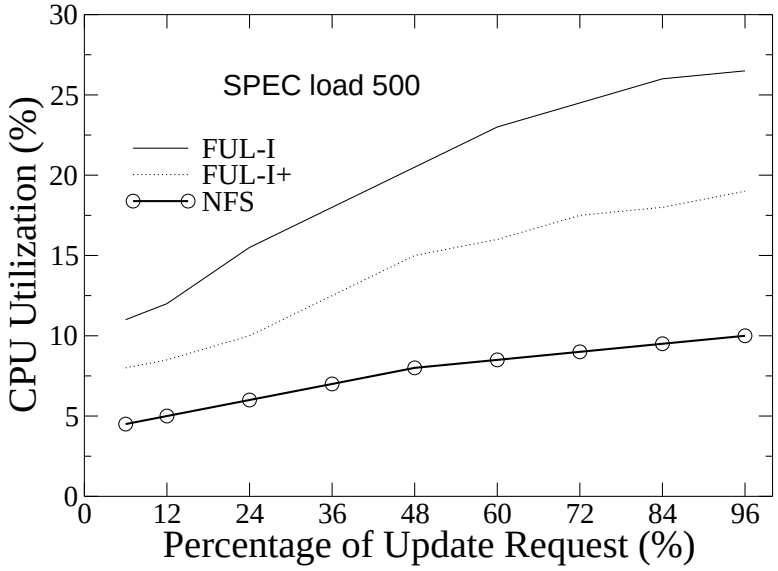


Figure 4.11: CPU utilization comparison among file update logging schemes as the percentage of update requests in the input workload is varied.

The main workload we used in the experiments is SPEC SFS 3.0 (SFS97_R1), the Standard Performance Evaluation Corp.'s benchmark for measuring NFS throughput and response time. Its operation mix closely matches real-world NFS workloads. SPECsfs includes a representative mix of different types of NFS requests, with 12% of the requests updating the file system and the remaining requests being read-only. Unsurprisingly, the file update logging overhead is more pronounced when the percentage of update requests in the input workload is higher. To stress test our logging schemes, we varied the update request percentage from 12% to 96%, but kept fixed the distribution among different types of update requests and the distribution among different types of read-only requests.

SFS benchmark directly interacts with the NFS server through a UDP socket, rather than through system calls. As a result, SFS benchmark cannot be used to evaluate client-side logging overhead. FOr this, we used the SDET [26] benchmark

instead. SDET represents the workload typically seen in a software development environment. The evaluation also uses the Harvard NFS traces [18] and some micro benchmarks.

4.8.2 Syscall logging overhead on NFS client

On the 400MHz client machine, we ran the SDET benchmark using 32 scripts and generated a total file set size of 55 MBytes. The client-side logging incurred 4.08% of CPU overhead. The kernel logging buffer required was about 12 MBytes. The total NFS traffic in this run was 97 MBytes, and the traffic resulting from client-side log was 3MBytes. The client-side log size was 3MBytes, the the server undo log size was 80 MBytes. With client-side logging turned on, the NFS server's throughput dropped by 5% from 1907 to 1811. This decrease in throughput is mainly due to the additional CPU overhead used for system call logging.

4.8.3 Effectiveness of Contamination Analysis and Damage Repair

To evaluate the effectiveness of RFS's automated damage repair procedure, we ran the SDET benchmark with two clients until the undo log size reached 100 MBytes. There were 87 processes involved and 985 files modified. We randomly picked some processes as the root processes. Different numbers of root processes thus correspond to different contamination levels, i.e., different scopes of contamination. In Table 4.3, *All* means that all processes are contaminated and all update operations need to be undone. Similarly *high* and *low* mean the proportion of operations to be undone is large or small. Contamination level is decided both by the number of initial root processes and the propagation of contamination.

Contami--nation Level	Contami--nated Processes	Contami--nated Files	Contami--nated Blocks	Contamina--tion Analysis (secs)	Damage Repair (secs)	Total (secs)
all	87	985	0	76	95	171
high	77	751	0	76	87	163
low	1	0	1	75	1	76

Table 4.3: The repair time is dependent on the contamination level, whereas the contamination analysis time is not.

Table 4.3 shows that the contamination analysis time does not depend on the the contamination level. This is because the contamination analysis module needs to read in and parse the entire syscall logs and file update log in order to determine the status of contamination. In contrast, the repair time depends on how many undo records are selected by the contamination analysis module. From this result, the contamination analysis and damage repair time for a 700-MByte undo log, or one-day's SPECsfs run, is estimated to be between 9 to 20 minutes, depending on the contamination level. We believe this is much faster than a manual repair process that is able to repair the system at the same level of precision.

4.8.4 Forwarding latency of traffic interceptor

For each NFS packet, the request interceptor adds a small forwarding delay, ranging from 0.2 ms to 1.5 ms with different packet size. The throughput of the protected NFS server is unaffected, as long as the log converter is able to convert redo records into undo records in time. When the converter fails to keep up with the input load, the request interceptor will drop NFS packets, and the system throughput decreases.

4.8.5 Overwrite logging performance of RFS-O

In RFS-O it takes about 5ms for the *log converter* to process a NFS update request, so it can process about 200 update requests per second at most. The default update request percentage in SPECsfs is 12%. We varied this percentage in SPECsfs and measured the throughput of the protected NFS server. The result is shown in Figure 4.7. Each percentage corresponds to a SPECsfs benchmark run. The load we generated was 700 for each run which generated about 7GB of initial file set size. Note that it's normal for SPECsfs to have throughput slightly higher than the load specified. Unless specified otherwise, on the protected NFS server, the operating system and the testing directories reside on one disk; on the logging server, the operating system, mirror image, RFS undo log and client system call log all reside on one disk.

When update request percentage is below 30%, we observed no significant throughput difference between the vanilla NFS server and RFS-O. However, beyond 30%, the performance degradation of RFS-O was more and more pronounced. When update request percentage is 96%, the system throughput dropped to half (from 600 to 300). Most of the performance cost of RFS-O lies in the log converter, which spends over 90% of the time processing write requests. Log conversion is I/O bound, and adding one more disk to the logging server to hold half of the SPECsfs working directories eliminated the performance degradation of RFS-O, as shown by the curve labeled "with RFS, two disks on the mirror NFS server."

4.8.6 Hardware Requirement of RFS-O

To further understand the hardware requirement of the logging server which runs log converter and mirror file system, we compared CPU and disk usage of the protected NFS server and the logging server. For a file update request, the logging

server needs an additional read for the prior image and an additional write to the undo log, i.e., three disk accesses in total. The bottleneck of both NFS and the logging server lies in disk access – Figure 4.8 shows the CPU and disk utilization comparison between the protected NFS server and the logging server. It also shows the CPU usage for the traffic interceptor machine, which was always less than 5% regardless of the update request percentage, indicating that a low-end machine with a small amount of memory is sufficient. The CPU load of the logging server is comparable with that of the protected NFS server. At the default NFS update request percentage, 12%, the disk load on the logging server is 55% of that of the protected NFS server.

4.8.7 Performance characteristics of append logging

Under *append logging*, random writes may become sequential writes to the *overwrite pool* if the free virtual blocks are contiguous. On the other hand, sequential reads may become random reads if consecutive file blocks are overwritten non-sequentially and thus get dispersed in the *overwrite pool*. As a result, a server doing *append logging* may perform better than in-place updates for workloads dominated by random writes, but perform worse for workloads dominated by sequential reads after random writes.

To illustrate this characteristic of append logging, we performed the following experiments using a server machine with smaller (256MB) memory and a client machine with 128MB memory. The server machine ran either a vanilla NFS server or RFS-I+. The client machine ran a generic NFS client. First we created a 500MB file on the server using sequential writes from the client. In this setup, there is no cache hit on either the client side or the server side. The sequential write throughput for both vanilla NFS and RFS-I+ was 11MB/sec.

Then we performed a sequence of random write operations (each of size 4096

bytes) until the size of the *overwrite pool* reached 2GB, which produced sufficient disk layout difference between the vanilla NFS server and RFS-I+. Under a vanilla NFS server the disk utilization was 100% and the write throughput was 1.54MB/sec. With RFS-I+, the disk utilization was 22.5% and the write throughput was 10.23MB/sec. Overall, the disk access efficiency of RFS-I+ was 30 times *higher* than the vanilla NFS. This result shows that the append strategy behaves similarly to a log structured file system, which has the advantage of converting random writes into sequential writes.

Finally, we performed a sequence of sequential read operations against the 500MB file with a request size of 4096 bytes. Under the vanilla NFS server, the disk utilization was 18.6% and the read throughput was 4.76 MB/sec; with RFS-I+, the disk utilization was 94.4% and the read throughput dropped to 0.87 MB/sec. Overall, the disk access efficiency of RFS-I+ was 28 times *worse* than the vanilla NFS server. Periodic cleaning can mitigate the loss of sequential read locality by moving the current versions of those file blocks that have become read-only from the *overwrite pool* to the *base image*.

4.8.8 Throughput and latency comparison of four prototypes

We use SPECsfs as the workload to compare the four RFS prototypes. As expected, the system throughput decreases as the update percentage of the workload increases(Section 4.8.1). At 12%, it is about 700 ops/sec. The throughput goes down to 500 ops/sec at 96%. The initial working set size is proportional to the request rate in the input workload, 7GB for a load of 700 ops/sec and 5GB for a load of 500 ops/sec. Therefore the measurement results corresponding to different offered loads may not be directly comparable. Moreover, the performance tends to decrease when the system is overloaded. For example, the throughput at an offered load of 700 ops/sec may be 400 ops/sec but reaches 500 ops/sec when the input

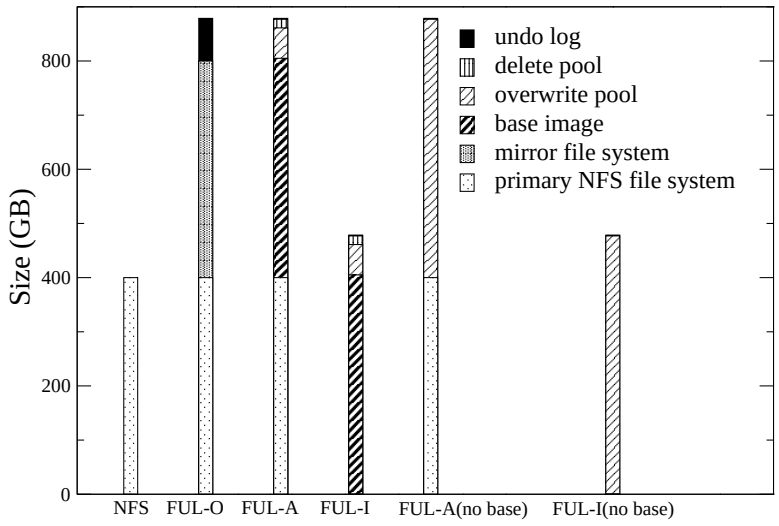


Figure 4.12: The storage requirement break-down for various file update logging schemes. RFS-I and RFS-I+ require additional storage for old data. RFS-O and RFS-A double the storage requirement because two file servers are involved - the protected file server and the logging server.

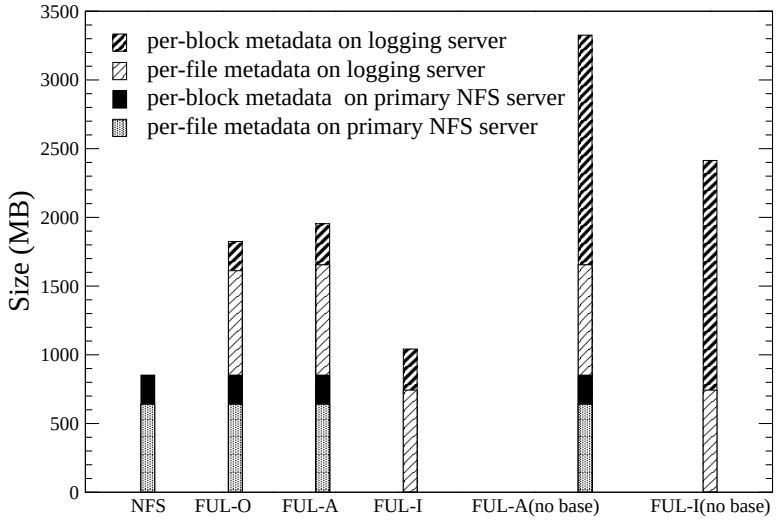


Figure 4.13: The break-down of metadata used in various file update logging schemes. The metadata includes both user-level metadata and kernel-level metadata, as shown in Figure 4.2

load is 500 ops/sec.

Figure 4.9 shows that all four prototypes have almost the same throughput when the update percentage in the input workload is less than 12%. RFS-A performs the same as the vanilla NFS server because the logging server is never the bottleneck, and its primary NFS server is identical to the vanilla NFS server. As the update request percentage increases, the performance of RFS-O is limited by the prior image read and write. Surprisingly, even with all the extra processing due to file update logging, RFS-I and RFS-I+ actually over-perform the vanilla NFS server in throughput by around 7%. The reason is that in this experiment, disk is the system bottleneck. The advantage of the append logging in processing random write requests, as discussed in Section 4.8.7, is significant enough that it results in a small but significant overall performance improvement. Another block-level versioning system, Clotho [25], reported similar performance gain for similar reason. The

in-kernel packet interception mechanism in RFS-I+ has no effect on disk access efficiency, therefore RFS-I+ performs roughly the same as RFS-I in all cases.

Figure 4.10 shows the average per-request latency of the vanilla NFS server, RFS-I and RFS-I+. The per-request latency of RFS-O and RFS-A is similar to the vanilla NFS because the NFS reply comes directly from the protected NFS server. Each latency number represents the average of three measurements. The three upper latency curves correspond to the case when the system is heavily loaded (SPEC load 500). The three lower latency curves correspond to the case when the system is lightly loaded (SPEC load 200). The results for RFS-I and RFS-I+ are similar, because the RFS-I+'s kernel optimizations do not have any significant effect on the latency. When the percentage of update requests is no more than 36%, the average latency of RFS-I and RFS-I+ is similar to that of the vanilla NFS server. As the update request percentage increases further, the per-request latency of RFS-I and RFS-I+ becomes higher than that of the vanilla NFS server. The latency penalty of RFS-I and RFS-I+ increases with the update request percentage. In one extreme case (at 96% and load 200), the latency penalty is about 7 msec, which corresponds to a 200% latency overhead. However, this latency penalty actually decreases with the input load. For example, the percentage latency penalty is reduced to 10-30% at load 500.

The latency penalty of RFS-I and RFS-I+ comes from the extra processing associated with file update logging. With append logging, the logging server may need to issue multiple requests to the local NFS daemon to serve one NFS request from the client. The additional requests could be issued to reassemble read/write requests, to read the prior image when a *write* request is not aligned, to verify the file type of an object to be deleted, etc. These requests do not require additional disk bandwidth, but they do increase the request processing latency. As the update request percentage increases, more blocks are overwritten and reside in the *overwrite*

pool. Consequently there is a high chance that multiple local requests are needed to reassemble one client request. This leads to increases in the average per-request latency.

4.8.9 CPU utilization comparison of RFS-I and RFS-I+

Figure 4.11 compares the CPU utilization of vanilla NFS, RFS-I and RFS-I+. RFS-O and RFS-A are excluded because they require a separate server. When the input SPECsfs load is 500 ops/sec, the throughputs of NFS, RFS-I and RFS-I+ are comparable, but the CPU utilization of RFS-I and RFS-I+ is about 170% and 85% higher than the vanilla NFS server, respectively. These results suggest that user level append logging indeed consumes additional CPU resource, and the kernel optimizations in RFS-I+ effectively reduces the CPU consumption by eliminating a large portion of the context switching and memory copying overhead. When CPU is the performance bottleneck, RFS-I+ should out-perform RFS-I. To substantiate this claim, we modify the SPEC workload so that it is read-only, with a high buffer hit ratio. We also upgrade the network connection from 100 Mbps to 1000 Mbps. As a result, disk and network are no longer the system bottleneck. With an initial working set size of 300MB and a SPECsfs input load of 7000 ops/sec, the measured throughput of NFS, RFS-I and RFS-I+ are 6560, 4166, and 5441 respectively; RFS-I+ out-performs RFS-I by 30%.

4.8.10 Storage and Memory Overhead

A major concern for fine-grained file update logging is its storage/memory requirement. Figure 4.2 shows the data structures used by RFS-O, RFS-A and RFS-I.

	RFS-I	+Intelligent Demultiplex	+ In-Kernel Reply Modification	+ Write Payload Decoupling(RFS-I+)
Path-0	0%	47%	56%	56%
Path-1	0%	1%	6%	6%
Path-2	100%	52%	38%	38%
Context Switch	1	0.52	0.41	0.41
Memory Copy	1	0.81	0.31	0.07

Table 4.4: This table shows the incremental context switch and memory copy saving due to each of the three optimizations in RFS-I+. The first column is for RFS-I without kernel module. The last column is for RFS-I+ with all three optimizations from the kernel module applied. The first three rows show the percentage of NFS packets taking difference paths. The last two rows show normalized context switching and memory copying overhead.

Figure 4.12 and 4.13 show the detailed breakdown of the storage space requirements for their data and metadata, when driven by a two-week Harvard EECS trace from Oct 14 of 2001 to Oct 28 of 2001. The NFS bar corresponds to the vanilla NFS server configuration. The RFS-I bar represents both RFS-I and RFS-I+, as RFS-I+'s in-kernel optimization has no effect on these data structures.

Most of the storage requirement for file update logging arises from the need to store the current and old data. Figure 4.12 shows that the storage requirements of all four prototypes are similar except that RFS-O and RFS-A have a protected NFS server, which almost doubles the storage requirement. The amount of historical data being generated is modest, only 80GB in this two weeks trace. The file system size of the protected NFS server is about 400GB and file server is more than 90% full. The total size of data that have been created and updated during these two weeks is around 25GB. Therefore the storage space cost of doing comprehensive

versioning is about 3 to 4 times of the size of updated and created data. This case study represents a huge and slowly-changing file system. For a small but busy file system the ratio could be higher.

The *undo log* in RFS-A and RFS-I contains only pointers to old data, and is only 1.6GB in size, too small to be visible in the figure. The compactness of the undo log substantiates the claim that file update logging at the NFS command level is space-efficient.

Although the storage requirement for metadata is relatively small, metadata may still compete with normal file data for the buffer cache space. The *base image* is introduced to reduce the memory consumption of the *block map*. To demonstrate the effectiveness of the *base image*, we also report the storage requirements when the base image is not used in the two "(no base)" cases. Files in the *delete pool* are renamed from the *base image*. If there is no *base image*, the *delete pool* won't exist either. Both old and new data would have to be stored in the *overwrite pool*. Therefore, the size of the *overwrite pool* in the "no base" case is equal to the sum of *base image*, *overwrite pool* and *delete pool*. The size of the *block map* is proportional to the size of the *overwrite pool*. It is 87MB in the current design and would be 1372MB in the "no base" design.

Figure 4.13 shows that the design of the *base image* greatly reduces the metadata size. Overall the metadata size of RFS-I is only 22.3% higher than that of a vanilla NFS server. This overhead is independent of the server size, but may vary with file size distribution. It slowly increases with the logging window size and/or the frequency of write and delete requests, both of which require a larger overwrite pool and delete pool. For example, when the access pattern remains the same and the logging window is increased to four weeks, this metadata overhead becomes 32.5%.

The analysis above ignores the size of client syscall log. It is not possible to

get client syscall logs for the existing NFS traces. According to the experiment that evaluates client-side syscall logging overhead 4.8.2, it is about 4% of the total storage used by the logging server and it does not use memory space.

To add more workload diversity, we did another simpler experiment (measuring only RFS-O's storage requirements) using a different NFS trace. The trace is collected from the graduate student home directory server (over 250 users) in the Computer Science Department of Stony Brook University, for a period of 8 hours and 48 minutes that was spread over the last week of the Fall 2001 semester. During this tracing period, there were 1,863,971 NFS requests, and among them 51,313 (2.7%) requests were updates (e.g., `write` and `setattr`). We used this trace to analyze the storage requirement of RFS-O.

The resulting RFS-O's undo log size for this trace is 259,762,779 bytes, or around 260 MBytes. The majority of the undo log, 97%, is attributed to the prior images of file updates. Similar to the first experiment, we cannot get the client-side syscall log size and assumed it to be 4% of the undo log size in RFS-O. According this trace, the per-day undo logging requirement is about 709 MBytes. Therefore, a 40GB disk can be used to maintain a protection window of 8 weeks for a NFS system of comparable size.

4.8.11 Effectiveness of Packet Interception Optimization

Table 4.4 quantifies the contribution of each of the three optimizations in RFS-I+ that aim to reduce the number of context switches and memory copying operations associated with each NFS request and reply. We used the default NFS operation mix of the SPECsfs benchmark as the workload, in which 18% of requests are read, 9% are write, 11% are readdir, 28% are lookup, and 11% are getattr. We also assumed the average data payload size for read and write requests to be 6KB, and the average readdir reply size to be 2KB. In RFS-I, all the NFS packets take

Path-2, which involves two context switches and two memory copying operations. With each additional optimization, more packets take the less expensive Path-0 and Path-1.

Table 4.4) shows that *intelligent demultiplexing* reduces the context switch overhead significantly, while *in-kernel reply modification* and *write payload decoupling* greatly reduce the memory copying overhead. When all three optimizations are applied, the context switch overhead is reduced from a normalized value of 1 to 0.41 and the memory copying overhead is almost completely eliminated (from a normalized value of 1 to 0.07). Note that the memory copying overhead is calculated based on bytes and not packets because different packets may have significantly different size. Moreover, when the system is loaded, RFS-I+'s in-kernel packet interception mechanism can actually deliver several packets to the user-level logging module in one shot. Consequently the context switch overhead can thus be amortized over multiple requests and further reduced.

4.9 Conclusion

The RFS project augments existing network file servers in such a way that post-intrusion or post-error damage repair can be more accurate (because every update can be rolled back,) and faster (because both determination of the extent of damage and undo of corrupted effects can be automated.) The ability to keep track of inter-process dependencies represents an important research contribution to intrusion tolerant systems design. Through a fullly-operational RFS prototype, we show that the time to repair a network file server after a malicious attack or an operational error is reduced to the level of minutes or hours with most of the useful work being preserved.

From a security standpoint, a major weakness of the RFS architecture is that

it is vulnerable to denial of service attacks. If an attacker keeps updating even a single file block, it will eventually fill up the undo log and effectively disable the protection provided by RFS. One simple solution is to increase the log disk space and to support early warning to system administrators so that the log disk never becomes full. A more sophisticated solution is to regulate the undo log disk space consumption rate from individual users, so that a user can only consume as much undo log disk space as his/her quota permits, and thus never has a chance to exhaust the entire undo log disk space.

File update logging is a critical building block for quickly repairing damage to a file system due to malicious attacks or innocent human errors. So far it has not been incorporated into mainstream operating systems because of the concern of additional storage requirements, performance overhead, and the implementation complexity. Given the dramatic improvements in the cost efficiency of magnetic disk technology, disk cost is no longer an issue. Measurements on a real-world NFS trace show that a $200 200GB disk can easily support a one-month logging window for a large NFS server with 400GB of storage space and an average load of 34 requests/sec. The performance overhead and implementation complexity associated with file update logging, however, remain significant barriers to its deployment in practice.

In this paper we experimented with four prototypes with different file update logging implementations. RFS-O and RFS-A incorporate legacy NFS file servers for better performance at additional hardware cost. RFS-O is superb in simplicity but not as good in write performance. In RFS-A and RFS-I we designed a novel user-level file update logging scheme that is both efficient and portable. We make comprehensive evaluation regarding their latency, throughput, and CPU usage characteristics using standard benchmarks, NFS traces, and synthetic workloads. Below is a summary of RFS's file update logging scheme:

- Logging at a higher level of abstraction, such as NFS requests and replies, tends to produce a much more compact log than logging at a lower level of abstraction, such as disk accesses and responses, and is also more portable and flexible.

- RFS-O shows a run-time throughput penalty of less than 6% when update request percentage in the input workload is below 30%.

- RFS-A incurs close to zero latency and throughput penalty compared with a vanilla NFS server, and is thus the best choice for IT environments where performance is critical, mirroring the file image is not an issue, and minimum disruption to the primary NFS server is important.

- As shown in RFS-I, the user-level file update logging, when integrated with normal NFS processing capabilities, can have comparable throughput as a vanilla NFS server. It does incur 3$\sim$5 msec of latency penalty when the update request percentage is above 36%. The update request percentage in typical NFS workloads, as specified in the SPECsfs benchmark, is less than 12%.

- As shown in RFS-I+, if portability can be slightly compromised, a small kernel module can effectively reduce the context switch and memory copy overhead associated with the user-level implementation of the file update logging.

Chapter 5

Mariner: A Repairable Storage System

Mariner is a high performance repairable IP storage system designed to support continuous snapshotting and replication features for its clients without sacrificing performance. It provides storage access over iSCSI interface to its clients. The clients are typically NFS/CIFS servers and DBMS servers. The window for continuous data protection is configurable and depends on the amount of disk space allocated for historical data. *Mariner* does not require any modifications to its clients except provisioning of iSCSI drivers. *Mariner* features several unique innovations:

- 1. Comprehensive block-level versioning techniques that provide access to old data without significant performance degradation for current data access.
- 2. Integration of the block-level versioning scheme with a track-based disk logging technique [13] that is able to reduce the end-to-end latency of a synchronous disk write to less than 0.5 msec, the best result ever reported in the literature, and supports fast crash recovery.

- 3. Transparent reliable link-layer multicasting that exploits the VLAN support in modern Ethernet switches to perform in-network packet duplication and remove the bandwidth/latency penalty associated with replication.
- 4. Fault tolerance design that keeps *Mariner* available upon any single point of failure. The fault tolerance design is derived from a more generic fault-tolerance model for a 1-client-N-server system
- 5. User-level versioning file system architecture that provides end users the ability to navigate through file versions on a repairable storage server using the standard OS-supported file system interface.

This chapter focuses on the block-level versioning techniques and the fault tolerance design of *Mariner* . We briefly describe the integration with track-based logging and the transparent reliable link-layer multicast. The complete implementation details and evaluation results can be found at [42]). The user-level versioning file system is on-going work. We lay out only the basic design to show how does it integrate with block-level versioning.

Figure 5.1 shows the system architecture. The **client** node is a storage client such as file server or DBMS server. In addition, the client node has a central role in the fault tolerance design. There are two types of local storage nodes: the **primary** node, and the **trail** node. The primary node supports access to the current data. It is a standard storage server enhanced with features supporting fault tolerance. The trail node supports access to the historical data as well as the fault tolerance protocol. Thus, the trail node is a special storage server that uses own logging and versioning software.

Mariner handles consistency among multiple storage replicas but not among multiple storage clients. It assumes no sharing of mutable data among storage clients; or if they do, it is the upper-level software's responsibilities (file system

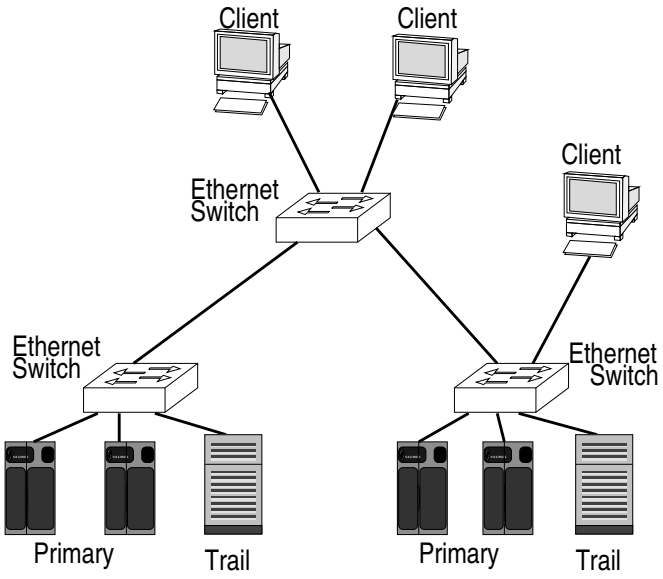


Figure 5.1: Mariner System Architecture

or DBMS) to handle the consistency issue. Essentially, every client and the storage node that it connects to, constitute an independent 1-client-N-server system. For the sake of simplicity, we consider only one client in most of the discussions. Section 5.5.1 shows an overview of such 1-client-N-server system with storage organization and data flows. However, the advantage of *transparent link-layer multicasting* probably shows only in the context of large scale storage systems with multiple clients.

5.1 Comprehensive block-level versioning

Mariner's comprehensive block-level versioning is designed to provide storage snapshot at any point in time within a **protection window**[1]. The snapshots, however, are not directly accessible. They are set up upon user request and torn down

[1] A protection window is the time period in which any update is undo-able.

after the user finishes accessing them. There are certain overheads associated with setting up a snapshot and there are some resources being used to maintain an accessible snapshot. This usage model is designed to cope with (conceptually) infinite number of snapshots and to maintain a good performance for current data access.

Mariner does not support version branching. In general snapshots are read-only although temporary write is supported while a snapshot is set up to be accessible. This flexibility is very useful because for upper level applications such as file system or DBMS, a storage snapshot has only crash consistency, i.e., as if the system crashed at that particular point in time. To provide useful information at file system or database level, the file system or DBMS need to perform an fsck-like procedure, which requires modifying to the snapshot. All temporary updates to an accessible snapshot are discarded after the snapshot is torn down. Part of the resources associated with an accessible snapshot are for supporting these temporary writes.

5.1.1 Design Issues

While designing mariner's comprehensive versioning, we had an ambitious goal of not only to match but to outperform the standard (non-versioning) storage, especially on write request latency. We adapted the fast logging techniques from the original trail [13], including the self-describing log record format and the techniques to detect disk head position. For each write request, a log record is written to a place close to current disk head. Observing from a large scale, logging disk is used sequentially in "rounds". In each round the log disk head moves from the first track to the last track, and then wraps back to the first track for another round.

To ensure every write within the protection window is undo-able, we should either (1) keep the before image data inside each log record together with the payload (current data) with the payload; or (2) make sure that the earlier log record (R') - that contains the before-image data in its payload - does not get recycled

before $timestampofR + protectionwindow$. With 1, the write performance will be degraded because of additional before-image read and write. With 2, the log records containing current data can never be recycled. To prove it, assuming some current data was recycled at time T and there was an overwrite to the data at T+1, the before-image for the overwrite is not available and this violates the assumption of 2. This means that the log storage will be fragmented with current data, which would make log storage write performance less efficient. It also means the a big portion of the log storage will be wasted on retaining current data because current data read is usually better served by current data storage on which sequence logical blocks translate to sequential physical blocks.

The trade-off between policy 1 and policy 2 applies to both file systems and storage systems. RFS-O([75] (Section 4.1.2) used policy 1 and its performance is much worse than standard (non-versioning) file system. The RFS-A/RFS-I([76], Section 4.1.2) used policy 2, their performance matches standard file system. Note that even when log storage is sub-optimal, log disk still has some advantage in handling writes because random writes can be converted to sequential or close to sequential writes.

In mariner, we tried to find a sweet spot between policy 1 and policy 2 to achieve even better performance. According some trace studies [17], most data in the storage system are cold data, i.e., data that is not likely to be overwritten again or frequently. Some data are cold data from the beginning (i.e., write once data); some data was active in the beginning but cooled down over time. If we can identify a **current data warm window**, where most overwrites should have happened, if they are ever going to happen, we can recycle the current data from the log storage after passing this window. Setting the *current data warm window* size to 0 is equivalent to policy 1; setting the size to infinity is equivalent to policy 2. By tuning this window size properly, we could improve log storage usage without too much overhead

on before-image read.

We had also considered another alternative to strike a balance. As described in the beginning of this section, the log disk are used in rounds. We could recycle a current data block whenever it is scanned in the next round by the block allocation algorithm. Essentially the current data is recycled upon the needs of free blocks. This approach definitely guarantees the log disk performance. But it has some other drawbacks. Firstly there is no guarantee on how long a current data block stays on the log disk. It could be longer or shorter than the *current data warm window*. While it does not hurt to stay longer, staying shorter could cause excessive before-image-read overhead. Secondly unlike a window size, this approach is less flexible to tune. We didn't implement this policy but considered it as one of the hints to help decide the size of *current data warm window*. That is, when block allocation algorithm hits a current data block, quite often the current data block should be recyclable.

Keeping some current data out of the log storage also has other pros and cons. One advantage is that it reduce the foot print of the versioning metadata that maps a current data block number to its physical block number on the log storage. One disadvantage is that if the before-image-read involves contacting another server, it bring network overhead in addition to the disk overhead; it also brings more complexity to the fault-tolerance design since the logging server is no longer a stand-alone server and needs support from another server.

5.1.2 Data Logging and Garbage Collection

Before getting into the details of the versioning techniques, we first clarify three terms - logical block, physical block and local physical block. These terms are often used differently in different contexts(e.g. file system, disk geometry). In the context of block-level versioning, the versioning device is a virtual block device based on log storage, which consists of multiple log disks with the same capacity.

The **logical block** refers to the block number of the versioning device as used in user's read/write request. The **physical block** refers to the block of the log storage. The **local physical block** refers to the block on a log disk. We use a simple strip-mapping between the *physical block* and the *local physical block*. Assuming there are N disks, the *local physical block number* LPBN on *i*th log disk corresponds to the *physical block number* $PBN = LPBN * N + i$.

The self-describing log record may contain batched multiple requests. Each request includes sequence id, the logical block number (LBN), size, timestamp, payload and before-image related information. Given a write request, if the before-image is still on the log storage (could be a different log disk), the log record will contain the pointer to those before-image, i.e., the PBN of before-image data. Otherwise, the log record contains the actual before-image data read from current data storage (maybe on a different server).

A log record is recyclable if all the local physical blocks in all of its requests are recyclable. A local physical block is recyclable if it contains current data that is written more than *current data warm window* ago; or if it contains old data that is overwritten more than *protection window* ago. For example, if a logical block L was written to physical block P1 at time T1, and later was written to physical block P2 at time T2, and was not written again; P1 can be recycled at time $T2 + protectionwindow$, and P2 can be recycled at $T2 + currentdatawarmwindow$. Unlike the FIFO-based garbage collection in the original *Trail* [13], mariner's garbage collection cannot guarantee that the physical blocks under current disk head is always free. Although given proper *protection window*, *current data warm window*, and log disk capacity, the block allocation need not look too far for free blocks upon each allocation.

5.1.3 Metadata Organizations

To support historical data access, *Mariner* maintains a *block map* that maps $< LBN, timestamp >$ to $< PBN >$. This index data structure is organized as an external B tree. The B tree entry is recycled with the same policy as the *physical block*, as described above.

To implement the *protection window* and the *current data warm window*, an **allocation map** is used to keep track of the block usage indexed by LPBN (*local physical block number*). Each entry contains a flag and a timestamp. The flag indicates whether it contains current data or old data. For current data, the timestamp corresponds to the time of the last write. For old data, the timestamp corresponds to the time of the overwrite. Given the example in Section 5.1.2. At time T1, the allocation map[P1] has current data flag and timestamp T1. At time T2, the allocation map[P2] has current data flag and timestamp T2; in the mean time, the allocation map[P1] is updated with old-data flag and timestamp of T2, As shown in this example, each write may cause updates to two allocation map entries. The entry for new data is returned by block allocation algorithm. The entry for current data can be found via the *block map*.

Accessing *block map* for normal data write has one potential problem. The *block map* is organized as external B-tree due to its large size. The access time could be long due to disk seeks. To speed up the look-up of current data block, *Mariner* added another versioning metadata structure called **current-block map**. This map contains mapping from LBN to PBN. This map is much smaller because only current data is included. In fact only the current data that stays on the log storage are needed. If a look-up didn't find any entry, the before-image needs to be read from current data storage. Therefore reducing *current data warm window* size could further decrease the size of *current-block map*. Hopefully the current-block map will be smaller enough to fit into memory.

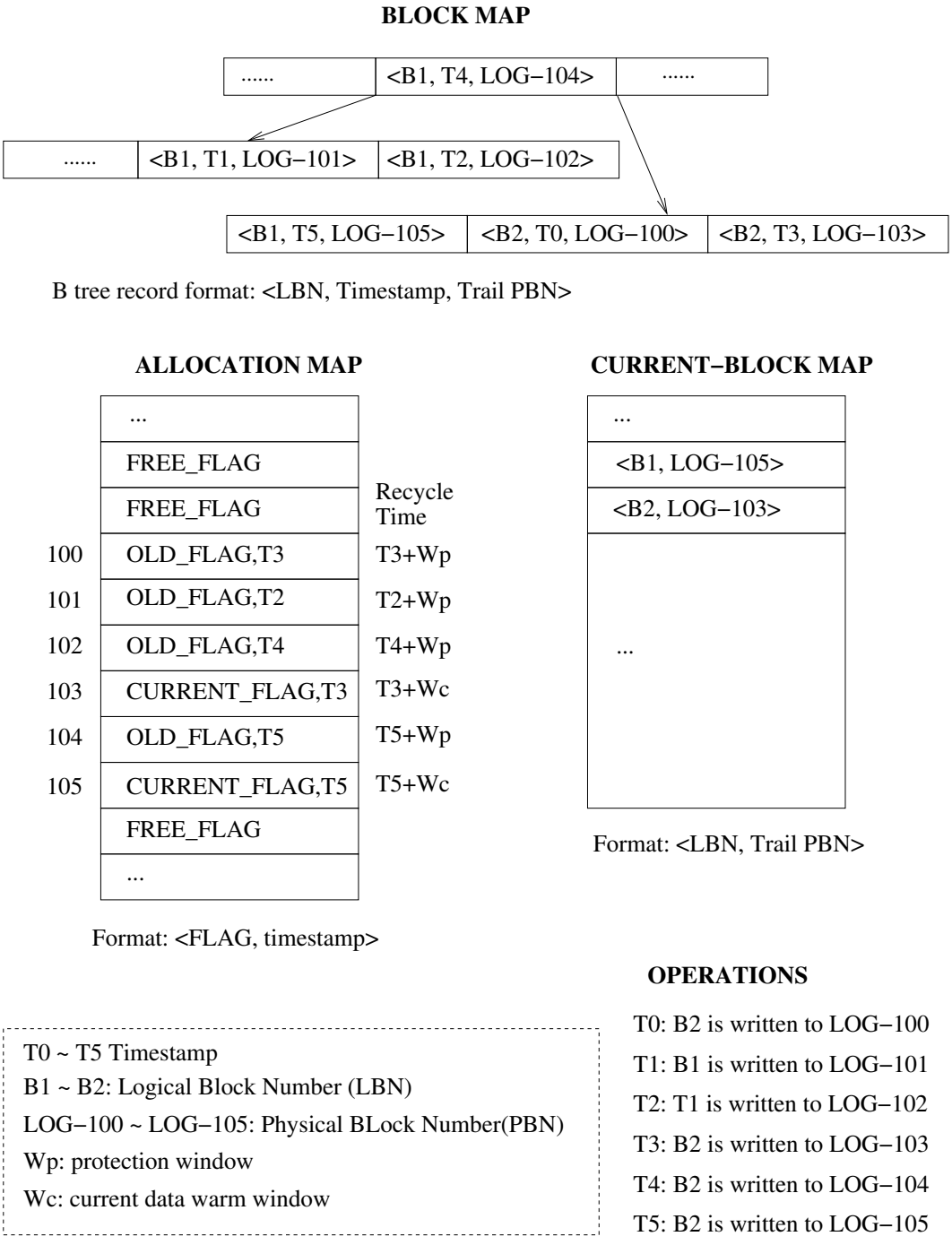


Figure 5.2: An example to illustrate the versioning metadata.

All metadata structures are flushed to the disk periodically, together with sequence id of the last log record that has been incorporated into these metadata.

5.2 Track-Based Logging

The *Trail* node plays a key role in *Mariner* by providing a low-latency and high-throughput synchronous disk write, which is essential to support both continuous data protection, and fault-tolerance. The synchronous writes are stored on *Trail* log device as trail log records. The *Trail* log records are self-describing. Upon failures, *Trail* can quickly identify the latest log records and use them to recover the data and metadata consistency among all *Mariner* nodes. (Section 5.6.1).

The core technique is to accurately estimate the disk head position and log data to the free space close to that location. The destination of each log record is decided as late as possible before it is scheduled to disk. To estimate the disk head position, *Trail* requires pre-computed on disk geometry and hardware characteristics information(e.g. rotation speed). *Trail* also conducts re-calibration periodically. To have free space available near the disk head location most of the time, Trail utilizes the log disk in circular fashion and whenever disk head is idle and current track has reached high utilization, *Trail* moves the disk head to the next track to guarantee that the disk head is on an empty track and ready for next disk write, hence the name *track-based logging*. *Trail* log device is consists of multiple log disks. When a log record needs to be stored, usually a log disk which can provide lowest write latency is selected. The space utilization of different log disks is kept roughly balanced but not as strict as stripping device. Consequently, each write operation incurs very little rotational latency and zero seek delay, and can be completed under 500 μsec. The paper [42] contains implementation details and evaluation results.

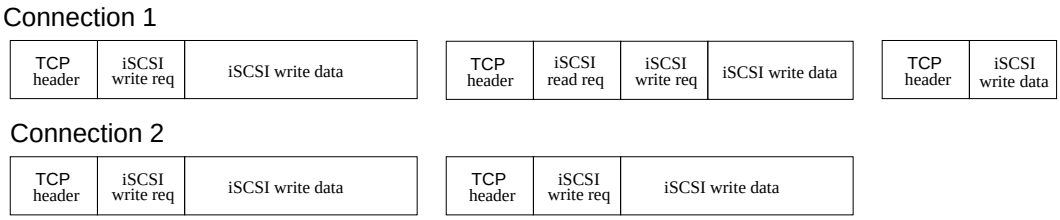


Figure 5.3: Skew in location of common data because of dissimilarity in TCP stream contents. Connection 1 `write` data gets displaced because of interspersed READ requests.

5.3 Transparent Reliable Multicast

A *Mariner*'s client needs to send each disk write to multiple storage servers for replication and versioning. The client is connected to the storage servers through iSCSI protocol, which in turn is built on top of TCP. A separate TCP connection is needed between client and each storage servers. The connection oriented nature of TCP requires that data sent over all the connections be sent independently, even though the data being transmitted over these connections is largely the same. *Mariner* exploits the VLAN support in modern Ethernet switches to build a transparent mirroring and reliable multicast (TRM) mechanism that greatly reduces the performance overhead of data replication on an Ethernet-based storage area network.

Logically, TRM integrates data replication logic with a software layer that sits below the TCP/IP stack and constantly look for common payload among TCP connections going to different destinations. When packets from different TCP connections share the same payload, TRM sends that payload as an Ethernet multicast packet that eventually is delivered to all the associated destination nodes. For example, after TRM detects that three packets carrying the same payload are sent to

Node 1, 2, and 3 over three independent TCP connections, it constructs an Ethernet multicast packet, which consists of the TCP/IP headers of the original three packets and one copy of the payload, and sends it out over a spanning tree to reach Node 1, 2, and 3. This spanning tree is constructed through either IGMP snooping or VLAN set-up [58]. When each receiver node receives the Ethernet multicast packet, its TRM layer reconstructs the original TCP/IP packet, and forwards it up through the TCP/IP stack. ACKs for each TCP connection are independently sent back from the receivers to the sender as unicast Ethernet packets. If for some reasons, the Ethernet multicast packet does not reach a particular receiver node, TCP's retransmission mechanism ensures that eventually the corresponding TCP/IP packet is retransmitted to that receiver via unicast. Essentially TRM relies completely on TCP for packet transport reliability. By leveraging tree-based link-layer multicast, TRM is able to reduce the additional latency/bandwidth penalty associated with data replication to close to zero, because data is actually replicated inside the network.

There are three key operations in TRM: (a) Duplication of data that is subsequently sent over individual TCP connections to mirrored storage devices, (b) Monitoring TCP connections for duplicate data and constructing multicast packets by using that data, and (c) Reconstructing the TCP streams based on the data and metadata received over multicast.

The task of identifying common payload among TCP connections is complicated by the fact that TCP does not preserve packet boundaries. If the amount of data transmitted over different TCP connections is different, the common data shared among these TCP connections may appear in different locations within the packets containing them. For instance, when multiple TCP connections corresponding to different iSCSI sessions carry `write` as well as `read` requests, only `write`

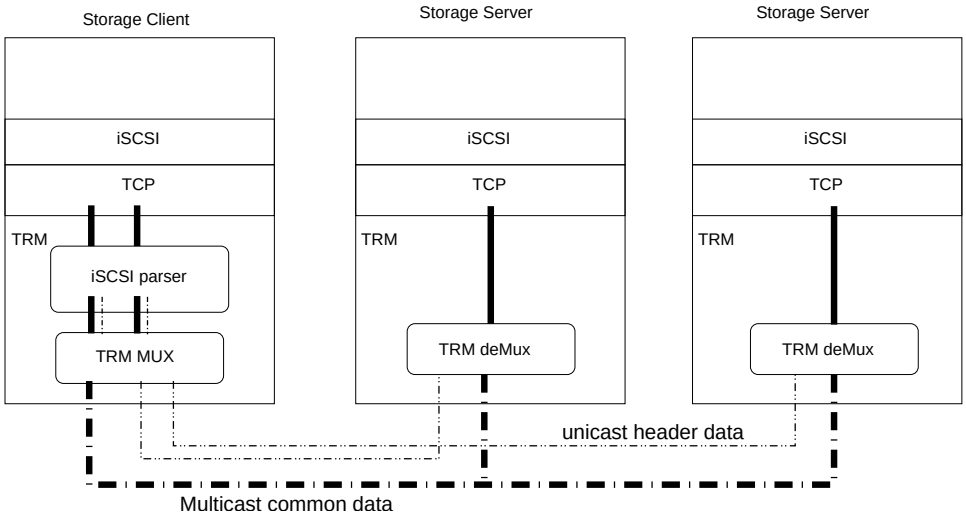


Figure 5.4: Architectural decomposition of iSCSI-based TRM. An iSCSI protocol parser keeps track of each TCP connection corresponding to different iSCSI sessions. The parser provides the description of each packet to the TRM multiplexer, which splits each TCP stream into multicast and unicast substreams. Each storage node and *Trail* node in *Mariner* is augmented with a TRM de-multiplexer that reconstructs the TCP stream from the received unicast data and multicast data.

data need to be duplicated, but not `read` requests. Therefore, the set of packets going through the connections to the primary/secondary storage nodes are quite different from those going through the connection to the *Trail* node. Owing to this skew, it is impossible to merge all packets containing common data because common data may appear in very different locations within these packets. Figure 5.3 illustrates the difficulties in merging packets with skewed common payload.

TRM exploits the semantics of the iSCSI protocol's to focus only on disk write payloads, and significantly increases the chance of successful packet merging without requiring complicated comparison logic. In particular, TRM employs an iSCSI parser to analyze each iSCSI request to pin-point the precise location of the payload

of each `write` request. It then passes each packet parsed to a TRM multiplexer along with a description of the packet. Based on this description, the TRM MUX identifies the common data and sends it over to the storage/*Trail* nodes using a single multicast packet. Then the TRM MUX sends the connection-specific payload portion of each packet as a unicast packet to each peer of the TCP connection. Reconstruction of each TCP stream is fairly straightforward. The storage/*Trail* nodes are augmented with a TRM deMux layer, which is responsible for receiving the multicast data and the connection specific unicast data. Using the information embedded in multicast data and the unicast data, the TRM deMux layer reconstructs the original TCP packets and passes them to the TCP/IP stack for further processing. The software architecture of iSCSI-based TRM is shown in Figure 5.4. For complete implementation details and evaluation results, please see [40].

5.4 User-Level Versioning File System

Mariner's block-level versioning provides storage snapshots at any point in time. While this may be enough for many users, others may also prefer some high level information such as how many versions a file has within a period of time. So can we support traditional versioning file system features based on block-level versioning storage?

One possibility is to modify existing file system to take advantage of the versioning storage. This approach is probably too intrusive, complex and not portable. Instead, we propose a user-level versioning file system (UVFS) approach. The UVFS should be able to work with any block-level versioning storage and standard file system.

A unique feature of UVFS is that it uses pathname, rather than inode number, to identify a file or directory. At user level it is difficult to trace through the file system

metadata structures. In contrast pathname is a much more universal and portable concept across file systems. This scheme does have a minor drawback - it cannot continuously trace the modifications to a file after it has been renamed.

Using pathname as identifier leads to several observations. Due to hard links, one inode can be associated with multiple UVFS objects. If a file is renamed, it becomes a different UVFS object. A pathname can be created and then deleted and then created again, corresponding to many incarnations of the UVFS object with different inodes. An incarnation is represented by a triple value of $< pathname, Tstart, Tend >$. The Tstart and Tend represent one pair of the creation and deletion time.

In this section, **snapshot** refers to the object or system image at a point in time; **version** refers to a modification of a object as indicated by the last modification time. The number of *snapshots* is unlimited but the number of *versions* is limited. Both *snapshot* and *version* are represented with a tuple value $< pathname, T >$. The block-level versioning storage system provides access to **snapshots**. The UVFS provides information about **versions**. The version information is provided by six closely related operations. Most of the operations are recursive and based on other operations.

- *1. Access a snapshot of a pathname*. To provide snapshot access at file system level, first we request the versioning storage to create a snapshot device with certain timestamp. The snapshot device is mounted with the file system. Due to the crash consistency nature of the snapshot device, an fsck is needed. Instead of the usual fsck, *Mariner* implements a customized fast fsck [43]. This fast fsck restores the file system level consistency of the storage snapshot device before the file system is mounted. Once the file system is mounted, it can be used to access any pathname at the requested snapshot time.
- *2. List all versions of an incarnation within a time range*: UVFS uses *last*

modify time attribute to identify different versions of an incarnation. Given an incarnation $< pathname, Tstart, Tend >$ and a duration $< D1, D2 >$. The UVFS first refines the time range to be $T1 =< max(D1, Tstart), T2 = min(D2, Tend) >$. The algorithm iterates on the time range, starting from $< pathname, T1, T2 >$. The UVFS accesses the snapshot $< pathname, T2 - \delta >$, and retrieves its the last modification time, say T. If $T \geq T1$, the UVFS finds one version $< pathname, T >$ and adds it to the list of versions. The next iteration continues at $pathname, T1, T$. If $T < T1$, we have found all relevant versions.

- *3. List all incarnations of a pathname within a time range*: The creation or deletion of a pathname modifies its parent directory. Therefore one can discover incarnations of a pathname from parent directory's version changes. The algorithm does recursion on the pathname. Given a pathname and a duration $< /a/b/c/d, D1, D2 >$, UVFS first solve the problem for $< /a/b/c, D1, D2 >$. Then for each each incarnation of $< /a/b/c.Tstart, Tend >$, we use operation 2 to find all versions of /a/b/c in the range, such as $< /a/b/c, T1 >$, $< /a/b/c/, T2 >$, ... $< /a/b/c, Tn >$. By accessing the snapshot of $< /a/b/c, Ti >$ and $< /a/b/c, Tj >$ and comparing their content, we know whether d was created or deleted at time Tj. The list T1, T2, ... Tn must have contained all the creation time and deletion time of /a/b/c/d, this way we found all the incarnations of /a/b/c/d from Tstart to Tend. Combining the results for all incarnations of /a/b/c from D1 to D2, we got all incarnations of /a/b/c/d from D1 to D2. The recursion stops when it is reach root directory ("/"), which by definition has only one incarnation $< /, -infinity, infinity >$.
- *4. List all versions of a pathname within a time range* This operation is based on operation 2 and operation 3. First we find all the incarnations. Then we

find all the versions of each incarnation. Finally we sum up all the versions from all the incarnations. It is possible that the versions of one incarnation are not logically related to versions of another incarnation. It is up to the user to decide how to use the output of this operation.

- *5. List all pathnames under a parent pathname within a time range*: This operation is based on all the above operations. The algorithm does a recursion on sub-directories similar to the breadth-first search. Given a pathname /a/b/c and duration D1, D2. UVFS discovers every version of the input pathname with operation 2, e.g., $< /a/b/c, T1 >$, $< /a/b/c, T2 >$, ... ,$< /a/b/c, Tn >$. From these versions, UVFS discovers all the all pathnames under /a/b/c during $< D1, D2 >$, in a way similar to operation 3. The recursion continues for each pathname under /a/b/c. The recursion stops when a pathname is a file or an empty directory.
- *6. List all versioning information of the whole file system* Conceptually this operation could be simply built upon 4 and 5 - first find all the pathnames under root directory (operation 5) and then find all the versions of each file (operation 4). The versions of all directories should have already been found in step 5. In practice, the implementation should be smarter so that it does not repeatedly search versions of directories.

Finally we describe the implementation of snapshot access in more detail. Other operations are pure algorithms using only getattr() and readdir() system call. Assuming an NFS environment, the NFS server is the *Mariner* client node. To access a snapshot, the user on a NFS client specifies a pathname and a timestamp in a request. The request is sent to the NFS server and then to the storage server with comprehensive block-level versioning - the *Trail* node. When the *Trail* node receives the request, it creates a virtual device corresponding to the timestamp, does the fast fsck [43], and adds the virtual device to the list of iSCSI targets. The iSCSI

initiator on the client node connects to the new target device and generates a new iSCSI device. The NFS server then mounts this new iSCSI device and and exports it to the NFS client. Finally, the NFS client creates a local directory and mounts the NFS server directory. The above procedure involves collaborations of three UVFS daemons, one on NFS client, one on NFS server, and the third on the *Trail* node. They communicate with one another through a proprietary protocol.

To optimize the performance of the proposed user-level versioning file system, *Mariner* employs various forms of caching. At the NFS server side, *Mariner* it caches the modification times for each file, and uses them to determine which snapshots should be used to answer a particular query. In addition, *Mariner* retains the mount connections between NFS clients and servers and the virtual devices created at the *Trail* node after servicing the corresponding snapshot access query, so as to reuse them for future queries.

5.5 Fault Tolerance Model for a 1-Client-N-Server Storage System

In this section and the next few sections, we are going to describe a fault-tolerance model for a 1-client-N-server storage system and how to apply this model to *Mariner*, including implementation details and evaluation results.

This fault tolerance model is for fail-stop system. The failures being covered include ”software failure/machine crash”, ”power failure”, ”network failure”, and ”disk failure”. From the failure recovery’s viewpoint, all failures can be classified into two categories: ”failure with data loss” and ”failure without data loss”. For the sake of simplicity, we do not step into details of partial data loss.

5.5.1 Overview of the 1-client-N-server system

Before getting into the details of the recovery model, we first describe the characteristics of the storage system that this model is designed for. As shown in figure 5.5, the storage system is composed of one storage client and N storage servers. Each storage server knows the identity (IP address or DNS name) of the client. The client waits for the server to establish connection. The storage client accepts read and write request from application and distributes it to N servers. A common example of such system is an NFS/DBMS server (the storage client) and N back-end replication storage servers. In the fault tolerance model, the client also acts as the central management point. It is responsible for admitting servers, detecting/handling failures and co-ordinating node repair. The storage servers do simple replication to improve system availability and read performance. All write requests are logged by some servers. Logs are essential for the fault tolerance model, but they could also be used for other purposes such as comprehensive versioning and low-latency-write

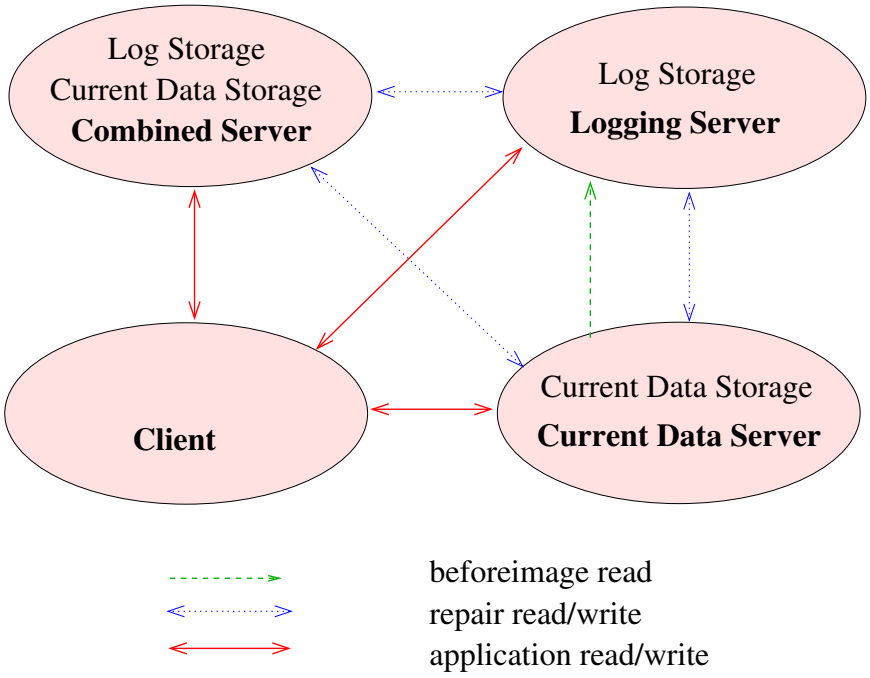


Figure 5.5: Overview of the 1-client-N-server system

(Section 5.1, 5.2).

In our model, a server can have current data storage, which stores current image of data; and log storage, which stores the combined redo/undo log for recent writes; or both. A server with only current data storage is called **current-data server**. A server with only log storage is called **logging server**. A server with both kinds of storage is called **combined server**. If a server is configured to do logging while serving read/write request, we state that it is in **RDWR-LOG** state. If a server does not need to do logging while serving read/write request, we state that it is in **RDWR** state. Obviously not all servers can play all the roles. For example, a logging server cannot be in the RDWR state and a current-data server cannot be in the RDWR-LOG state.

Log records are recycled after certain time. Because of this, a logging server usually does not have complete current data image (unless all data is modified recently). This kind of server is derived from Mariner's trail server 5.2. Such logging server needs another RDWR to provide before-image for undo-based logging 5.1. The RDWR server is called the before-image server of the logging server.

Read requests are dispatched to one of the available RDWR servers. Write requests are replicated to all the available servers through a modified **2-phase-commit** protocol (Figure 5.6). In phase 1 the write request is sent to all servers but only committed on RDWR-LOG servers. On RDWR servers it is treated as a pre-write. In phase 2 after receiving confirmation that the write request has been committed to the log storage, client sends notification to RDWR servers to commit the pre-write. Each write request bears a unique **global sequence number** which grows monotonically. Whenever possible, the notification is piggy-backed through subsequent read or write requests. The modified 2-phase commit protocol has a different goal as the traditional 2-phase commit protocol (all-or-nothing). That is to guarantee that a successful write request is logged on at least one log storage before committed to

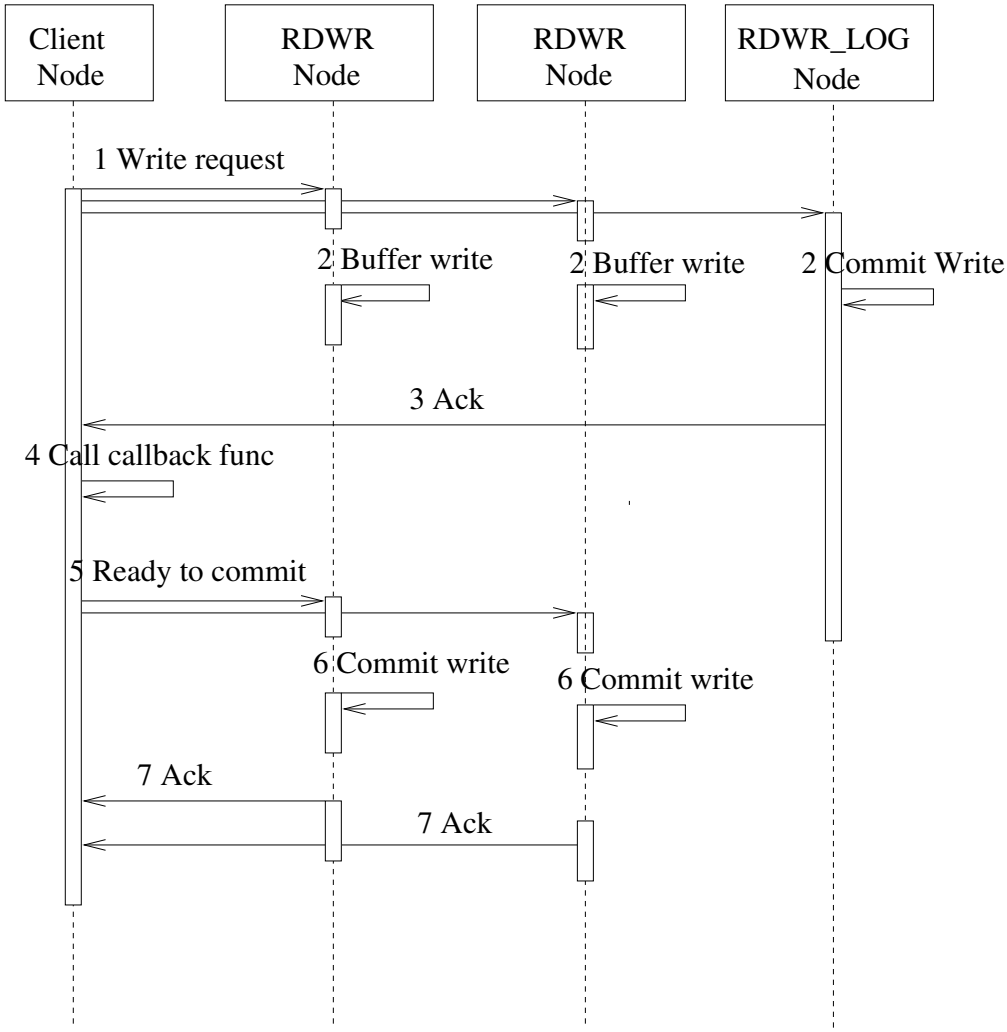


Figure 5.6: The modified 2-phase commit protocol

current data storage.

There are other data flows in the system, as shown in Figure 5.5. A logging server may need to read the before-image, a repairee may need to read repairer's current data and log records, a repairer may need to update repairee's current data storage and log storage.

5.5.2 Datainfo

Each server's data status is represented by a **datainfo**, which is a triple value of **(recyclepoint, syncpoint, snappoint)**. All the values are some particular global sequence number. The **snappoint** represents the last write committed to the current data storage. The **syncpoint** represents the last write committed to the log storage. The **recyclepoint** represents the most recent log record that has been recycled. All three points are initialized to -1. The maximum *snappoints* or *syncpoints* of all servers is called **system progress point**.

The *datainfo* is normally maintained in memory but the on-disk copy is also needed in case of failures. There is no extra overhead to maintain the *syncpoint* on disk. The *global sequence number* is stored in each log records, the *syncpoint* essentially is the largest *global sequence number* in the log records on disk. The *snappoint*, however, cannot take such free ride to the current data storage. To limit performance penalty, the *snappoint* is flushed to disk periodically instead of upon each write request. Therefore, the on-disk *snappoint* is not accurate.

The only persistent metadata on client is an **active-server-list**. Upon each server node joining and leaving the system, client commits the update to the *active-server-list* to disk. If the list is lost due to client disk failure, it is conservatively reset to include all servers. The reason that client needs this list is because one of the active servers should have the *system progress point.* When client fails and restarts, only after all the active servers has joined, client can compute the *system progress*

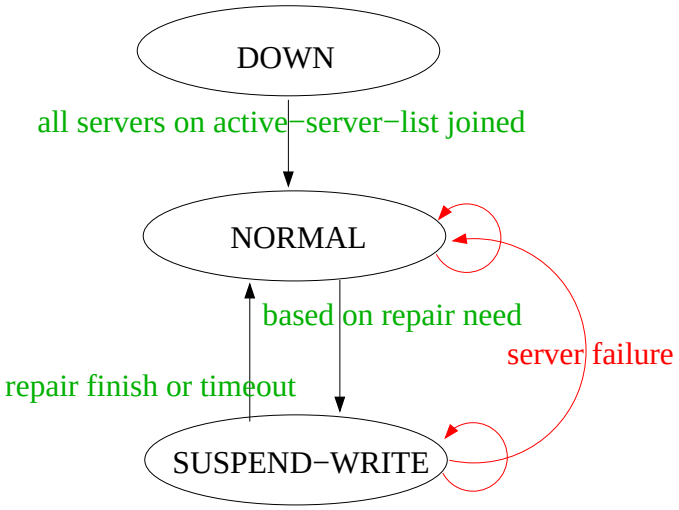
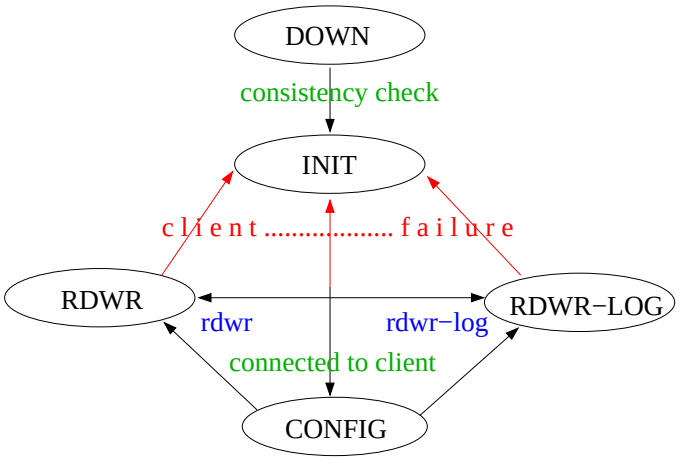


Figure 5.7: Client state transition

point and start responding read/write requests. Maintaining an *active-server-list* to compute *system progress point* is a much better choice than maintaining the *system progress point* directly. Firstly, the *active-server-list* changes very rare. The *system progress point* changes so often that it is not feasible to maintain it accurately on disk, just like the *snappoint*. Moreover, the *system progress point* on the client becomes meaningless if some active server had data loss.

The storage system is available for reading if client is up, one of the current-data server or combined server is up and the server's *snappoint* equals *system progress point*. The system is available for write if client is up, one of the logging or combined server is up and the server's *syncpoint* equals *system progress point*.



The figure omits the links from each state
to the DOWN state upon crash failures

Figure 5.8: Server state transition

5.5.3 State Transition and Control Messages

In this section we describe the client and server's state and their transitions as shown in Figure 5.7 and Figure 5.8. We also describe the control messages passed between client and server.

Client can be in three possible states:

- DOWN:

  DOWN state is for self initialization, mostly to compute the *system progress point*. Client enters this state after restarting from crash failures. Initially client tries to retrieve the active-server-list from disk. Upon disk failure and data loss, the list is reset to include all the servers. Client then initializes the listen socket and waits for new connections from the servers. Client waits all servers on the *active-server-list* to join with their NODE-INSERT message and calculates the *system progress point*. If any server had data loss, the *system progress point* is not guaranteed to be found among active servers. In

that case, the list is also reset to include all servers and client has to wait all the servers to join.

- NORMAL:
  Client enters NORMAL state from DOWN state after the *system datainfo* is computed. Client may also enter NORMAL state from SUSPEND-WRITE state after a repairee has caught up with *system progress point*, or after client stayed too long in the SUSPEND-WRITE state.

  In this state, client schedules the read/write requests, detects and handles failures(Section 5.5.6, admits newly recovered servers, schedules repair (Section 5.5.5 and assigns roles to each server. Upon receiving any control messages or detecting any failure, client does *system configuration*. It examines each active server's *datainfo*. If the *datainfo* is up to date, client configures its next state to be RDWR or RDWR-LOG through SET-STATE-RDWR/SET-STATE-RDWR-LOG messages. If a *logging server* is configured to RDWR-LOG state, a before-image server is assigned through SET-BEFOREIMAGE message. For a server whose *datainfo* is not up to date, client schedules a repair through REPAIR or BEINGREPAIRED message.

- SUSPEND-WRITE:
  SUSPEND-WRITE state is entered when client stops serving new write request and waits for a repairee to be brought up to date. Client enters this state from NORMAL, after an repairee's datainfo is very close to *system progress point* (e.g. $<$ 100 log records missing) or after the client realizes that the repair speed is not fast enough(e.g. $<$ new write throughput/2). This state is set so that the repairee can completely catch up *system progress point* and start serving read/write request, which is not easy if new write requests constantly comes in during the repair. In this state, client behaves the same as in

NORMAL state, except that it does not serve write request.

Server can be in one of the following five states:

- DOWN:
  DOWN is a state for self-consistency check, mostly to discover its own *datainfo*. Server enters this state after restart from crash failures. The details of self-consistency check is described in Section 5.6.1.

- INIT:
  INIT is a state when server waits to join the client. Server enters this state from the DOWN state after self-consistency check, or from any other states after client failure is detected. (Section 5.5.6. In this state, server keeps trying to connect to client and register with NODE-INSERT message.

- CONFIG:
  CONFIG is a state where server waits for client's message regarding next state or next task. The messages could be SET-STATE-RDWR, SET-STATE-RDWR-LOG, REPAIR, BEINGREPAIRED, and SET-BEFORE-IMAGE. Server enters this state from the INIT state after a client has joined it.

- RDWR:
  RDWR is a state where server handles read/write request without logging. A current-data server or combined server may enter this state (from RDWR-LOG or CONFIG after receiving SET-STATE-RDWR message. The server sends back SET-STATE-ACK to client to acknowledge that it is ready to serve read/write requests. In this state, control messages are handled in the same way as in CONFIG state.

- RDWR-LOG:
  RDWR is a state where server handles read/write request with Logging. A

logging server or combined server may enter this state (from RDWR or CONFIG) after receiving SET-STATE-RDWR-LOG message. Upon switching to this state, a server needs to get prepared to start logging (For example, in mariner implementation, upon this message a combined server would reset the log file, recyclepoint, and syncpoint). If the server is a logging server, after receiving this message it establishes a data channel to read from the before-image server. After all preparations are done, the server sends back a SET-STATE-ACK message to the client. In this state, control messages are handled in the same way as in the CONFIG and RDWR states. If the server encounters failures in accessing before-image server, a BEFORE-IMAGE-FAIL message is sent to the client.

Next we describe the control messages passed between client and server. Each control messages can have some parameters, other than those listed explicitly, there is a timestamp associated with each message. The messages sent from client to server include:

- SET-STATE-RDWR:
  This message set a current-data server or combined server to RDWR state.

- SET-STATE-RDWR-LOG (before-image-server-id):
  This message set a logging server or combined server to RDWR-LOG state. For logging server, a before-image server id is also provided.

- REPAIR (repairee-id, repairee-datainfo) This message instructs a server to repair another server. Upon this message, the repairer examines it's own *datainfo* compare it against repairee's datainfo and tries to remotely update repairee's current data storage or log storage. Section 5.5.5 contains more detailed description. The repair may finish successfully or stop upon repairee

failures. Regardless, the actual progress is reported to client via REPAIR-ACK message.

- BEINGREPAIRED (repairer-id) This message is handled almost the same way as REPAIR. The only difference is that it is initiated by the repairee rather than the repairer. The repairee tries to access repairer's current data storage or log storage remotely to update its own storage.

- SET-BEFOREIMAGE (before image-server-id) This message provides logging node with a new before image server id. It could be triggered if either a logging server or the client detected failure of the before-image server. Note that these two failures do not necessarily occur simultaneously due to network partition problem.

- SET-SERVER-SYNCPOINT (syncpoint) This messages updates a repairee's datainfo after client receives an REPAIR-ACK message. The reason that it is sent by the client rather than the repairer is because we want to conform to our simple message passing model - control messages are only between client and server, there is no server-server communication.

The messages sent from server to client include:

- NODE-INSERT (datainfo) This message is for a server node to join the storage system with its own *datainfo*. This message is sent when a server is in INIT state.

- SET-STATE-ACK This message acknowledges SET-STATE-RDWR or SET-STATE-RDWR-LOG message indicating that a server is ready for serving read/write requests. This acknowledgment is needed because the read/write requests are sent over data channel in different TCP connections and can be re-ordered with control messages.

- REPAIR-ACK (repairee-datainfo) This message acknowledges that REPAIR message has been processed and the repair operation has finished. The repair progress is returned via repairee-datainfo parameter.

- BEINGREPAIRED-ACK (repairee-datainfo) This message is similar to REPAIR-ACK, but acknowledging BEINGREPAIRED message instead.

- BEFOREIMAGE-FAIL (before-image-server-id) This message is sent when a logging server detects the failure of its before-image server.

5.5.4 Consistent view

Client and server may reach different views regarding each other's state. An obvious example is when network is down, all nodes consider themselves alive but other nodes dead. In this section we prove that (1) eventually client and server come to consistent view; and (2) the transient inconsistency does not matter. Table 5.2 and Table 5.1 enumerate all possible views and how do inconsistent views transit to consistent views.

We skipped client state SUSPEND-WRITE in both tables because SUSPEND-WRITE is almost same as NORMAL state. The only difference is that write requests are suspended in that state, which is completely a local data channel issue. As far as client's view of server and server's view of client are concerned, SUSPEND-WRITE state is equivalent to NORMAL state. In other words, the NORMAL state in this subsection should be understood as "NORMAL-OR-SUSPEND-WRITE" state.

For the sake of simplicity, the recovery model assumes when a node dies, it does not join the system again before a certain duration(for example, 10 seconds) has passed. The assumption makes sure that by the time a node rejoins the system, its failure must have been detected by other nodes through the missing heart-beat

message. In other words, it is not possible for a server to fail and restart quickly without being noticed.

	client state DOWN	client state NORMAL
server state DOWN	1 DOWN	6 DOWN→7
server state INIT	2 DOWN	7 DOWN→8
server state CONFIG	3 NORMAL→2	8 NORMAL
server state RDWR	4 NORMAL→2	9 NORMAL
server state RDWR-LOG	5 NORMAL→2	10 NORMAL

Table 5.1: Server's view of client when client and server are in different states.

Table 5.1 shows a server's view of a client when the client and the server are in various states. All the views are numbered. 2, 8, 9 and 10 are consistent with the client's state. Views 3, 4, 5 are inconsistent views but they will transition to 2 shortly after the server detects client failure. The inconsistency at 3, 4, 5 indicates a normal delay between failure occurrence and failure detection. Transition from 6 to 7 is an internal transition after the server finishes self-consistency check. Transition from 7 to 8 is because the server has re-connected to the client. The inconsistency at 6 and 7 doesn't matter because it simply means server is trying to rejoin the system.

Table 5.2 shows a client's view of a server when the client and the server are in various states. Views 1,3, 10, 12, 16, and 22 are consistent with the server's state. Views 2, 4, 5, 6 are inconsistent and eventually transition to 1 or 3 shortly after the client detects server failure. The inconsistency indicates a normal delay between failure occurrence and failure detection.

Views 7 and 8 are inconsistent because client does not distinguish between

	server state DOWN	server state INIT	server state CONFIG	server state RDWR	server state RDWR--LOG
client state DOWN	1 DOWN	7 DOWN →10	9 DOWN →7	13 DOWN →7	18 DOWN →7
client state DOWN	2 CONFIG →1		10 CONFIG		
client state NORMAL	3 DOWN	8 DOWN →12	11 DOWN →8	14 DOWN →8	19 DOWN →8
client state NORMAL	4 CONFIG →3		12 CONFIG	15 CONFIG →16	20 CONFIG →22
client state NORMAL	5 RDWR →3			16 RDWR	21 RDWR →22
client state NORMAL	6 RDWR--LOG→3			17 RDWR--LOG →16	22 RDWR--LOG

Table 5.2: Client's view of a server when client and the server's are in various states. Note that when client is in DOWN state, it still accepts server node to compute the *system syncpoint.* But no servers will be configured to RDWR or RDWR-LOG state. Client does not handle read/write request in DOWN state,

server's DOWN or INIT state. Transition from 7 and 8 indicates that eventually the server connects to the client. Client's view of server does not include CONFIG/RDWR/RDWR-LOG while server is in INIT state. The reason is because we assumed servers will pause for some time before trying to reconnect to clients, therefore it must have been detected as dead by the client.

Views 9, 13, 18 are inconsistent because client detects a server failure (maybe network) although the server is not dead. The client stops sending heart beat messages to the server. Eventually the server will consider the client as dead and transition to 7. Similar situation exists for 11, 14, and 19.

Views 15, 17, 20, 11 are inconsistent because a server has received a client's SET-STATE-RDWR/RDWR-LOG message(upon which server changes its state) but the client hasn't received the SET-STATE-ACK (upon which client changes its view of the server). Since no read/write request are sent during the state change, the transient inconsistency does not matter.

5.5.5 Repair

In this section, first we show some invariants of server *datainfo* in Table 5.3. Then we describe two repair methods and how to apply them. Finally we give a set of primitives that helps to implement the repair operation.

There are two possible repair methods, **full-copy** and **log-replay**. The *full-copy* method can be used only to repair current data storage. It simply copies each block from repairer's current data storage to the repairee's current data storage. The log-replay method can be used to repair both current data storage and log storage. When it is used to repair log storage, it can bring the repairee's syncpoint more up to date; it can also bring repairee's recyclepoint to earlier time if the repairee has lost some old log records. Table 5.4 lists the criterion for selecting the repairer.

logging server	syncpoint >= recyclepoint >= -1, snappoint == -1
current-data server	syncpoint == recyclepoint == -1, snappoint >= -1
combined server	syncpoint >= recyclepoint >= -1, syncpoint >= snappoint >= -1

Table 5.3: Datainfo invariants

repair current data storage by full-copy	repairee snappoint < repairer snappoint
replay current data storage by log replay	repairer recyclepoint <= repairee snappoint < repairer syncpoint
repair log storage by log-replay adding older versions of data	repairer recyclepoint < repairee recyclepoint <= repairer syncpoint
repair log storage by log-replay adding newer data updates	repairer recyclepoint <= repairee syncpoint < repairer syncpoint

Table 5.4: Repairer qualification

One issue that warrants further discussion is the inaccuracy of *snappoint*. Section 5.5.2 has discussed inaccuracy of on-disk *snappoint* for performance reasons. Another reason that can cause inaccuracy is the *full-copy* repair if new write requests are serving while blocks being copied to the repairee. Suppose the repairer's *snappoint* moved from S1 to S2 during the *full-copy*, the repairee's snappoint will be set to S1. The repairee will contain some data between S1 and S2, but not not necessarily complete or in any order.

Regardless of the reason for inaccuracy, it is always on the conservative side. That is, all data updates before the *snappoint* must have been received. Also the

inaccuracy exists only on nodes that needs to be repaired. When a server returns to RDWR, RDWR-LOG state, the in-memory snappoint is always accurate. To cope with the *snappoint* inaccuracy, we require the the repair operation should be idempotent. Fortunately data updates to current data storage are naturally idempotent.

The recovery model does not specify the exact policy to select repairer and repair method. Log storage can only be repaired via log-replay. For current data storage, full-copy should be preferred upon complete data loss or after a server missed too many updates. Otherwise, log-replay is usually preferred. Usually the obsolete current data storage gets repaired, but it is not always necessary to repair the log storage. Sometimes, it is because the missing historical data is not that important. Sometimes, it is because the server is reconfigured to be a current data server.

If a repair operation could not complete because of failures in reading repairer log or updating repairee storage, the actual progress is notified to client. Eventually client will arrange another repair operation to bring the repairee up to date, probably with a different repairer. Table 5.5 summaries the primitives needed to implement repair operation and being-repaired operation.

Each repair brings repairee's data more up to date. But the *system progress point* could be changing constantly as new write requests being served during the repair. It may take multiple repair operations to get a repairee up to date. Client should throttle the bandwidth for new write requests to be lower than the repair bandwidth. When a repairee is close to *system progress point*, client suspends new write requests for a short period of time.

A repair can be initiated by either repairee or repairer. It is called a repair operation if initiated by repairer and being-repaired operation otherwise. In a being repaired operation, repairee read repairer's data remotely and updates its own data.

used by	repair primitive
being-repaired operation	setup data channel to read repairer current data storage
being-repaired operation	setup data channel to read repairer log storage
repair op	setup the data channel to update repairee current data storage
repair op	setup the data channel to update repairee log storage
log-replay	find out all relevant repairer log records according datainfo
log-replay	read next repairer log record
log-replay	play a log record on repairee
full-copy	find out all current data storage blocks
full-copy	read next block from repairer's current data storage
full-copy	write a data block to repairee's current data storage

Table 5.5: Primitives needed for repair operation

In a repair operation, the repairer reads its own data and updates repairee's storage remotely. This design is chosen for its simplicity, not because that it is not possible to make repair a collaborative effort of both repairee and repairer.

5.5.6 Failure detection and handling

There are five failure detection methods in our recovery model - all of them cannot distinguish whether the failure is because of software crash, power failure or network partitioning. There is no need for the distinction too because it does not affect failure handling.

- **1. Before-image read failure.** Server detects before-image server failure if before-image read encounters failure of time-out.
- **2. Repair failure.** In repair operation, repairer detects repairee failure if it encounters error updating repairee's storage. Similarly in being-repaired operation, repairee detects repairer failure if it encounters error reading repairer's storage.
- **3. Control channel failure.** If client or server cannot send control message successfully(TCP error), they consider the other node dead.
- **4. Heart-beat timeout.**
- **5. 2-phase-commit failure.** A server is considered dead if the write request sent to that server during 2-phase-commit failed or timed-out, unless it is because of the before-image reading problem.

The before-image read failure and repair failure are detected by one server regarding another server. The fact that a server A reports failure on accessing server B does not mean that server B must be dead, there could be another server C that can

access server B. The handling of this type of failure is simple: servers reports the failure through BEFORE-IMAGE-FAIL, REPAIR-ACK, or BEINGREPAIRED-ACK message; and client reconfigures the before-image server and reschedules repairer for a repairee during *system configuration*.

The control channel failure and heart-beat timeout detection methods are used by both client and server regarding each other. If a server detects client as dead, it simply tries to reconnect and rejoin the client. If client detects a server is dead, client closes the control channel and data channel to this server, removes it from the *active-server-list* and commit the new *active-server-list* to the disk.

The 2-phase-commit failure is detected by client. The failure handling also involves closing control/data channel and update *active-server-list*. There are some very subtle issues in the failure handling of this type, which will be discussed in detail the next subsection.

5.5.7 Subtlety of 2-phase-commit failure handling

If a write request sent to all RDWR-LOG nodes during the first phase of the 2-phase-commit were failed or timed-out, client aborts the 2-phase-commit for this request - all RDWR nodes will not receive this write request. The subtlety is that client does not know whether the *aborted write request* has been committed on any RDWR-LOG servers. The failure could have occurred either before the commit or after the commit but before the reply reached client.

To guarantee that the all storage servers have a consistent image, client injects an ”undo write request” to cancel any potential effect of the aborted request. The data in undo write request can be read from RDWR server. The undo write request bears a new global sequence number. Client reconfigures a new set of RDWR-LOG servers and send the undo write request to them.

Only after the undo write request is committed to all the new RDWR-LOG

servers, the dead RDWR-LOG servers are removed from *active-server-list* and the change is committed to client's disk. At this point, the failure handling is complete. The original *aborted write request* returns error, and client is ready to handle new write requests.

Imagine an adverse situation where client crashes during failure handling. After client restart, the on-disk *active-server-list* should contain the original dead RDWR-LOG servers and client needs to wait for all of them joining the system. Depending on whether the *aborted write request* has been committed on the old set of RDWR-LOG servers and the undo write request have been committed on the new set of RDWR-LOG servers, there can be four scenarios as illustrated in Table 5.6.

	aborted request committed on server M	aborted request not committed on any
undo request committed on server N	1. N repairs M cancels aborted request	2. N repairs all other servers, but the undo request is a NOP
undo request not committed on any	3. M repairs other servers aborted request takes effect	4. repair goes usual way, aborted request has no effect

Table 5.6: Recovery scenarios if there is an *aborted write request* and the client crashed while committing corresponding undo write request.

In all four cases all servers will eventually reach consistency. Only in case (3), the final system state will contain data from the *aborted write request*. This is acceptable because when a client crashes the upper I/O level should not have received any return code from mariner for the *aborted write request*. The upper I/O level should time-out and assume that the disk data status is unknown with regards

to the *aborted write request*. If client did not crash, when the old set of RDWR-LOG server recovers, The recovery scenario will be either case 1) or case 2).

Note that case 2 may create a hole in the global sequence number. Suppose the *aborted write request* has sequence number Seq1, the undo write request should have sequence Seq1 + 1. In the case 2), Seq1 will be missing from all server's log file. This does not impose a big problem for our repair algorithm, as long as the *recyclepoint* on the new logging server is set properly. If a server starts logging from a empty log file, usually the *recyclepoint* should be set to $firstlogrecord'ssequencenumber - 1$. However, if the first log record is a undo write record, it should be $firstlogrecord'ssequencenumber - 2$.

5.6 Map the model to Mariner

In this section, we illustrate how this model is tailored for Mariner, what kind of flexibility an implementation could chose within the framework. We first describe mariner's specific server characteristics, storage organization, system configuration policy, and repair policy. Then we discuss the server self-consistency check in detail.

Mariner has two types of server nodes: **primary** server and **trail** server. Primary node uses conventional storage device as current data storage and is usually configured as current-data server. It also has some small log storage. The log storage is only used for fault tolerance purpose and is not organized for providing historical data. The logging method on primary is simple and not optimized. Only when there is no trail node available, primary node will be configured as combined node and perform logging. For the sake of simplicity, primary just uses normal file as log storage. Primary node, while not doing logging, can act as before-image server. The primary log file is exported through NFS to repairees.

Trail node is a special logging server which supports high performance redo-undo logging. The log records are kept for relatively long time to support historical data access. Trail logging needs before-image server. Trail server does not have current data storage, it uses block device as log storage. The structure of trail log storage is more complicated and not exported. In our implementation, when a primary node needs to be repaired with trail log, trail node takes initiative.

Mariner uses simple system configuration policies and repair policies following the general requirements (Section 5.5.1, 5.5.5). Trail node with current *datainfo* is configured as RDWR-LOG node. Primary node with current *datainfo* is configured to be RDWR node unless there is no trail node in RDWR-LOG state. In that case one primary node will be configured as RDWR-LOG node. The before-image server for trail node should be a primary node in RDWR state.

Repairs are never scheduled in parallel. At most one server is being repaired at a time. Repairs are not intentionally delayed. First server that needs to be repaired is repaired by the first qualified server. We implemented only primary-trail and trail-primary repair, both are initiated by trail node. We did not implement trail-trail repair or primary-primary repair.

5.6.1 Self-consistency check

The first task of a failed server after restart is to conduct self-consistency check. The goal is to find out the *datainfo* and maintain the consistency between data and metadata. The steps are:

- 1 Check whether the server storage has been corrupted. For the sake of simplicity, we only examine the superblock signature to decide where a server's storage been corrupted. The decision could be overridden manually for more flexibility. Upon storage corruption, the datainfo is reset to -1, -1, -1.

- 2 If the storage has not been corrupted, check whether the server has been shutdown cleanly with all metadata flushed to the disk. This is checked by reading a **crash bit** in the log disk superblock. This bit is set and flushed to disk when a server is started and ready to accept write request. This bit got cleaned on disk when a server is shutdown cleanly. Unless with scheduled system shutdown, this bit should always been set on the disk.

- 3 If the storage has not been shutdown cleanly, check where are the latest log records that have not been incorporated into the on-disk metadata and update metadata with information from these log records. For this step, Primary and trail server have very different algorithm due to their individual logging scheme and metadata organization. Next we will describe them in detail.

Logging on the primary node is simple: each write request is logged in a separate record, which includes both before-image and current image. The log record is appended to a log file. The metadata on the primary node is also very simple. The only important information is *datainfo* located in the beginning of the log file. The self-consistency check is simple. The snap point cannot be accurate and does not need to be accurate(Section 5.5.5). Recyclepoint and syncpoint could be refreshed according the first and last log records in the log file.

The reason that we can simply look at the beginning and the end of the primary log for the first and last log record is because a new log file is created whenever the primary server is configured to RDWR-LOG state. The log records in the log file never got recycled individually. This strategy is applicable because in primary logging is only considered a temporary backup logging when trail node is not available. It is not used as long-term logging to support historical data access.

Trail logging is much more complex and so is its self-consistency check algorithm. Trail log records are chained backwards. The versioning metadata are

periodically flushed to disk. During the self-consistency check first we identify the last log record committed to the disk. Then with the backward chain, we can find all the log records up to the point of last metadata sync. Finally we replay these log records from the oldest to the newest to bring the metadata up to date. During log record replay, the versioning metadata is updated in a similar way as during normal write request, except that it does not need to run block allocation algorithm.

5.6.2 Identifying the last log record on trail

Identifying the last log record is a relatively complex component in trail's self-consistency check. Before stepping into its details, we first discussed another strongly related topic - the block allocation.

Free blocks are identified according the garbage collection policy (Section 5.1.2) using *allocation map* (Section 5.1.3). The block allocation follows a set of rules to limit the number of blocks and bytes that self-consistency check needs to look at.

- 1 Each log disk is used in **rounds**. Within each *round*, trail only looks forwards (from low block number to high block number) for free blocks and never look back. After it reached the end of physical log disk, the block allocation will wrap around and start looking from the beginning of the log disk again.

- 2 Upon trail finishes one *round* and starts a new **round**, trail flushes the position of last log record to the disk.

- 3 When allocating for a log record, there is a limit (FREE-STRIPS-SKIP-LIMIT) on how many big-enough free block strips can be skipped from last log record's position.

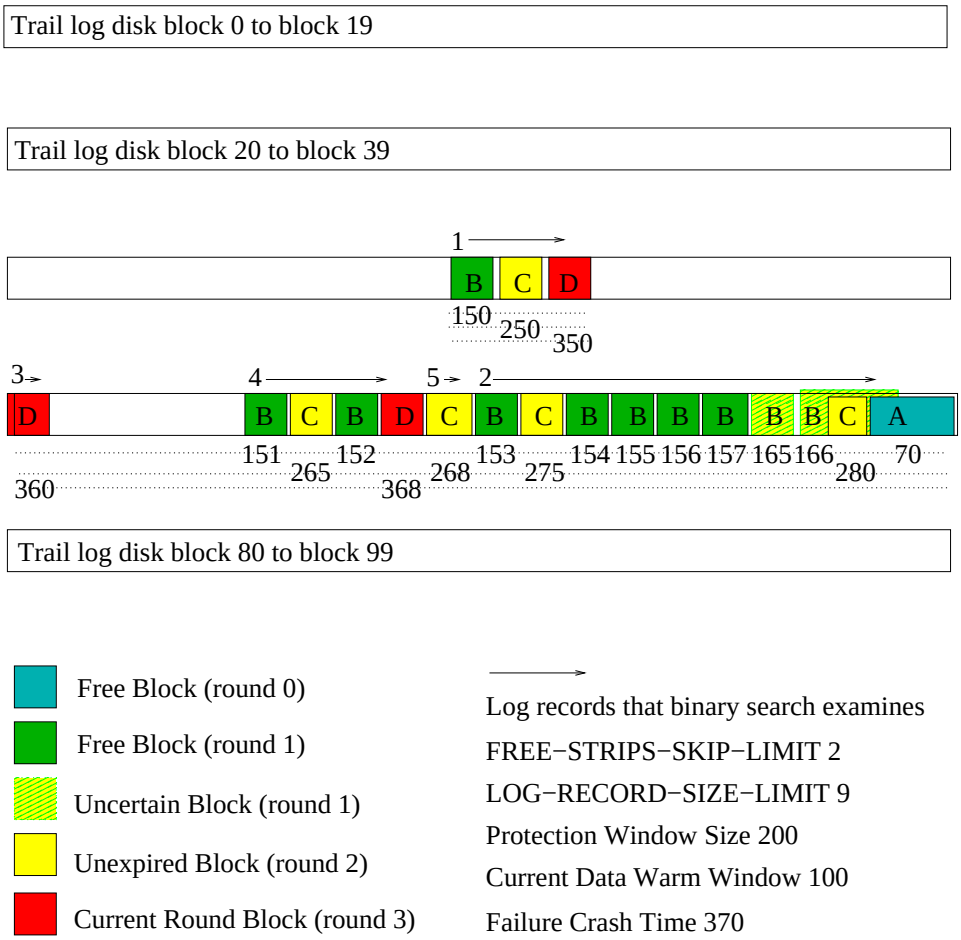


Figure 5.9: Finding the last log record on a trail log disk

- 4 There is a limit(LOG-RECORD-SIZE-LIMIT) on the log record size.
- 5 A log record is aligned with disk block boundary. Each log record starts with a header that has a unique signature, which can be guaranteed not to appear in data payload([13]). This allows us to easily identify whether a block contains a valid log record.

A pseudo binary search algorithm is used to identify the last log record on each log disk. The final result is the latest one among the results from all log disks. On each log disk, first the superblock is read to identify the information about the last *round*. The search for last log record starts from a middle position, read in blocks and look for two signs to reduce the range in the next iteration.

- sign-A A valid log record in the last *round*. This is easy to identify given the header signature, header sequence number and the last *round* information.
- sign-B A range of blocks with enough (FREE-STRIPS-SKIP-LIMIT) of large freespace strips but without any valid log records in the last *round*. By large strip, we mean that its size should be larger than LOG-RECORD-SIZE-LIMIT. According block allocation constraints 3 and 4, this indicates that we have found the end of the log disk i.e., no more newer log record beyond this point.

Upon sign-A the next iteration will move to the right half of the range. Upon sign-B the next iteration will move to the left half of the range. To confirm sign-A, we need only information from the one block. But to confirm sign-B we need to keep some states, including the number of large freespace strips that have been encountered(**num-large-free-strips**), and the size of current freespace strip (**free-strip-current-size**). We also remembers the position of last valid log record in **last-log-record-position**.

While searching for sign-B, the self-consistency check actually emulates the behavior of block allocation algorithm. While checking whether a log record has expired, the "current time" should be the time when the block allocation examines the block, which is not known to the self-consistency check algorithm. But we could have an estimation. The time should fall between "last known valid log record's timestamp" and "server fail time". Compared to a log record's life span, the difference between the lower and upper bounds should be relatively small.

Given an iteration range [R1, R2], we start examining blocks from the middle (R1 + R2)/2 until we identified sign-A or sign-B or we reached R2. If we reached R2, it is equivalent as we found sign-B. For the ease of discussion, we assume a block that is being examined has block number K; and if it contains a log record header, the log record length is L. There could be several possibilities:

- Block K contains a header of last *round*, i.e., sign-A is confirmed. This iteration ends and next iteration starts for [K + L, E]. The internal states *num-large-free-strips* and *free-strip-current-size* is reset to 0. The *free-strip-start* is reset to K + L + 1. K is assigned to the *last-log-record-position*.

- Block K contains a header from previous *round* and it has not expired. We skip this log record and continue current iteration at block K + L. Since we found some non-free blocks, the *free-strip-current-size* is reset to 0.

- Block K contains a header from previous *round* and we are not sure whether it has expired. The self-consistency enters an special *uncertain* state. To be conservative, the L blocks will not be counted into freespace strips. The *free-strip-current-size* is reset to 0. Since these blocks could also be expired, we cannot skip the next L-1 blocks. Current iteration continues at block $K + 1$. If we do encounter another valid header within next L-1 blocks, the *uncertain* state stops immediately. Otherwise the *uncertain* state automatically stops

after L-1 blocks.

- BLOCK K contains no valid header. If we are not in the *uncertain* state, it is considered a free block. We increment *free-strip-current-size*. If it reaches LOG-RECORD-SIZE-LIMIT, we increment the *num-large-free-strips* and reset the *free-strip-current-size* to 0. Current iteration continues at block $K + 1$.

- BLOCK K contains an expired header. If we are in the *uncertain* state, the *uncertain* state stops. We increment the *free-strip-current-size*. If it reached LOG-RECORD-SIZE-LIMIT, we increment *num-large-free-strips* and reset the free-strip-current-size to 0. Current iteration continues at block $K + 1$.

The final output of the binary search is the *last log record position*. Figure 5.9 shows an example of the pseudo binary search. There are five rounds.

5.7 Implementation of the fault-tolerance model

Figure 5.10 shows the software architecture of Mariner's fault-tolerance implementation, it shows each software components and its interactions. The **client daemon** and the **server daemon** are the user level daemons that implement the generic 1-client-N-server recovery model. The *server daemon* on the trail nodes and the *server daemon* on the primary nodes are the same. Control messages(Section 5.5.3 are passed between these daemons through TCP.

Each server exports a virtual block device to client through iSCSI protocol. The modified 2-phase commit protocol is implemented based on UNH-ISCSI [51]. The virtual device on primary node may also be exported to trail through iSCSI for before-image read and repair. The virtual block device is implemented by

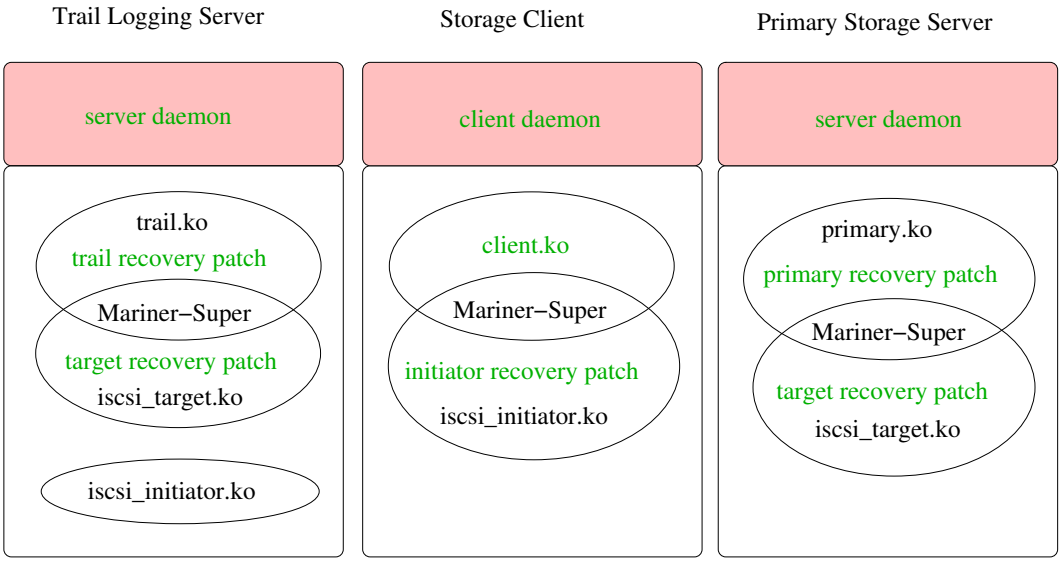


Figure 5.10: Fault tolerance implementation

trail kernel module on trail, and by *primary kernel module* on primary, for different read/write/logging schemes. The **trail recovery patch** and the **primary recovery patch** handles self-consistency check, repair, communication with the user level daemons, and communication with the modified 2-phase-commit protocol(Section 5.5.1). These server kernel modules and patches are mostly mariner specific. The **client kernel module** handles communication with *client daemon* and with the 2-phase-commit protocol. The **initiator recovery patch** and the **target recovery patch** are added to the UNH-ISCSI implementation to implement the the modified two-phase commit protocol. All server nodes run the same iSCSI target and the recovery patch. The additional vanilla iSCSI-initiator on trail is needed for before-image read and repair.

Note that client can see many iSCSI devices, one for each iSCSI target. Ideally the application should see only one device offered by mariner storage system and the *client kernel module* could have implemented another layer of virtual device

to distribute the read/write to the corresponding iSCSI devices. In practice for the sake of transparent reliable multicast (as explained in Section 5.3), the request dispatching and duplication is implemented inside iSCSI initiator. We did not provide the virtual block device on client. device on client. The read or write to any of the iSCSI devices are treated in the same way by the *initiator kernel patch*.

Sever daemons communicates with server kernel patches through IOCTLs on the virtual block devices. To make the communication interface uniform, **client kernel module** also implements a virtual device (/dev/mariner-client) just for the sake of of issuing ioctl. The set of IOCTLs are the same for all server nodes but different for client. The mariner kernel module and iSCSI module interacts through a data structure called **mariner super**. It is owned by mariner kernel module but exported to iSCSI module through EXPORT_SYMBOL. The *mariner super* contains all fault tolerance related information. It has a copy in both user and kernel space. Some of the IOCTLs are dedicated to keep the two copies consistent.

5.8 Evaluation of mariner's fault-tolerance implementation

In this section, we describe the evaluation of the fault tolerance implementation by testing failure of each node in our system. Each failure could be due to various reasons. Table 5.7 shows the possible combinations of failure scenarios. It is tedious to enumerate all cases. We select five representative tests as marked on the table.

In experiment 1) to 4), we start the mariner system and a **testing flow**. Then we induce failure manually, and show the recovery sequence until the whole system is back to normal. The *testing flow* consists of continuous write request (4K size) with 10ms sleeping time after each request. Experiment 5) uses file system commands

	trail	primary1	primary2	client
network failure	3			
reboot or death of control daemons			1	4
power off or system crash with unsynced metadata	5			
disk failure with data loss		2		

Table 5.7: This table shows the possible combinations of failure scenarios and the test cases that are used in the evaluation section

(mkfs, cp) to verify trail self-consistency check algorithm(Section 5.6.1).

This evaluation plan included only single failure scenarios. The fault tolerance model has been designed with single failure in mind. Although the model tries to be generic and in fact can handle many multiple failure cases, it probably needs some work to handle arbitrary multiple failure scenarios. The mariner implementation and the testbed setup also pose restrictions to exercising multiple failure cases. For example, we did not implement trail-trail repair and primary-primary repair; the testbed is configured with only one trail node.

The focus of the evaluation is not about read/write throughput and latency. Consequently the hardware/software setup is not tuned for performance. On trail node, the two log disks are actually two physical disk partitions. The *trail kernel module* used an old prototype. In the recently two years, other mariner team members have made significant improvement to trail logging performance ([42]).

5.8.1 System setup and failure induction method

Our evaluation testbed consists of one client node, one trail node and two identical primary nodes: primary1 and primary2. All of the four nodes are Dell PowerEdge

SC1450 machines with 2.8GHz CPU and 1GB memory. The hard disks used in the testbed are 250GB Maxtor 7L250S0 SATA disks. Each server node has two disks, one as the system disk and another as data disk. Trail is configured with two log disks with two partitions on the data disk. Each log disk is of size 200MB. The primary node's current storage size is also configured to be 200MB. Therefore the mariner storage system provides 200MB of current data storage and 200 400 MB of historical data storage.

All machines run Fedora 3 with Linux kernel 2.6.11. We use UNH iSCSI implementation (version 1.6.0) [51] as the base for iSCSI initiator and iSCSI target. Each node is equipped with two network cards. The eth0 (130.245.134.xxx) is used for external network communication. The eth3 (169.254.0.xxx) is used for internal network communication among the four mariner nodes. Both mariner control messages and iSCSI communication uses the internal network interface eth3.

The network failure and recovery is induced with *ifconfig eth3 down* and *ifconfig eth3 up*. All the nodes are powered by a remote controlled power switch. Each node's power can be turned on and off through a web interface. Complete disk data loss is emulated by writing 0 to the current data storage and the log storage.

5.8.2 Normal case

As a baseline for comparison we first shows the normal situation without failure. Figure 5.11 shows the throughput of the *testing flow*, which is about 150KBytes/Second. This is not the highest throughput we can get from the testbed because there is 10ms sleep time between each 4KBytes write request. In each second there are about 36 write requests and 360ms of sleeping time.

The reason that we keep this 10ms gap between requests is because of some issues in the implementation. The modified 2-phase commit protocol is initially implemented without full consideration of fault tolerance. Moreover, it is mixed

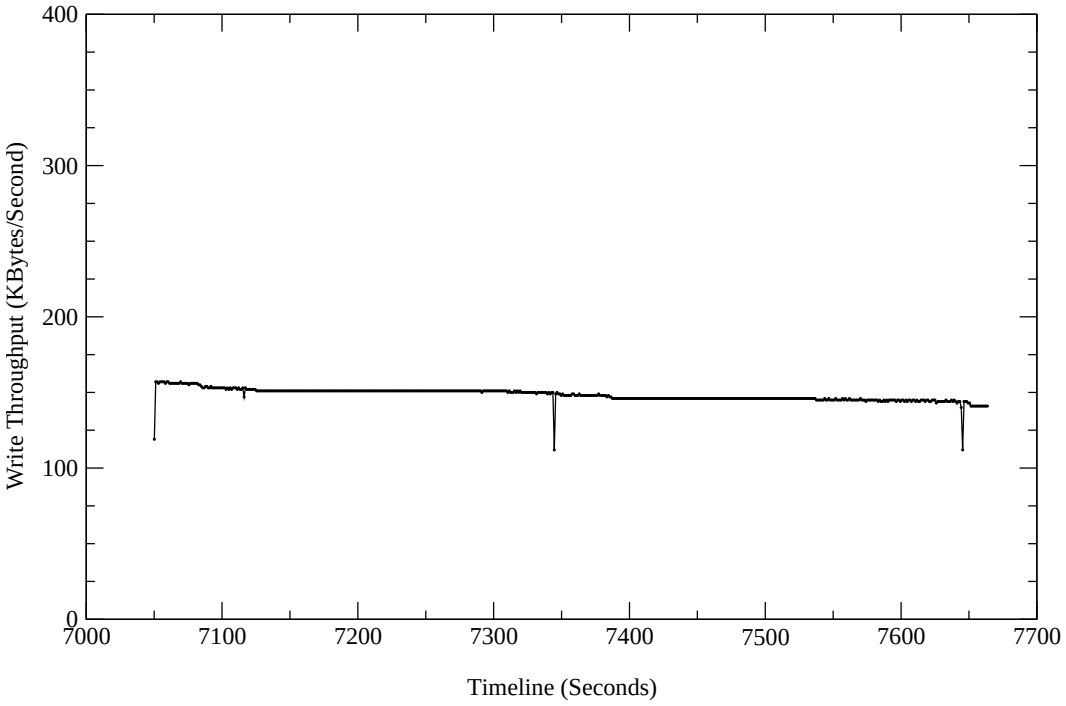


Figure 5.11: Throughput of *testing flow* in the absence of failure

with other modifications that are added to support TRM(Section 5.3). The error handling of the the 2-phase-commit protocol, of TRM, and of iSCSI are not correctly integrated. It is a well known problem that the interaction of error handling from different storage system components is often incorrect [31]. Unfortunately mariner seems suffers the problem too. For the system to function properly, mariner needs to detect a failure and close the corresponding iSCSI connection before the iSCSI's original error handling get triggered. Without fully understanding all the intricacies, one way to get around the issue is to reduce the testing load by adding the 10ms sleeping time.

Before the first write request is issued, the *client daemon* admits all server nodes. Trail nodes is assigned as RDWR-LOG node. Primary1 and primary2 is assigned as RDWR nodes. Primary1 is also assigned as trail's before-image server. The *client daemon* informs the *client kernel module* regarding each server's state through IOCTLs. Upon each write request, the *initiator recovery patch* consults the *client kernel module* regarding each server's state and use this information in the modified 2-phase commit protocol.

5.8.3 Primary2 fail recovery

Primary2 has the simplest role in the mariner system and its failure has least impact to the system. Figure 5.12 shows throughput change of the *testing flow* as well as the time line of failure detection, handling and recovery.

From left to right, the first vertical line represents failure detection and handling. The heart-beat interval is 5 seconds. Client detects primary2's failure few seconds after we turn off the power. The *client daemon* calls an IOCTL to inform the *client kernel module* that primary2 is down. Client also tears down the iSCSI connection to primary2, which takes about 1 to 2 seconds.

The second vertical line represents the beginning of recovery. It starts from

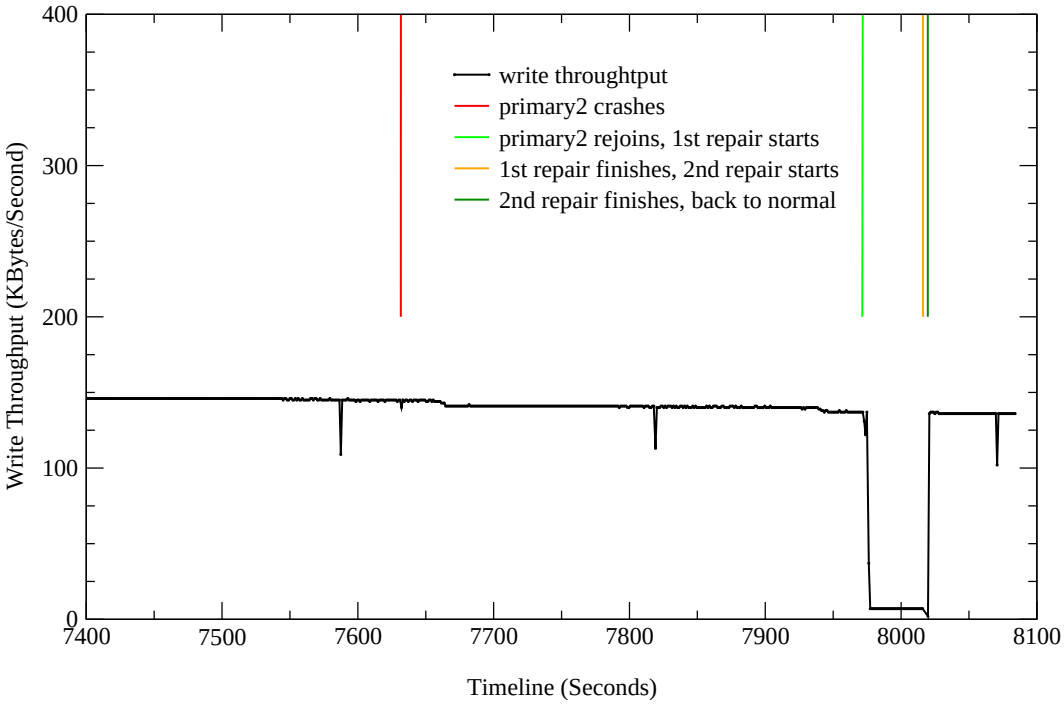


Figure 5.12: Primary2's recovery after crash failure

client receiving NODE-INSERT message until first repair is scheduled. On primary2, after it reboots, the *primary kernel module* is installed and does self-consistency check. The *server daemon* sends NODE-INSERT to client and client establishes a new iSCSI connection. Then client checks that the primary2's data is out of date. A REPAIR message is sent to trail node to repair primary2. After trail receives the REPAIR message, it establishes an iSCSI connection to the repairee (primary2). With the iSCSI management commands *iscsi_config up* and *iscsi_config down*, it takes 2 to 4 seconds to establish and tear down the iSCSI connection. This sets a lower time limit on a repair operation. In the first repair operation, trail node took 3.4 seconds to establish the iSCSI connection. In the next 41 seconds, the repair operation found 11814 log records to be replayed and updated primary2's current data storage through the iSCSI connection. The repair throughput is 1091KBytes/Second. In the mean time, the write throughput of the *testing flow* dropped drastically to 7KBytes due to resource competition

The third vertical line represents the end of first repair and beginning of second repair. When client receives trail's REPAIR-FINISH message, it shows primary2's datainfo is still not up to date because of there are new write requests being served during the repair. However, the difference is small, only 79 write requests were missing, less than a pre-defined threshold (100). Therefore client enters SUSPEND-WRITE state. It schedules the second repair while suspending new write requests. Hopefully the second repair does not take long and the system can go back normal as soon as possible. Originally we had considered to let trail take more control over multiple repairs. In the end, it turns out that client is in a much better position to coordinate.

The *testing flow*'s throughput dropped close to 0 during second repair because the write requests were suspended. It didn't completely go to 0 because of the way throughput is computed. The application tries to computes it once a second, but

when write request is suspended the application does not get to run. The second repair took 3.6 seconds with 79 log records being replayed. The repair throughput is much lower than the first repair because most time were spent on setting up and tearing down the iSCSI connection. A small optimization could be to not tearing down in the first repair and then setting it up again in the second repair.

The fourth vertical line represents the end of second repair. The client goes back to NORMAL state. The primary2 is assigned RDWR node. The state updates are send to kernel via IOCTLs and the system is back to normal. The *testing flow*'s throughput goes back to 136KBytes/Second, still lower than the initial throughput of 156KBytes/Second. This is similar to the normal case as we have discussed in Section 5.8.2.

5.8.4 Primary1 fail recovery with complete data loss

The primary1's fail scenario (Figure 5.13 shares many common aspect as primary2's fail scenario. In this section, we will focus on the differences.

First differences is that this failure involves complete data loss. This is recognized by the self-consistency check because of missing superblock signature. As a result, the datainfo is reset to -1. During the repair, much more log records need to be replayed, both from the data loss and from the down time. First repair took 109.7 seconds and replayed 30526 log records. The repair throughput is 1139KBytes/Second, slightly higher than the first repair operation in primary2's recovery, because the cost of establishing iSCSI connection is amortized over more log records.

Second difference is that there are three repair operations instead of two. This is because that the first repair operation took longer and missed more new write requests. When client receives the REPAIR-FINISH message for the first repair, the repairee's datainfo indicates 315 log records missing, which is not close enough

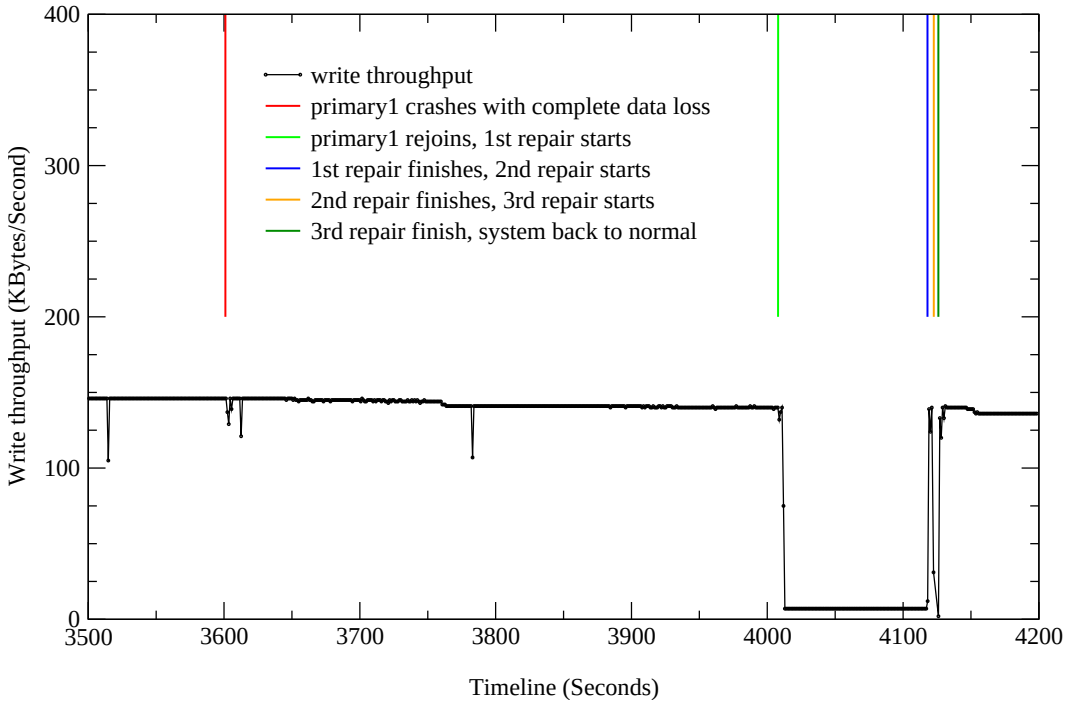


Figure 5.13: Primary recovery after power failure with complete data loss

for client to enter SUSPEND-WRITE state (need to be less than 100). When client receives the REPAIR-FINISH from the second repair, only 3 log records are missing, therefore client entered SUSPEND-WRITE state and scheduled the last (third) repair operation. We can imagine that if the initial number of missing log record is very big, it could take more than 3 rounds of repair. But since the repair throughput (>1000 KBytes/Second) is much higher than the new write throughput (<10 KBytes/Second), it is fairly easy for the repairee to catch up and the number of repairs cannot be very big.

Another subtle difference is that compared to Figure 5.12, the testing flow shows bigger write throughput drop around the first and last vertical line. This is because that there are before-image server re-assignment in the case of primary failure. After client detected the failure of primary1, primary2 is assigned as trail's before-image server. When primary1 is back to normal, primary1 is re-assigned as before-image server. Trail reads before-image through iSCSI. For each new before-image server assignment, trail tears down the iSCSI connection to the old server and set up the iSCSI connection to the new server. This could take seconds as described in Section 5.8.2 and causes disturbance to the write throughput. Not all write requests requires reading before-image. If some request does require before-image read during the before-image server transition, trail returns failure and let upper level retry mechanism steps in.

5.8.5 Trail fail recovery

Figure 5.14 shows the trail fail scenario. Although the basic vertical lines shows similar steps of failure detection, recovery and repair, the duration of each step aren't quite consistent with previous two figures(Figure 5.13, 5.12). In addition, the write throughput of the *testing flow* looks very different.

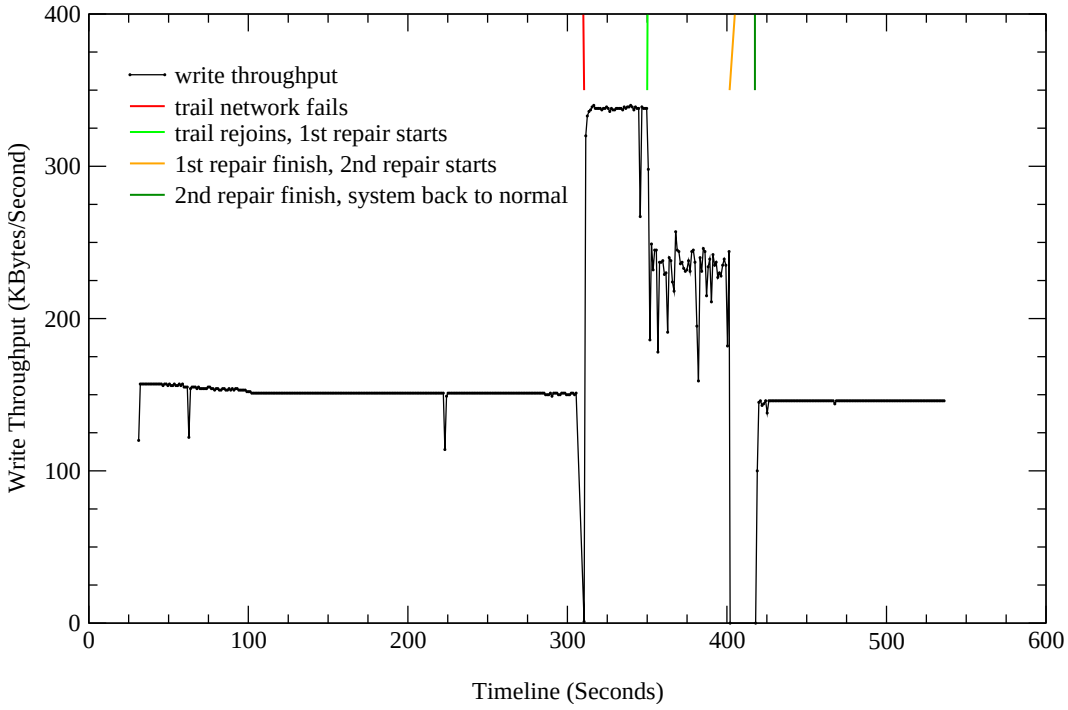


Figure 5.14: Trail recovery after network failure

5.8.5.1 Write throughput drop around failure detection

First difference is that the write throughput dropped to 1KBytes/Second around first vertical line, which is much more prominent compared to previous figures. The reason is that RDWR-LOG server is in a critical position in the 2-phase commit protocol. The system cannot handle any write request from the time that a RDWR-LOG server failed until the failure is detected and a new RDWR-LOG server is assigned. Any other server failure (RDWR server, before-image server) may not block the progress completely.

5.8.5.2 Write throughput increase during trail down time

Second difference is about the write throughput increased to 340KBytes during the trail down time, where primary1 is assigned as RDWR-LOG server. In this experiment the RDWR-LOG server's performance determines the system performance. This shows that that primary node has better logging performance than trail, which is contrary to our claim about trail's super logging performance(Section 5.2).

The contradiction is caused by implementation issues both on the trail side and on the primary side. On the trail side, as mentioned in the beginning of Section 5.8, we used an early trail implementation prototype which includes the complicated versioning algorithm but not any performance tune up as described in [42]. Especially it did not incorporate the track-based logging techniques. On the primary side, the primary uses log files rather than log disks. That means each log write may not be committed to the disk when it returns. Therefore it is not fair to compare trail's synchronous logging performance with primary's asynchronous logging performance. Initially we used log file instead of log disk on primary node for the ease of implementation. We hoped that by opening the log file with O_DIRECT flag, each file write will become synchronous write. Later we realized that the

O_DIRECT flag is not supported by the Linux-2.6.11. Hence the primary logging is still not synchronous logging. However this does not pose big threat to mariner's correctness under single failure situation. Presumably, the primary node only does logging when the trail node is failed. Therefore the primary node should will not fail again while doing logging.

5.8.5.3 High write throughput and low repair throughput during first repair

During the first being-repaired operation, the write throughput is 200 250KBytes, much better than the 7KBytes in Figure 5.13. In the mean time, trail was disconnected from the network for only 40 seconds, missed only 3260 log records, yet the first being-repaired operation took 49 seconds. The repair throughput is 260KBytes, much worse than primary's failure case. This difference is mainly caused by the same reason that trail logging performance is not as good as primary logging performance. The fact that both repair and being-repaired operation are conducted by trail further increases trail's load.

During being-repaired operation, various trail versioning data structures needs to be updated, the repair throughput should be comparable to the trail logging throughput. The repair throughput is actually a little higher because there is no 10ms gap between each log records and there is no need to read before-image. Trail node reads log records sequentially from the primary's log file mounted through NFS. Since the trail's being-repaired throughput is limited by trail's own logging performance and the sequential read is very efficient with file system pre-fetch, the primary node still has enough bandwidth left to serve new write requests.

During repair operation(Figure 5.13, 5.12), the repair can be done faster because there are no versioning metadata updates. The repair involves only reading log records from the log disk and updating primary node's current data storage. With current implementation of trail logging and repair, the repair naturally gets

higher priority over serving new write requests. Therefore the repair throughput is relatively high and the write throughput is very low.

5.8.5.4 Write throughput drops to 0 during second repair

While the high write throughput may be desirable for application, it also means long repair time. For mariner system, each write request served during the node failure and repair time costs more system resources than that served during normal status. We needs a good balance between system throughput during repair and the overall system throughput. Ideally there should be a mechanism to explicitly control the resources used for repair and for serving new requests. Right now we only use a very simple method. Whenever client discovers that the repair throughput is not significantly ($>$200%) higher than than the write throughput, the client will suspend new write request to let the repair finish as soon as possible.

This explains why the write throughput dropped to 0 during second repair. In fact, due to some implementation limitations, in this experiment we actually kill the *testing flow* process and restart the iSCSI initiator. The client re-admit all server nodes and then schedule the second repair. The whole procedure takes about 3 seconds. Therefore the third vertical line is not quite vertical.

5.8.6 Client fail recovery

Figure 5.15 shows the client failure scenario. The *testing flow* stopped when we turns off the power and manually restarted when client node is back to normal. This clearly matches the the write throughput line.

A less obvious observation is that the client recovery involves two repairs. The first repair is on primary1 and the second on primary2. If client crashed in the middle of the modified 2-phase commit protocol, it is possible that trail has committed

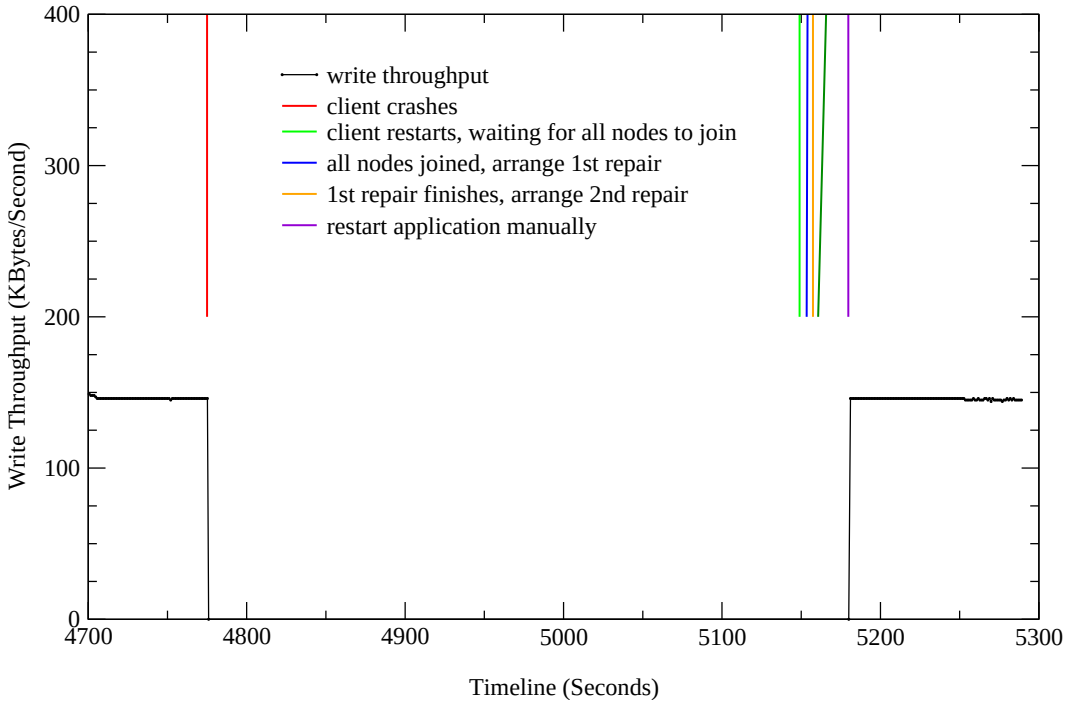


Figure 5.15: System recovery after client crashed

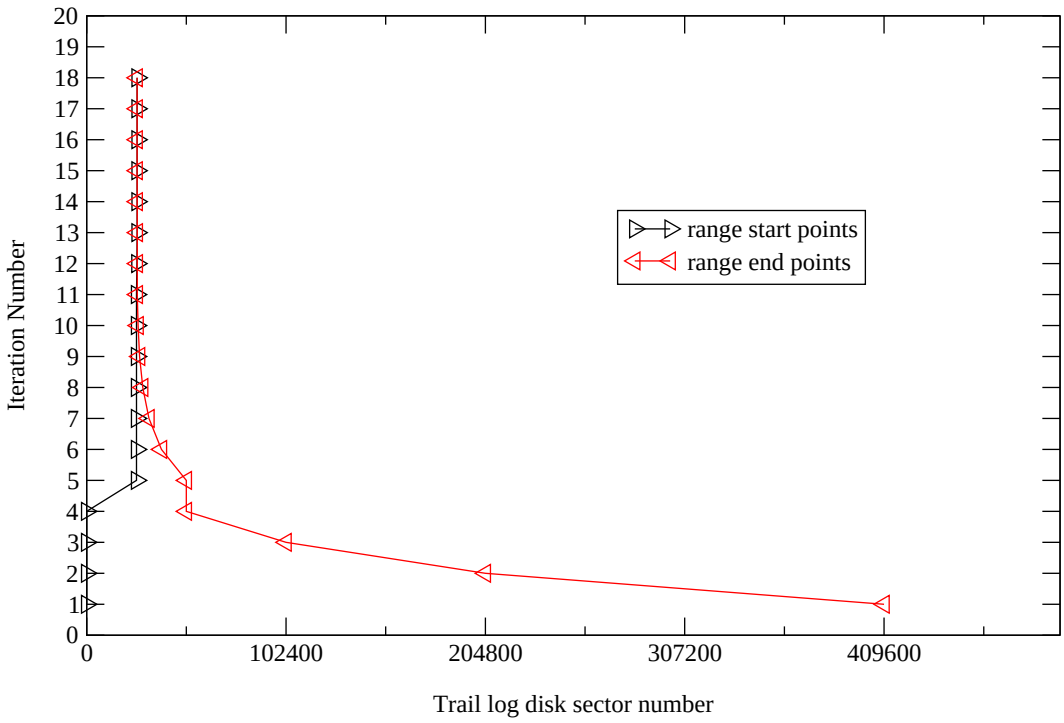


Figure 5.16: Binary search to find last log record on trail

the last request, but the acknowledgment has not reached client; or that the acknowledgment has reached client but has not been forwarded to the primary nodes. Therefore the pre-write request on primary1 and primary2 could not get committed. When primary1 and primary2 rejoins client, client schedules two repairs to replay the last write request on both primary nodes.

5.8.7 Trail self-consistency check

Previous evaluations focused on the interaction between different nodes. Note that previous trail failure is network failure. The trail node didn't crash and the self-consistency check(Section 5.6.1 is not exercised. In the section we examine the correctness and performance of trail's self-consistency check with file system level

commands. We skip the self-consistency check of primary node and client node because they are very simple. Trail's periodic metadata flush is turned off for the ease of experiment.

We create an ext3 file system on the 200MB mariner device and mount the file system on /mnt/mariner. Then we copy a testing directory from /root/test into /mnt/mariner/test. The testing directory has 2 sub-directories and 52 files, with a total size of 360KBytes. In total, it took 45 seconds. We then power-cycle the trail node. After trail node reboots, the *trail kernel module* is installed and the self-consistency check is done.

Figure 5.16 shows the binary search to find the last log record on log disk 0. The search on log disk 1 is similar. If a valid log record is found, the range start point increases. If end of log disk is found, the range end point decreases. The start point and end point merge after 18 iterations. 244 sectors were read on log disk 0 and 233 sectors read on log disk 1. The total time spent to find last log record is about one second. On average the time should be logarithmic to the size of log disk. In the worse case it could be linear if the log disks have too much unexpired log records.

The second step is to find all log records which has not been incorporated in the versioning metadata on disk. Starting from the last log record and following the backward chain, the second steps read 6401 sectors, found and stored the position of the 6401 log records to be replayed. The second step took 9 seconds.

The third and last step is to replay the log records to update the versioning metadata and flush them to the disk, which takes about 0.6 second. After the self consistency check and other initializations of the *trail kernel module*, we mount the mariner device again and use "diff /mnt/mariner/test /root/test" to verify the correctness of the self-consistency check.

5.8.8 Summary

We evaluated the fault-tolerance model and mariner's implementation with comprehensive single failure scenarios. The failed node gets repaired and the system always returns to a consistent state. The system remains available except for a short period of 10 seconds upon trail node's recovery. These experiments illustrated the complexity and the effectiveness of the fault-tolerance model. In the mean time, they also exposed the implementation limitations and several places where performance and system availability could be improved.

5.9 Lessons

5.9.1 Lessons on Mariner's overall design

The most important experience that we learned from Mariner project is that repairability can be seamlessly integrated with many desired features of a storage system. If designed properly, it does not necessarily come with much extra cost in performance or complexity.

At the core of Mariner's design is a fast and self-describing trail logging method. To fulfill the promise of a a repairable system, the logging method is extended with a more sophisticated block allocation algorithm to ensure certain *protection window*. We also introduced another concept of *current data warm window* to separate hot data and cold data to improve logging performance. The block level versioning metadata is build on top of the log record to provide access of any point-in-time image. A user level versioning file system is build on top of the versioning storage to provide the historical versions at file system level. It can be used either directly by user or by the repair program. Mariner shares same client syscall logging and dependency analysis scheme with RFS. There is a small difference in server side

file update logging. With mariner, the file update logging have neither real data (like in RFS-O) or data pointer (like in RFS-A/RFS-I/RFS-I+). With actual data being maintained by the versioning storage, the file update log is only used for dependency analysis and metadata (directory/attribute) repair.

In addition to repairability, the core logging technique also provides good support for the recoverability 1. The system can tolerate arbitrary hard failures 1 with high level of availability. Because trail logging is very fast, we can afford to put it in the critical path of write request handling. Therefore we do not need another simple logging just for the sake of fault tolerance, which is a common practice. One example is the journaling file system. Another example is Veritas's comprehensive versioning product RTP [66] which uses a dirty block logging to record recent modifications. Because trail logging is self-describing, it is the only data structure that needs to be written to disk synchronously. All other metadata structures can be flushed to disk in a delayed manner to reduce the impact to system performance. When hard failures occur, trail's self consistency check could restore metadata from the information in the log record.

5.9.2 Lessons on Mariner's fault tolerance design

Other than conveniently taking advantage of the trail logging techniques, Mariner's fault tolerance design focuses on simplicity, uniformity, flexibility, and extensibility.

5.9.2.1 Simplicity

First simplification is that the basic model is built for a 1 client N server system instead of the M client N server system as Mariner actually is. This is because we assume no data sharing among multiple clients. Even though they might share data servers, separate partitions are used by each client. This greatly simplified the fault

tolerance design.

Second simplification is that we decided not to use a separate manager node to take care of failure detection, node insertion, node deletion, and repair etc. Instead we let the storage client taking care of the management duty. This approach removed the issue of manager node failure and the need of manager-client communication. It also brings convenience for the fault tolerance design to be integrated with the 2-phase commit protocol. Client can conveniently suspend data flows while performing certain management tasks. There are probably some minor disadvantages such as when client node is down, server node cannot be repaired. But they are rarely needed and can be fixed by the collaboration of multiple clients.

5.9.2.2 Uniformity

Same failure handling and node repair schemes are used for all types of failures(network failure, power failure, node crash, disk failure with and without data loss). The failure detection methods for different types of failures are not completely the same but still share many common components. Therefore the fault tolerance model is extensible to new failure types. In fact, it has now covered all hard failure types.

Another uniformity shows in the control channel communication pattern. All the control messages are passed between client and server. There is no server-server communication. While this may occasionally increases the number of control messages passed around, it is consistent with our principle of keeping client as the central controlling point; and it greatly simplifies the control channel connection establishment and the failure detection/handling.

5.9.2.3 Flexibility

Mariner's recovery model has two layers - a generic model and some system-specific components. The generic model works for a 1-client-N-server storage system which have some server nodes that support self-describing logging. The system-specific components decide the exact node configuration policies, repair policies, logging schemes and self-consistency check algorithms. There is a clear interface between the two layers.

The system-specific components are either individual modules for a certain functionality or pure policies geared towards particular storage system characteristics. For example, the trail node and the primary node have very different logging and self consistency check schemes. But they share the same control daemon (which belongs to the generic model). The only difference that client cares about regarding trail node and primary node is that trail node is the preferred logging node and repair node.

These system-specific components do not change node state and do not involve control message communications. Therefore they do not have global impact to the correctness of the fault tolerance model. It is easy to change the system-specific components according different storage system characteristics or different workloads.

5.9.2.4 Extensibility

Other than the flexibility provided by the system-specific components, we also try to make the generic model itself extensible. In the original design, high availability wasn't a goal. Similar to the RFS, the system stops responding to new write request during repair time. Later we extended the model so that the system is available for write even at repair time. This extension is made possible because in general we try

to have modular design and make less assumptions. This is especially true for the repair operation:

In the repair operation, a repairer replays some log records (or copies data blocks) to bring repairee's data status more up to date. The repairer itself does not need to have most current data - it only needs to be more up to date than the repairee. The repair operation does not need to finish successfully as originally planned. If it fails in the middle, the progress is recorded and the repair can be resumed with same or different repairer at a later time. The basic repair unit (replaying one log record/copying one data block) is idempotent. The data status of neither repairee nor repairer needs to be accurate - it just needs to be conservative.

Without any modification, the original model will just keep scheduling repair operations to handle the new write requests. It can only stop if there is some quiet time without write requests coming. The system performance will surely suffer. To make the system works better, we only need two adjustments. One is for client to suspend write request when repair is close to finish. Another is for client to limit the write request bandwidth when repair speed is slower than new write request. There is no modification needed on the repair operation itself.

Another possible extension of the fault tolerance model is to handle multiple failures. Current model already can handle many multiple failure scenarios and has only few places assuming single failure. We still need careful examination to ensure the extended model works for arbitrary multiple failure scenarios. But we do believe that it is doable with reasonable adjustments.

5.9.3 Implementation Complexity and Limitations

Our fault tolerance model itself is simple and efficient. It has been analyzed and evaluated with comprehensive single failure scenarios. The failed node all gets repaired and the system always returns to a consistent state. However we encountered

extra implementation complexities and limitations when the fault tolerance model needs to be integrated with existing software layers such iSCSI and TRM(the transparent reliable multicast protocol).

One example of the limitation is that at operating system command interface it takes up to two seconds to setup or tear down an iSCSI connection. This definitely reduces the high-availability promise that the fault tolerance model could deliver.

One example of the complexity is how to get error handling right. It is a well known fact that in a storage system it difficult to get error handling right. What is even harder is to make the error handling of different layers of software working correctly and consistently for one goal (fault tolerance of the whole system). Because that we don't have enough time to fully investigate and integrate the error handling of iSCSI and TRM, the current implementation has several limitations as discussed in Section 5.8.2 and Section 5.8.5.4.

Chapter 6

TBBT: A Scalable Trace Replay for File Server Evaluation

File system traces are used to characterize and model workloads, to study new file/storage management algorithms and heuristics, and to identify interesting access patterns suitable for performance optimization. Surprisingly, however, they are rarely used to evaluate the performance of actual file servers. The reason is that a tool that can accurately replay file access traces against a live file server is relatively challenging to build. This chapter describes the design, implementation, and our evaluation of the Trace-Based file system Benchmarking Tool (TBBT), the first comprehensive NFS trace replay tool [74].

TBBT automatically detects and repairs missing operations in an NFS trace, derives a file system image required to successfully replay the trace, ages the file system image appropriately, and initializes the file server under test with that image. TBBT then drives the file server with a workload that is derived from replaying the trace according to user-specified parameters. TBBT can scale a trace temporally or

spatially to meet the needs of simulations and it can do so without violating dependencies among file system operations. Empirical experiments using a large NFS trace show that TBBT can produce qualitatively different throughput and latency results than SPECsfs, a widely used industrial-strength file system benchmark.

6.1 Introduction

Modern file systems are typically optimized to take advantage of the workload characteristics they are designed to serve. Accordingly, the performance of a file system must be evaluated with respect to its target workload. The ideal benchmarking workload should represent the way that actual applications use the file system. It should also be effective in predicting system performance in the target environment, scalable so as to simulate the system under different loads, easy to generate, and reproducible.

At present, the most common workloads for file system evaluation are synthetic benchmarks. These benchmarks are designed to re-create the characteristics of particular environments. Many synthetic benchmarks are parameterized, making it possible to tailor the resulting workload to specific requirements. In recent years synthetic benchmarks have improved significantly in terms of realism and the degree to which they can be tailored to a specific application. Yet synthetic benchmarks cannot always mimic file access traces collected from real-world environments. Many time-varying and site-specific factors are difficult, if not impossible, for a synthetic benchmark to capture. For example, recent file access trace analysis showed that modern file servers handle a variety of workloads with widely divergent characteristics [17, 54, 69]. An additional barrier is the time required to develop a high-quality benchmark, which is often months or years. As a result, synthetic benchmarks may not keep pace with changes in the workloads of their target

environments.

In contrast to synthetic benchmarks, replay traces - those extracted from the system being evaluated – are by definition representative of that system's workload. Given that disk, network, and web access traces have been used extensively to evaluate storage systems, network protocols, and web servers respectively, we see no reason why design engineers cannot usefully employ file access traces to evaluate file systems. Indeed, we believe that trace replay may constitute a better basis for file system workload generation and thereby performance evaluation.

Replaying an NFS trace against a live file system/server is non-trivial:

- Because a file system is stateful, a trace replay tool must be context sensitive. The context for each request in the trace must be properly set up in the file system under test prior to the trace replay. For example, a file open request can be successfully replayed only if the associated file already exists.

- The disk layout of a file system significantly impacts the performance of the file system. For example, how to properly "age" a file system [60] to accurately reflect the performance degradation of real-world file systems due to disk space fragmentation remains problematic.

- Because a trace could be collected on a file system whose performance diverges widely from that of the target file system, a trace replay tool must be capable of scaling the dispatch rate of trace requests to meet specific benchmarking requirements. This scaling up or down must occur without violating any inter-request dependencies.

In this paper we present the design, implementation, and evaluation of a novel NFS trace player, TBBT, and how it addresses each of these three issues. TBBT can infer the directory hierarchy of the file system underlying the trace, construct a

file system image with the same hierarchy, replay the trace at a user-specified rate, and gather performance measurements. Because traces do not carry physical layout information, it is impossible for TBBT to incorporate the actual aging effects in the construction of the initial file system image. However, TBBT does support an artificial method that allows users to incorporate a degree of file aging into the initial file system image used in their simulation. TBBT also allows its users to scale up the trace to simulate additional clients and/or higher-speed clients without violating the dependencies among file access requests in the trace. Finally, TBBT has a robust error-tracing capability: it can automatically detect and repair inconsistencies from incomplete traces.

Although designed to overcome the limitations of synthetic benchmarks, TBBT has its own limitations (Section 6.6). In fact, TBBT is not designed to replace synthetic benchmarks, but instead to complement them.

The rest of this section is organized as follows. Section 6.2 reviews related work in synthetic benchmarks, file system trace collection, simulation, and trace replay. Section 6.3 describes the design issues of the various components of TBBT and discusses the challenges in file system trace replay. Section 6.4 discusses the implementation details of TBBT. Section 6.5 presents the results of an evaluation study of the TBBT prototype. Section 6 concludes the paper with a summary of this research and directions for future work.

6.2 Related Work

In this section, we describe related work in file system trace research, file system trace replay, synthetic benchmarks, and file system aging. We also describe a disk trace player that has a remarkable technique for issuing requests with accurate timing.

Ousterhout's file system trace analysis [52] and the Sprite trace analysis [7] motivated many research efforts in log-structured file systems, journaling, and distributed file systems. More recent trace studies have demonstrated that file system workloads vary widely depending on the applications they serve, continue to evolve historically, and consequently raise new issues for researchers to address: Roselli et al. measured a range of workloads and showed that, in contrast to findings from earlier studies, file sizes have become larger and that large files are often accessed randomly [54]. Vogels showed that workloads on personal computers differ from most previously studied workloads [69]. More recently, Mesnier et al. demonstrated a strong relationship between file names and other file attributes and the lifespan, size and access patterns of files [49].

Gibson et al. used a trace replay approach to evaluate two network storage architectures: Networked SCSI disks and NASD [27]. Two traces were used: one was a week-long NFS trace from University of California, Berkeley [15] and the other was a month-long AFS trace from Carnegie Mellon University. The traces were decomposed into many *client-minutes*, each of which represented one minute of activity from a client. Specific *client-minutes* were selected, mixed, and scaled to represent different workloads. Their paper did not reveal how they initialized the file system or how they handled dependency issues. Rather than implementing a full-fledged and accurate trace replay mechanism, their trace play tool was limited to the functionality required by their research.

There are two types of synthetic benchmarks. The first type generates a workload by using real applications. Examples include the Andrew Benchmark [33], SSH-Build [57], and SDET [26]. The advantage of such benchmarks is that they capture application dependencies between file system operations as well as the application think-time. The disadvantage is that the benchmark sample size is usually relatively small and does not represent the workload of a large, general purpose

networked file server.

The second type of synthetic benchmark directly generates a workload through the system-call interface or the network file system protocol. Examples include SPECsfs [62] and Postmark [36]. These benchmarks are easy to scale and fairly general-purpose, but have difficulty simulating a diverse and changing workload, operational dependencies at the application-level, and think-time. Recent research on file system benchmarking focuses on building flexible synthetic benchmarks to give users control over the workload patterns or building more complex models to emulate dependencies among file system operations (hBench [12], Fstress [5], FileBench [45]).

This paper compares our trace replay work with SPECsfs, a widely-used general-purpose benchmark for NFS servers [62]. Both SPECsfs and TBBT bypass the NFS client and access the server directly. However, SPECsfs attempts to re-create a typical workload based on characterization of real traces. Unfortunately, the results do not resemble any NFS workload we have observed. Furthermore, we question whether a typical workload actually exists – each NFS trace we have examined has unique characteristics.

Smith developed an artificial aging technique to create an aged file system image by running a workload designed to simulate the aging process [60]. This workload is created from file system snapshots and traces. File systems that have been aged using this technique exhibit more realistic aging effects but are not closely related to the benchmark run. The only relevant factor is that the file system is relatively full and the free space is fragmented. This technique can be used for benchmarks that have a relatively small data set and do not have a dedicated initialization phase. Usually these benchmarks are micro-benchmarks or small macro-benchmarks such as SSH-Build. This technique is not applicable for benchmarks that take full control of a logical partition and has its own initialization procedure, such as SPECsfs.

Smith's aging technique requires writing 80 GB of data (which requires several hours of run time) to age a 1 GB file system for the equivalent of seven months. This makes this method impractical for large file systems. TBBT's aging technique is less realistic, but runs two orders of magnitude more quickly.

Buttress developed a disk I/O generation tool specifically designed to issue requests with accurate timing [6]. In benchmarking disk I/O systems, it is important to generate I/O accesses that meet exactly the timing requirements. However, timing accuracy (issuing I/Os at the desired time) at high I/O rates are difficult to achieve on stock operating systems. TBBT suffers the same problem when a *timestamp-based* timing policy (as described in Section 6.3.4) is used to generate file system requests. Buttress generated I/O workloads with microsecond accuracy at I/O throughputs comparable to those of high-end enterprise storage arrays. Buttress's timing control technique is flexible, portable, and provides a simple interface for load generation. TBBT could incorporate Buttress's technique to improve the timing accuracy of its request dispatching procedure. Like other disk-level benchmarks (such as IObench [71], Bonnie [9], and lmbench [47]), Buttress does not need to handle the complications arising from dependencies among file system operations.

6.3 Design Issues

TBBT translates an NFS trace to a standard format, corrects omissions in the trace, and calculates the initial file system image required for successful replay. It then creates the initial file system image according to user-configurable aging parameters, and replays NFS requests in the trace against the file server being tested. In this section, we discuss each of these steps in more detail.

6.3.1 Trace Transformation

Field	Description
`callTime`	Timestamp of the call
`respTime`	Timestamp of the response
`opType`	NFS operation type
`opParams`	Request parameters (specific to the `opType`)
`opReturn`	Values returned by the operation

Table 6.1: Each TBBT trace record contains the time at which an NFS request is made, the time of the corresponding response, the type of NFS operation in the request, the request's input parameters, and the associated return values.

TBBT uses a trace format that consists of a pair of request and reply - <`callTime`, `respTime`, `opType`, `opParams`, `opReturn`>. These are described in Table 6.1. The call and response are paired through an RPC message exchange ID. The `opType` is equivalent to the NFS procedure number in the original trace. The `opParams` and `opReturn` are similar to the corresponding NFS procedure parameters and return values. TBBT currently handles NFSv2 and NFSv3 but may be extended to handle NFSv4 and other network storage protocols in the future.

An important aspect of the TBBT trace format is the creation of the TBBT trace. This operation requires more than simply reformatting the original trace. Consider the way that TBBT rewrites each NFS *filehandle*. In the NFS protocol, a *filehandle* is used to identify a specific file or directory. However, it is possible for a single object to have more than one *filehandle* (because many implementations embed information such as the object version number and file system mount point inside

the *filehandle*). To make matters worse, some NFS operations (such as `create`, `lookup`, and `remove`) use a name to identify a file instead of using a *filehandle*. For example, in the case of `create` or `mkdir`, the *filehandle* is not known to the client because the file or directory does not yet exist. To avoid any potential for ambiguity, TBBT assigns TBBT-IDs to *all* of the files and directories that appear in the trace. The NFS server might use a different *filehandle* for a particular file every time the trace is replayed, but the TBBT-ID will never change.

TBBT also inserts additional information into the trace records to facilitate trace replay. For example, neither a `remove` request nor a `remove` reply contains the *filehandle* of the removed file, which is needed for file system dependency analysis during trace replay (as discussed in Section 6.3.4). The same problem exists for `rmdir` and `rename`. For all three operations, TBBT infers the TBBT-ID of the object in question from the parent's TBBT-ID, the object name and the file system image, and inserts it into the associated trace record.

TBBT trace rewriting also handles errors or omissions in the original trace. The most common error is packet loss. The traces we use for our experiments are reportedly missing as many as 10% of the NFS calls and responses during periods of burst traffic [17]. The number of lost calls and responses can be estimated by analyzing the progression of RPC exchange IDs (XIDs), which are typically generated by using a simple counter. This, however, does nothing to tell us *what* was lost.

In many cases, the contents of missing calls may be inferred – although not always with complete certainty. For example, if we observe the request sequence `remove A; remove A` and each of these requests has a successful reply, then it is clear that there must be a `create`, `rename`, `symlink`, or `link` request between the two `remove` requests – during the interval when file "A" is removed the first and second times, another file named "A" must have appeared – but this event is missing from the trace. If we simply replayed the trace without correcting

this problem, then a second `remove A` would fail instead of succeed. Such a discrepancy is called a *replay failure*. The correction is to insert NFS operations to replace the missed packets. Note that it is also a replay failure if the replay of an operation returns successfully while the original trace recorded failure, or if both original and replay return failure but for different reasons.

We use a table-driven heuristic approach to select corrective operations and insert them into the replay stream. Enumerating all possible combinations of operations, trace return codes, and replay return codes would require an enormous table. In practice, however, the combinations we have actually encountered all fall into approximately thirty distinct cases. Table 6.2 illustrates a small number of unexpected replay failures and the rules we use to resolve them.

Op	**Replay error**	**Corrective Op(s)**
`create`	file already exists	`remove`
`remove`	file does not exist	`create`
`rmdir`	directory not empty	`remove` and/or `rmdir`
`getattr`	permission denied	`setattr`

Table 6.2: Examples of trace corrections. In these examples, the operation was observed to succeed in the trace, but would fail during replay. To prevent the failure, corrective operations are are added to replay to ensure that the observed operation will succeed.

Note that there is frequently more than one way to augment the trace in order to prevent the problem. For example, if a file cannot be created because it already exists in the file system, we could either `rename` the file or `remove` it. We cannot determine which of these two operations are missing (or whether there are additional operations that we missed as well) but we can observe that `removes` are

almost always more frequent than `renames` and therefore always choose to correct this problem via a `remove`.

A similar problem is that we can not accurately determine the correct timestamp for each corrective operation. Therefore the inserted operations might not perfectly recreate the action of the missing packets. There are also lost packets which do not lead to replay failures and therefore cannot be detected. Since the overall number of lost RPC messages is small (approaching 10% only in extreme situations, and typically much smaller) the total number of corrective operations is always much smaller than the operations taken verbatim from the original trace.

Potential replay failures are detected and corrected through a simulated *pre-play*. The pre-play executes the trace requests one-by-one in a synchronous fashion. Replay failures are detected by comparing the return value of original request in the trace and the return value of the pre-play. The corrective operations are generated in accordance with the rules shown in the trace correction table. Corrections are then inserted into the trace with appropriate timestamps.

6.3.2 Creating the Initial File System Image

To replay calls from a file access trace, the tested server must be initialized with a file system image similar to that of the traced server so that it can respond correctly to the trace requests. There are two factors to be considered while creating the initial file system image: the logical file system hierarchy and the physical disk layout. While the former is essential for correct trace replay, the latter is crucial to the performance characteristics of the file system. Ideally, one could take a file system snapshot of the traced server before a trace is collected. In practice, however, this is often impractical because it may cause service degradation. Moreover, most file system snapshotting tools capture only the file system hierarchy but not the physical layout. TBBT approximates the traced server's file system image using

information from the NFS trace. It then constructs (and ages) the image through the native file system of the tested server.

The idea of extracting the file system hierarchy from an NFS trace is not new [8, 19]. However, because earlier tools were developed mainly for the purpose of trace studies, the extracted file system hierarchy may not be sufficiently complete to permit trace replay. For example, if operations such as `symlink`, `link` and `rename` are not handled properly, the dynamic changes to the file system hierarchy during tracing cannot be properly captured.

TBBT's file system hierarchy extraction tool produces a *hierarchy map*. Each entry in the hierarchy map contains the following fields: *TBBT-ID*, *path*, *createTime*, *deleteTime*, *size*, and *type*. Each hierarchy map entry corresponds to one file system object under one path. File system objects with multiple hardlinks have multiple paths and may appear in multiple hierarchy map entries, but have the same TBBT-ID in each entry. If a path exists before trace collection starts, its createTime is set to 0 (to indicate that TBBT must create this object before the trace replay begins), and the size field gives the object's size at the time when the trace began. The type field indicates whether the file is a regular file, a directory, or a symbolic link.

The file system hierarchy extracted from an NFS trace is not necessarily a complete snapshot of the traced file system because only files that are referenced in the trace appear in the TBBT *hierarchy map* and many workloads are highly localized. In traces gathered from our own systems, we observed that in many cases only a small fraction of a file system is actually accessed during the course of a day (or even a month). The fact that only active files appear in the TBBT *hierarchy map* may have a serious effect on the locality of the resulting file system. To alleviate this problem, TBBT augments the extracted file system hierarchy with additional files. Details about how these objects are created are given in Section 3.3.2.

TBBT populates the target server file system by traversing the hierarchy map in a breadth-first or depth-first order, creating each file, directory, or link as it is encountered. This approach yields a nearly ideal physical disk layout for the file system hierarchy: free space is contiguous, data blocks of each file are allocated together and therefore likely to be physically contiguous, data is close to the corresponding metadata, and files under the same directory are grouped together. As a result, the real world effects of concurrent access and file system aging are not captured. TBBT's artificial aging technique is designed to emulate these effects.

6.3.3 Artificially Aging a File System

The effect of aging centers on fragmented free space, fragmented files, and declustered objects (objects which are often accessed together but are located far from each other on the disk). TBBT's aging mechanism is purely synthetic and is not meant to emulate the actual file system aging process (as emulated in Keith Smith's work [60]) It focuses on emulating the fragmentation of file blocks and free space, but the mechanism is extensible to include declustering effects among related file system objects. .

An important design constraint is that TBBT must be capable applying aging effects to any file system without resorting to a raw disk interface. Using only the standard system call interface makes it easier to integrate a file system aging mechanism into other file system benchmarking tools.

Aging is related to file system block allocation algorithms. Some of our analysis assume a FFS-like block allocation policy. This policy divides a file partition into multiple *cylinder groups*, each of which has a fixed number of free inodes and free blocks. Files under the same directory are preferably clustered in one group.

File System Aging Metrics

To the best of our knowledge, there are no standard metrics to quantify the effect

of aging on a file system. Before presenting our file system aging metrics, we define several basic terms that we use.

A file system *object* is a regular file, directory, symbolic link, or a special device. The *free space object* is an abstract object that contains all of the free blocks in the file system. A *fragment* is a contiguous range of blocks within an object. The *fragment size* is the number of blocks within a fragment, and the *fragment distance* is the number of physical blocks between two adjacent fragments of the same object. The *block distance* is the number of physical blocks between two adjacent logical blocks in an object. The *inode distance* is the number of physical blocks between an object's inode and its first block and the *parent distance* is the number of physical blocks between the first block of an object and that of its parent directory. The block used in these definitions is file system block(4K by default).

If we assume that the policy goal for file block allocation is to have sequential physical blocks, then the effect of file system aging (in terms of the fragmentation it causes) can be quantified in terms of the physical distance between consecutive blocks of a file. *Average fragment distance*, *average block distance* and *average fragment size* are calculated over all fragments/blocks that belong to each file within a file system partition, and are related to one another as follows: *average fragment distance* = *average block distance*$\times$*average fragment size*. Because the calculation of these metrics is averaged over the number of blocks or segments in a file, files of different sizes are weighted accordingly. *Average block distance* describes the overall degree of file fragmentation. Either of the other two metrics helps further distinguish between the following two types of fragmentation: a large number of small fragments that are located relatively close to each other, or a small number of large fragments that are located far away from each other. *Average inode distance* can be considered as a special case of *average block distance* because it measures the distance between a file's inode and its first block.

In an aged file system, both free space and allocated space are fragmented. The *average fragment size* of the special *free space object* reflects how fragmented the free space portion of a file partition is. The file system aging effect can also be quantified by the degree of clustering among related files, e.g., files within the same directory. The *average parent distance* is meant to capture the proximity of a directory and the files it contains and, indirectly, the proximity of files within the same directory. Alternatively, one can compute *average sibling distance* between each pair of files within the same directory.

These metrics provide a simplistic model; they do not capture the fact that logical block distances do not equate to physical seek time nor do they reflect the non-commutative nature of rotational delays. (The latter explains the differences in time that may be required to move the disk head from position A to position B than from B to A.) This simplistic model does have several benefits, however: it is both device and file-system independent, and does provide intuition for the performance of the file system.

File System Aging Techniques

File deletions account for most free space fragmentation. Fragmented files, in contrast, are caused by two reasons: 1) when a file grows in a context of free space fragmentation and the absence of contiguous free blocks to allocate, and 2) when the interleaving of append operations to several files causes blocks associated with these different files to be interleaved as well. There are techniques that mitigate the fragmentation effect of interleaved appends. These include dividing a logical partition into *cylinder groups* and then placing files in different cylinder groups [46]. Another technique is to preallocate contiguous blocks when a file is opened for writing. Despite these optimizations, file fragmentation may still materialize if interleaved appends occur within the same group or if the file size is more than the pre-allocated size.

The aging effects become more pronounced when inode and block utilization between cylinder groups are not balanced. To reduce the declustering effect, an FFS-like policy tries to place files under the same directory in one group, and to allocate one file's inode and data blocks in the same group. But this policy also tries to keep balanced utilizations among different *cylinder groups*. Once the utilization of a group is too high, allocation switches to another cylinder group, if available. The imbalanced usage is usually caused by a highly skewed directory tree where some directory has many small files or files of very large size.

TBBT relies on interleaved appending as the primary file system aging technique, and uses file deletion only to fragment the free space. TBBT's initialization procedure populates a file system partition with the initial hierarchy derived from the input trace and additional synthetic objects to fill all available space. These synthetic objects are used both to populate the incomplete file system hierarchy and to occupy free space. All of the objects get fragmented because of interleaved appending. At the end of the initialization, the synthetic objects that occupy the free space are deleted to make fragmented free space available. To initialize a 1GB file system partition with 0.1GB of free space, we write exactly 1GB of data and then delete 0.1GB of data. In contrast, Smith's aging technique writes around 80GB of data, and deletes around 79GB of data.[1]

To determine the set of synthetic objects to be added to a file system and to generate a complete TBBT hierarchy map, TBBT takes four parameters. The first two parameters are *file size distribution* and *directory/file ratio*, which are similar to

[1]Note that our choice of terminology and examples in this discussion assume that the underlying file system uses an FFS-like strategy for block allocation. Our methodology works just as well with other strategies, such as LFS, although for LFS instead of fragmenting the free space, we create dead blocks for the cleaner to find and reorganize.

SPECsfs's file system initialization parameters. The third parameter is the *distortion factor*, which determines the degree of imbalance among directories in terms of directory fan-out and the file size distribution within each directory. The fourth parameter is the *merge factor*, which specifies how extensively synthetic objects are commingled with the initial file system image. A low *merge factor* means that most directories are dominated by either synthetic objects or extracted objects, but not both.

To create fragmentation, TBBT interleaves the append operations to a set of files, and in each append operation adds a certain number of blocks to the associated file. To counter the file pre-allocation optimization technique, each append operation is performed in a separate open-close session. File blocks written in an append operation are likely to reside in contiguous disk blocks. However, blocks that are written in one append operation to a file may be far away from those blocks that are written in another append operation to the same file. The expected distance between consecutive fragments of the same file increases with the total size of files that are appended concurrently. By controlling the **interleaving scope** - the total size of files involved in interleaved appending – and the number of blocks in each append operation, TBBT can control the *average block distance* and *average fragment size* of the resulting file system. We assume that large files tend to be written in larger chunks. Instead of directly using the number of blocks in each append operation to tune *average fragment size*, we use **append operations per file**, which specifies the number of appending operations used to initialize one file. The minimum size of each fragment is 1 block. Usually the average file size is around 10 blocks. Therefore, a very large value for *append operations per file* may only affect some large files.

The declustering effect is described by *average inode distance* and *average parent/sibling distance*. To create this effect, TBBT may add a zero-sized synthetic

object to create a skewed directory hierarchy and then provoke imbalanced usage among cylinder groups. To increase the *average parent/sibling distance*, rather than select files randomly, TBBT interleaves files from different directories.

In summary, given a TBBT hierarchy map, TBBT's file system aging mechanism tries to tune: *average block distance* and *average fragment size* of a normal file, *average fragment size* of the special *free space object*, *average inode distance* and *average parent/sibling distance*. *Average block distance* is tuned via the *interleaving scope*. *Average fragment size* is tuned via the *append operations per file*. Different aging effects could be specified for different files, including the special *free space object*. We have not implemented controls for *average inode distance* and *average parent/sibling distance* in the current TBBT prototype. Randomization is used whenever possible to avoid regular patterns. TBBT's aging technique can be used to initialize the file system image for both trace-based and synthetic workload-based benchmarking.

6.3.4 Trace Replay

When replaying requests in an input trace, TBBT respects the semantics of the NFS protocol. Sending requests in strict, timestamped sequence is not always feasible. For example, given sequence1 in Table 6.3, if the create reply comes at time 3 during the replay, it is impossible to send the write request at time 2. TBBT's trace player provides flexible policies to handle request timing issues. For SPECsfs-like synthetic benchmarks, multiple processes are used to generate requests against multiple disjointed directories, and in each process requests are executed synchronously without any concurrency. As a result, the SPECsfs load generation policy is much simpler.

Workload scaling is a common feature in synthetic benchmarks. It is also desirable for the TBBT trace player so that one trace can be used to evaluate file

systems/servers with a wide range of performance, or to combine different traces. However, there is no absolute guarantee regarding workload fidelity after scaling it artificially.

Ordering and Timing Policy

TBBT's trace player provides two ordering policies to determine the relative order among requests: *conservative order* and *FS dependency order*. Both guarantee the replay can proceed to completion, and both apply the same modifications to the initial file system hierarchy at the end of trace play. TBBT's trace player also provides two timing policies: *full speed* and *timestamp-based*, to determine the exact time at which requests are issued. In the *full speed* policy, requests are dispatched as quickly as possible, as long as the chosen ordering policy is obeyed. In the *timestamp based* policy, requests are dispatched as close to their timestamps as possible without violating the ordering policy.

When the *conservative order* policy is used, a request is issued only after all prior requests (e.g., requests with earlier timestamps) have been issued and all prior replies have been received. The *conservative order* captures some of the concurrency inherent in the trace although it will not generate a workload with higher concurrency. In contrast, there is no concurrency in the workload generated by each process of SPECsfs's load generator. Because of differences in the traced server and tested server, it is impossible to guarantee that the order of replies in the trace replay is exactly the same as that in the trace. The disadvantage of *conservative order* is that processing latency variations in the tested server may inadvertently affect its throughput. For example, in sequence2 of Table 6.3, if the create latency during replay is three times higher than the latency in the original trace, the request issue ordering becomes sequence3, which has a lower throughput than sequence2.

T	**sequence1**	**sequence2**	**sequence3**	**sequence4**
0	creat A req	creat A req	creat A req	creat A req
1	creat A ack	creat A ack		write B req
2	write A req	write B req		write B ack
3	write A ack	write B ack	creat A ack	creat A ack
4			write B req	
5			write B ack	

Table 6.3: Examples illustrating the ordering issue in trace replay. The first column represents normalized time. Other columns represent NFS request sequence examples. The create latency is 1 on the traced server and 3 on the tested server. In Sequence1 there is an FS-level dependency because both operations involve the same file. Sequence2 has no FS-level dependency but may have an application-level dependency. Sequence3 is the result of replaying sequence2 by *conservative order*. Sequence4 is the result of playing sequence2 by *FS dependency order*.

The *FS dependency order* policy uses a read/write serialization algorithm to discover the dependencies of each request on other earlier requests and replies in the trace. Accordingly, the request issue ordering for sequence2 in Table 6.3 becomes sequence4, which results in higher throughput than sequence3. Conceptually, the file system hierarchy is viewed as a shared data structure and each NFS request is a read or write operation on one or more parts of this structure. If an NFS operation modifies some part of the structure that is accessed by a later operation in the trace, then the latter operation cannot be started until the first one has finished. For example, it is dangerous to overlap a request to create a file and a request to write some data to that file; if the write request arrives too soon, it may fail because the file does not yet exist. In many cases it is not necessary to wait for the response, but instead to ensure that the requests are made in the correct order. The exceptions are replies from `create`, `mkdir`, `symlink`, and `mknod`. These replies are regarded as write operations to the newly created object and therefore must be properly serialized with respect to subsequent accesses to these newly created objects. Table 6.4 summarizes the file system objects that are read or written by each type of request and reply. Because concurrent access to the same file system object is infrequent in real NFS traces, the granularity of TBBT's dependency analysis is an individual file system object. For finer-granularity dependency analysis, inode attributes and each file block could be considered separately.

The *FS dependency order* may be too aggressive because it only captures the dependencies detectable through the shared file system data structure but does not discover application-level dependencies. For example, in Table 6.3, if the application logic is to `write` some debugging information to the log file `B` after each successful `create A` operation, then the write operation indeed depends on the create operation and should be sent after receiving the create request's reply. In this

case, ordering requests based on FS-level dependencies is not sufficient. In general, *conservative order* should be used when *FS dependency order* cannot properly account for many application-level dependencies.

request/reply	**shared data structure set**	**type**
REQ : `read/readdir/ getattr/readlink` obj	obj	'read'
REQ : `write/setattr/commit` obj	obj	'write'
REQ : `lookup` parent, name([obj])	parent, [obj]	'read'
REQ : `create/mkdir` parent, name	parent	'write'
REPLY: `create/mkdir` obj	obj	'write'
REQ : `remove/rmdir` parent, name([obj])	parent, [obj]	'write'
REQ : `symlink` parent, name, path	parent	'write'
REPLY: `symlink` [obj]	[obj]	'write'
REQ : `rename` parent1, name1, parent2, name2([obj2])	parent1, parent2, [obj2]	'write'
`all other replies`	empty	-

Table 6.4: The file system objects that are read or written by different requests and replies. The notation [obj] means that the object may not exist and therefore the associated operation might return a failure.

Workload Scaling

TBBT can scale traces up or down, spatially or temporally. To spatially scale up a trace, the trace and its initial file system image are cloned several times, and then each cloned trace is replayed against a separate copy of the initial image. A spatial scale-up is analogous to the way that synthetic benchmarks run multiple

load-generation processes. To spatially scale down, the trace is decomposed into multiple sub-traces, where each sub-trace accesses only a proper subset of the initial file system image. Not all traces can be easily decomposed into such sub-traces, but it is typically not a problem for traces collected from file servers that support a large number of clients and users.

Temporally scaling up or down a trace is implemented by issuing the requests in the trace according the scaled timestamp and observing the chosen ordering policy. An ordering policy from above bounds the temporal scaling of a given trace. The two scaling approaches can be combined to scale a trace. For example, if the required speed-up factor is 12, it can be achieved by a spatial scale-up factor of 4 and a temporal scale-up factor of 3.

6.4 Implementation

Trace transformation and initial file hierarchy extraction are implemented in Perl. Trace replay is implemented in C. Each trace is processed in three passes. The first pass transforms the collected trace into TBBT's trace format, with the exception that, in replies to `remove, rmdir`, and `rename`, the TBBT-ID field is not available in the first pass. The second pass corrects trace errors by a pre-play of the trace. The third pass extracts the *hierarchy map* and adds the TBBT-ID to the replies to `remove, rmdir, rename`. Each successful or failed directory operation may contain information about a <parent, child> relationship from which the *hierarchy map* is built. Hierarchy extraction consumes a great deal of CPU and memory, especially for large traces. An incremental version of hierarchy extraction is possible and will greatly improve its efficiency.

Similar to SPECsfs [62], TBBT's trace player bypasses the NFS client and sends NFS requests directly to the tested file server via user-level RPC. The software

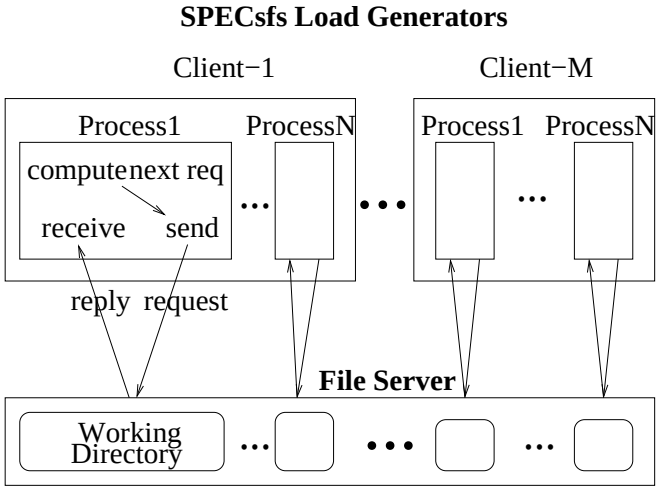


Figure 6.1: SPECsfs uses multiple independent processes to generate requests targeted at disjointed directories.

architecture of TBBT player, however, is different from that of SPECsfs. As shown in Figure 6.1, the workload generator of SPECsfs on each client machine uses a multi-process software architecture, with each process dispatching NFS requests using synchronous RPC. In contrast, TBBT uses a 3-thread software structure, as shown in Figure 6.2, which is more efficient because it reduces context switching and scheduling overhead. The *I/O thread* continuously reads trace records into the *operation queue*, a cyclic memory buffer. The *send thread* and *receive thread* send NFS requests to and receive replies from the tested NFS server using asynchronous RPC. The *operation queue* is also known as the *look-ahead window*. The size of *look-ahead window* should be several times larger than the theoretical concurrency bound of the input trace to ensure that the *send thread* is always able to find enough concurrent requests at run time.

The *send thread* determines whether an NFS request in the input trace is ready for dispatch by checking whether (1) it follows the ordering policy; (2) the request's

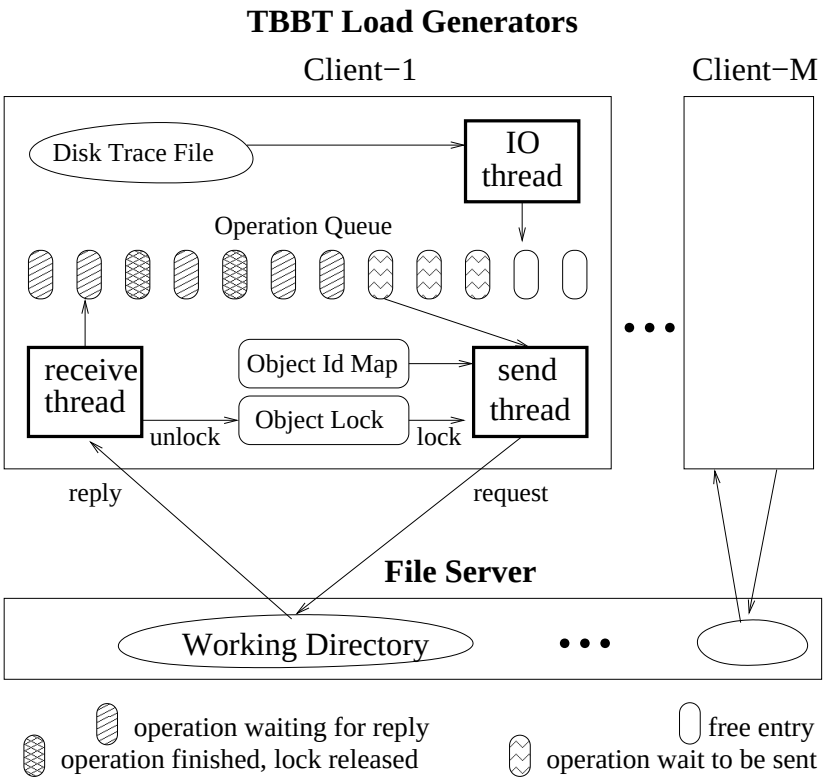


Figure 6.2: TBBT uses a three-thread process to read and replay traces stored on disk.

timestamp is larger than the current time-stamp, and (3) the number of outstanding requests to a given server exceeds the given threshold. The second check is only for the *timestamp-based* policy. The third is to avoid overloading the test file server. If a file server is overloaded, performance degrades. The first check is straightforward in the case of *conservative order*. For *FS dependency order*, we use *object locking* as illustrated in Figure 6.2. Before dispatching an NFS request, the *send thread* acquires the read/write lock(s) on all the object(s) associated with the request (and sometimes the reply). Some locks are released after the request is dispatched, other locks are released after the reply is received. Since all locks are acquired by the *sending thread* and there is only one *sending thread*, lock contention and atomicity issues are avoided. Each lock is not a real operating system lock, but instead a flag associated with a file system object.

During trace replay, requests are pre-determined rather than computed on the fly according to current replay status as in some synthetic benchmarks. This means that a robust trace player must react to transient server errors or failures in ways that sustain trace playing for as long as possible. This requires the trace player 1) to identify subsequent requests in the trace that are affected by a failed request, directly or indirectly, and then skipping them, and 2) to contain the side effects of various run-time errors. For example, because a `create` request is important for a trace replay to continue, it will be re-tried multiple times if the request fails; however, a failed `read` request will not be retried so as not to disrupt the trace replay process.

6.5 Evaluation

In this section, we examine the validity of the trace-based file system benchmarking methodology, analyze to what extent we may scale the workload, explore the difference between the evaluation results from TBBT and SPECsfs and conclude

with a measure of the run-time cost of our TBBT prototype.

The NFS traces used in this study were collected from the EECS NFS server (EECS) and the central computing facility (CAMPUS) at Harvard over a period of two months in 2001 [17]. The EECS workload is dominated by metadata requests and has a read/write ratio of less than 1.0. The CAMPUS workload consists almost entirely of email and is dominated by reads. The EECS trace and the CAMPUS trace grow by 2 GBytes and 8 GBytes per day, respectively. Most of the Harvard traces have a packet loss ratio of between 0.1-10%.

We used TBBT to drive two NFS servers. The first is the Linux NFSv3 and the second is a repairable file system called RFS. RFS augments a generic NFS server with fast repairability and without modifying the NFS protocol or the network file access path [75]. The same machine configuration is used for post-collection trace processing, hosting the test file systems, and running TBBT trace player and SPECsfs benchmark. The machine has a 1.5-GHz Pentium 4 CPU, 512-MBytes of memory, and one 40-GByte ST340016A ATA disk drive with 2MB on-disk cache. The operating system is RedHat 7.1.2 with Linux kernel 2.4.7.

6.5.1 Validity of Trace-Based Evaluation

An ideal trace analysis and replay tool should be able to faithfully recreate the initial file system image and its disk layout and replay the requests in the trace with accurate timing. In this section, we evaluate how successful TBBT is in approximating this ideal. Because the Buttress project has already solved the trace replay timing problem, this issue is omitted here.

Extraction of File System Hierarchy

To understand how different the extracted file system hierarchy is from the actual file system hierarchy, we measured the number of disjointed directory subtrees, the number of directories, the number of files, and the total file system size of the

derived file system hierarchy. Figure 6.3 shows the results for the EECS trace from 10/15/2001 to 10/29/2001. The Y-axis is in logarithmic scale. The total file system size on the EECS server is 400 GB, but only 42 GB are revealed by this 14-day trace (especially during the first several days). We expect the rate will slow further if additional weeks are added. Even at a rate of 7GB per week, however, we would only discover about 84 GB at the end of two months, or about 21% of the total file system size. This indicates that when the initial file system hierarchy is not available, the hierarchy extracted from the trace may be only a small fraction of the real hierarchy. It is therefore essential to introduce artificial file objects to achieve a comparable disk layout.

Effectiveness of Artificial Aging Techniques

In the following experiments, both file system and disk prefetch were enabled, and the file system aging metrics were calculated using disk layout information obtained from the *debugfs* utility available for the ext2 and ext3 file systems. We applied our aging technique to two test file systems. The first is a researcher's home directory (which has been in continuous use for more than 1.5 years) and the second is the initial file system image generated by the SPECsfs benchmark.

From the researcher home directory, we selected two subdirectories, *dir1* and *dir2*. For each subdirectory, we created three different versions of the disk image. The first version is a *naturally aged* version, which we obtained by copying the original subdirectory's disk image using `dd`. The second version is a *synthetically aged* version, which we obtained by applying our aging technique to the original subdirectory. The third version represents a *linearized* version of the original subdirectory's disk image using `cp -r` and thus also corresponds to the optimal disk image without any aging effect. The file system buffer cache was purged before each test. For the three versions of each subdirectory, we measured the elapsed time of the `grep -r`, which is a disk-intensive command that typically spends at

least 90% of its execution time waiting for the disk. Therefore aging was expected to have a direct effect on the execution time of the `grep` command.

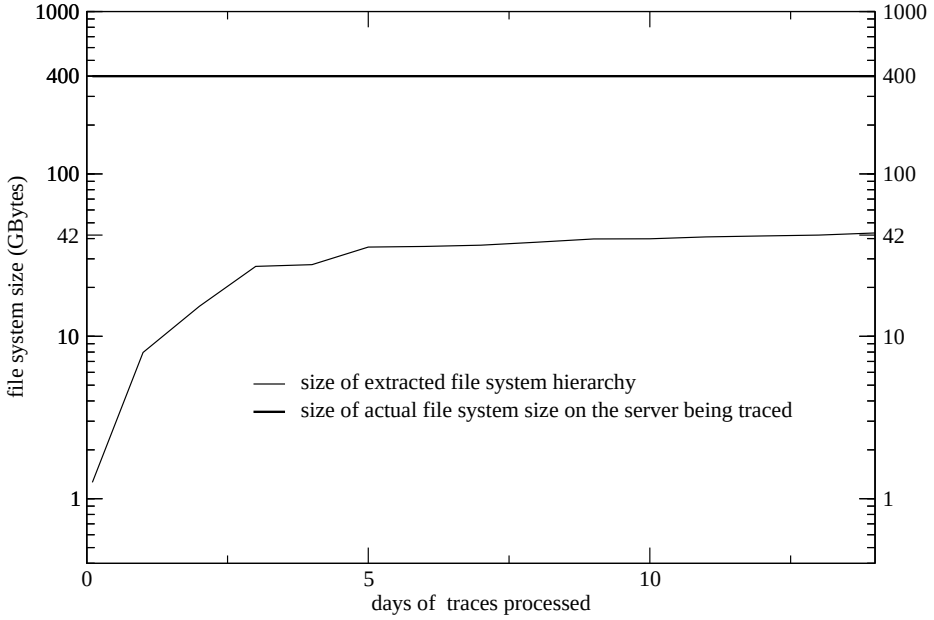


Figure 6.3: The total size of file system hierarchy discovered over time. Longer traces provide more information about the file system hierarchy, but with rapidly diminishing returns.

Figures 6.4 and 6.5 show that the proposed aging technique has the anticipated impact on the performance of `grep` for *dir1* and *dir2*: more interleaving and finer-grained appends result in more fragmentation in the disk image, which leads to lower performance.

Moreover, with proper aging parameter settings, it is actually possible to produce a synthetically aged file system whose `grep` performance is the same as that of the original naturally aged file system. For Figure 6.4, the <*interleaving scope*, *append operations per file*> pairs that correspond to these cross-over points are <2,8>, <4,2>, and <16,1>. For Figure 6.5, the <*interleaving scope*, *append*

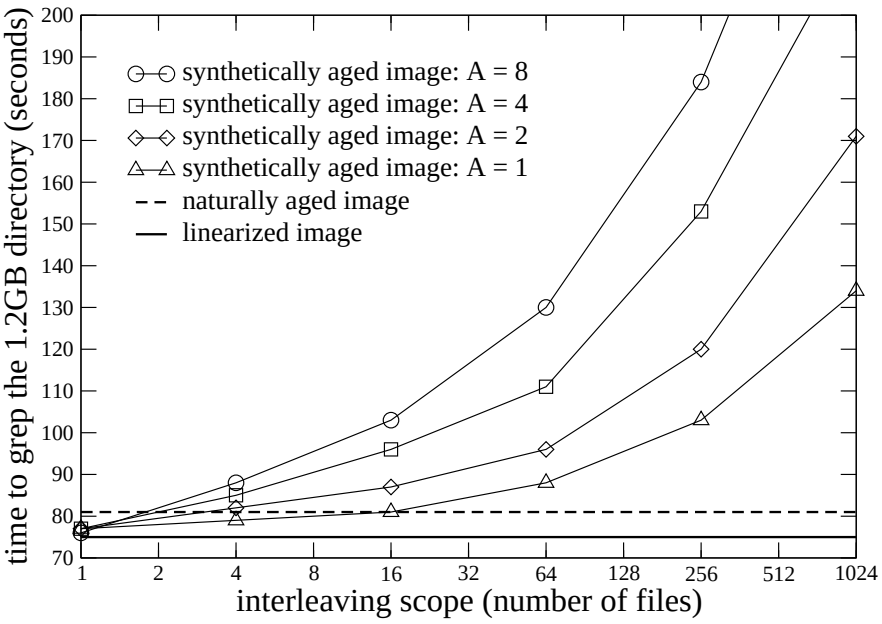


Figure 6.4: The elapsed time to complete the command `grep -r` on *dir1* (a research project directory) consistently increases with the *interleaving scope* and the *append operations per file* parameter. Different curves correspond to different values of the *append operations per file* parameter. For example, "A = 8" means the *append operations per file* is 8. The lines for the *naturally aged* and *linearized* image are flat because no synthetic aging is applied to these two cases.

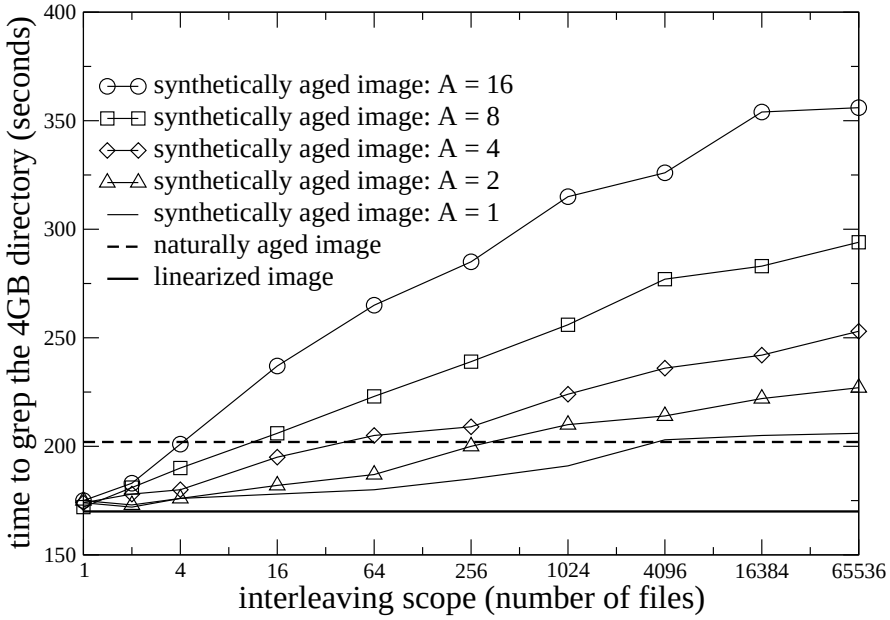


Figure 6.5: This figure shows the results of conducting an experiment similar to that shown in Figure 6.4 on a much larger research project directory. The two figures show that the aging parameters create qualitatively similar but quantitatively different aging effects on different file system data.

operations per file> pairs that correspond to these cross-over points are <4,16>, <16,8>, <64, 4>, <256, 2>, and <4096, 1>. These results demonstrate that the proposed aging technique can indeed produce a realistically aged file system image. However, the question of how to determine aging parameters automatically remains open.

Figures 6.4 and 6.5 also show that the `grep` performance of the original naturally-aged image is not very different from that of the linearized image; the impact of natural aging is not more than 20%. TBBT's aging technique can generate much more dramatic effects, but it is not clear whether such aging occurs in practice.

To show that the proposed aging technique can be used together with a synthetic benchmark such as SPECsfs, we ran the SPECsfs benchmark on the image initialized by SPECsfs itself and the image initialized by the aging technique with different parameters. We used an *append operations per file* value of 4 and varied the *interleaving scope* value. We then measured the average read latency and the initial image creation time. The results are shown in Table 6.5. As expected, the average read latency increased as the initial file system image aged more drastically. The difference (before and after running SPECsfs) in average block distance shows the aging effect produced by the run itself. Finally, the time required to create an initial file system image in general increased with the degree of aging introduced. SPECsfs takes more time to create the initial image even though the net aging effect is much smaller. The reason is because SPECsfs uses multiple processes to initialize the file system image therefore its disk access pattern during the initialization is not as sequential as TBBT.

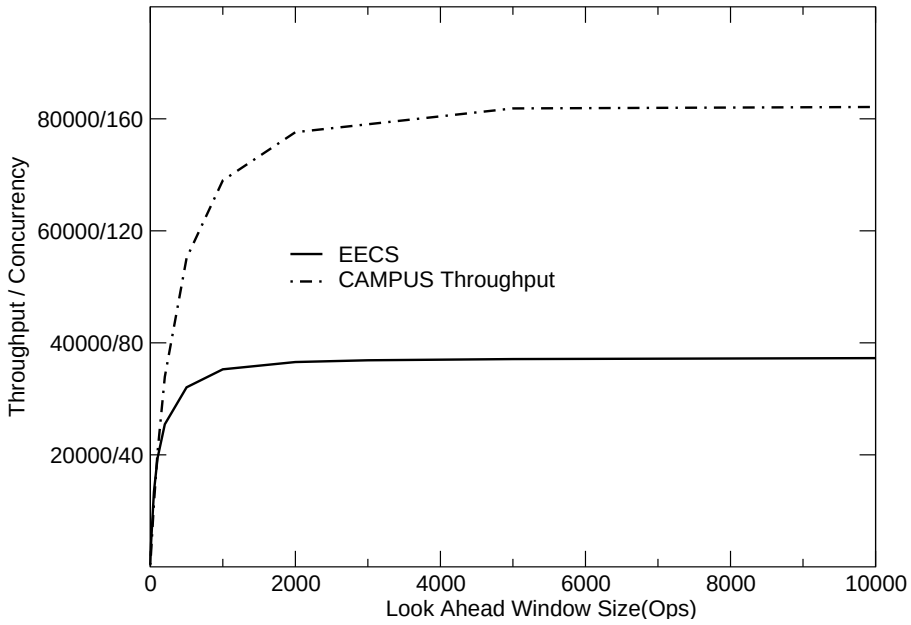


Figure 6.6: The impact of the *look-ahead window* size on the concurrency and thus the throughput of the workload that TBBT can generate from the EECS and CAMPUS trace.

	SPECsfs initialization	scope =512	scope =8192	scope =65536
average block distance before SPECsfs run	2	20	2180	6230
average block distance after SPECsfs run	817	828	2641	6510
average read latency	3.24 msec	3.24 msec	3.31 msec	4.45 msec
time to create initial image	683 sec	330 sec	574 sec	668 sec

Table 6.5: Results of applying the proposed file system aging techniques to SPECsfs. First column gives result of using SPECsfs's own initialization procedure. Other three columns show the result of using TBBT's aging technique to create SPECsfs run's initial file system image.

6.5.2 Workload Scaling

To study the maximum concurrency available in a trace, we conducted a simulation study. The simulation assumed that the reply for each request always comes back successfully after a pre-configured latency. The throughput result is given in Figure 6.6. In this simulation, there were two factors that limited the maximum concurrency: the *look-ahead window* in which future requests are examined during the simulation, and the per-request latency at the server. For per-request latency, we used the latency numbers in Table 6.6. Figure 6.6 shows the correlation between the maximum throughput that can be generated and the look-ahead window size.

The simulation results show that even for a lightly loaded workload such as the EECS trace (30 requests/sec) and a modest *look-ahead window* size (4000), there is enough concurrency to drive a file server with a performance target of 37000

requests/sec using temporal scaling.

NFS 10/21/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
throughput	33	30	189	180	1231	1807
getattr	5.1	0.6	0.9	1.5	2.1	0.7
lookup	2.9	0.9	0.8	2.0	2.0	1.2
read	9.6	3.1	5.3	4.8	5.4	4.7
write	9.7	2.2	4.4	3.8	4.6	2.5
create	0.5	0.7	0.7	0.9	17.3	0.7

Table 6.6: Per-operation latency and overall throughput comparison between TBBT and SPECsfs for an NFS server using the EECS 10/21/2001 trace. "T" means TBBT, "S" means SPECsfs.

6.5.3 Comparison of Evaluation Results

We conducted experiments to evaluate two file servers, NFS and RFS, using both TBBT and the synthetic benchmark SPECsfs. In these experiments, the tested file system was properly warmed up before performance measurements were taken. We first played the EECS trace of 10/21/2001, and tried to tune the parameters of the SPECsfs benchmark so that they matched the trace's characteristics as closely as possible. We also changed the source code of SPECsfs so that its file size distribution matched the file size distribution in the 10/21/2001 trace. The maximum throughput of the Linux NFS server under SPECsfs is 1231 requests/sec, and is 1807 requests/sec under TBBT. The difference is a non-trivial 31.8%. In terms of per-operation latency, Table 6.6 shows the latency of five different operations under

the original load (30 requests/sec), under a temporally scaled load with a speed-up factor of 6, and under the peak load. The per-operation latency numbers for TBBT and for SPECsfs were qualitatively different in most cases.

RFS 10/21/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
throughput	32	30	187	180	619	1395
getattr	4.0	0.7	2.2	1.2	3.2	0.8
lookup	4.4	0.7	2.8	1.3	2.6	1.0
read	10.8	3.3	8.4	4.1	18.1	4.9
write	11.6	5.4	7.4	4.0	11.1	2.8
create	0.7	1.0	5.1	1.3	16.3	1.2

Table 6.7: Performance results for RFS server using EECS 10/21/2001 trace.

NFS 10/22/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
Throughput	16	15	191	187	2596	4125
getattr	4.7	0.5	0.7	0.7	1.02	0.7
lookup	2.8	0.6	0.5	0.8	1.01	0.6
read	10.3	2.1	19.7	3.1	7.4	4.2
write	7	1.0	6.3	1.2	3.8	3.0
create	0.5	0.9	1.2	0.5	7.9	0.7

Table 6.8: Performance results for NFS server using EECS 10/22/2001 trace.

The same experiment, using RFS instead of the default Linux NFS server, is

shown in Table 6.7. The maximum throughput of the RFS server was 619 requests/sec for SPECsfs versus 1385 requests/sec for TBBT – a difference of more than a factor of two. Again there was no obvious relationship between the average per-operation latency for SPECsfs and TBBT.

To determine whether these differences are consistent across traces taken from different days, we ran the 10/22/2001 EECS trace against the LINUX NFS server. The 10/22/2001 trace was dominated by metadata operation (80%) while the 10/21/2001 trace had substantial read/write operations (60%). The SPECsfs configuration was again tuned to match the access characteristics of the 10/22/2001 trace. The results in Table 6.8 show that the difference between TBBT and SPECsfs in throughput and per-operation latency is still quite noticeable.

In all the trace replay experiments, the percentage of failed requests, i.e., those that return a different value that that in the original trace, is less than 1%. This means that TBBT can successfully replay traces recorded from one server against other servers. The fact that the latency and throughput measurements from TBBT deviated substantially from those from SPECsfs for different file servers and for different traces suggests that the trace-based file system/server evaluation methodology is indeed a valuable tool in gauging the performance of file servers under site-specific workloads.

6.5.4 Implementation Efficiency

TBBT's post-collection trace processing algorithm can process 2.5 MBytes of trace or 5000 requests per second. TBBT's initialization time increases with the total file system size as well as with the degree of file system aging. This is because the more drastic the aging effect TBBT, the less the disk access locality in its file system population process. Table 6.5 shows TBBT's initialization time is also affected by *average block distance level*. Overall TBBT's aging techniques are very efficient.

The initialization speed is more than two orders of magnitude faster than Smith's aging technique.

The run-time efficiency of TBBT's trace replay is mainly determined by disk I/O and CPU requirements. Each trace entry is represented by fewer than 100 bytes. The disk bandwidth requirement of TBBT's trace replay is fairly small, and the disk access pattern is large sequential read from the single *I/O thread*. Therefore the local disk is unlikely to be a bottleneck.

The CPU load of TBBT comprised of the send thread, receive thread, and the network subsystem inside the OS. When the Linux NFS server runs under a trace at peak throughput (1807 requests/sec), the measured CPU utilization and network bandwidth consumption for TBBT's trace player are 15% and 60.5 Mbps, respectively. When the same Linux NFS server runs under a SPECsfs benchmark at peak throughput (1231 requests/sec), the measured CPU utilization and network bandwidth consumption for the SPECsfs workload generator are 11% and 37.9 Mbps, respectively. These results suggest that TBBT's trace player is actually more efficient than SPECsfs's workload generator (in terms of CPU utilization per NFS operation) despite the fact that TBBT requires additional disk I/O for trace reads and incurs additional CPU overhead for dependency detection and error handling. TBBT out-performs SPECsfs because TBBT's trace player uses only three threads, whereas SPECsfs uses multiple processes and thus incurs excessive context switching and process scheduling overhead.

6.6 Limitations

There are several limitations associated with the proposed trace-driven approach to file system evaluation. First, for a given input workload, TBBT assumes the trace gathered from one file system is similar to that from the file system under

test. Unfortunately, this assumption does not always hold because even under the same client workload, different file servers based on the same protocol may produce very different traces. For example, file mount parameters such as read/write/readdir transfer sizes could have a substantial impact on the actual requests seen by an NFS server. Second, there is no guarantee that the heuristics used to scale up a trace actually make sense in practice. For example, if the bottleneck of a trace is accesses to a single file or directory, then identifying and cloning these accesses when replaying the trace is not feasible. Thirdly, it is generally not possible to deduce the entire file system hierarchy or its on-disk layout by passive tracing. Therefore the best one can do is to estimate the size distribution of those files that are never accessed during the tracing period and to apply synthetic aging techniques to derive a more realistic initial file system image. Again the file aging techniques proposed in this paper are not meant to reproduce the actual aging characteristics of the trace's target file system, but instead to provide users the flexibility to incorporate some file aging effects into their evaluations. Finally, trace-based evaluations are not as flexible as those based on synthetic benchmarks when it comes to exploring the entire workload space. Consequently, TBBT should be used to complement synthetic benchmarks rather than replace them.

6.7 Conclusion

The prevailing practice of evaluating the performance of a file system/server is based on synthetic benchmarks. Modern synthetic benchmarks do incorporate important characteristics of real file access traces and are capable of generating file access workloads that are representative of their target operating environments. However, they rarely fully capture the time-varying and often subtle characteristics of a specific site's workload. In this paper, we advocated a complementary

trace-driven file system evaluation methodology that compare the performance of a file system/server on a site by driving it with file access traces collected from that site. To support this methodology, we developed TBBT, the first comprehensive NFS trace analysis and replay tool. TBBT is a turn-key system that can take an NFS trace, properly initialize the target file server, drive it with a scaled version of the trace, and report latency and throughput numbers. TBBT addresses most, if not all, of the trace-driven workload generation problems, including correcting tracing errors, automatic derivation of initial file system from a trace, aging the file system to a configurable extent, preserving the dependencies among trace requests during replay, scaling a trace to a replay rate that can be higher or lower than the speed at which the trace is collected, and gracefully handling trace collection errors and implementation bugs in the test file system/server. Finally, we showed that all of these features can be implemented efficiently such that a single trace replay machine can stress a file server with state-of-the-art performance.

Our experiments demonstrated TBBT's usefulness for file system researchers. TBBT's most promising application, however, may be as a site-specific benchmarking tool for comparing competing file servers that use the same protocol. One could use TBBT to compare two or more file servers for a particular site by first collecting traces on the site, and then testing the performance of each of the file servers using the collected traces. Assuming traces collected on a site are indeed representative of that site's workload, comparing file servers using such a procedure may well be the best possible approach.

As for future work, we plan to extend TBBT to other network file access protocols. Although the current TBBT prototype can only replay NFS traces, its internal trace format is sufficiently generic to support traces collected on SMB, CIFS, and AFS servers. We plan to develop a converter that can translate CIFS traces collected from a SAMBA server into TBBT's internal format, and then use TBBT to

play back the resulting trace against a Windows-based CIFS server.

TBBT has been used by file system researchers from Harvard, CMU, Florida State University, Umass-Amherst and Texas A&M University. It is available at http://www.ecsl.cs.sunysb.edu/TBBT.

Chapter 7

Conclusions

In this chapter we first summarize the repairable system with a focus on the comparison of the repairable file system and the repairable storage system. Then we highlight the contributions of this dissertation, including both the repairable file/storage system and the traced-based file system benchmarking tool(Chapter 6). Finally we outline future research directions.

7.1 Summary of the repairable file and storage system

In this dissertation we expanded the horizon of traditional fault tolerance study by raising and addressing the issue of ”soft failures” - the data loss and recovery time caused by human mistakes, malicious attacks and untrusted software. The main approach to protecting against soft failures is to add versioning. This is in contrast to many hard failures addressed by traditional fault tolerance research in which the protection approach is to add redundancy. As the price of hardware drops quickly, soft failures play a larger and larger role in the cost of ownership.

To address soft failures, we proposed a repairable system framework. It consists of 1) comprehensive versioning techniques to keep all the data available during a protection window, and 2) dependency tracking/repair schemes to quickly identify the changes to be made and remove the damage to the system.

There are three aspects which are vital for the success of a repairable system: the accuracy of dependency tracking, the performance for current data access, and the integration with other fault tolerance techniques. While most of this dissertation effort has focused on the last two aspects, there has been excellent work by others - inspired by our original dependency tracking scheme - that strives to improve dependency tracking accuracy.

We developed two repairable systems. RFS is a repairable file system based on NFS protocol but with most of its functionalities implemented at the user level. In the process of optimizing system performance, we developed novel file system versioning techniques and built four prototypes: RFS-O, RFS-A, RFS-I and RFS-I+. Thorough evaluations were conducted on versioning techniques, dependency tracking schemes, and repairs with benchmarks, traces and adversaries. The result proves that a repairable system can match the performance of traditional file/storage system without much hardware cost. It also shows that NFS protocol is a very good interface to implement the functionality of a repairable system for both efficiency reasons and portability reasons.

Mariner is a repairable storage system based on iSCSI protocol and integrated with many other advanced features. From the perspective of comprehensive versioning, the storage block interface is much simpler than the file system interface. Therefore we could afford to implement it inside the kernel and integrate with Trail's fast logging techniques. The performance goal is not only to match but to exceed that of standard storage systems. At the core of Mariner is an efficient disk logging scheme. While we have performed some analysis with traces and have

found promising initial empirical results, more solid evaluations are still needed for the logging scheme.

Rather than focusing mainly on the performance aspects (as in the RFS project) of Mariner, we conducted more research on the integration of repairability and recoverability so that the system tolerates well both soft and hard failures. We discovered that the logging scheme used in a repairable system can greatly facilitate the tolerance of hard failures(true for both RFS and Mariner) and that recoverability can be obtained without performance cost. But due to the versioning metadata structures, there are additional complexities in the recoverability design. We proposed a generic fault tolerance model for 1-client-N-server storage system and applied this model to Mariner. Evaluation results show that Mariner can tolerate any single hard failure. The system remains available for both read and write requests during failure and recovery time. Overall the integration experience is natural, smooth, but requires careful thinking.

To connect Mariner's block level versioning ability to the proposed dependency tracking framework, we need a versioning file system on top of the versioning storage system. We have proposed a simple user-level versioning file system that serves this purpose. It has been implemented by other colleagues and is now under evaluation.

Having summarized both RFS and Mariner, let's now examine the essential commonalities and differences between the repairable file system and the repairable storage system. In addition to sharing the repairable system framework, RFS and Mariner use the same principle to solve one of the key issues in the design of a comprehensive versioning system: the size of versioning metadata, especially if that metadata must persist in memory. One technique often used is to increase the block (also known as "extent") size. Other techniques include delicate data structures, combining snapshotting and journaling, etc. In RFS and Mariner, the

metadata footprint is greatly reduced by separating cold data and warm data. RFS uses a base image, Mariner uses a beforeimage server. The implementation details are absolutely different but they share the same spirit.

As to the differences, the basic fact is of course that one functions at the file system level and the other at the storage system level. Many other differences naturally follow. A key issue that we'd like to point out is *whether logging can be accomplished without an extra disk seek for each update request.* It is important because of the strong performance requirement of a repairable system. Combining logging and the update request is relatively easy on the repairable storage system because of the simple update interface. There is only fully-aligned block write. The write data can be stored in the log record and the current data block map is used to keep track of the location of each logical block. The file system update interface is much more complex. There are directory and attribute updates in addition to the file data write. The file data write itself is also more complex because of the alignment issues. The complexity of the file system update interface makes it unfeasible to fully integrate the update request into the log record. But we also do not want to spend an extra synchronous disk write to commit the log record. Therefore, in RFS, the log record is committed in a delayed and batched fashion so that the cost is amortized. However, this does make RFS's recovery from hard failures less ideal. The last few writes could be lost because of unflushed log records.

The main lesson that we learned from this dissertation is that repairability is ready to be integrated into mainstream file and storage systems. The versioning functionality can be added at different interfaces; the NFS and block device interfaces are two good candidates. The repairable storage system is a simpler and more general solution that can protect more than just the NFS server. The repairable file system has a better connection to dependency tracking, it could be faster to repair, and it could perform better with file-system-metadata-update intensive workloads.

7.2 Dissertation Contributions

The research contributions of this dissertation are:

- Repairable System Framework

  We proposed a repairable file and storage system framework that can repair a system quickly with most of the useful work preserved after malicious attaches, honest human errors, or untrusted software. In the mean time the additional repairability should not degrade the system performance during normal time.

- RFS

  RFS is the first known repairable file system that can selectively undo undesirable side effects due to an attack or operator error. The time to repair a network file server after a malicious attack or an operational error is reduced to the level of minutes or hours with most of the useful work being preserved. The performance overhead (for the repairability capability) is less than 10%.

- Mariner

  Mariner is a repairable storage system based on SAN and iSCSI. In Mariner the repairability is seamlessly integrated with many other advanced features including low-latency write, reliable multicast, and fault tolerance. The system is expected to not only match but outperform standard iSCSI based SAN storage systems. We have evaluated Mariner's fault tolerance implementation with comprehensive single-failure scenarios. The system is available for read and write upon any single point of failure. The failed nodes all get repaired and the system always returns to a consistent state.

- TBBT: Trace-driven file system Benchmarking Toolkit

  TBBT is the first comprehensive NFS trace analysis and replay tool. TBBT

addresses most, if not all, of the trace-driven workload generation problems, including correcting tracing errors, automatic derivation of initial file system from a trace, aging the file system to a configurable extent, preserving the dependencies among trace requests during replay, scaling a trace to a replay rate that can be higher or lower than the original speed, and gracefully handling trace collection errors and implementation bugs in the test file system/server.

7.3 Directions for Future Research

The current research on repairable system is far from complete. As a short term goal we need to address the system scaling issue. For the soft failures that the repairable file and storage system is designed for, the appropriate protection should be on the level of hours and days if not weeks. Our current evaluation workloads have been around minutes and hours. Some of the system strategies such as the separation of warm data and cold data, has not been adequately evaluated due to the small scale of the test. To scale the system for workload 10 times higher, we may need to further optimize the versioning metadata organization to reduce its memory footprint ; we also need to scale up the TBBT trace player so that longer traces can be used for evaluation. After the system is scaled up, we could then study the relationship between *protection window*, *current data warm window* and system performance. This kind of long duration workload study is missing in most of the research on comprehensive versioning systems. It may expose system design issues which have caused the failure of many CDP startups.

More work could be done for tolerating arbitrary failure scenarios in both repairable file system and repairable storage system. One extension is to handle multiple failures, as discussed in Section 5.9. Another extension is to address ”site failure” - which can only be protected with remote replications. We performed

some rudimentary investigation and found that we probably can borrow from the Seneca remote replication protocol [34].

One future research topic is to enable a smooth transition of the versioning frequency to expand the protective power of a repairable file or storage system. As time goes by, the probability of undetected errors and intrusions decreases - such is the importance of frequent versioning. The current protection window represents a setting where the versioning frequency is either "as much as possible" (within the window) or 0 (outside the window). For example, we could store every version within a day, then every minute within a week, every hour within a month, etc. Such smooth transitioning may speed up the access time to very old data. It also enhances the protective power of the system. We have mentioned that the protection provided by repairable system is complementary to that provided by the archival system, backup system, or periodic checkpointing system such as Netapp's NFS Filer. However, it is not straightforward how to combine and integrate the protection from all these systems. If the versioning frequency can be adjusted to function smoothly over time and if the adjustments are automatic, versioning frequency could make the repairable system a complete solution for data protection.

In the area of dependency tracking, how to track application level dependencies and maintain application level consistencies in the face of failures remain as challenging topics, and they are becoming increasingly

important, especially in the context of federated applications.

Conceptually the repairable storage system can be used to protect DBMS servers. But there are also existing techniques that directly protect DBMS servers such as [53]. In this dissertation, we have compared the pro and cons of implementing versioning at the file system level vs. the storage system level. Similarly it is worthwhile to investigate and compare versioning at the database level vs the storage level.

There are many ways the current repairable system framework can be extended. Two promising paths: 1) extend the repairable file system to work with other network file access protocols such as CIFS; 2)extend the repairable storage system to work with other SAN protocols such as Fiber Channel.

As for future roles for the file system trace play tool TBBT, we plan to extend it to other network file access protocols. Although the current TBBT prototype can only replay NFS traces, its internal trace format is sufficiently generic to support traces collected on SMB, CIFS, and AFS servers. We plan to develop a converter that can translate CIFS traces collected from a SAMBA server into TBBT's internal format, and then use TBBT to playback the resulting trace against a Windows-based CIFS server.

Bibliography

[1] *The Advanced Maryland Automatic Network Disk Archiver*. (http://www.-amanda.org/).

[2] *Home of the tripwire open source project*. (http://www.tripwire.org/).

[3] *Server message block protocol (smb)*. (http://ourworld.compuserve.com/-homepages/timothydevans/smb.htm).

[4] *NFS: Network file system protocol specification.* Sun Microsystems, Mar 1989.

[5] D. Anderson and J. Chase. Fstress: A Flexible Network File Service Benchmark. Technical Report TR-2001-2002, Duke University, May 2002.

[6] E. Anderson. A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST'04)*, March 2004.

[7] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Monterey, CA, October 1991.

[8] M. A. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, San Fransisco, CA, January 1992.

[9] T. Bray. The Bonnie Disk Benchmark, 1990. http://www.textuality.com/bonnie/.

[10] A. Brown and D. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, July 2001.

[11] A. Brown and D. Patterson. Embracing failure: A case for recovery-oriented computing (roc). In *2001 High Performance Transaction Processing Symposium*, October 2001.

[12] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*, 1997.

[13] T. Chiueh and L. Huang. Track-based disk logging. In *Proceedings of 2002 International Conference on Dependable Systems and Networks*, 2002.

[14] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for linux. In *USENIX Annual Technical Conference (Freenix)*, 2004.

[15] M. D. Dahlin. A quantitative analysis of cache policies for scalable network file systems. In *First OSDI*, pages 267–280, 1994.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In

Proceedings of 5th Symposium on Operating Systems Design and Implementation, pages 211–224, New York, NY, USA, Dec 2002.

[17] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[18] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[19] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Seventeenth Annual Large Installation System Administration Conference (LISA'03)*, pages 73–85, San Diego, CA, October 2003.

[20] D. P. et al. Recovery-oriented computing (roc): Motivation, definition, techniques, and case studies. In *UC Berkeley Computer Science Technical Report*, March 2002.

[21] H. P. et al. Snapmirror: file system based asynchronous mirroring for disaster recovery. In *Conference on File and Storage Technologies*, pages 28–30, Monterey, CA, January 2002.

[22] P. A. et al. Surviving information warfare attacks on databases. In *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, pages 110–123, May, 1997.

[23] P. L. et al. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 8:7–40, Jan 2001.

[24] P. L. et al. Intrusion confinement by isolation in information systems. In *IFIP WG 11.3 13th Working Conference on Database Security*, pages 26–28, July 1999.

[25] M. D. Flouris and A. Bilas. Clotho: Transparent data versioning at the block i/o level. In *21st IEEE Conference on Mass Storage Systems and Technologies*, April 2004.

[26] S. L. Gaede. *Perspectives on the SPEC SDET benchmark*. Lone Eagle Systems Inc., January 1999. (http://www.specbench.org/osg/sdm91/sdet/-SDETPerspectives.html).

[27] G. A. Gibson, D. F. Nagle, K. Amiri, F. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–284. ACM Press, 1997.

[28] A. Goel, W. chang Feng, D. Maier, W. chi Feng, and J. Walpole. Forensix: A robust, high-performance reconstruction system. In *ICDCSW '05: Proceedings of the Second International Workshop on Security in Distributed Computing Systems (SDCS) (ICDCSW'05)*, pages 155–162, Washington, DC, USA, 2005. IEEE Computer Society.

[29] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[30] D. Grune, B. Berliner, and J. Polk. Concurrent Versions System. http://www.cvshome.org/.

[31] H. S. Gunawi, C. Rubio-Gonzlez, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. Eio: error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[32] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *USENIX winter 1994 conference*, pages 235–246, Chateau Lake Louise, Banff, Canada, January 1994.

[33] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[34] M. Ji, A. Veitch, and J. Wilkes. Seneca: Remote mirroring done write. In *USENIX 2003 Annual Technical Conference*, Jun 2003.

[35] R. J.T. *Human error*. Cambridge University Press, New York, 1990.

[36] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance, October 1997.

[37] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Trans. Comput. Syst.*, 23(1):51–76, 2005.

[38] P. Leach and D. Perr. *CIFS: A common internet file system*. Microsoft Interactive Developer, November 1996.

[39] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[40] S. Lin, M. Lu, and T. cker Chiueh. Transparent reliable multicast for ethernet-based storage area networks. In *The 6th IEEE Symposium on Network Computing and Apllications*, July 2007.

[41] P. Liu. Architectures for intrusion tolerant database systems. In *submitted*, pages 110–123, 2002.

[42] M. Lu, S. Lin, and T. cker Chiueh. Efficient logging and replication techniques for comprehensive data protection. In *24th IEEE Conference on Mass Storage Systems and Technologies*, September 2007.

[43] M. Lu, S. Lin, and T. cker Chiueh. An incremental file system consistency checker for block-level cdp systems. In *27th International Symposium on Reliable Distributed Systems*, Oct 2008.

[44] D. Mazires. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, pages 261–274, June 2001.

[45] R. McDougall. A new methodology for characterizing file system performance using a hybrid of analytic models and a synthetic benchmark. In *Work in Progress Session in 3rd Usenix Conference on File and Storage Technologies*, March 2004.

[46] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[47] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[48] I. Mendocino. RealTime: Near-Instant Recovery to Any Point in Time. http://www.mendocino.com/.

[49] M. Mesnier, E. Thereska, D. Ellard, G. R. Ganger, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 44–51, May 2004.

[50] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March 2004.

[51] U. of New Hampshire InterOperability Laboratory. iSCSI Initiator and Target Reference Implementations for Linux. http://sourceforge.net/projects/unh-iscsi/.

[52] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP'85)*, pages 15–24, Orcas Island, WA, December 1985.

[53] D. Pilania and T. cker Chiueh. Design, implementation and evaluation of an intrusion resilient database system. In *Proceedings of International Conference on Data Engineering (ICDE 2004)*, Tokyo, Japan, April 2005.

[54] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *USENIX 2000 Technical Conference*, pages 41–54, San Diego, CA, 2000.

[55] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[56] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.

[57] M. Seltzer, G. Ganger, M. K. McKusick, K. Smith, C. Soules, and C. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.

[58] S. Sharma, K. Gopalan, S. Nanda, and T. cker Chiueh. Viking: A multi-spanning-tree ethernet architecture for metropolitan area and cluster networks. In *Proceedings of IEEE INFOCOM*, 2004.

[59] A. Smirnov and T. cker Chiueh. A portable implementation framework for intrusion-resilient database management systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004.

[60] K. A. Smith and M. I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[61] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *2nd USENIX Conference on File and Storage Technologies*, pages 43–58, Mar 2003.

[62] SPEC SFS (System File Server) Benchmark, 1997. http://www.spec.org/osg/sfs97r1/.

[63] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 OSDI Conference*, October 2000.

[64] W. Sun, Z. Liang, V.N.Venkatakrishnan, and R.Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.

[65] S. Sundararaman, G. Sivathanu, and E. Zadok. Selective versioning in a secure disk system. In *Proceedings of the 17th USENIX Security Symposium*, pages 259–274, San Jose, CA, July-August 2008. USENIX Association.

[66] I. Symantec. Veritas netbackup realtime protection administrator's guide , version 6.5, april 2008. .

[67] M. Szeredi. Filesystem in Userspace. http://www.fuse.sourceforge.net/.

[68] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[69] W. Vogels. File System Usage in Windows NT. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, SC, December 1999.

[70] P. Widmayer, B. Becker, and et al. *An asymptotically optimal multiversion b-tree*. Very Large Data Bases Journal, 1996.

[71] B. L. Wolman and T. M. Olson. IOBENCH: a system independent IO benchmark. *Computer Architecture News*, 17(5):55–70, September 1989.

[72] I. XOsoft. Enterprise rewinder: Product suite for continuous data protection (cdp). http://www.xosoft.com/.

[73] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

[74] N. Zhu, J. Chen, and T. Chiueh. TBBT:Scalable and Accurate Trace Replay for File Server Evaluation. In *Proceedings of the FAST '05 Conference on File and Storage Technologies*, Sanfrancisco, California, Dec 2005.

[75] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, June 2003.

[76] N. Zhu and T. Chiueh. Portable and efficient continuous data protection for network file servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'07)*, pages 687–697, 2007.